SpeedShop User’s Guide

007–3311–006

New Features

SpeedShop User's Guide 007–3311–006

This revision of the manual includes bug fixes and the following updates for the SpeedShop 1.4 release:

- Support in `ssrun`(1) for displaying Message Passing Interface (MPI) trace experiments.
- A conversion command, `ssfilter`(1), that translates MPI experiment files into vampir trace format.
- A new command, `ssaggregate`(1), that combines multiple experiment files into a single experiment file.
- Support for hardware counter experiments on systems with R12000 processors.

Record of Revision

Version	*Description*
1.3.2	August 1998 Brings the manual into conformance with the 1.3.2 version of the SpeedShop software.
1.4	April 1999 Supports the 1.4 version of the SpeedShop software.

Contents

Page

About This Guide

The *SpeedShop User's Guide* describes and illustrates methods for measuring program performance using SpeedShop commands such as `ssrun`(1) and `prof`(1). It contains tutorials that generate performance statistics for C and Fortran programs.

This manual is a user's guide for the SpeedShop performance tools, release 1.4.

It contains the following chapters:

- Chapter 1, page 1, provides a general introduction to performance analysis concepts and techniques, plus an overview of the SpeedShop tools.
- Chapter 2, page 13, provides a tutorial on how to collect performance data and generate reports for a C program.
- Chapter 3, page 31, provides a tutorial on how to collect performance data and generate reports for Fortran programs running on single-processor machines.
- Chapter 4, page 49, describes the types of experiments that can be performed using SpeedShop tools.
- Chapter 5, page 65, describes how to use the `ssusage`(1) command to collect information about a program's machine resource usage.
- Chapter 6, page 67, explains in detail how to set up and run experiments using `ssrun`(1), and explains how to use caliper points to generate reports for part of a program.
- Chapter 7, page 93, explains how to generate reports from performance data using `prof`(1).
- Chapter 8, page 121, explains how to use `pixie`(1) and `prof` directly, without invoking `ssrun`(1).
- Chapter 9, page 133, explains how to use the `thrash`(1) and `squeeze`(1) commands to determine the memory usage, or working set, of your application. It also includes commands to print performance data files.

Related Publications

The following documents contain additional information that may be helpful:

- *Developer Magic: Performance Analyzer User's Guide*
- *C Language Reference Manual*
- *C++ Language System Library*
- *C++ Language System Overview*
- *C++ Language System Product Reference Manual*
- *C++ Programmer's Guide*
- *ProDev ProMP User's Guide*
- *Developer Magic: Debugger User's Guide*
- *Developer Magic: Static Analyzer User's Guide*
- *Developer Magic: ProDev WorkShop Overview*
- *Fortran 77 Language Reference Manual*
- *MIPSpro 7 Fortran 90 Commands and Directives Reference Manual*
- *Fortran Language Reference Manual*, Volumes 1–3

Obtaining Publications

Silicon Graphics maintains publications information at the following World Wide Web site:

`http://techpubs.sgi.com/library`

The preceding website contains information that allows you to browse documents online, order documents, and send feedback to Silicon Graphics.

To order a printed Silicon Graphics document, call 1 800 627 9307.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and part number of the document with your comments.

You can contact us in any of the following ways:

- Contact your customer service representative and ask that a PV be filed.
- Send electronic mail to the following address:

  `techpubs@sgi.com`

- Send a facsimile to the attention of “Technical Publications” at fax number +1 650 932 0801.
- Use the Suggestion Box form on the Technical Publications Library World Wide Web page:

  `http://techpubs.sgi.com/library/`

- Call the Technical Publications Group, through the Technical Assistance Center, using the following number:

  1 800 800 4SGI

- Send mail to the following address:

Technical Publications
Silicon Graphics, Inc.
1600 Amphitheatre Pkwy.
Mountain View, California 94043–1351

We value your comments and will respond to them promptly.

Introduction to Performance Analysis [1]

This chapter provides a brief introduction to performance analysis techniques for Silicon Graphics systems and describes how to use them with SpeedShop to solve performance problems. It includes the following sections:

- Sources of Performance Problems. This section provides a general overview of potential performance problems. See Section 1.1, page 1.
- Fixing Performance Problems. This section discusses how you can use SpeedShop to determine what the problems are. See Section 1.2, page 3.
- SpeedShop Tools. This section lists SpeedShop commands, experiment types, and libraries. See Section 1.3, page 3.
- Using SpeedShop Tools for Performance Analysis. This section steps you through the steps to take when using SpeedShop. See Section 1.4, page 8.

1.1 Sources of Performance Problems

To tune a program's performance, you need to determine its consumption of machine resources. At any point in a process, there is one limiting resource controlling the speed of execution. Processes can be slowed down by any of the following:

- CPU speed and availability
- I/O processing
- Memory size and availability
- Bugs

Performance problems may span the entire run of a process, or they may occur in just a small portion of the program. For example, a function that performs a lot of I/O processing might be called regularly as the program runs, or a particularly CPU-intensive calculation might occur in just one portion of the program. When there are performance problems in a small portion of the program, collect data for just that part of the program.

Because programs exhibit different behavior during different phases of operation, you need to identify the limiting resource for each phase. A program can be I/O-bound while it reads in data, CPU-bound while it performs computation, and I/O-bound again in its final stage while it writes out data.

Once you have identified the limiting resource in a phase, you can perform an in-depth analysis to find the problem. After you have solved that problem, you can check for other problems within the same or other phases—performance analysis is an iterative process.

1.1.1 CPU-Bound Processes

A *CPU-bound* process spends its time in the CPU and is limited by CPU speed and availability. To improve performance on CPU-bound processes, streamline your code using one or more of the following techniques:

- Modifying algorithms
- Reordering code to avoid interlocks
- Removing nonessential steps
- Blocking to keep data in cache and registers
- Using alternative algorithms

1.1.2 I/O-Bound Processes

An *I/O-bound* process has to wait for I/O to complete and may be limited by disk access speeds or memory caching. To improve the performance of I/O-bound processes, try one of the following techniques:

- Improving overlap of I/O with computation
- Optimizing data usage to minimize disk access
- Using data compression

1.1.3 Memory-Bound Processes

A *memory-bound* program continuously swaps out pages of memory. Page thrashing is often due to accessing virtual memory on a haphazard rather than strategic basis. To fix a memory-bound process, try to improve the memory reference patterns by increasing local references, or, if possible, decrease the memory used by the program. For more information on paging activity, see the `osview`(1) man page or the `-w` option on the `sar`(1) man page.

1.1.4 Bugs

Certain bugs can cause performance problems. Examples include:

- The program is unnecessarily reading the same information from the same file more than once.
- Floating point exceptions are slowing down the program.
- Old code has not been completely removed.
- The program is leaking memory (making `malloc()` calls without the corresponding calls to `free()`).

1.2 Fixing Performance Problems

The SpeedShop performance tools described in this manual can help you to identify specific performance problems described later in this chapter. However, the techniques described in this manual are only a part of performance tuning. Other areas that you can tune, but that are outside the scope of this document, include graphics, I/O, the kernel, system parameters, memory, and real-time system calls.

Although it may be possible to obtain short-term speed increases by relying on unsupported or undocumented quirks of the compiler, it is a bad idea to do so. Any such “features” may break in future compiler releases. The best way to produce efficient code that will remain efficient is to follow good programming practices. In particular, choose good algorithms and leave the details to the compiler.

1.3 SpeedShop Tools

The SpeedShop tools allow you to run experiments and generate reports that track down the sources of performance problems. SpeedShop consists of a set of commands that can be run in a shell, an application programming interface (API) to provide some control over data collection, and a number of libraries to support the commands.

This section provides an overview of the tools by first discussing the main commands, then providing more detail on additional commands, experiment types, libraries, the SpeedShop API, and supported programs and languages.

1.3.1 Main Commands

SpeedShop provides the commands listed in Table 1.

Table 1. SpeedShop Main Commands

Command	Description
`ssusage`	Collects information about your program’s use of machine resources. Output from `ssusage` can be used to determine where most resources are being spent.
`ssrun`	Allows you to run experiments on a program to collect performance data. It establishes the environment to capture performance data for an executable, creates a process from the executable (or from an instrumented version of the executable) and runs it. Input to `ssrun` consists of an experiment type, control flags, the name of the target, and the arguments to be used in executing the target.
`prof`	Analyzes the performance data you have recorded using `ssrun` and provides formatted reports. `prof` detects the type of experiment you have run and generates a report specific to the experiment type. You can also use the `cvperf` command to display the data through the WorkShop graphic user interface.

1.3.2 Additional Commands

SpeedShop provides the additional commands shown in Table 2.

Table 2. SpeedShop Additional Commands

Command	Description
`pixie`	Makes basic block counting experiments possible. If you use `ssrun`, you will not usually need to call `pixie` directly.
`fbdump`	Prints out the formatted contents of compiler feedback files generated by `prof`.

Command	Description
`squeeze`	Allocates a region of virtual memory and locks the virtual memory down into real memory, making it unavailable to other processes.
`thrash`	Allows you to allocate a block of memory, then access the allocated memory to explore system paging behavior.
`ssdump`	Prints out formatted performance data that was collected while running `ssrun`. This program is included for SpeedShop debugging purposes. You do not normally need to use it.

1.3.3 Experiment Types

The following are the most popular experiments using the `ssrun` command. (For the complete list of experiments, see the `ssrun`(1) man page.)

- Providing information on a program's CPU usage using statistical program counter sampling with `pcsamp` experiments.

  Data is measured by periodically sampling the program counter of the target executable when it is executing in the CPU. The program counter shows the address of the currently executing instruction in the program. The data that is obtained from the samples is translated to a time that can be displayed at the function, source line, and machine instruction levels. The actual *CPU time* is calculated by multiplying the number of times a specific address is found in the PC by the amount of time between samples. (For a definition of CPU time, wall-clock time, and process virtual time, see the Glossary.)

- Displaying information from a variety of hardware counters using statistical sampling with `hwc` experiments.

  Hardware counter experiments are available on R10000 and R12000 systems that have built-in hardware counters. Data is measured by counting each time the specified hardware counter exceeds its maximum value, or overflows. You can specify the hardware counter and the overflow interval you want to use. (For more information on the hardware counter experiments, see Section 4.6, page 55.)

- Displaying a program's CPU time (see the Glossary) by statistical call stack profiling with `usertime`.

Data is measured by periodically sampling the call stack. The program's call stack data is used to attribute *exclusive* user time to the function at the bottom of each call stack (that is, the function being executed at the time of the sample), and to attribute *inclusive* user time to all the functions above the one currently being executed. Exclusive time is the execution time of a given function but not any functions that function calls, while inclusive time is the execution time both of a given function and of any functions called by that function.

- Displaying a program's best possible time by counting basic blocks with `ideal`.

  Data is measured by counting the number of executions of each *basic block* and calculating an ideal CPU time for each function. This involves instrumenting the program to divide the code into basic blocks, which are consecutive sequences of instructions with a single entry point, a single exit point, and no branches into or out of the sequence. Instrumentation also records a count of all dynamic (function-pointer) calls. You can compare this ideal time with the times returned by other experiments to measure the performance of your code against its potential (see Section 4.4.4, page 54). Because an exact count of every instruction in your program is recorded, you can also use the `ideal` experiment to determine the efficiency of your algorithm and identify any code that is not executed.

- Tracing floating-point exceptions with `fpe`.

  A floating-point exception trace collects each floating-point exception, including the exception type and the call stack, at the time of the exception. `prof`(1) generates a report showing inclusive and exclusive floating-point exception counts.

1.3.4 SpeedShop Libraries

Versions of the SpeedShop libraries `libss.so` and `libssrt.so` are available to support applications built using shared libraries (called *dynamic shared objects*, or DSOs) only and the old 32-bit, new 32-bit, or 64-bit application binary interfaces (ABIs).

Table 3 provides information about the different SpeedShop libraries.

Table 3. SpeedShop Libraries

Library	Description
`libss.so`	A shared library (DSO) that supports `libssrt.so`. The `libss.so` data normally appears in experiment results generated with `prof`.
`libssrt.so`	A shared library (DSO) that is linked in to the program you specify when you run an experiment. All the performance data collection with the SpeedShop system is done within the target processes by exercising various pieces of functionality using `libssrt`. Data from `libssrt.so` does not normally appear in performance data reports generated with `prof`.
`libfpe_ss.so`	Supplements the standard `libfpe.so` for the purposes of collecting floating-point exception data. See the `fpe_ss`(3) man page for more information.
`libmalloc_ss.so`	Inserts versions of `malloc` routines from `libc.so.1` that allow tracing all calls to `malloc`, `free`, `realloc`, `memalign`, and `valloc`. See the `malloc_ss`(3) man page for more information.
`libpixrt.so`	A shared library (DSO) used by programs that have been processed by the `pixie`(1) command.

1.3.5 API

The SpeedShop application programming interface (API) allows you to use `ssrt_caliper_point` to set caliper points in your source code. See Section 6.8, page 86, for information on using caliper points. For information on other API functions, see the `ssapi`(3) man page.

1.3.6 Supported Programming Models and Languages

The SpeedShop tools support programs with the following characteristics:

- Shared libraries (DSOs).
- Unstripped executables.
- Executables that call `fork`(2), `sproc`(2), `system`(3F), or `exec`(2).

- Executables using supported techniques for opening, closing, and delay-loading DSOs.
- C, C++, Fortran (Fortran 77 and Fortran 90), or Ada 95 source code.
- Power Fortran and Power C source code. `prof` understands the syntax and semantics of the multiprocessing run time and displays the data accordingly.
- `pthreads`, supported with data on a per-program basis.
- Message Passing Interface (MPI) or other message-passing paradigms. Currently supported by providing data on the behavior of each process. The behavior of the MPI library itself is monitored just like any other user-level code.
- The OpenMP collection of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism.

1.4 Using SpeedShop Tools for Performance Analysis

Performance tuning typically consists of

- Examining machine resource usage
- Breaking down the process into phases
- Identifying the resource bottleneck within each phase
- Correcting the cause of the bottleneck

Generally, you run the first experiment to break your program down into phases and run subsequent experiments to examine each phase individually. After you have solved a problem in a phase, you should re-examine machine resource usage to see if there is further opportunity for performance improvement.

The general steps for a performance analysis cycle are as follows:

1. Build the application.
2. Run experiments on the application to collect performance data.
3. Examine the performance data.
4. Generate an improved version of the program.
5. Repeat as needed.

To accomplish this using SpeedShop tools, do the following:

- Use `ssusage` to capture information on your program's use of machine resources.
- Use `ssrun` to capture different types of performance data over either your entire program or parts of the program. `ssrun` can be used in conjunction with `dbx`(1) or `cvd`(1), the WorkShop debugger.
- Use `prof` to analyze the data and generate reports.

1.4.1 Using `ssusage` to Evaluate Machine Resource Use

To determine overall resource usage by your program, run the program with `ssusage`. The results of this command allow you to identify high-user CPU time, high-system CPU time, high I/O time, and a high degree of paging. The `ssusage`(1) command has the following format:

`ssusage` *executable_name executable_args*

From the `ssusage` output, you can decide which experiments to run to collect data for further study. For more information on `ssusage`, see Chapter 5, page 65, or see the `ssusage`(1) man page.

1.4.2 Using `ssrun` and `prof` to Gather and Analyze Performance Data

This section describes the steps involved in a performance analysis cycle when using the line-based interface to the SpeedShop tools: the `ssrun` and `prof` commands.

You can also call the commands individually. For example, if you are planning to perform basic block counting experiments that involve instrumenting the executable, you can do this by calling `ssrun` with the appropriate experiment type.

To perform a performance analysis, follow these general steps:

1. Build the executable.

   You can usually build the executable as you would normally. See Section 6.1, page 67, for information on how to build the executable.

2. Specify caliper points if you want to analyze data for only a portion of your program. See Section 1.4.3, page 11, for more information.

3. To collect performance data, issue the `ssrun` command with the following parameters:

   `ssrun` *flags exp_type executable_name executable_args*

   The following options are available with the `ssrun` command:

flags	One or more valid flags. For a complete list of flags, see the `ssrun`(1) man page.
exp_type	Experiment name.
executable_name	Executable name.
executable_args	Arguments to the executable

   Use the information in the following list to determine which experiments to run. Each performance problem is followed by one or more experiment types:

Problem	**Experiments**
High-user CPU time	`usertime`, `pcsamp` (four variants), hardware counter experiments, or `ideal`
High-system CPU time	If floating-point exceptions are suspected: `fpe` trace
High I/O time	`ideal`, then examine counts of I/O routines
High paging rates	`ideal`, then `prof -cordfb` and `cord` to rearrange procedures. For more information on rearranging code regions, see the *MIPSpro Compiling and Performance Tuning Guide*.

   For each process of the executable, the experiment data is stored in a file with a name in the following form:

   executable_name.exp_type.id

   The experiment ID consists of one or two letters designating the process type, followed by the process ID number. An example of a name is:

   ```
   generic.ideal.m10966
   ```

   See Table 4 for letter codes and descriptions.

Table 4. Letter Codes in Process Experiment ID Numbers

Letter Codes	Description
`m`	Master process created by `ssrun`
`p`	Process created by a call to `sproc()`
`f`	Process created by a call to `fork()`
`s`	Process created by a call to `system()`
`e`	Process created by a call to `exec()`
`fe`	Process created by a call to `fork()` and `exec()`

For more information on the `ssrun` command, see Chapter 6, page 67, or view the `ssrun`(1) man page.

4. To generate a report from the experiment, issue `prof` with the following parameters:

   `prof` *options data_file*

options	One or more valid options. For a complete list of flags, see the `prof`(1) man page.
data_file	The name of the file in which the experiment data was recorded.

   For more information on using `prof`, see Chapter 7, page 93, or see the `prof`(1) man page.

1.4.3 Collecting Data for Part of a Program

If you have a performance problem in only one part of your program, consider collecting performance data for just that part. You can do this by setting caliper points around the problem area when running an experiment, then using the `prof -calipers` option to generate a report for the problem area or using the calipers time line in the `cvperf`(1) window of WorkShop to view the area through a graphic user interface.

You can record caliper points using one of the following methods:

- Direct calls to the SpeedShop API.
- The caliper signal environment.

- A debugger such as the ProDev WorkShop debugger.
- Periodic caliper points with pollpoint caliper points.

For more information on using calipers, see Section 6.8, page 86.

Tutorial for C Users [2]

This chapter provides a tutorial that shows you how to gather and analyze performance data in a C program, using SpeedShop tools. The tutorial covers these topics:

- Section 2.1, page 13, introduces the sample program and explains the different scenarios in which it will be used.
- Section 2.2, page 15, steps you through the necessary setup for running the experiment.
- Section 2.3, page 15, steps you through five different experiments, discussing first how to do the experiments, then how to interpret the results.

Note: Because of inherent differences between systems and because of concurrent processes that may be running on your system, your experiment will produce different results from the one in this tutorial. However, the basic structure of the results should be the same.

2.1 Tutorial Overview

This tutorial uses a sample program called `generic.` There are two versions of the program:

`generic` directory	Contains files for the n32-bit ABI
`generico32` directory	Contains files for the (old) 32-bit ABI

When you work with the tutorial, choose the version of `generic` most appropriate for your system. A good guideline is to choose the version that corresponds to the way you expect to develop your programs.

This tutorial was written and tested using the version of `generic` in the `generic` directory.

2.1.1 Contents of the `generic` Program

The `generic` program was designed as a test and demonstration application. It contains code to run scenarios that each test a different area of SpeedShop. The version of `generic` used in this tutorial performs scenarios that:

- Build a linked list of structures

- Use a lot of user time
- Scan a directory and run the `stat` command on each file
- Perform file I/O
- Generate a number of floating-point exceptions
- Load and call a routine in a DSO

2.1.2 Output from the `generic` Program

Output from the program looks like the following:

```
0:00:00.000 ======== (27173)             Begin script Fri  06 Feb 1998
15:03:31.
        begin script ‘ll.u.cvt.d.i.f.dso’
0:00:00.002 ======== (27173)        start of linklist Fri  06 Feb 1998
15:03:31.
        linklist completed.
0:00:00.003 ======== (27173)         start of usrtime Fri  06 Feb 1998
15:03:31.
        usertime completed.
0:00:25.572 ======== (27173)         start of cvttrap Fri  06 Feb 1998
15:03:57.
        cvttrap completed, y = 2.60188e+14, z = 2.60188e+14.
0:00:25.806 ======== (27173)         start of dirstat Fri  06 Feb 1998
15:03:57.
        dirstat of /usr/include completed, 304 files.
0:00:26.618 ======== (27173) start of iofile -- stdio Fri  06 Feb 1998
15:03:58.
        stdio iofile on /unix completed, 7307988 chars.
0:00:26.864 ======== (27173)        start of fpetraps Fri  06 Feb 1998
15:03:58.
        fpetraps completed.
0:00:26.865 ======== (27173)          start of libdso Fri  06 Feb 1998
15:03:58.
dlslave_init executed
dlslave_routine executed
        slaveusertime completed, x = 5000000.000000.
        libdso: dynamic routine returned 13
        end of script ‘ll.u.cvt.d.i.f.dso’
0:00:27.972 ======== (27173)               End script Fri  06 Feb 1998
15:03:59.
```

2.2 Tutorial Setup

Copy the program to a directory where you have write permission and compile it so that you can use it in the tutorial.

1. Change to the `/usr/demos/SpeedShop` directory.
2. Copy the appropriate `generic` directory and its contents to a directory where you have write permission:

   `cp -r generic` *your_dir*
3. Change to the directory you just created:

   `cd` *your_dir*`/generic`
4. Compile the program, by entering:

   ```
   make all
   ```

   This provides an executable for the experiment.

2.3 Analyzing Performance Data

This section explains how to run the following experiments on the `generic` program, generate the experiment's results, and interpret the results:

- `usertime`. As a first cut at optimization, this may be the most useful experiment. It breaks down a program into its functions and returns the *CPU time* used in each. See Section 2.3.1, page 16.
- `pcsamp`. This experiment uses a different method to return the CPU time. See Section 2.3.2, page 19.
- `dsc_hwc`. This experiment counts the number of times a required data item was not in secondary data cache. If the data item is not in secondary data cache, it must be fetched from memory, which requires more time. See Section 2.3.3, page 22.
- `ideal`. This experiment calculates the best time achievable. See Section 2.3.4, page 24.
- `fpe`. This experiment counts the number of floating-point exceptions in each function. See Section 2.3.5, page 29.

You can follow the tutorial from start to finish, or you can choose the experiment you want to perform.

2.3.1 A `usertime` Experiment

This section explains how to perform a `usertime` experiment. The `usertime` experiment allows you to gather data on the amount of CPU time spent in each function in your program. For more information on `usertime`, see Section 4.2, page 50. For definitions of *CPU time*, *wall-clock time*, and *process-virtual time*, see the Glossary.

2.3.1.1 Performing a `usertime` Experiment

From the command line, enter the following:

```
ssrun -usertime generic
```

This command starts the experiment. Output from `generic` and from `ssrun` is printed to `stdout`, as shown in the following example. A data file is also generated. The name consists of the process name (`generic`), the experiment type, `usertime`, and the experiment ID. In this example, the file name is `generic.usertime.m10981`.

```
0:00:00.000 ======== (10981)            Begin script Mon  02 Feb 1998
11:05:02.
        begin script ‘ll.u.cvt.d.i.f.dso’
0:00:00.002 ======== (10981)        start of linklist Mon  02 Feb 1998
11:05:02.
        linklist completed.
0:00:00.003 ======== (10981)         start of usrtime Mon  02 Feb 1998
11:05:02.
        usertime completed.
0:00:22.948 ======== (10981)         start of cvttrap Mon  02 Feb 1998
11:05:25.
        cvttrap completed, y = 2.60188e+14, z = 2.60188e+14.
0:00:23.156 ======== (10981)         start of dirstat Mon  02 Feb 1998
11:05:25.
        dirstat of /usr/include completed, 304 files.
0:00:23.937 ======== (10981) start of iofile -- stdio Mon  02 Feb 1998
11:05:26.
        stdio iofile on /unix completed, 7307988 chars.
0:00:24.777 ======== (10981)        start of fpetraps Mon  02 Feb 1998
11:05:27.
        fpetraps completed.
0:00:24.777 ======== (10981)          start of libdso Mon  02 Feb 1998
11:05:27.
dlslave_init executed
```

```
dlslave_routine executed
        slaveusertime completed, x = 5000000.000000.
        libdso: dynamic routine returned 13
        end of script ‘ll.u.cvt.d.i.f.dso’
0:00:25.866 ======== (10981)               End script Mon  02 Feb 1998
11:05:28.
```

2.3.1.2 Generating a Report

To generate a report on the data collected, enter the following at the command line:

`prof` *your_output_file_name* `> usertime.results`

The `prof` command prints results to `stdout`. Because of line width restrictions, the DSO, file name, and line number information at the end of each line is wrapped to the next line in the following sample output.

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Mon Feb  2 11:07:15 1998
   prof generic.usertime.m10981
                 generic (n32): Target program
                      usertime: Experiment name
                         ut:cu: Marching orders
                 R4400 / R4000: CPU / FPU
                             1: Number of CPUs
                           175: Clock frequency (MHz.)
  Experiment notes--
          From file generic.usertime.m10981:
        Caliper point 0 at target begin, PID 10981
                   /usr/demos/SpeedShop/progs.etc/linpack.demos/c/generic
        Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of statistical callstack sampling data (usertime)--
                           809: Total Samples
                             0: Samples with incomplete traceback
                        24.270: Accumulated Time (secs.)
                          30.0: Sample interval (msecs.)
-------------------------------------------------------------------------
Function list, in descending order by exclusive time
-------------------------------------------------------------------------
[index]  excl.secs excl.%   cum.%  incl.secs incl.%   samples procedure(dso:file,line)

   [4]     22.770  93.8%   93.8%     22.770  93.8%        759  anneal
(generic: generic.c, 1573)
   [6]      1.020   4.2%   98.0%      1.020   4.2%         34  slaveusrtime
(dlslave.so: dlslave.c, 22)
   [9]      0.210   0.9%   98.9%      0.210   0.9%          7  cvttrap
(generic: generic.c, 317)
  [12]      0.120   0.5%   99.4%      0.120   0.5%          4  __read
(libc.so.1: read.s, 20)
  [14]      0.090   0.4%   99.8%      0.090   0.4%          3  _xstat
(libc.so.1: xstat.s, 12)
  [10]      0.030   0.1%   99.9%      0.180   0.7%          6  iofile
(generic: generic.c, 464)
  [11]      0.030   0.1%  100.0%      0.150   0.6%          5  fread
(libc.so.1: fread.c, 34)
   [1]      0.000   0.0%  100.0%     24.270 100.0%        809  __start
(generic: crt1text.s, 101)
   [2]      0.000   0.0%  100.0%     24.270 100.0%        809  main
(generic: generic.c, 101)
   [3]      0.000   0.0%  100.0%     24.270 100.0%        809  Scriptstring
(generic: generic.c, 184)
   [5]      0.000   0.0%  100.0%     22.770  93.8%        759  usrtime
(generic: generic.c, 1377)
  [15]      0.000   0.0%  100.0%      0.090   0.4%          3  dirstat
(generic: generic.c, 348)
  [16]      0.000   0.0%  100.0%      0.090   0.4%          3  _stat
(libc.so.1: stat.c, 31)
  [13]      0.000   0.0%  100.0%      0.120   0.5%          4  _read
(libc.so.1: readSCI.c, 27)
   [7]      0.000   0.0%  100.0%      1.020   4.2%         34  libdso
(generic: generic.c, 619)
   [8]      0.000   0.0%  100.0%      1.020   4.2%         34  dlslave_routine
(dlslave.so: dlslave.c, 7)
```

2.3.1.3 Analyzing the Report

The report shows information for each function. The meanings of the column headings are described below:

- The `index` column assigns a number to each function.
- The `excl.secs` column shows how much time, in seconds, was spent in the function itself (exclusive time). For example, less than one hundredth of a second was spent in `__start`, but 0.03 of a second was spent in `fread`.

- The `excl.%` column shows the percentage of a program's total time that was spent in the function. The `anneal` function consumed 93.8% of the program's time.
- The `cum.%` column shows the percentage of the complete program time that has executed in the routines listed so far.
- The `incl.secs` column shows how much time, in seconds, was spent in the function and descendents of the function. For example, 0.21 seconds were spent in `cvttrap` and the functions that were called by it.
- The `incl.%` column shows the cumulative percentage of inclusive time spent in each function and its descendents. For example, 93.8% of the time was spent in `anneal` and all the functions that were called through it.
- The `samples` column shows how many samples were taken when the process was executing in the function and in all of the function's descendants.
- The `procedure (dso:file, line)` columns list the function name, its DSO name, its file name, and its line number. For example, the top line reports statistics for the function `anneal`, the DSO `generic`, in the file `generic.c`, at line 1573.

2.3.2 A pcsamp Experiment

This section explains how to perform a `pcsamp` experiment. The `pcsamp` experiment allows you to gather information on actual CPU time for each function in your program. For more information on `pcsamp`, see Section 4.3, page 51. For definitions of *CPU time*, *wall-clock time*, and *process-virtual time*, see the Glossary.

From the command line, enter the following:

```
ssrun -fpcsamp generic
```

This starts the experiment. The `f` prefix is added to `pcsamp` for this program because the program runs quickly and does not gather much data using the default `pcsamp` experiment name; adding the `f` prefix results in more data samples. Output from `generic` and from `ssrun` is printed to `stdout`, as shown in the example below.

A data file is also generated. The name consists of the process name (`generic`), the experiment type (`fpcsamp`), and the experiment ID. In this example, the file name is `generic.fpcsamp.m11140`.

```
0:00:00.000 ======== (11140)             Begin script Mon  02 Feb 1998
10:58:41.
        begin script 'll.u.cvt.d.i.f.dso'
0:00:00.003 ======== (11140)        start of linklist Mon  02 Feb 1998
10:58:41.
        linklist completed.
0:00:00.004 ======== (11140)         start of usrtime Mon  02 Feb 1998
10:58:41.
        usertime completed.
0:00:22.437 ======== (11140)         start of cvttrap Mon  02 Feb 1998
10:59:03.
        cvttrap completed, y = 2.60188e+14, z = 2.60188e+14.
0:00:22.638 ======== (11140)         start of dirstat Mon  02 Feb 1998
10:59:03.
        dirstat of /usr/include completed, 304 files.
0:00:23.407 ======== (11140) start of iofile -- stdio Mon  02 Feb 1998
10:59:04.
        stdio iofile on /unix completed, 7307988 chars.
0:00:23.750 ======== (11140)        start of fpetraps Mon  02 Feb 1998
10:59:04.
        fpetraps completed.
0:00:23.751 ======== (11140)          start of libdso Mon  02 Feb 1998
10:59:04.
dlslave_init executed
dlslave_routine executed
        slaveusertime completed, x = 5000000.000000.
        libdso: dynamic routine returned 13
        end of script 'll.u.cvt.d.i.f.dso'
0:00:24.778 ======== (11140)               End script Mon  02 Feb 1998
10:59:05.
```

2.3.2.1 Generating a Report

To generate a report on the data collected, and to redirect the output to a file, enter the following:

`prof` *your_output_file_name* `> pcsamp.results`

Output similar to the following is generated:

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Mon Feb  2 11:01:36 1998
   prof generic.fpcsamp.m11140
                 generic (n32): Target program
```

```
                      fpcsamp: Experiment name
               pc,2,1000,0:cu: Marching orders
                R4400 / R4000: CPU / FPU
                            1: Number of CPUs
                          175: Clock frequency (MHz.)
  Experiment notes--
          From file generic.fpcsamp.m11140:
        Caliper point 0 at target begin, PID 11140
                /usr/demos/SpeedShop/linpack.demos/c/generic
        Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of statistical PC sampling data (fpcsamp)--
                        23828: Total samples
                       23.828: Accumulated time (secs.)
                          1.0: Time per sample (msecs.)
                            2: Sample bin width (bytes)
-------------------------------------------------------------------------
Function list, in descending order by time
-------------------------------------------------------------------------
 [index]      secs    %    cum.%   samples  function (dso: file, line)

     [1]    22.279  93.5%  93.5%     22279  anneal (generic: generic.c, 1573)
     [2]     0.975   4.1%  97.6%       975  slaveusrtime (dlslave.so: dlslave.c, 22)
     [3]     0.201   0.8%  98.4%       201  __read (libc.so.1: read.s, 20)
     [4]     0.198   0.8%  99.3%       198  cvttrap (generic: generic.c, 317)
     [5]     0.121   0.5%  99.8%       121  _xstat (libc.so.1: xstat.s, 12)
     [6]     0.010   0.0%  99.8%        10  __open (libc.so.1: open.s, 23)
     [7]     0.010   0.0%  99.9%        10  __write (libc.so.1: write.s, 20)
     [8]     0.010   0.0%  99.9%        10  __sigfillset (libc.so.1: sigfillset.c, 11)
     [9]     0.010   0.0%  99.9%        10  _ecvt_r (libc.so.1: ecvt.c, 70)
    [10]     0.003   0.0% 100.0%         3  fread (libc.so.1: fread.c, 34)
    [11]     0.003   0.0% 100.0%         3  dirstat (generic: generic.c, 348)
    [12]     0.002   0.0% 100.0%         2  _doprnt (libc.so.1: doprnt.c, 285)
    [13]     0.001   0.0% 100.0%         1  memcpy (libc.so.1: bcopy.s, 329)
    [14]     0.001   0.0% 100.0%         1  _readdir (libc.so.1: readdir.c, 135)
    [15]     0.001   0.0% 100.0%         1  _read (libc.so.1: readSCI.c, 27)
    [16]     0.001   0.0% 100.0%         1  __sinf (libm.so: fsin.c, 93)
             0.002   0.0% 100.0%         2  **OTHER** (includes excluded DSOs, rld, etc.)

            23.828 100.0% 100.0%     23828  TOTAL
```

2.3.2.2 Analyzing the Report

The report has the following columns:

- The `secs` column shows the amount of CPU time, in seconds, that was spent in the function.
- The `%` column shows the percentage of the total program time that was spent in the function.
- The `cum.%` column shows the percentage of the complete program time in functions that have been listed so far.
- The `samples` column shows how many samples were taken when the process was executing in the function.
- The `function (dso:  file, line)` columns list the function, its DSO name, its file name, and its line number.

2.3.3 A Hardware Counter Experiment

Note: This experiment can be performed only on systems that have built-in hardware counters (machines with the R10000 or R12000 class of CPU).

This section takes you through the steps to perform a hardware counter experiment. There are a number of hardware counter experiments, but this tutorial describes the steps involved in performing the `dsc_hwc` experiment. This experiment captures information about secondary data cache misses. For more information on hardware counter experiments, see Section 4.6, page 55.

2.3.3.1 Performing a Hardware Counter Experiment

From the command line, enter:

```
ssrun -dsc_hwc generic
```

This starts the experiment. Output from `generic` and from `ssrun` is printed to `stdout`. A data file is also generated. The name consists of the process name (`generic`), the experiment type (`dsc_hwc`), and the experiment ID. In this example, the file name is `generic.dsc_hwc.m294398`.

2.3.3.2 Generating a Report

To generate a report on the data collected and redirect the output to a file, enter the following:

`prof` *your_output_file_name* > `dsc_hwc.results`

The report should look similar to the following listing:

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Mon Feb  2 11:11:44 1998
   prof generic.dsc_hwc.m294398
                 generic (n32): Target program
                       dsc_hwc: Experiment name
                 hwc,26,131:cu: Marching orders
               R10000 / R10010: CPU / FPU
                            16: Number of CPUs
                           195: Clock frequency (MHz.)
  Experiment notes--
          From file generic.dsc_hwc.m294398:
        Caliper point 0 at target begin, PID 294398
               /usr/demos/SpeedShop/linpack.demos/c/generic
        Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of R10K perf. counter overflow PC sampling data (dsc_hwc)--
                             6: Total samples
       Sec cache D misses (26): Counter name (number)
                           131: Counter overflow value
                           786: Total counts
-------------------------------------------------------------------------
Function list, in descending order by counts
-------------------------------------------------------------------------
 [index]        counts     %   cum.%   samples  function (dso: file, line)

     [1]           131  16.7%  16.7%     1  init2da (generic: generic.c, 1430)
     [2]           131  16.7%  33.3%     1  genLog (generic: generic.c, 1686)
     [3]           131  16.7%  50.0%     1  _write (libc.so.1: writeSCI.c, 27)
                   393  50.0% 100.0%     3  **OTHER** (includes excluded DSOs, rld, etc.)

                   786 100.0% 100.0%         6  TOTAL
```

2.3.3.3 Analyzing the Report

The information immediately preceding the function list displays the following:

- The `Total samples` is the number of times the program counter was sampled. It is sampled once for each *overflow,* or once each time the hardware counter exceeds the specified value.

- The `Counter name (number)` indicates the hardware counter used in the experiment. In this case, hardware counter 26 counts the number of times a value required in a calculation was not available in secondary cache. For a complete list of the hardware counters and their numbers, see Table 7, page 59.
- The `Counter overflow value` is the number at which the hardware counter overflows, or exceeds its preset value. In this case, the value is 131, which is the default. You can change the overflow value by setting the `_SPEEDSHOP_HWC_COUNTER_OVERFLOW` environment variable to a value larger than 0.
- The `Total counts` is the total number of times a value was not in secondary cache when needed. This value is determined by multiplying the total number of samples by the overflow value; extra counts that do not cause an overflow are not recorded.

The function list has the following columns:

- The `counts` column shows the number of times a data item was not in secondary cache when needed for a calculation during the execution of the function. As with `Total counts` (described earlier), a function's `counts` value is determined by multiplying its `samples` value by the overflow value.
- The `%` column shows the percentage of the program's overflows that occurred in the function.
- The `cum.%` shows the percentage of the program's overflows that occurred in the functions listed so far. A function might have a low number in its `%` column but a high value in its `cum.%` column if it executed late in the program.
- The `samples` column shows the number of times the program counter was sampled during execution of the function. A sample is taken for each overflow of the hardware counter.
- The `function (dso:  file, line)` columns show the function name, the DSO, the file name, and line number of the function.

2.3.4 An `ideal` Experiment

This section takes you through the steps to perform an `ideal` experiment. The times returned represent an idealized computation. This experiment ignores potential floating-point interlocks and memory latency time (cache misses and memory bus contention). The CPU times returned will always be lower than the times for an actual run. For more information on the `ideal` experiment, see Section 4.4, page 52.

2.3.4.1 Performing an `ideal` Experiment

From the command line, enter

```
ssrun -ideal generic
```

This starts the experiment. First the executable, `rld`, and the DSOs are instrumented using `pixie`(1). This entails making copies of the libraries and executables, giving the copies an extension that depends on the ABI, and inserting information into the copies. The extension is `.pixie` for the executable, `.pix32` for all old 32-bit libraries, `.pixn32` for all n32 libraries, and `.pix64` for all 64-bit libraries.

Output from `generic` and from `ssrun` is printed to `stdout`. A data file is also generated. The name consists of the process name (`generic`), the experiment type (`ideal`), and the experiment ID. In this example, the file name is `generic.ideal.m10966`, and the following is written to `stdout`:

```
Beginning libraries
        /usr/lib32/libssrt.so
        /usr/lib32/libss.so
        /usr/lib32/libm.so
        /usr/lib32/libc.so.1
Ending libraries, beginning "generic"
0:00:00.001 ======== (10966)            Begin script Mon  02 Feb 1998
11:28:03.
        begin script ‘ll.u.cvt.d.i.f.dso’
0:00:00.048 ======== (10966)       start of linklist Mon  02 Feb 1998
11:28:03.
        linklist completed.
0:00:00.072 ======== (10966)        start of usrtime Mon  02 Feb 1998
11:28:03.
        usertime completed.
0:00:25.057 ======== (10966)        start of cvttrap Mon  02 Feb 1998
11:28:28.
        cvttrap completed, y = 2.60188e+14, z = 2.60188e+14.
0:00:25.377 ======== (10966)        start of dirstat Mon  02 Feb 1998
11:28:28.
        dirstat of /usr/include completed, 304 files.
0:00:26.232 ======== (10966) start of iofile -- stdio Mon  02 Feb 1998
11:28:29.
```

```
        stdio iofile on /unix completed, 7307988 chars.
0:00:27.716 ======== (10966)        start of fpetraps Mon  02 Feb 1998
11:28:31.
        fpetraps completed.
0:00:27.717 ======== (10966)          start of libdso Mon  02 Feb 1998
11:28:31.
Beginning libraries
Ending libraries, beginning "./dlslave.so"
dlslave_init executed
dlslave_routine executed
        slaveusertime completed, x = 5000000.000000.
        libdso: dynamic routine returned 13
        end of script 'll.u.cvt.d.i.f.dso'
0:00:30.021 ======== (10966)               End script Mon  02 Feb 1998
11:28:33.
```

In the output section that starts with `Beginning libraries` and ends with `Ending libraries, beginning "generic"` tells you that `ssrun` is instrumenting first the libraries listed in the executable and then the `generic` executable itself. The text `beginning "./dlslave.so"` is added when the DSO `dlslave.so` is opened by `dlopen`(3C).

2.3.4.2 Generating a Report

To generate a report on the data collected, enter the following at the command line:

`prof` *your_output_file_name* `> ideal.results`

This command redirects output to a file called `ideal.results`. The file contains results that look similar to the following partial listing. Because of line length restrictions, the DSO, file name, and line number have been wrapped to the next line.

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Mon Feb  2 13:23:25 1998
   prof generic.ideal.m10966
                 generic (n32): Target program
                         ideal: Experiment name
                         it:cu: Marching orders
                 R4400 / R4000: CPU / FPU
                             1: Number of CPUs
                           175: Clock frequency (MHz.)
  Experiment notes--
```

```
          From file generic.ideal.m10966:
        Caliper point 0 at target begin, PID 10966
           /usr/demos/SpeedShop/linpack.demos/c/generic.pixie
        Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of ideal time data (ideal)--
                    2062563179: Total number of instructions executed
                    3929944273: Total computed cycles
                        22.457: Total computed execution time (secs.)
                         1.905: Average cycles / instruction
-------------------------------------------------------------------------
Function list, in descending order by exclusive ideal time
-------------------------------------------------------------------------
 [index]   excl.secs   excl.%     cum.%        cycles  instructions    calls  function
(dso: file, line)

     [1]      21.453    95.5%     95.5%    3754320037    1971220024   1  anneal
(generic: generic.c, 1573)
     [2]       0.829     3.7%     99.2%     145001152      75000732   1  slaveusrtime
(dlslave.so: dlslave.c, 22)
     [3]       0.171     0.8%    100.0%      30000081      16000054   1  cvttrap
(generic: generic.c, 317)
     [4]       0.001     0.0%    100.0%        101504         58124   1  init2da
(generic: generic.c, 1430)
     [5]       0.001     0.0%    100.0%         91200         384001600  _drand48
(libc.so.1: drand48.c, 116)
     [6]       0.001     0.0%    100.0%         89072         55011 447  fread
(libc.so.1: fread.c, 34)
     [7]       0.000     0.0%    100.0%         74854         47366  53  _doprnt
(libc.so.1: doprnt.c, 285)
     [8]       0.000     0.0%    100.0%         64035         29479 628  __sinf
(libm.so: fsin.c, 93)
     [9]       0.000     0.0%    100.0%         32355          7182   9  offtime
(libc.so.1: time_comm.c, 180)
    [10]       0.000     0.0%    100.0%         17112         11916 305  _readdir
(libc.so.1: readdir.c, 135)
    [11]       0.000     0.0%    100.0%         16168         10334   1  iofile
(generic: generic.c, 464)
    [12]       0.000     0.0%    100.0%         15232         12544 448  _read
(libc.so.1: readSCI.c, 27)
    [13]       0.000     0.0%    100.0%         14530          8498 326  memcpy
(libc.so.1: bcopy.s, 329)
    [14]       0.000     0.0%    100.0%         10735          6446   1  dirstat
```

```
(generic: generic.c, 348)
    [15]       0.000     0.0%    100.0%          6535          2831 106  strlen
(libc.so.1: strlen.s, 58)
    [16]       0.000     0.0%    100.0%          6364          4242 304  _xstat
(libc.so.1: xstat.s, 12)
    [17]       0.000     0.0%    100.0%          6363          3636 303  _cerror
(libc.so.1: cerror.s, 30)
.
.
.
   [129]       0.000     0.0%    100.0%             5             3   1  get_exit_status
(libss.so: sswrap_assembly.s, 6)
   [130]       0.000     0.0%    100.0%             4             2   1  __readenv_sigfpe
(libc.so.1: stubfpestart.c, 3)
   [131]       0.000     0.0%    100.0%             4             2   1  crtninit.s
(generic: crtninit.s, 3)
   [132]       0.000     0.0%    100.0%             1             1   1  __istart
(generic: crt1tinit.s, 14)
```

2.3.4.3 Analyzing the Report

The columns in the report provide the following information

- The `excl.secs` column shows the minimum number of seconds that might be spent in the function under ideal conditions. For example, 21.453 seconds is optimal for the `anneal` function, the way it is currently written. The `pcsamp` experiment actually timed this function at 22.279 seconds.
- The `excl.%` column represents how much of the program's total time was spent in the function.
- The `cum.%` column shows the cumulative percentage of time spent in the functions listed so far.
- The `cycles` column reports the number of machine cycles used by the function. For example, 3,754,320,037 CPU clock cycles were spent in the `anneal` function.
- The `instructions` column shows the number of instructions executed by a function. For example, the `anneal` function executed 1,971,220,024 instructions.
- The `calls` column reports the number of calls made to the function. For example, there was just one call to the `anneal` function.

- The `function (dso:  file, line)` column lists the function, its DSO name, its file name, and the line number. For example, the first line reports statistics for the function `anneal` in the file `generic.c` in the `generic` executable on line 1573.

2.3.5 An `fpe` Trace

This section takes you through the steps to perform a floating-point exception (`fpe`) trace, which identifies functions in which floating-point exceptions have occurred. For more information on the `fpe` trace, see Section 4.7, page 61.

2.3.5.1 Performing an `fpe` Trace

From the command line, enter

```
ssrun -fpe generic
```

Output from `generic` and from `ssrun` is printed to `stdout`. A data file is created with a name generated by concatenating the process name (`generic`), the experiment type (`fpe`), and the experiment ID. In this example, the file name is `generic.fpe.m12213`.

2.3.5.2 Generating a Report

To generate a report on the data collected, enter the following at the command line:

`prof` *your_output_file_name* `> fpe.results`

The report should look similar to the following partial listing:

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Mon Feb  2 13:26:33 1998
   prof generic.fpe.m12213
                 generic (n32): Target program
                           fpe: Experiment name
                        fpe:cu: Marching orders
                 R4400 / R4000: CPU / FPU
                             1: Number of CPUs
                           175: Clock frequency (MHz.)
  Experiment notes--
          From file generic.fpe.m12213:
        Caliper point 0 at target begin, PID 12213
           /usr/demos/SpeedShop/linpack.demos/c/generic
        Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of FPE callstack tracing data (fpe)--
                             4: Total FPEs
                             0: Samples with incomplete traceback
-------------------------------------------------------------------------
Function list, in descending order by exclusive FPEs
-------------------------------------------------------------------------
 [index]  excl.FPEs excl.& cum.% incl.FPEs incl.%  function  (dso:file)

   [1]         4 100.0%  100.0%        4 100.0%  fpetraps (generic: generic.c, 405)
   [2]         0   0.0%  100.0%        4 100.0%  __start (generic: crt1text.s, 101)
   [3]         0   0.0%  100.0%        4 100.0%  main (generic: generic.c, 101)
   [4]         0   0.0%  100.0%        4 100.0%  Scriptstring (generic: generic.c, 184)
```

2.3.5.3 Analyzing the Report

The report shows information for each function. The function names are shown in the right-hand column of the report. The remaining columns are described below:

- The `excl.FPEs` column shows how many floating-point exceptions were found in the function. Four floating-point exceptions were found in `fpetraps`.
- The `excl.%` column shows the percentage of the total number of floating-point exceptions that were found in the function.
- The `cum.%` column shows the percentage of floating-point exceptions in the functions that have been listed so far. The list is sorted by the number of floating-point exceptions, with the most in the top line and the least in the bottom line. Because all of the exceptions are in the first function listed in this example, all entries in this column are 100%.
- The `incl.FPEs` columns shows how many floating-point exceptions were generated by the function and the functions it called.
- The `incl.%` column provides the percentage of the program’s total number of floating-point exceptions in this function and the functions it calls. Because `fpetraps` is called through all of the other functions, they are all listed as 100%.

Tutorial for Fortran Users [3]

This chapter provides two tutorials for using the SpeedShop tools to gather and analyze performance data in a Fortran program. The first tutorial covers the following topics:

- Section 3.1, page 31, introduces the sample program and explains the different scenarios in which it will be used.
- Section 3.2, page 32, leads you through the necessary setup for running the experiment.
- Section 3.3, page 33, steps you through different experiments, discussing first how to do the experiments, then how to interpret the results.

The second tutorial creates a Message Passing Interface (MPI) experiment. The experiment file is generated by SpeedShop and displayed by the WorkShop performance analyzer. See Section 3.4, page 45.

Note: Because of inherent differences between systems and also due to concurrent processes that may be running on your system, your experiment will produce different results from the one in this tutorial. However, the basic structure of the results should be the same.

3.1 Tutorial Overview

This tutorial is based on a standard benchmark program called `linpackup.` There are two versions of the program: the `linpack` directory contains files for the n32-bit ABI, and the `linpacko32` directory contains files for the o32-bit ABI. Each `linpack` directory contains versions of the program for a single processor (`linpackup`) and for multiple processors (`linpackd`). When you work with the tutorial, choose the version of the program that is most appropriate for your system. A good guideline is to choose whichever version corresponds to the way you expect to develop your programs.

This tutorial was written and tested using the single-processor version of the program (`linpackup`) in the `linpack` directory.

The `linpack` program is a standard benchmark designed to measure CPU performance in solving dense linear equations. The program focuses primarily on floating-point performance.

Output from the `linpackup` program looks like the following:

```
   .
   .
   .
        norm. resid      resid           machep         x(1)          x(n)
     5.35882395E+00  7.13873405E-13  2.22044605E-16  1.00000000E+00  1.00000000E+00


       times are reported for matrices of order   300
         dgefa      dgesl      total     mflops       unit      ratio
    times for array with leading dimension of 301
     3.720E+00  4.000E-02  3.760E+00  4.835E+00  4.136E-01  6.714E+01
     3.780E+00  3.000E-02  3.810E+00  4.772E+00  4.191E-01  6.804E+01
     3.730E+00  4.000E-02  3.770E+00  4.822E+00  4.147E-01  6.732E+01
     3.730E+00  4.000E-02  3.770E+00  4.822E+00  4.147E-01  6.732E+01

    times for array with leading dimension of 300
     3.800E+00  4.000E-02  3.840E+00  4.734E+00  4.224E-01  6.857E+01
     3.810E+00  4.000E-02  3.850E+00  4.722E+00  4.235E-01  6.875E+01
     3.770E+00  4.000E-02  3.810E+00  4.772E+00  4.191E-01  6.804E+01
     3.782E+00  4.000E-02  3.822E+00  4.757E+00  4.205E-01  6.825E+01
```

3.2 Tutorial Setup

Copy the program to a directory where you have write permission and compile it so that you can use it in the tutorial.

1. Change to the `/usr/demos/SpeedShop` directory.
2. Copy the appropriate `linpack` directory and its contents to a directory in which you have write permission:

   `cp -r` *linpack your_dir*

3. Change to the directory you just created:

   `cd` *your_dir/linpack*

4. Compile the program by entering:

   ```
   make all
   ```

   This provides an executable for the experiment.

3.3 Analyzing Performance Data

This section list the steps you need to perform the following experiments on the `linpackup` program, generate the experiment's results, and interpret the results:

- The `usertime` experiment. It returns the *CPU time* (see the Glossary for a definition) used by each routine in your program. See Section 3.3.1, page 33.
- The `pcsamp` experiment. It returns CPU time for each routine in your program. See Section 3.3.2, page 37.
- Hardware counter experiment. In a hardware counter experiment, the program counter is sampled every time a hardware counter exceeds a specified limit. In the experiment performed in this section, the hardware counter keeps track of the number of times a data item required in a calculation was not present in secondary data cache. When a data item is not in cache, it must be retrieved from memory, which is a more time-consuming process. See Section 3.3.3, page 39.
- The `ideal` experiment. This experiment calculates the best time achievable. See Section 3.3.4, page 41.

3.3.1 A `usertime` Experiment

This section lists the steps you need to perform a `usertime` experiment. The `usertime` experiment allows you to gather data on the amount of *CPU time* spent in each routine in your program. For more information on `usertime`, see Section 4.2, page 50. For definitions of *CPU time*, *wall-clock time*, and *process-virtual time*, see the Glossary.

3.3.1.1 Performing a `usertime` Experiment

From the command line, enter the following:

```
ssrun -v -usertime linpackup
```

This starts the experiment. The `-v` flag tells `ssrun` to print a log to `stderr`.

Output from `linpackup` and from `ssrun` is printed to `stdout`, as shown in the following example. A data file is also generated. The name consists of the process name (`linpackup`), the experiment type (`usertime`), and the experiment ID. In this example, the filename is `linpackup.usertime.m12205`.

```
ssrun: target PID 12205
ssrun: setenv _SPEEDSHOP_MARCHING_ORDERS ut:cu
ssrun: setenv _SPEEDSHOP_EXPERIMENT_TYPE usertime
ssrun: setenv _SPEEDSHOP_TARGET_FILE linpackup
ssrun: setenv _RLD_LIST libss.so:libssrt.so:DEFAULT
 Please send the results of this run to:

 Jack J. Dongarra
 Mathematics and Computer Science Division
 Argonne National Laboratory
 Argonne, Illinois 60439

 Telephone: 312-972-7246

 ARPAnet: DONGARRA@ANL-MCS

     norm. resid      resid           machep         x(1)          x(n)
  5.35882395E+00  7.13873405E-13  2.22044605E-16  1.00000000E+00  1.00000000E+00


    times are reported for matrices of order   300
      dgefa      dgesl      total     mflops       unit      ratio
 times for array with leading dimension of 301
  3.960E+00  4.000E-02  4.000E+00  4.545E+00  4.400E-01  7.143E+01
  3.960E+00  4.000E-02  4.000E+00  4.545E+00  4.400E-01  7.143E+01
  3.970E+00  4.000E-02  4.010E+00  4.534E+00  4.411E-01  7.161E+01
  3.960E+00  4.000E-02  4.000E+00  4.545E+00  4.400E-01  7.143E+01

 times for array with leading dimension of 300
  3.910E+00  4.000E-02  3.950E+00  4.603E+00  4.345E-01  7.054E+01
  3.880E+00  8.000E-02  3.960E+00  4.591E+00  4.356E-01  7.071E+01
  3.930E+00  4.000E-02  3.970E+00  4.579E+00  4.367E-01  7.089E+01
  3.922E+00  3.800E-02  3.960E+00  4.591E+00  4.356E-01  7.071E+01
```

3.3.1.2 Generating a Report

To generate a report on the data collected, enter the following at the command line:

```
prof your_output_file_name > usertime.results
```

The `prof` command interprets the type of experiment you have performed and prints results to `stdout`. The following report shows partial `prof` output. Most lines have been wrapped because of line width restrictions:

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Mon Feb  2 13:37:38 1998
   prof linpackup.usertime.m12205
               linpackup (n32): Target program
                      usertime: Experiment name
                         ut:cu: Marching orders
                 R4400 / R4000: CPU / FPU
                             1: Number of CPUs
                           175: Clock frequency (MHz.)
  Experiment notes--
          From file linpackup.usertime.m12205:
        Caliper point 0 at target begin, PID 12205
                 /usr/demos/SpeedShop/linpack.demos/fortran/linpackup
        Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of statistical callstack sampling data (usertime)--
                          2777: Total Samples
                             0: Samples with incomplete traceback
                        83.310: Accumulated Time (secs.)
                          30.0: Sample interval (msecs.)
-------------------------------------------------------------------------
Function list, in descending order by exclusive time
-------------------------------------------------------------------------
 [index]  excl.secs excl.%   cum.%  incl.secs incl.%    samples  procedure
(dso: file, line)

     [5]     78.090  93.7%   93.7%     78.090  93.7%       2603  daxpy
(linpackup: linpackup.f, 495)
     [6]      2.730   3.3%   97.0%      2.730   3.3%         91  matgen
(linpackup: linpackup.f, 199)
     [4]      1.920   2.3%   99.3%     79.680  95.6%       2656  dgefa
(linpackup: linpackup.f, 221)
     [8]      0.270   0.3%   99.6%      0.270   0.3%          9  dscal
(linpackup: linpackup.f, 670)
     [9]      0.180   0.2%   99.9%      0.180   0.2%          6  idamax
(linpackup: linpackup.f, 700)
    [10]      0.090   0.1%  100.0%      0.090   0.1%          3  dmxpy
(linpackup: linpackup.f, 826)
     [7]      0.030   0.0%  100.0%      0.810   1.0%         27  dgesl
```

```
(linpackup: linpackup.f, 324)
     [1]      0.000   0.0%  100.0%     83.310 100.0%       2777  __start
(linpackup: crt1text.s, 101)
     [2]      0.000   0.0%  100.0%     83.310 100.0%       2777  main
(libftn.so: main.c, 76)
     [3]      0.000   0.0%  100.0%     83.310 100.0%       2777  linp
(linpackup: linpackup.f, 3)
```

3.3.1.3 Analyzing the Report

The report shows information for each function. The function names are show in the right-hand column of the report. The remaining columns are described below:

- The `index` column, which enumerates the routines in the program, provides an index number for reference.
- The `excl.secs` column shows how much time, in seconds, was spent in the routine itself (exclusive time). For example, less than one hundredth of a second was spent in `linp`, but 1.92 seconds were spent in `dgefa`.
- The `excl.%` column shows the percentage of a program's total time that was spent in the function.
  For example, the `daxpy` routine consumed 93.7% of the program's time.
- The `cum.%` column shows the percentage of the complete program time that has been spent in the routines that have been listed so far. For instance, when the `dgefa` routine completes, 99.3% of the program has completed by the routines listed so far.
- The `incl.secs` column shows how much time, in seconds, was spent in the function and descendents of the function. For example, 0.81 seconds were spent in `dgesl` and the routines that were called from it.
- The `incl.%` column shows the cumulative percentage of inclusive time spent in each routine and its descendents. For example, 1% of the time was spent in `dgesl` and all the routines that were called from it.
- The `samples` column provides the number of samples taken from the function and all of its descendants.
- The `procedure (dso:file,line)` column lists the routine name, its DSO name, its file name, and its line number. For example, the top line reports statistics for the routine `daxpy`, the DSO name `linpackup`, in the file `linpackup.f`, at line 495.

Note: Many functions shown here have only one or two hits. The data for those functions is not statistically significant.

3.3.2 A `pcsamp` Experiment

This section lists the steps you need to perform a `pcsamp` experiment. The `pcsamp` experiment allows you to gather information on actual CPU time for each source code line, machine line, and function in your program. For more information on `pcsamp`, see Section 4.3, page 51. For definitions of *CPU time*, *wall-clock time*, and *process-virtual time*, see the Glossary.

3.3.2.1 Performing a `pcsamp` Experiment

From the command line, enter the following:

```
ssrun -pcsamp linpackup
```

This starts the experiment.

Output from `linpackup` and from `ssrun` is printed to `stdout`, as shown in the following example. A data file is also generated. The name consists of the process name (`linpackup`), the experiment type (`pcsamp`), and the experiment ID. In this example, the file name is `linpackup.pcsamp.m12333`.

```
.
.
.
       norm. resid      resid           machep         x(1)          x(n)
    5.35882395E+00  7.13873405E-13  2.22044605E-16  1.00000000E+00  1.00000000E+00
.
.
.
```

3.3.2.2 Generating a Report

To generate a report on the data collected, enter the following at the command line:

`prof` *your_output_file_name* `> pcsamp.results`

The `prof` command interprets the type of experiment you have performed and prints results to `stdout`. The following report shows partial `prof` output, and most lines have been wrapped because of line width restrictions:

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Mon Feb  2 13:52:27 1998
   prof linpackup.pcsamp.m12333
               linpackup (n32): Target program
                        pcsamp: Experiment name
               pc,2,10000,0:cu: Marching orders
                 R4400 / R4000: CPU / FPU
                             1: Number of CPUs
                           175: Clock frequency (MHz.)
  Experiment notes--
          From file linpackup.pcsamp.m12333:
        Caliper point 0 at target begin, PID 12333
         /usr/demos/SpeedShop/linpack.demos/fortran/linpackup
        Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of statistical PC sampling data (pcsamp)--
                          8272: Total samples
                        82.720: Accumulated time (secs.)
                          10.0: Time per sample (msecs.)
                             2: Sample bin width (bytes)
-------------------------------------------------------------------------
Function list, in descending order by time
-------------------------------------------------------------------------
 [index]      secs    %    cum.%   samples  function (dso: file, line)

     [1]    77.440  93.6%  93.6%      7744  daxpy (linpackup: linpackup.f, 495)
     [2]     2.690   3.3%  96.9%       269  matgen (linpackup: linpackup.f, 199)
     [3]     1.940   2.3%  99.2%       194  dgefa (linpackup: linpackup.f, 221)
     [4]     0.370   0.4%  99.7%        37  idamax (linpackup: linpackup.f, 700)
     [5]     0.210   0.3%  99.9%        21  dscal (linpackup: linpackup.f, 670)
     [6]     0.060   0.1% 100.0%         6  dmxpy (linpackup: linpackup.f, 826)
             0.010   0.0% 100.0%         1  **OTHER** (includes excluded DSOs, rld, etc.)

            82.720 100.0% 100.0%      8272  TOTAL
```

3.3.2.3 Analyzing the Report

The report has the following columns:

- The `secs` column shows the amount of CPU time spent in the routine.
- The `(%)` column shows the percentage of the total program time that was spent in the function.

- The `cum.%` column shows the percentage of the complete program time that has been spent by the routines listed so far.
- The `samples` column shows how many samples were taken when the process was executing in the function.
- The `function (dso:file, line)` columns list the routine name, its DSO name, its file name, and its line number. For example, the first line reports statistics for the routine `daxpy`, in the DSO `linpackup`, in the file `linpackup.f`, at line number 495.

3.3.3 A Hardware Counter Experiment

Note: This experiment can be performed only on systems that have built-in hardware counters (the R10000 and R12000 classes of machines).

Hardware counters keep track of a variety of hardware information. For a complete list of hardware counter experiments, see the `ssrun`(1) man page.

This section lists the steps you need to perform a hardware counter experiment. The tutorial describes the steps involved in performing the `dsc_hwc` experiment. This experiment allows you to capture information about secondary data cache misses. For more information on hardware counter experiments, see Section 4.6, page 55.

3.3.3.1 Performing a Hardware Counter Experiment

From the command line, enter the following:

```
ssrun -dsc_hwc linpackup
```

This starts the experiment. Output from `linpackup` and from `ssrun` will be printed to `stdout`. A data file is also generated. The name consists of the process name (`linpackup`), the experiment type (`dsc_hwc`), and the experiment ID. In this example, the filename is `linpackup.dsc_hwc.m438011`.

3.3.3.2 Generating a Report

To generate a report on the data collected, enter the following at the command line:

```
prof your_output_file_name > dsc_hwc.results
```

Output similar to the following is generated:

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Mon Feb  2 13:56:59 1998
   prof linpackup.dsc_hwc.m438011
               linpackup (n32): Target program
                       dsc_hwc: Experiment name
                 hwc,26,131:cu: Marching orders
               R10000 / R10010: CPU / FPU
                            16: Number of CPUs
                           195: Clock frequency (MHz.)
  Experiment notes--
          From file linpackup.dsc_hwc.m438011:
        Caliper point 0 at target begin, PID 438011
         /usr/demos/SpeedShop/linpack.demos/fortran/linpackup
        Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of R10K perf. counter overflow PC sampling data (dsc_hwc)--
                          2929: Total samples
       Sec cache D misses (26): Counter name (number)
                           131: Counter overflow value
                        383699: Total counts
-------------------------------------------------------------------------
Function list, in descending order by counts
-------------------------------------------------------------------------
 [index]    counts     %   cum.% samples  function (dso: file, line)

     [1]    309029  80.5%  80.5%    2359  daxpy (linpackup: linpackup.f, 495)
     [2]     46636  12.2%  92.7%     356  dgefa (linpackup: linpackup.f, 221)
     [3]     25938   6.8%  99.5%     198  matgen (linpackup: linpackup.f, 199)
     [4]      1310   0.3%  99.8%      10  idamax (linpackup: linpackup.f, 700)
     [5]       131   0.0%  99.8%       1  _FWF (libfortran.so: wf90.c, 47)
     [6]       131   0.0%  99.9%       1  memset (libc.so.1: bzero.s, 98)
               524   0.1% 100.0%       4  **OTHER** (includes excluded DSOs, rld, etc.)

                383699 100.0% 100.0%      2929  TOTAL
```

3.3.3.3 Analyzing the Report

The information immediately above the function list displays the following:

- The `Total samples` is the number of times the program counter was sampled. It is sampled once for each *overflow,* or each time the hardware counter exceeds the specified value.

- The `Counter name (number)` indicates the hardware counter used in the experiment. In this case, hardware counter 26 counts the number of times a value required in a calculation was not available in secondary cache. For a complete list of the hardware counters and their numbers, see Table 7, page 59.
- The `Counter overflow value` is the number at which the hardware counter overflows, or exceeds its preset value. In this case, the value is 131, which is the default. You can change the overflow value by setting the `_SPEEDSHOP_HWC_COUNTER_OVERFLOW` environment variable to a value larger than 0.
- The `Total counts` is the total number of times a value was not in secondary cache when needed. This value is determined by multiplying the total number of samples by the overflow value; extra counts that do not cause an overflow are not recorded.

The function list has the following columns:

- The `counts` column shows the number of times a data item was not in secondary cache when needed for a calculation during the execution of the routine. As with `Total counts` (described earlier), a routine's `counts` value is determined by multiplying its `samples` value (described later) by the overflow value.
- The `%` column shows the percentage of the program's overflows that occurred in the routine.
- The `cum.%` shows the percentage of the program's overflows that occurred in the routines listed so far. For example, although the `matgen` routine had only 6.8% of the program's overflows, by the time it is encountered in the routine list, 99.5% of the program's total overflows have been recorded.
- The `samples` column shows the number of times the program counter was sampled during execution of the routine. A sample is taken for each overflow of the hardware counter.
- The `function (dso:  file, line)` columns show the name, the DSO, the file name, and line number of the routine.

3.3.4 An `ideal` Experiment

This section provides the steps you need to perform an `ideal` experiment. The times returned represent an idealized, best-case computation. This experiment ignores interlocks and memory latency time (cache misses and memory bus contention). The CPU times returned will always be lower than for an actual run. For more information on collecting ideal-time data and basic block counting, see Section 4.4, page 52.

3.3.4.1 Performing an `ideal` Experiment

From the command line, enter the following:

```
ssrun -ideal linpackup
```

This starts the experiment. First the executable and libraries are instrumented using `pixie`. This entails making copies of the libraries and executables, giving them an extension that depends on the ABI, and inserting information into the copies. The extension is `.pixie` for the executable, `.pix32` for all 32 libraries, `.pixn32` for all n32 libraries, and `.pix64` for all 64 libraries.

Output from `linpackup` and from `ssrun` is printed to `stdout`, as shown in the following example. A data file is also generated. The name consists of the process name (`linpackup`), the experiment type (`ideal`), and the experiment ID. In this example, the file name is `linpackup.ideal.n11596`.

```
Beginning libraries
./libssrt.so.pixn32 is up to date.
./libss.so.pixn32 is up to date.
./libfortran.so.pixn32 is up to date.
./libffio.so.pixn32 is up to date.
./libftn.so.pixn32 is up to date.
./libm.so.pixn32 is up to date.
./libc.so.1.pixn32 is up to date.
Ending libraries, beginning "linpackup"
.
.
.
```

3.3.4.2 Generating a Report

To generate a report on the data collected, enter the following at the command line:

`prof` *your_output_file_name* `> ideal.results`

The `prof` command redirects output to a file called `ideal.results`. The file should contain results that look something like the following. Most lines have been wrapped because of line length restrictions.

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Mon Feb  2 14:04:20 1998
   prof linpackup.ideal.m11596
               linpackup (n32): Target program
                         ideal: Experiment name
                         it:cu: Marching orders
                 R4400 / R4000: CPU / FPU
                             1: Number of CPUs
                           175: Clock frequency (MHz.)
  Experiment notes--
          From file linpackup.ideal.m11596:
        Caliper point 0 at target begin, PID 11596
         /usr/demos/SpeedShop/linpack.demos/fortran/linpackup.pixie
        Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of ideal time data (ideal)--
                    4911547956: Total number of instructions executed
                    9700441338: Total computed cycles
                        55.431: Total computed execution time (secs.)
                         1.975: Average cycles / instruction
-------------------------------------------------------------------------
Function list, in descending order by exclusive ideal time
-------------------------------------------------------------------------
 [index]   excl.secs   excl.%     cum.%       cycles  instructions   calls
function  (dso: file, line)

     [1]      52.073    93.9%     93.9%    9112833799    4637546756  772633
daxpy (linpackup: linpackup.f, 495)
     [2]       1.937     3.5%     97.4%     339051600     163885662      18
matgen (linpackup: linpackup.f, 199)
     [3]       1.020     1.8%     99.3%     178526333      72336088      17
dgefa (linpackup: linpackup.f, 221)
     [4]       0.180     0.3%     99.6%      31463770      17658342    5083
dscal (linpackup: linpackup.f, 670)
     [5]       0.166     0.3%     99.9%      28990712      15670260    5083
idamax (linpackup: linpackup.f, 700)
     [6]       0.045     0.1%    100.0%       7839357       3605134       1
dmxpy (linpackup: linpackup.f, 826)
     [7]       0.009     0.0%    100.0%       1499774        695929      17
dgesl (linpackup: linpackup.f, 324)
     [8]       0.000     0.0%    100.0%         54065         30649      53
_sd2udee (libffio.so: sd2udee.c, 104)
     [9]       0.000     0.0%    100.0%         44650         28904       1
```

```
linp (linpackup: linpackup.f, 3)
    [10]       0.000     0.0%    100.0%         37376         26716      31
_wrfmt (libfortran.so: wrfmt.c, 56)
    [11]       0.000     0.0%    100.0%         11448          8427     159
_IEEE_BINARY_SCALE_I4 (libfortran.so: ieee_binary_scale_r_n.c, 41)
    [12]       0.000     0.0%    100.0%          8784          5116     492
nvmatch (libc.so.1: getenv.c, 46)
    [13]       0.000     0.0%    100.0%          8586          4596       6
getenv (libc.so.1: getenv.c, 25)
    [14]       0.000     0.0%    100.0%          7850          4718       4
_get_next_unit (libfortran.so: fortunit.c, 171)
    [15]       0.000     0.0%    100.0%          7370          5003      15
_FWF (libfortran.so: wf90.c, 47)
    [16]       0.000     0.0%    100.0%          6012          3586      21
_pack (libffio.so: _pack.c, 51)
    [17]       0.000     0.0%    100.0%          5293          2273      16
strlen (libc.so.1: strlen.s, 58)
    [18]       0.000     0.0%    100.0%          3992          2888       8
_map_to_dv (libfortran.so: dopexfer.c, 964)
    [19]       0.000     0.0%    100.0%          3757          2289       9
fflush (libc.so.1: flush.c, 377)
    [20]       0.000     0.0%    100.0%          3394          2352      28
_fwch (libfortran.so: fwch.c, 61)
    [21]       0.000     0.0%    100.0%          3248          1680       8
_stride_dv (libfortran.so: dopexfer.c, 495)
    [22]       0.000     0.0%    100.0%          2809          1908      53
_IEEE_EXPONENT_I4_R (libfortran.so: ieee_exponent_n.c, 73)
    [23]       0.000     0.0%    100.0%          2649          1607      18
_unpack (libffio.so: _unpack.c, 54)
    [24]       0.000     0.0%    100.0%          2544          1908     159
__is_nan64 (libfortran.so: inline.h, 343; compiled in ieee_binary_scale_r_n.c)
    [25]       0.000     0.0%    100.0%          2404          1696      15
setup_format (libfortran.so: f90io.h, 451; compiled in wf90.c)
    [26]       0.000     0.0%    100.0%          2052          1296      54
second_ (linpackup: second.c, 8)
    [27]       0.000     0.0%    100.0%          1988          1764      28
_sw_endrec (libfortran.so: wf.c, 906)
    [28]       0.000     0.0%    100.0%          1712          1252      15
_xfer_iolist (libfortran.so: dopexfer.c, 150)
    [29]       0.000     0.0%    100.0%           848           636      53
__is_nan64 (libfortran.so: inline.h, 343; compiled in ieee_exponent_n.c)
    [30]       0.000     0.0%    100.0%           795           530      53
isdigit (libc.so.1: ctypefcns.c, 62)
```

```
    [31]       0.000     0.0%    100.0%          736           72       8
_tripcnt (libfortran.so: dopexfer.c, 1172)
    [32]       0.000     0.0%    100.0%          694          425       3
_s2ui (libffio.so: s2uboiz.c, 394)
.
.
.
   [130]       0.000     0.0%    100.0%            1            1       1
__istart (linpackup: crt1tinit.s, 14)
```

3.3.4.3 Analyzing the Report

The report has the following columns:

- The `excl.secs` column shows the minimum number of seconds that might be spent in the routine under ideal conditions. For example, 52.073 seconds is optimal for the `daxpy` routine. The `pcsamp` experiment (see Section 3.3.2, page 37) times this routine at 77.44 seconds.
- The `excl.%` column represents how much of the program's total time was spent in the routine.
- The `cum.%` column shows the cumulative percentage of time spent in the routines listed so far.
- The `cycles` column reports the number of machine cycles used by the routine. For example, 91,12,833,799 CPU clock cycles were spent in the `daxpy` routine.
- The `instructions` column shows the number of instructions executed by a routine. For example, the `dgefa` routine executed 72,336,088 instructions.
- The `calls` column reports the number of calls to the routine. For example, there was just one call to the `dmxpy` routine.
- The `procedure (dso:file, line)` column lists the name, the DSO name, the file name, and the line number for the routine.

3.4 MPI Tracing tutorial

The following steps generate tracing data for an MPI program:

1. First, set the `MPI_RLD_HACK_OFF` environment variable to prevent SpeedShop confusion over the organization of the DSOs.

```
% setenv MPI_RLD_HACK_OFF=1
```

2. Compile the `matmul.f` source file and include the MPI library:

   ```
   % f90 -o matmul matmul.f -lmpi
   ```

3. Now run the `ssrun` command as part of the `mpirun`(1) command on the executable file to generate experiment files:

   ```
   % mpirun -np 4 ssrun -mpi matmul
   ```

   The result will be a series of experiment files, one for each process (the identifier begins with an `f`) and one for the master process (the identifier begins with an `m`):

   ```
   matmul.mpi.f9587021
   matmul.mpi.f9905720
   matmul.mpi.f9930637
   matmul.mpi.f9930718
   matmul.mpi.m9951566
   ```

4. Finally, display an experiment file with the WorkShop `cvperf`(1) command. Remember, you cannot use `prof` to display an MPI trace experiment.

   ```
   % cvperf matmul.mpi.f9587021
   ```

   To display the output, select either `MPI Stats View (Graphs)` or `MPI Stats View (Numerical)` from the `Views` menu. See Figure 1, page 47 for an illustration of the `MPI Stats View (Graphs)`.

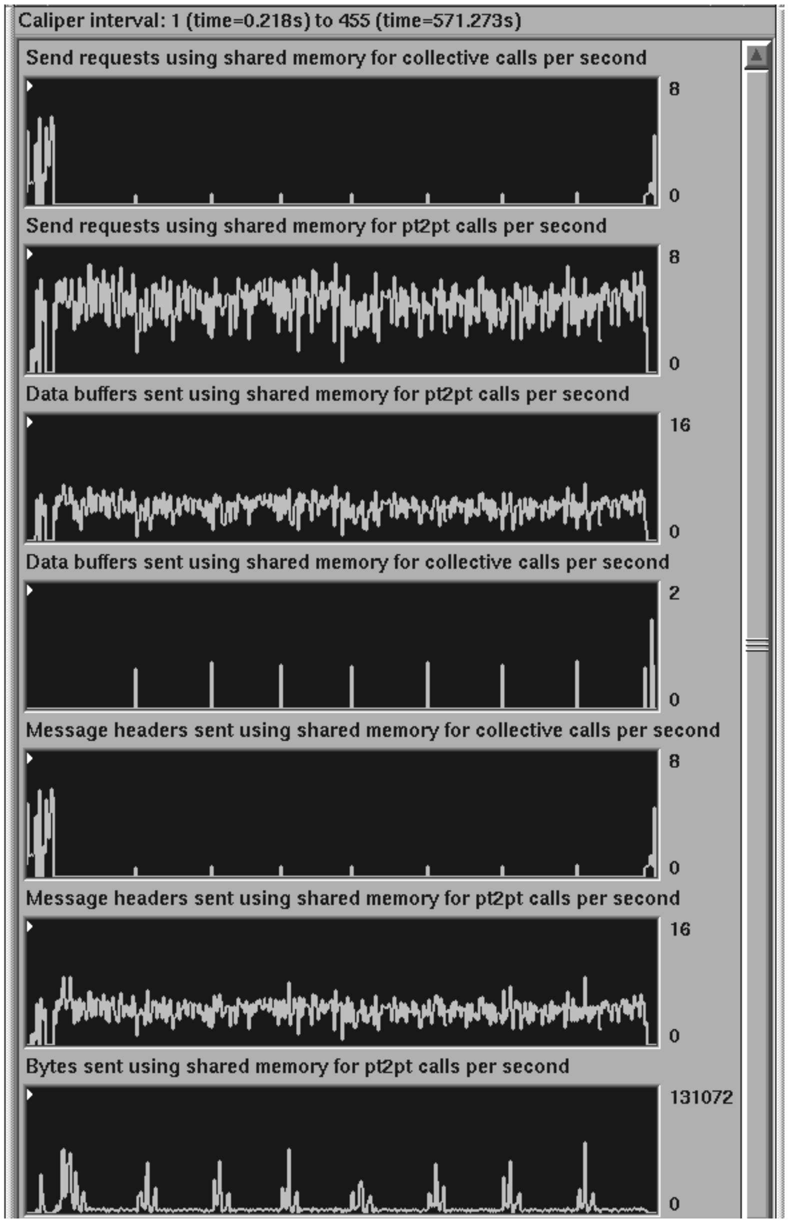

Caliper interval: 1 (time=0.218s) to 455 (time=571.273s)
Send requests using shared memory for collective calls per second
8
0
Send requests using shared memory for pt2pt calls per second
8
0
Data buffers sent using shared memory for pt2pt calls per second
16
0
Data buffers sent using shared memory for collective calls per second
2
0
Message headers sent using shared memory for collective calls per second
8
0
Message headers sent using shared memory for pt2pt calls per second
16
0
Bytes sent using shared memory for pt2pt calls per second
131072
0

Figure 1. An MPI Experiment in `cvperf`

Experiment Types [4]

This chapter provides detailed information on each experiment type available within SpeedShop. It contains the following sections:

- Selecting an experiment. See Section 4.1, page 49.
- The `usertime` experiment. See Section 4.2, page 50.
- The `pcsamp` experiment. See Section 4.3, page 51.
- The `ideal` experiment. See Section 4.4, page 52.
- The I/O trace experiment. See Section 4.5, page 54.
- The hardware counter experiments. See Section 4.6, page 55.
- The floating-point exception trace experiment. See Section 4.7, page 61.
- Heap trace experiments. See Section 4.8, page 61.
- Combining multiple experiment files into one file. See Section 4.9, page 62.

For information on how to run the experiments described in this chapter, see Chapter 6, page 67.

4.1 Selecting an Experiment

Table 5 shows the possible experiments you can perform using the SpeedShop tools and the reasons why you might want to choose a specific experiment. The Clues column shows when you might use an experiment. The Data Collected column indicates performance data collected by the experiment. For detailed information on the experiments, see the relevant section in the remainder of this chapter.

Table 5. Summary of Experiments

Experiment	Clues	Data Collected
`usertime`	Slow program, nothing else known. Not CPU-bound.	Inclusive and exclusive CPU time for each function by sampling the callstack at 30-millisecond intervals.
`pcsamp`	High user CPU time.	Actual CPU time at the source line, machine instruction, and function levels by sampling the program counter at 10 or 1-millisecond intervals using basic block counting.
`ideal`	CPU-bound.	Ideal CPU time at the function, source line, and machine instruction levels.
`_hwc`	High user CPU time.	On R10000 and R12000 class machines, counts at the source line, machine instruction, and function levels of various hardware events, including: clock cycles, graduated instructions, primary instruction cache misses, secondary instruction cache misses, primary data cache misses, secondary data cache misses, translation lookaside buffer (TLB) misses, and graduated floating-point instructions.
`fpe`	High system time. Presence of floating-point operations.	All floating-point exceptions, with the exception type and the callstack at the time of the exception.

4.2 `usertime` Experiment

The `usertime` experiment is a good experiment with which to begin performance analysis of your program. It returns CPU time (see the glossary) for each function while your program runs.

It uses statistical call stack profiling to measure inclusive and exclusive user time. This experiment takes a sample every 3 milliseconds. Data is measured by periodically sampling the callstack. The program's callstack data is used to do the following:

- Attribute exclusive user time to the function at the bottom of each callstack (that is, the function being executed at the time of the sample).

- Attribute inclusive user time to all the functions above the one currently being executed (those involved in the chain of calls that led to the function at the bottom of the callstack executing).

The time spent in a procedure is determined by multiplying the number of times an instruction for that procedure appears in the stack by the average time interval between call stacks. Call stacks are gathered when the program is running; hence, the time computed represents user time, not time spent when the program is waiting for a CPU. User time shows both the time the program itself is executing and the time the operating system is performing services for the program, such as I/O.

User time runs should incur a program execution slowdown of no more than 15%. Data from a `usertime` experiment is statistical in nature and shows some variance from run to run.

Note: For this experiment, o32 executables must explicitly link with `-lexc`.

4.3 `pcsamp` Experiment

The `pcsamp` experiment estimates the actual *CPU time* (see the glossary) for each source code line, machine code line, and function in your program. The `prof` listing of this experiment shows exclusive PC sampling time. This experiment is a lightweight, high-speed operation that makes use of the operating system.

CPU time is calculated by multiplying the number of times an instruction or function appears in the PC by the interval specified for the experiment (either 1 or 10 milliseconds).

To collect the data, the operating system regularly stops the process, increments a counter corresponding to the current value of the PC, and resumes the process. The default sample interval is 10 milliseconds. If you specify the optional `f` prefix to the experiment, a sample interval of 1 millisecond is used.

By default, the experiment uses 16-bit counters. If the optional `x` suffix is used, a 32-bit counter size will be used. Using a 32-bit bin provides more accurate information, but requires additional memory and disk space.

- 16-bit bins allow a maximum of 65,536 counts.
- 32-bit bins allow over 4 billion counts.

PC sampling runs should slow the execution time of the program down no more than 5 percent. The measurements are statistical in nature, meaning they exhibit variance inversely proportional to the running time.

4.4 `ideal` Experiment

The `ideal` experiment returns information on the fastest possible execution time for your program. Although your program will never match ideal time, it is a good tool for finding the bottlenecks in your program. Compare the results returned by the `ideal` experiment with those returned by the `usertime` or `pcsamp` experiment (for more information, see Section 4.4.4, page 54).

The `ideal` experiment gathers information by instrumenting the executables and any DSOs to count basic blocks and dynamic (function-pointer) calls.

You can also use an `ideal` experiment file to optimize the way your program is organized. For more information on reordering code regions, see the *MIPSpro Compiling and Performance Tuning Guide*.

4.4.1 How SpeedShop Prepares Files

To permit block counting, SpeedShop does the following:

- Divides the code into basic blocks, which are sets of instructions with a single entry point, a single exit point, and no branches into or out of the set.
- Inserts counter code at the beginning of each basic block to increment a counter each time that basic block is executed.

The target executable, `rld`(1), and all the DSOs are instrumented. Instrumented files with an extension `.pix*`, where `*` depends on the ABI, are written to the current working directory or to the directory specified by the `_SPEEDSHOP_OUTPUT_DIRECTORY` environment variable, if set.

After instrumentation, `ssrun` executes the instrumented program. Data is generated as long as the process exits normally or receives a fatal signal that the program does not handle.

4.4.2 How SpeedShop Calculates Ideal CPU Time

The `prof` command uses a machine model to convert the block execution counts into an idealized, exclusive CPU time at the function, source line, or machine instruction levels. By default, the machine model corresponds to the machine on which the target was run; the user can specify a different machine model for the analysis.

Note that the execution time of an instrumented program is three to six times longer than an uninstrumented one. This timing change may alter the behavior of a program that deals with a graphical user interface (GUI) or depends on events such as `SIGALRM` that are based on an external clock. Also, during analysis the instrumented executable might appear to be CPU-bound, whereas the original executable was I/O-bound.

Basic block counts are translated to ideal CPU time displayed at the function, source line, and assembly instruction levels.

4.4.3 Inclusive Basic Block Counting

The basic block counting explained in the previous section allows you to measure ideal time spent in each procedure, but it does not propagate the time up to the caller of that procedure. For example, basic block counting may tell you that procedure `sin(x)` took the most time, but significant performance improvement can only be obtained by optimizing the callers of `sin(x)`. Inclusive basic block counting solves this problem.

Inclusive basic block counting calculates cycles just like regular basic block counting and then propagates it in proportion to its callers. The cycles of procedures obtained using regular basic block counting (called exclusive cycles) are divided up among its callers in proportion to the number of times they called this procedure. For example, if `sin(x)` takes 1000 cycles, and its callers, procedures `foo()` and `bar()`, call `sin(x)` 25 and 75 times respectively, 250 cycles are attributed to `foo()` and 750 to `bar()`. By propagating cycles this way, `__start()` usually ends up with all the cycles counted in the program. (It is possible to write code that makes determining the complete call graph impossible, in which case you may end up with parts of the call graph disconnected.)

The assumption can be very misleading. If `foo` calls `matmult` 99 times for 2–by–2 matrices, while `bar` calls it once for 100–by–100 matrices, the inclusive time report will attribute 99% of `matmult()`'s time to `foo()`, but actually almost all the time could derive from the one call from `bar()`.

To generate a report that shows *inclusive time* (see the glossary), specify the `-gprof` option to the `prof` command.

4.4.4 Using `pcsamp` and `ideal` Together

The `ideal` experiment can be used together with the `pcsamp` experiment to compare actual and ideal times spent in the CPU. A major discrepancy between `pcsamp` CPU time and `ideal` CPU time indicates one or more of the following situations:

- Cache misses and floating-point interlocks in a single process application.
- Secondary cache invalidations in an application with multiple processes that is run on a multiprocessor.

A comparison between basic block counts (`ideal` experiment) and PC profile counts (`pcsamp` experiment) is shown in Table 6.

Table 6. Basic Block Counts and PC Profile Counts Compared

Basic Block Counts	PC Profile Counts
Used to compute ideal CPU time.	Used to estimate actual CPU time.
Data collection by instrumentation.	Data collection by the kernel.
Slows program down by factor of three.	Has minimal impact on program speed.
Generates an exact count of every instruction.	Generates statistical, inexact counts.

4.5 I/O Trace Experiment

The I/O trace experiment shows you the level of I/O activity in your program by tracing various I/O system calls, for example `read`(2) and `write`(2).

The `prof` output of an I/O trace experiment yields the following information:

- The number of I/O system calls executed.
- The number of calls with an incomplete traceback.
- The number of I/O-related system calls from each function in the program.
- The percentage of I/O-related system calls from each function in the program.

- The percentage of I/O-related system calls encountered so far in the list of functions.
- The number of I/O-related system calls made by a given function and by all the functions ultimately called by that given function. For example, the `main` function will probably include all of the program's I/O calls with complete tracebacks.
- The percentage of I/O-related system calls made by a given function and by all the functions ultimately called by that given function.
- The DSO, file name, and line number for each function.

The following `ssrun` command creates an I/O trace experiment file from the executable file `generic`:

```
% ssrun -io generic
```

4.6 Hardware Counter Experiments

The experiments described in this section are available for systems that have hardware counters (R10000 and R12000 class machines). Hardware counters allow you to count various types of events, such as cache misses and counts of issued and graduated instructions.

A hardware counter works as follows: for each event, the appropriate hardware counter is incremented on the processor clock cycle. For example, when a floating-point instruction is graduated in a cycle, the graduated floating-point instruction counter is incremented by 1.

These experiments are detailed by nature. They return information gathered at the hardware level. You probably want to run a higher level experiment first. Once you have narrowed the scope, you can use hardware counter experiments to pinpoint the area to be tuned.

4.6.1 Two Tools for Hardware Counter Experiments

There are two tools that allow you to access hardware counter data:

- `perfex`(1) is a command-line interface that provides program-level event information. For more information on `perfex`, see the `perfex`(1) man page. For more information on hardware counters, see the `r10k_counters`(1) man page.

- SpeedShop allows you to perform the hardware counter experiments described in the next section (Section 4.6.2).

4.6.2 SpeedShop Hardware Counter Experiments

In the SpeedShop hardware counter experiments, overflows of a particular hardware counter are recorded. (Each hardware counter is configured to count from zero to a number designated as the overflow value. When the counter reaches the overflow value, the system resets it to zero and increments the number of overflows at the present program instruction address.) Each experiment provides two possible overflow values; the values are prime numbers, so any profiles that seem the same for both overflow values should be statistically valid.

The hardware counter experiments show where the overflows are being triggered in the program, at the function, source-line, and individual instruction level. When you run `prof` on the data collected during the experiment, the overflow counts are multiplied by the overflow value to compute the total number of events. These numbers are statistical, meaning they are not precise. The generated reports show exclusive hardware counts: that is, information about where the program counter was. They do not show the callstack to get there.

Hardware counter overflow profiling experiments should incur a slowdown of execution of the program of no more than 5%. Count data is kept as 32-bit integers only.

The available hardware experiments are described in the following sections.

4.6.3 The `[f]gi_hwc` Experiment

The `[f]gi_hwc` experiment counts overflows of the graduated instruction counter. The graduated instruction counter is incremented by the number of instructions that were graduated on the previous cycle. The experiment uses statistical PC sampling based on an overflow interval of 32,771. If the optional `f` prefix is used, the overflow interval is 6,553.

4.6.4 The `[f]cy_hwc` Experiment

The `[f]cy_hwc` experiment counts overflows of the cycle counter. The cycle counter is incremented on each clock cycle. The experiment uses statistical PC sampling based on an overflow interval of 16,411. If the optional `f` prefix is used, the overflow interval is 3,779.

4.6.5 The [f]ic_hwc Experiment

The `[f]ic_hwc` experiment counts overflows of the primary instruction cache miss counter. The counter is incremented one cycle after an instruction fetch request is entered into the miss handling table. The experiment uses statistical PC sampling based on an overflow interval of 2,053. If the optional `f` prefix is used, the overflow interval is 419.

4.6.6 The [f]isc_hwc Experiment

The `[f]isc_hwc` experiment counts overflows of the secondary instruction cache miss counter. The secondary instruction cache miss counter is incremented after the last 16-byte block of a 64-byte primary instruction cache line is written into the instruction cache. The experiment uses statistical PC sampling based on an overflow interval of 131. If the optional `f` prefix is used, the overflow interval is 29.

4.6.7 The [f]dc_hwc Experiment

The `[f]dc_hwc` experiment counts overflows of the primary data cache miss counter. The primary data cache miss counter is incremented on the cycle after a primary cache data refill is begun. The experiment uses statistical PC sampling based on an overflow interval of 2,053. If the optional `f` prefix is used, the overflow interval is 419.

4.6.8 The [f]dsc_hwc Experiment

The `[f]dsc_hwc` experiment counts overflows of the secondary data cache miss counter. The secondary data cache miss counter is incremented on the cycle after the second 16-byte block of a primary data cache line is written into the data cache. The experiment uses statistical PC sampling, based on an overflow interval of 131. If the optional `f` prefix is used, the overflow interval is 29.

4.6.9 The [f]tlb_hwc Experiment

The `[f]tlb_hwc` experiment counts overflows of the translation lookaside buffer (TLB) counter. The TLB counter is incremented on the cycle after the TLB miss handler is invoked. The experiment uses statistical PC sampling based on an overflow interval of 257. If the optional `f` prefix is used, the overflow interval is 53.

4.6.10 The [f]gfp_hwc Experiment

The `[f]gfp_hwc` experiment counts overflows of the graduated floating-point instruction counter. The graduated floating-point instruction counter is incremented by the number of floating-point instructions that graduated on the previous cycle. The experiment uses statistical PC sampling based on an overflow interval of 32,771. If the optional `f` prefix is used, the overflow interval is 6,553.

4.6.11 The prof_hwc Experiment

The `prof_hwc` experiment allows you to set a hardware counter to use in the experiment and to set a counter overflow interval using the following environment variables:

`_SPEEDSHOP_HWC_COUNTER_NUMBER`

The value of this variable can be between 0 and 31. Hardware counters are described in the *MIPS R10000 Microprocessor User's Manual*, Chapter 14, and on the `r10k_counters`(1) man page. The hardware counter numbers are provided in Section 4.6.11.1, page 58.

`_SPEEDSHOP_HWC_COUNTER_OVERFLOW`

The value of this variable can be any number greater than 0. Some numbers may produce data that is not statistically random, but rather reflects a correlation between the overflow interval and a cyclic behavior in the application. You may want to do two or more runs with different overflow values.

The default counter is the primary instruction-cache miss counter; the default overflow interval is 2,053.

The experiment uses statistical PC sampling based on the overflow of the specified counter, at the specified interval. Note that these environment variables cannot be used for other hardware counter experiments. They are examined only when the `prof_hwc` experiment is specified.

4.6.11.1 Hardware Counter Numbers

The possible numeric values for the `_SPEEDSHOP_HWC_COUNTER_NUMBER` variable are shown in the following tables. Table 7, page 59, gives the hardware counter numbers for systems with R10000 processors, and Table 8, page 60, gives them for systems with R12000 processors.

Table 7. R10000 Hardware Counter Numbers

0	Cycles
1	Issued instructions
2	Issued loads
3	Issued stores
4	Issued store conditionals
5	Failed store conditionals
6	Decoded branches
7	Quadwords written back from secondary cache
8	Correctable secondary cache data array ECC errors
9	Primary instruction-cache misses
10	Secondary instruction-cache misses
11	Instruction misprediction from secondary cache way prediction table
12	External interventions
13	External invalidations
14	Virtual coherency conditions (or functional unit completions, depending on hardware version)
15	Graduated instructions
16	Cycles
17	Graduated instructions
18	Graduated loads
19	Graduated stores
20	Graduated store conditionals
21	Graduated floating-point instructions
22	Quadwords written back from primary data cache
23	TLB misses
24	Mispredicted branches
25	Primary data cache misses

26	Secondary data cache misses
27	Data misprediction from secondary cache way prediction table
28	External intervention hits in secondary cache
29	External invalidation hits in secondary cache
30	Store/prefetch exclusive to clean block in secondary cache
31	Store/prefetch exclusive to shared block in secondary cache

Table 8. R12000 Hardware Counter Numbers

0	Cycles
1	Decoded instructions
2	Decoded loads
3	Decoded stores
4	Miss Handling Table occupancy
5	Failed store conditionals
6	Resolved conditional branches
7	Quadwords written back from secondary cache
8	Correctable secondary cache data array ECC errors
9	Primary instruction-cache misses
10	Secondary instruction-cache misses
11	Instruction misprediction from secondary cache way prediction table
12	External interventions
13	External invalidations
14	ALU/FPU progress cycles
15	Graduated instructions
16	Executed prefetch instructions
17	Prefetch primary data cache misses
18	Graduated loads

19	Graduated stores
20	Graduated store conditionals
21	Graduated floating-point instructions
22	Quadwords written back from primary data cache
23	TLB misses
24	Mispredicted branches
25	Primary data cache misses
26	Secondary data cache misses
27	Data misprediction from secondary cache way prediction table
28	State of intervention hits in secondary cache
29	State of invalidation hits in secondary cache
30	Store/prefetch exclusive to clean block in secondary cache
31	Store/prefetch exclusive to shared block in secondary cache

4.7 Floating-Point Exception Trace

A floating-point exception trace collects each floating-point exception with the exception type and the callstack at the time of the exception. Floating-point exception tracing experiments should incur a slowdown in execution of the program of no more than 15%. These measurements are exact, not statistical.

The `prof` command generates a report that shows inclusive and exclusive floating-point exception counts.

4.8 Heap Trace Experiments

If you are running a heap trace experiment on a multiprocessor application, you will get an experiment file for each process and an additional experiment file for the master process. Each process experiment file can either contain a sample of the data from the whole application or its own data only, as follows:

- By default, the experiment file for each process will contain data from all processes.

- If you set the `_SSMALLOC_NO_BUFFERING` environment variable before executing `ssrun`, the experiment file for each process will contain only its own heap trace data.

4.9 Combining Multiple Experiment Files into One

The `ssaggregate`(1) command lets you combine the data from two or more experiment files of the same experiment type (such as `ideal`) into a single file. You can then view the new file with either `prof`(1) or the WorkShop performance analyzer, `cvperf`(1).

The `ssaggregate` command takes the following form:

`ssaggregate -e` *files* `-noverbose -o` *output_file*

The following example combines two `pcsamp` experiments into a single file and displays the file with `prof`:

```
% ssaggregate -e generic.pcsampx.f14636 generic.pcsamp.f14635 -o combo
% prof combo
```

The output from `prof` is as follows:

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Tue Nov 24 11:30:03 1998
   prof combo
                 generic (n32): Target program
                        pcsamp: Experiment name
               pc,2,10000,0:cu: Marching orders
                 R5000 / R5000: CPU / FPU
                             1: Number of CPUs
                           180: Clock frequency (MHz.)
  Experiment notes--
          From file combo:
        Caliper point 0 at target begin, PID 14635
                        /home/saffron02/speedshop/c/generic ll.u.cvt.d.i.f.dso ll.u.cvt.d.i.f.dso ll.u.cvt.d.i.f.dso
        Caliper point 0 at target begin, PID 14636
                        /home/saffron02/speedshop/c/generic ll.u.cvt.d.i.f.dso ll.u.cvt.d.i.f.dso ll.u.cvt.d.i.f.dso
        Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of statistical PC sampling data (pcsamp)--
                          4012: Total samples
                        40.120: Accumulated time (secs.)
                          10.0: Time per sample (msecs.)
```

```
                             2: Sample bin width (bytes)
-------------------------------------------------------------------------
Function list, in descending order by time
-------------------------------------------------------------------------
 [index]      secs    %    cum.%   samples  function (dso: file, line)

     [1]    37.480  93.4%  93.4%      3748  anneal (generic: generic.c, 1573)
     [2]     1.450   3.6%  97.0%       145  slaveusrtime (dlslave.so: dlslave.c, 22)
     [3]     0.490   1.2%  98.3%        49  _read (libc.so.1: read.s, 15)
     [4]     0.330   0.8%  99.1%        33  _xstat (libc.so.1: xstat.s, 12)
     [5]     0.300   0.7%  99.8%        30  cvttrap (generic: generic.c, 317)
     [6]     0.030   0.1%  99.9%         3  _write (libc.so.1: write.s, 15)
     [7]     0.010   0.0%  99.9%         1  fread (libc.so.1: fread.c, 27)
     [8]     0.010   0.0% 100.0%         1  _syscall (libc.so.1: syscall.s, 15)
             0.020   0.0% 100.0%         2  **OTHER** (includes excluded DSOs, rld, etc.)

            40.120 100.0% 100.0%      4012  TOTAL
```

By default, `ssaggregate` issues periodic status message while it is processing. The `-noverbose` option turns the status messages off. See the `ssaggregate`(1) man page.

Collecting Data on Machine Resource Usage [5]

This chapter describes how to collect machine resource usage data using the SpeedShop `ssusage`(1) command. Finding out the machine resources that your program uses can help you identify performance bottlenecks and determine which performance experiments you need to run. You can use the list in Section 1.4.2, page 9, to identify which experiments to run, based on the results of running `ssusage` on your program.

5.1 ssusage Syntax

The `ssusage` command has no options of its own. It takes the following form:

`ssusage` *executable_name* [*executable_args*]

executable_name	Name of the executable for which you want to collect machine resource usage data
executable_args	Arguments to your executable, if any

5.2 ssusage Results

The `ssusage` command prints output to `stderr`. For example, the `ssusage generic` command provides output similar to the following:

```
...
22.03 real, 18.18 user, 0.21 sys, 7 majf, 120 minf, 0 sw, 241 rb, 0
wb, 135 vcx, 648 icx
```

The last two lines of the output constitute the machine resource usage information that `ssusage` provides. Following is a description of each field from the report:

`real`	The real, or wall-clock, time in which the executable ran, in seconds.
`user`	User CPU time, excluding the time the operating system was performing services for the executable, in seconds.
`sys`	System CPU time, during which the system was performing services for the executable, in seconds.

`majf`	Major page faults that cause physical I/O.
`minf`	Minor page faults that require mapping only.
`sw`	Process swaps.
`rb/wb`	Physical blocks read or written. These are attributed to the process that first requests a block, but they do not necessarily directly correlate with the process's own I/O operations.
`vcx`	Voluntary context switches; those caused by the process' own actions.
`icx`	Involuntary context switches; those caused by the scheduler.

If the program terminates abnormally, a message is printed before the usage line.

Setting Up and Running Experiments: `ssrun` [6]

This chapter provides information on how to set up and run performance analysis experiments using the `ssrun` command. It consists of the following sections:

- Building Your Executable, see Section 6.1, page 67.
- Setting Up Output Directories and Files, see Section 6.2, page 69.
- Using Run-Time Environment Variables, see Section 6.3, page 70.
- Using Marching Orders, see Section 6.4, page 74.
- Running Experiments, see Section 6.5, page 77.
- Running Experiments on MPI Programs, see Section 6.6, page 82.
- Running Experiments on Programs Using Pthreads, see Section 6.7, page 86.
- Using Calipers, see Section 6.8, page 86.
- Effects of `ssrun`, see Section 6.9, page 90.

6.1 Building Your Executable

The `ssrun` command is designed to be used with normally built executables and default environment settings. However, there are some cases where you need to change the way you build your executable or set certain environment variables.

This section explains when to change the way you build your executable program. For information on setting environment variables, see Section 6.3, page 70.

- If you have used the `ssrt_caliper_point`(3) function provided in the SpeedShop libraries, you have to explicitly link in the SpeedShop libraries file, `libss.so`. For more information on setting caliper points, see Section 6.8, page 86.
- If you are planning to build your executable using the `-32` option to the `cc` command, and you want to run the `usertime` experiment, you must add

`-lexc` to the link line. For more information on `cc -32`, see the `cc`(1) man page.

- If you have built a stripped executable, you need to rebuild a nonstripped version to use with SpeedShop. For example, if you are using `ld` to link your C program, do not use the `-s` option. Using the `-s` option strips debugging information from the program object and makes the program unusable for performance analysis.
- If you have used compiler optimization level 3 and you are performing experiments that report function-level information, inlining can result in extremely misleading profiles. The time spent in the inlined procedure will show up in the profile as time spent in the procedure into which it was inlined. It is generally better to use compiler optimization level 2 or less when gathering an execution profile.

6.1.1 Special Information for MP Fortran Programs

If you are compiling MP Fortran programs, you may encounter anomalies in the displayed data:

- For all `f90`(1), `f77`(1), and `fort77`(1) MP compilations, parallel loops within the program are represented as subroutines with names relating to the source routine in which they are embedded. The naming conventions for these subroutines are different for 32-bit and 64-bit compilations.

  For example, in the `linpack` example program, most of the time is spent in the routine `DAXPY`, which can be parallelized. The name differences are as follows:

  - In an n32 or 64-bit MP version, the routine has the name `DAXPY`, but most of that work is done in the MP routine named `DAXPY.PREGION1`.
  - In an o32-bit version, the `DAXPY` routine is named `daxpy_`, and the MP routine is `_daxpy_519_aaab_`.

- If you perform an `ideal` experiment, the source annotations for 32-bit and 64-bit compilations with the `-g` option differ and are not correct in most cases.

  - In 64-bit source annotations, the exclusive time is correctly shown for each line, but the inclusive time for the first line of the loop (`do` statement) includes the time spent in the loop body. This same time appears on the lines comprising the loop's body, in effect representing a double-counting.

– In 32-bit source annotations, the exclusive time is incorrectly shown for the line comprising the loop's body. The line-level data for the loop-body routine (`_daxpy_519_aaab_`) doesn't refer to proper lines. If the program was compiled with the `-mp_keep` flag, the line-level data should refer to the temporary files that are saved from the compilation. But the temporary files do not contain that information, so no source or disassembly data can be shown. The disassembly data for the main routine does not show the times for the loop body.

– If the 32-bit program was compiled without the `-mp_keep` flag, the line-level data for the loop-body routine is incorrect. Most lines refer to line 0 of the file and the rest to other lines at seemingly random places in the file. Consequently, false annotations will appear on some lines. Disassembly correctly shows the instructions and their data, but the line numbers are wrong. This reflects essentially the same double-counting problem as seen in 64-bit compilations, but the extra counts go to other places in the file, rather than to the first line of the loop.

6.2 Setting Up Output Directories and Files

When you run an experiment, performance data files are written to the current working directory by default. They are named using the following convention:

executable_name.exp_name.exp_type.id

The experiment ID, *id*, consists of one or two letters (designating the process type) and the process ID number. See Table 4 for letter codes and descriptions.

The following are examples of data file names:

```
stat.ideal.m10966
engines.pcsamp.m14493
```

In a single-process application, `ssrun` generates a single performance data file. In a multiprocessor application, `ssrun` generates a performance data file for each process.

You can change the default file name or directory for performance data files using environment variables. See `_SPEEDSHOP_OUTPUT_DIRECTORY` and `_SPEEDSHOP_OUTPUT_FILENAME` in Table 9 for more information.

6.3 Using Run-Time Environment Variables

This section provides information about available environment variables, grouped by functionality:

- User Environment Variables, see Section 6.3.1, page 70.
- Process Tracking Environment Variables, see Section 6.3.2, page 71.
- Expert-Mode Environment Variables, see Section 6.3.3, page 72.

6.3.1 User Environment Variables

A number of environment variables are normally used to control the operation of SpeedShop. Table 9 lists these variables.

Table 9. General Environment Variables

Variable	Description
`_SPEEDSHOP_VERBOSE`	Causes a log of each program's operation to be written to `stderr`. If this variable is set to an empty string, only major events are logged; if it is set to a non-empty string, more detailed events are logged.
`_SPEEDSHOP_SILENT`	Suppresses all SpeedShop output other than fatal error messages. If both `_SPEEDSHOP_VERBOSE` and `_SPEEDSHOP_SILENT` are set, `_SPEEDSHOP_VERBOSE` is ignored.
`_SPEEDSHOP_CALIPER_POINT_SIG` *sig_num*	Causes the specified signal number to be used for recording a caliper point in the experiment.
`_SPEEDSHOP_REUSE_FILE_DESCRIPTORS`	Opens and closes the file descriptors for the output files every time performance data is to be written.

Variable	Description
`_SPEEDSHOP_HWC_COUNTER_NUMBER`	Specifies the counter to be used for `prof_hwc` experiments. Counters are numbered between 0 and 31, and are described in the *MIPS R10000 Microprocessor's User's Manual*, Chapter 14. Counter 0 counters are numbered 0-15, and counter 1 counters are numbered 16–31.
`_SPEEDSHOP_HWC_COUNTER_OVERFLOW`	Specifies the overflow value for the counter to be used in `prof_hwc` experiments. The value chosen can be any number greater than 0. Some choices may produce data that is not statistically random but reflects a correlation between the overflow interval and a cyclic behavior in the application. Users may want to do two or more runs with different overflow values.
`_SPEEDSHOP_OUTPUT_NOCOMPRESS`	Disables the compression of performance data.
`_SPEEDSHOP_OUTPUT_DIRECTORY`	Causes the output data files to be placed in the specified directory rather than the current working directory.
`_SPEEDSHOP_OUTPUT_FILENAME`	Causes the output file to be saved under the specified name. If `_SPEEDSHOP_OUTPUT_FILENAME` is set to `myfile`, the experiment file is named `myfile.`*suffix* (for example, `myfile.m12345`). If `_SPEEDSHOP_OUTPUT_DIRECTORY` is also specified, the directory is prepended to the file name you specify.

6.3.2 Process Tracking Environment Variables

A number of environment variables may be used for controlling the treatment of processes spawned from the original target. Table 10, page 72, lists these variables.

Table 10. Process Tracking Environment Variables

Variable	Description
`_SPEEDSHOP_TRACE_FORK [True|False]`	If True, specifies that processes spawned by calls to `fork()` will be monitored if they do not call `exec()`. If they do call `exec()` and `_SPEEDSHOP_TRACE_FORK_TO_EXEC` is not set to True, the data covering the time between the `fork()` and `exec()` will be discarded. It is True by default.
`_SPEEDSHOP_TRACE_FORK_TO_EXEC [True|False]`	If True, specifies that a process spawned by calls to `fork()` will be monitored, even if they also call `exec()`. It is False by default.
`_SPEEDSHOP_TRACE_EXEC [True|False]`	If True, specifies that a process spawned by calls to any of the various flavors of `exec()` will be monitored. It is True by default.
`_SPEEDSHOP_TRACE_SPROC [True|False]`	If True, specifies that a process spawned by calls to `sproc()` will be monitored. It is True by default.
`_SPEEDSHOP_TRACE_SYSTEM [True|False]`	If True, specifies that `system()` calls will be monitored. It is False by default.

6.3.3 Expert-Mode Environment Variables

A number of variables may be used for debugging and finer control of the operation of SpeedShop. Table 11, page 73, lists these variables.

Table 11. Expert-Mode Environment Variables

Variable	Description
`_SPEEDSHOP_SAMPLING_MODE`	Used for PC sampling and hardware counter profiling. If set to 1, generates data for the base executable only. If not set or set to a value other than 1, data is generated for the executable and all the DSOs it uses.
`_SPEEDSHOP_INIT_DEFERRED_SIG` *sig_num*	If specified, initialization of the experiment is not performed when the target process starts. Initialization is delayed until the specified signal is sent to the process. A handler for the given signal is installed when the process starts. It is the user's responsibility to ensure that it is not overridden by the target code.
`_SPEEDSHOP_SHUTDOWN_SIG` *sig_num*	If specified, termination of the experiment is not performed when the target process exits. Termination happens when the specified signal is sent to the process. A handler for the given signal is installed when the process starts, and it is the user's responsibility to ensure that it is not overridden by the target code.
`_SPEEDSHOP_EXPERIMENT_TYPE`	Passes the name of the experiment to the run–time DSO. It is normally set by `ssrun` but can be overwritten.
`_SPEEDSHOP_MARCHING_ORDERS`	Passes the marching orders of the experiment to the run–time DSO. The marching orders are usually set by `ssrun` from the experiment type, but they can be overwritten.
`_SPEEDSHOP_SBRK_BUFFER_LENGTH`	Defines the maximum size of the internal malloc (memory allocation) area used. This area is completely separate from the user's area and has a default size of 0x100000.

Variable	Description
`_SPEEDSHOP_FILE_BUFFER_LENGTH`	Defines the size of the buffer used for writing the experiment files. The default length is 8 KB. The buffer is used only for writing small records to the file; large records are written directly to avoid the buffering overhead.
`_SPEEDSHOP_DEBUG_NO_SIG_TRAPS`	Disables the normal setting of signal handlers for all fatal and exit signals.
`_SPEEDSHOP_DEBUG_NO_STACK_UNWIND`	Suppresses the stack unwind, as in `usertime` experiments and at caliper samples, for all experiments. The option is used as a workaround for various unwind bugs in `libexc`.

6.4 Using Marching Orders

Using marching orders is another method of specifying what experiment type you want to run. One of the benefits of using marching orders is that it lets you customize experiments.

Each experiment type corresponds to a marching orders specification. You can use marching orders in either of the following ways:

- The `_SPEEDSHOP_MARCHING_ORDERS` environment variable. The following example selects the `usertime` experiment:

  ```
  setenv _SPEEDSHOP_MARCHING_ORDERS ut:cu
  ```

- The `-mo` option on the `ssrun` command line. The following example selects the `pcsamp` experiment:

  ```
  ssrun -mo pc,2,10000,0:cu ...
  ```

- Adding marching orders to a predefined experiment by using the `_SPEEDSHOP_EXTRA_MARCHING_ORDERS` environment variable. The following example generates a useful resource usage graph when viewed with the `cvperf`(1) command:

  ```
  setenv _SPEEDSHOP_EXTRA_MARCHING_ORDERS hb
  ssrun -pcsamp a.out
  ```

If the marching orders on the command line differ from those specified with the environment variable, the command-line version takes precedence.

The number and meaning of the arguments for each marching order depend on the specific marching order. The following specifies PC sampling, using 16-bit bins, sampling every 10 ms, and sampling both the executable and all of its DSOs:

```
pc,2,10000,0
```

The following specifies call stack sampling every 10 ms, based on process virtual time plus system time spent on behalf of the process:

```
ut,10000,2
```

6.4.1 Defining the Base Experiment

The experiment specifier, with which a marching order begins, takes one of the following values:

`ut`		A time experiment that returns real time, virtual time, or user time. The default arguments are `30000,2`. The first argument is the interval between callstack samples in microseconds. The second argument is the timer type used to measure the intervals; the supported values are 0, 1, and 2, with the same meanings as for the second argument of `hb` (described later). The argument value `-1` is not valid for `ut`.
`pc`		A 16-bit or 32-bit PC sampling (`pcsamp`) experiment. The default arguments are `2,10000,0`. The first argument is the size of the sample count bins in bytes. The supported values are `2` (16 bits) and `4` (32 bits). The second argument is the sampling rate in microseconds. Supported values are 10000 (10-millisecond sample interval) and 1000 (1-millisecond sample interval). The third argument is the sampling mode:
	`0`	Selects the user executable and all its dynamic shared objects
	`1`	Selects only the user executable (without any dynamic shared objects)
`it`		A 32-bit `ideal` experiment. Only 4-byte (32-bit) counters are supported.
`mf`		A memory allocation and deallocation experiment that traces calls to `malloc`, `realloc`, `free`, `memalign`, and `valloc` routines.

There are no arguments to this marching order. The arguments to these routines and bad calls are recorded. Bad calls include `malloc` calls of 0 bytes, freeing invalid memory blocks, reallocating invalid memory pointers, and calling `memalign` with invalid arguments. (For descriptions of these routines, see the `malloc`(3) man page.

`fpe` A floating-point exceptions (`fpe`) experiment. There are no arguments. The call stack is sampled whenever a floating-point exception occurs.

`io` An I/O trace experiment. There are no arguments. The start time and end time for each of the following I/O system calls are recorded: `creat`(2), `open`(2), `read`(2), `write`(2), `close`(2), `pipe`(2), `dup`(2), `readv`(2), and `writev`(2).

`mpi` MPI experiment. There are no arguments. The beginning time, ending time, return value, and arguments are recorded.

Note: The output from this experiment can only be displayed by using the `cvperf`(1) user interface; it cannot be displayed through `prof`.

For a list of the routines traced, see .

`hwct` A hardware counter call stack profiling experiment (`_hwct`).

`hwc` A hardware counter profiling experiment (`_hwc`).

`hb` Heart beat data collection. System-wide, per-process, and MPI resource usage data is collected at regular time intervals. If the program creates multiple processes, data is collected for each process. If the process is using the MPI library, MPI library statistics are also recorded.

The default arguments are `1000000,2`. The first argument is the interval in microseconds between samples. The second argument is the time type to use, as follows:

`-1`	Use `alarm`(2) instead of `setitimer`(2) to deliver the periodic signal. In this case, the interval is rounded to the nearest second (periods of less than 1 second are rounded up to 1 second). The interval is in real (wall-clock) time.
`0`	Real (wall-clock) time.
`1`	Virtual time. The timer runs while the user program is executing.

`2`	User time. The timer runs while the user program is executing or the system is processing system calls made by the program.

`cu`	Caliper point usage data collection. It usually appears at the end of a marching order, and there are no arguments. Usage data is recorded at caliper points. As with the `hb` marching order, you can collect system-wide, per-process, and MPI resource usage data.

The `hb` marching order collects data based on time, and the `cu` marching order is based on caliper points that you can set anywhere in your source code. For more information on setting caliper points, see Section 6.8, page 86.

6.5 Running Experiments

This section describes how to use `ssrun` to perform experiments. For information on using `pixie` directly, see Chapter 8, page 121.

6.5.1 `ssrun` Syntax

The `ssrun` command takes the following form:

`ssrun` *ssrun_options* -*exp_type executable_name executable_args*

The arguments are as follows:

ssrun_options	Zero or more of the flags described in Table 12, page 78. These options control the data collection and the treatment of descendent processes or programs, and they specify how the data is to be externalized.
-exp_type	The experiment type. Experiments are described in detail in Chapter 4, page 49.
executable_name	The name of the program on which you want to run an experiment.
executable_args	Arguments to your program, if any.

The `ssrun` command generates a performance data file that is named as described in Section 6.2, page 69.

Table 12. Flags for the `ssrun` Command

Name	Result
`-hang`	Specifies that the process should be left waiting just before executing its first instruction. This allows you to attach the process to a debugger.
`-mo` *marching_orders*	Allows you to specify marching orders. If this option is used, the environment variable `_SSRUNTIME_MARCHING_ORDERS` is not examined.
`-name` *target_name*	Specifies that the target should be run with argv[0] set to *target_name*.
`-port` *hostname portno*	Specifies that the process is to be left waiting, and notifications of status are to be sent to the socket on the host named by hostname and the port specified by *portno*. When the process is ready, a message of the form "running *pid host*" will be sent to inform the requester of the PID of the target process and the host, which may be remote. A debugger can then attach to it and take control of its execution.
`-purify`	Can be used only when the Purify product is installed. Specifies that `purify` should be run on the target, and then runs the resulting "purified" executable. Note that `-purify` and SpeedShop performance experiments cannot be combined.
`-quiet`	Suppresses all output other than error messages. If `-quiet` is specified, the `_SPEEDSHOP_SILENT` environment variable is also set.
`-v`	Prints a log of the operation of `ssrun` to `stderr`. The same behavior occurs if the environment variable `_SPEEDSHOP_VERBOSE` is set to an empty string.
`-V`	Prints a detailed log of the operation of `ssrun` to `stderr`. The same behavior occurs if the environment variable `_SPEEDSHOP_VERBOSE` is set to a nonzero-length string. This option can be used to see how to set the various environment variables, and how to invoke instrumentation when necessary.
`-workshop`	Specifies special instrumentation so that the experiment files can be read by WorkShop's `cvperf` analyzer.
`-x` *display-id window-id*	Specifies that the process is to be left waiting and that the window of the WorkShop debugger requesting the creation (as specified by the *display-id* and *window-id* arguments on the command line) be informed of the PID of the target process. A debugger can then attach to it and take control of its execution.

6.5.2 ssrun Examples

This section provides examples of using ssrun with options and experiment types. For additional examples, see Chapter 2, page 13, or Chapter 3, page 31.

6.5.2.1 Example Using the pcsampx Experiment

The pcsampx experiment collects data to estimate the actual CPU time for each source code line, machine instruction, and function in your program. The optional x suffix causes a 32-bit bin size to be used, allowing a larger number of counts to be recorded. For a more detailed description of the pcsamp experiment, see Section 4.3, page 51.

The following example performs a pcsampx experiment on the generic executable:

```
ssrun -pcsampx generic
```

To see the performance data that has been generated, run prof on the performance data file, generic.pcsampx.12185, as shown in the following example:

```
prof generic.pcsampx.m12185
```

The report is printed to stdout. (This layout of this report has been altered slightly to accommodate presentation needs.) For more information on prof and the reports generated by prof, see Chapter 7, page 93.

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Mon Feb  2 15:08:14 1998
   prof generic.pcsampx.m12185
                 generic (n32): Target program
                       pcsampx: Experiment name
               pc,4,10000,0:cu: Marching orders
                 R4400 / R4000: CPU / FPU
                             1: Number of CPUs
                           175: Clock frequency (MHz.)
  Experiment notes--
          From file generic.pcsampx.m12185:
        Caliper point 0 at target begin, PID 12185
                        /usr/demos/SpeedShop/linpack.demos/c/generic
        Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of statistical PC sampling data (pcsampx)--
                          2729: Total samples
                        27.290: Accumulated time (secs.)
                          10.0: Time per sample (msecs.)
                             4: Sample bin width (bytes)
-------------------------------------------------------------------------
Function list, in descending order by time
-------------------------------------------------------------------------
 [index]      secs    %    cum.%   samples  function (dso: file, line)

     [1]    25.470  93.3%  93.3%      2547  anneal (generic: generic.c,
1573)
     [2]     1.100   4.0%  97.4%       110  slaveusrtime (dlslave.so: dlslave.c, 22)
     [3]     0.310   1.1%  98.5%        31  __read (libc.so.1: read.s, 20)
     [4]     0.240   0.9%  99.4%        24  cvttrap (generic: generic.c, 317)
     [5]     0.150   0.5%  99.9%        15  _xstat (libc.so.1: xstat.s,
12)
     [6]     0.010   0.0% 100.0%         1  __write (libc.so.1: write.s, 20)
     [7]     0.010   0.0% 100.0%         1  _morecore (libc.so.1: malloc.c, 632)

            27.290 100.0% 100.0%      2729  TOTAL
```

6.5.2.2 Example Displaying Data in WorkShop

To use WorkShop's graphic user interface to display the information gathered by `ssrun`, include the `-workshop` option on the `ssrun` command line, as shown in the following example:

```
ssrun -workshop -pcsampx generic
```

The result is a file viewable through the `cvperf` WorkShop command:

```
cvperf generic.pcsampx.m44800
```

6.5.2.3 Example Using the `-v` Option

To get information about how a SpeedShop experiment is set up and performed, you can supply the `-v` option to `ssrun`.

The following example performs another `pcsampx` experiment on the `generic` executable:

```
ssrun -v -pcsampx generic
```

The `ssrun` command writes the following output to `stderr`. It displays information as the command line is parsed and shows the environment variables that `ssrun` sets.

```
fraser 75% ssrun -v -pcsampx generic

ssrun: setenv _SPEEDSHOP_MARCHING_ORDERS pc,4,10000,0:cu
ssrun: setenv _SPEEDSHOP_EXPERIMENT_TYPE pcsampx
ssrun: setenv _SPEEDSHOP_TARGET_FILE generic
ssrun: setenv _RLD_LIST libss.so:libssrt.so:DEFAULT
...
```

The `_RLD32_LIST` environment variable is for new 32-bit programs. `_RLD64_LIST` is used for 64-bit programs. If neither is set, the value of `_RLD_LIST` is the default.

6.5.3 Using `ssrun` with a Debugger

To use the `ssrun` command in conjunction with a debugger such as `dbx` or the WorkShop debugger, you need to call `ssrun` with the `-hang` option and the name of your program.

Follow these steps to run the floating-point exceptions trace experiment on `generic`, and then run `generic` in a debugger.

1. Call `ssrun` as follows:

   ```
   ssrun -hang -fpe generic
   ```

   The `ssrun` command parses the command line, sets up the environment for the experiment, calls the target process using `exec`, and hangs the target process on exiting from the call to `exec`.

2. Note the process ID returned by `ssrun`.

3. Start your debugging session as follows:

   `cvd -pid` *process_id_number*

4. Attach the process to the debugger.

5. Run the process from the debugger.

You can also invoke `ssrun` from within a debugger. In this case, `ssrun` leaves the target hung on exiting the call to `exec` and informs the debugger of that fact.

You can also use a debugger to set calipers for the purpose of recording performance data for a part of your program. See Section 6.8, page 86, for more information on setting calipers.

6.6 Running Experiments on MPI Programs

The Message Passing Interface (MPI) is a library specification for message passing, proposed as a standard by a committee of vendors, implementors, and users. It allows processes to communicate by passing data messages to other processes, even those running on distant computers.

SpeedShop offers two types of experiments for MPI programs, the first of which can only be displayed in `cvperf`(1):

MPI tracing experiments	Traces the use of MPI send, receive, and synchronization routines and a few other routines. See the following section for more information.

Other SpeedShop experiments	Generates other SpeedShop experiments, such as `usertime` and `pcsamp`. For more information, see Section 6.6.2, page 85.

Note: Before executing the `ssrun` command on an MPI executable, you must set the `MPI_RLD_HACK_OFF` environment variable as follows:

```
% setenv MPI_RLD_HACK_OFF 1
```

6.6.1 Generating MPI Tracing Experiments

MPI tracing experiments tell you how many times, and at what points in the application, various routines from the MPI library are called.

You can use either of the following versions of the `ssrun` command on an executable named `a.out`:

```
% mpirun -np 4 ssrun -mpi a.out
% mpirun -np 4 ssrun -mo mpi:cu a.out
```

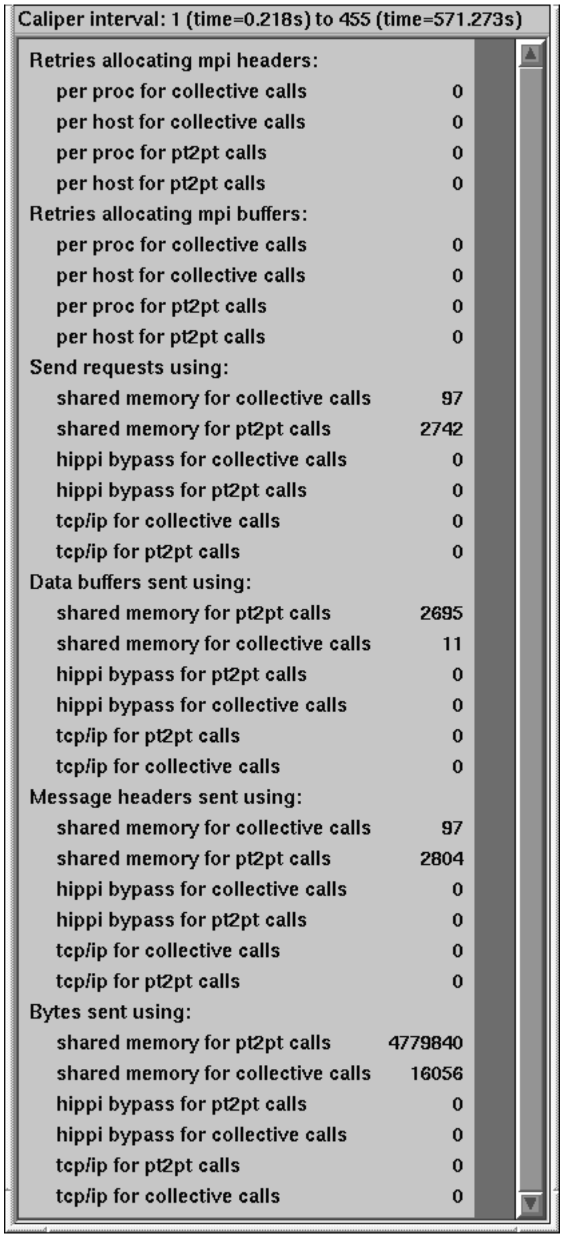
If you are running the application on four processors, you will see five output files: one for each processor and one for the master process. The identifier portions of the file names will start either with `m` for the master process or `f` (forked) for a process running on one of the processors. Names such as the following might be assigned to an executable with the name `verge`:

```
verge.mpi.m12345
verge.mpi.f12346
verge.mpi.f12347
verge.mpi.f12348
verge.mpi.f12349
```

The identifiers do not correspond to a processor number.

MPI output from the `ssrun` command can only be viewed in the `WorkShop Performance Analyzer` window. You can bring that window up with the `cvperf`(1) command. You can view the information in either chart or numerical format. Charts that do not contain data are not displayed. For an example of a portion of a numerical display, see Figure 2, page 84.

Note: The MPI tracing experiment does not track down communicators, and it does not trace all collective operations. These limitations may also affect the translation of some events by `ssfilter`(1).



```
Caliper interval: 1 (time=0.218s) to 455 (time=571.273s)
Retries allocating mpi headers:
    per proc for collective calls            0
    per host for collective calls            0
    per proc for pt2pt calls                 0
    per host for pt2pt calls                 0
Retries allocating mpi buffers:
    per proc for collective calls            0
    per host for collective calls            0
    per proc for pt2pt calls                 0
    per host for pt2pt calls                 0
Send requests using:
    shared memory for collective calls      97
    shared memory for pt2pt calls         2742
    hippi bypass for collective calls        0
    hippi bypass for pt2pt calls             0
    tcp/ip for collective calls              0
    tcp/ip for pt2pt calls                   0
Data buffers sent using:
    shared memory for pt2pt calls         2695
    shared memory for collective calls      11
    hippi bypass for pt2pt calls             0
    hippi bypass for collective calls        0
    tcp/ip for pt2pt calls                   0
    tcp/ip for collective calls              0
Message headers sent using:
    shared memory for collective calls      97
    shared memory for pt2pt calls         2804
    hippi bypass for collective calls        0
    hippi bypass for pt2pt calls             0
    tcp/ip for collective calls              0
    tcp/ip for pt2pt calls                   0
Bytes sent using:
    shared memory for pt2pt calls      4779840
    shared memory for collective calls   16056
    hippi bypass for pt2pt calls             0
    hippi bypass for collective calls        0
    tcp/ip for pt2pt calls                   0
    tcp/ip for collective calls              0
```

Figure 2. MPI Numerical Format

The following routines are traced:

`MPI_Barrier`	`MPI_Send`
`MPI_Bsend`	`MPI_Ssend`
`MPI_Rsend`	`MPI_Isend`
`MPI_Ibsend`	`MPI_Issend`
`MPI_Irsend`	`MPI_Sendrecv`
`MPI_Sendrecv_replace`	`MPI_Bcast`
`MPI_Recv`	`MPI_Irecv`
`MPI_Wait`	`MPI_Waitall`
`MPI_Waitany`	`MPI_Waitsome`
`MPI_Test`	`MPI_Testall`
`MPI_Testany`	`MPI_Testsome`
`MPI_Request_free`	`MPI_Cancel`
`MPI_Pcontrol`	

6.6.2 Generating Other Experiments for Programs Using MPI

If your program uses MPI, you must set up SpeedShop experiments that will be displayed in `prof` a little differently. There are two ways to accomplish this. The first method takes two steps:

1. Set up a shell script that contains the call to `ssrun` and the experiment you want to run.

   For example, if you have a program called `testit` and you want to run the `pcsampx` experiment with a script named `exp_script`, the process might look like the following:

   ```
   #!/bin/sh
   ssrun -pcsampx testit
   ```

2. Call `mpirun` with the script name using one of the following commands:

   ```
   % mpirun -np 6 exp_script
   % mpirun host1 2, host2 2 exp_script
   ```

The second method is to use one of the following:

```
% mpirun -np 6 ssrun -pcsampx testit
% mpirun host1 2, host2 2 ssrun -pcsampx testit
```

The master experiment file created on each MPI host might not contain performance data from the application (depending on the MPI version) but from a master program that spawns the members of an application group. You can choose to exclude that file from performance analysis.

When using `ssrun -ideal` or `ssrun -purify`, you should take care that the code for each separate host executes out of a different physical directory, not out of the same directory mounted by the network file system (NFS). During process creation, instrumentation is performed, and since different hosts may have different versions of the same named library (`libc.so.1`, for example), conflicts may occur. You may also need to use the `-d` option with `mpirun` to specify the directory on each host.

6.7 Running Experiments on Programs Using Pthreads

Pthreads is the multithreading model defined by the POSIX operating system standard (IEEE1003.1c-1995). This standard contains a set of interfaces and semantics for creating and managing threads within the POSIX operating system definition. The basic Silicon Graphics pthreads implementation consists of a library and a header file.

Applications using pthreads are specifically identified by SpeedShop. Performance data collection is done on a per-program basis, rather than on a per-pthread basis. Under IRIX 6.2, 6.3, and 6.4, SpeedShop creates as many experiment files as the number of `sproc`(2) system calls used by the pthreads library to create and manage the pthreads. In addition, `cm_usage` data is not supported, and `SIGTERM` is reserved to be used to terminate the application normally. You should analyze all the experiment files together via `prof` to get a valid profile for the code. Under IRIX 6.5, SpeedShop creates only one experiment file. For `usertime` and `fpe` experiments, however, you can specify the `-pthreads` option with `prof` to get per-pthread performance reports.

6.8 Using Calipers

In some cases, you may want to generate performance data reports for only a part of your program. You can do this by selecting caliper points to identify the area of your program or the time interval during execution for which you want to see performance data. When you run `prof`, you can specify a region for which to generate a report by supplying the `-calipers` option and the appropriate caliper numbers. For more information on `prof -calipers`, see Section 7.3.3, page 113.

Table 13, page 87, shows the different ways you can set caliper points.

Table 13. Setting Caliper Points

Use This Approach...	For These Benefits...
Explicitly link with the SpeedShop run-time and call `ssrt_caliper_point` to set a caliper sample.	Lets you set a caliper point at a specific location in the source program.
Set pollpoint caliper points at specified time intervals during program execution using the `_SPEEDSHOP_POLLPOINT_CALIPER_POINT` environment variable.	Lets you set caliper points at time intervals rather than at places in the code.
Define a signal to be used to set a caliper sample by specifying a signal as a value to the environment variable `_SPEEDSHOP_CALIPER_POINT_SIG` and then sending the target the given signal.	Useful if you want to be able to set a caliper point as your program is running.
Set a caliper sample trap in `dbx` or the WorkShop debugger. Setting a trap involves setting a breakpoint and evaluating the expression `libss_caliper_point(1)` when the process stops.	Useful if you are working with a debugger in conjunction with SpeedShop.

An implicit caliper point is always present at the start of execution of the process. A final caliper point is set when the process calls `_exit`. The implicit caliper point at the beginning of the program is numbered 0, the first caliper point recorded is numbered 1, and any additional caliper points are numbered sequentially.

In addition, caliper points are automatically set under the following circumstances to ensure that at least one valid set of data is recorded.

- When a fatal signal is received, such as `SIGQUIT`, `SIGILL`, `SIGTRAP`, `SIGABRT`, `SIGEMT`, `SIGFPE`, `SIGBUS`, `SIGSEGV`, `SIGSYS`, `SIGXCPU`, or `SIGXFSZ`. Note that this list does not and cannot include `SIGKILL`.
- When the program calls an `exec` function, such as `execve()` or `execvp()`.

- When an exit signal is received, such as `SIGHUP`, `SIGINT`, `SIGPIPE`, `SIGALRM`, `SIGTERM`, `SIGUSR1`, `SIGUSR2`, `SIGPOLL`, `SIGIO`, `SIGRTMIN`, or `SIGRTMAX`.

6.8.1 Setting Calipers with `ssrt_caliper_point`

To set caliper points with `ssrt_caliper_point`(3), follow these steps:

1. Insert calls to `ssrt_caliper_point` in your source code. Call the function with the argument `1` (meaning, True) and a string to help identify the caliper point in the experiment file later on.

   ```
   ...
   ssrt_caliper_point(1,"bgn_calc");
   ...
   ```

   You can insert one or more calls at any point in your code.

2. Link the SpeedShop library `libss.so` into your application.

   The library should be placed last on the link line.

3. Run your program with `ssrun` and the desired experiment type.

   For example, if you want to run the `ideal` experiment on `generic`:

   ```
   ssrun -ideal generic
   ```

   The caliper points you have set in the source file are recorded in the performance data file that is generated by `ssrun`.

6.8.2 Setting Time-Oriented Calipers

To add caliper points at a regular time interval into your experiment file, set the `_SPEEDSHOP_POLLPOINT_CALIPER_POINT` environment variable before you generate an experiment. It takes the following form:

`_SPEEDSHOP_POLLPOINT_CALIPER_POINT` *timer_type*, *timer_interval*

timer_type One of the following:

`0`	Real time. This is the total time a program spent while executing. It includes both time spent when a program is swapped out waiting for a CPU and the time the operating

	system is in control, performing some task for the program such as I/O or executing a system call.
`1`	Process virtual time. This is the time spent when the program is actually running. This does not include either the time spent when a program is swapped out waiting for a CPU or the time the operating system is in control, performing some task for the program such as I/O or executing another system call.
`2`	CPU time. This is process virtual time plus the time the system is running on behalf of the process. The system time could include performing I/O or executing other system calls.

timer_interval The interval, in seconds, at which a new caliper will be set.

The caliper points you have set with the `_SPEEDSHOP_POLLPOINT_CALIPER_POINT` environment variable are recorded in the performance data file that is generated by `ssrun`.

6.8.3 Setting Calipers with Signals

To set calipers with signals, follow these steps:

1. Set the `_SPEEDSHOP_CALIPER_POINT_SIG` variable to the signal number you want to use.

   Choose a signal that does not terminate the program. The signal should also not be caught by the target program; doing so would interfere with its triggering a caliper point.

   The following signals are good choices because they do not have system-defined semantics already associated with them:

   ```
   SIGUSR1 16      /* user defined signal 1 */
   SIGUSR2 17      /* user defined signal 2 */
   ```

2. Run `ssrun` with your program.

3. Enter a command such as `ps` or `top` to determine the process ID of `ssrun`. This is also the process ID of the program you are working on.

4. Send the signal you used in step 1 to the process using the `kill` command:

   `kill -`*sig_num pid*

   A caliper point is set at the point in the program where the signal was received by the SpeedShop run time.

6.8.4 Setting Calipers with a Debugger

From either `dbx` or the WorkShop debugger, you can set a caliper point anywhere it is possible to set a breakpoint: at a function entry or exit, a line number, an execution address, a watchpoint, or a pollpoint (timer-based). You can also attach conditions and or cycle counts.

Use the following procedure:

1. Set a breakpoint in your program where you want a caliper point.

2. When the process stops, evaluate the expression `ssrt_caliper_point`(3). The evaluation of the expression always returns zero, but a side effect of the evaluation is the recording of the appropriate data.

3. Resume execution of the process.

6.9 Effects of `ssrun`

When you call `ssrun`, the system performs the following operations for all experiments:

- Sets various environment variables like `_SPEEDSHOP_MARCHING_ORDERS` and `_SPEEDSHOP_EXPERIMENT_TYPE`.

  For more information on these environment variables, see Section 6.3, page 70.

- Inserts the SpeedShop libraries `libss.so` and `libssrt.so` as part of your executable using the environment variable `_RLD_LIST`.

- Invokes the target process by calling `exec()`.

- The SpeedShop run-time library writes the appropriate experiment data to the output file.

6.9.1 Effects of `ssrun -ideal`

When you run an `ideal` experiment, the following additional operations occur:

- `libpixrt.so` is inserted first in the executable's library list.
- `libssrt.so` and `libss.so` are inserted in the executable's library list.
- `ssrun` runs `pixie`(1) on all the libraries that the program uses, as well as on the executable.

  The generated pixified versions have an extension that depends on the ABI:

  – `.pixie` for the executable

  – `.pix32` for all o32 libraries

  – `.pixn32` for all n32 libraries

  – `.pix64` for all 64-bit libraries

  The generated files are written either to the current working directory or, if set, to the directory specified by the `_SPEEDSHOP_OUTPUT_DIRECTORY` environment variable. They include code that allows performance data to be collected for each function and basic block.

  For more information on the `ideal` experiment, see Section 4.4, page 52.

Analyzing Experiment Results: `prof` [7]

This chapter provides information on how to view and analyze experiment results. It consists of the following sections:

- Using `prof` to Generate Performance Reports, see Section 7.1, page 93.
- Using `prof` with `ssrun`, see Section 7.2, page 98.
- Using `prof` Options, see Section 7.3, page 106.
- Generating Reports for Different Machine Types, see Section 7.4, page 118.
- Generating Reports for Multiprocessed Executables, see Section 7.5, page 119.
- Generating Compiler Feedback Files, see Section 7.6, page 119.

7.1 Using `prof` to Generate Performance Reports

Performance data is examined using `prof`, a text-based report generator that prints to `stdout`.

To generate a report from performance data gathered during experiments recorded by `ssrun`(1) or `pixie`(1):

`prof` [*`options`*] *executable_name* [*`speedshop_data_file`*] | [*`pixie_counts_file`*]

7.1.1 `prof` Arguments

The arguments for `prof` when used with data files from `ssrun` or `pixie` are as follows:

options	Zero or more of the options described in Table 14, page 94.
executable_name	The name of the executable file created by the compiler.
speedshop_data_file	One or more names of performance data files generated by `ssrun`.

pixie_counts_file	One or more names of data files generated by `pixie` with `.Counts` suffixes.

7.1.2 `prof` Options

Table 14, page 94, lists `prof` options. For more information, see the `prof`(1) man page.

Table 14. Options for `prof`

Name	Result
`-archinfo`	Reports the number of times each register was used as a destination, base (integer registers only) or source, how many times each instruction opcode was used, and some detailed statistics concerning branches jumps, and how many delay slots were filled with no-op instructions. Works only with `ideal` experiments.
`-basicblocks`	Prints a list of all the basic blocks executed, ordered by the number of cycles spent in each basic block. Works only with ideal experiments.
`-b[utterfly]`	Causes `prof` to print a report showing the callers and callees of each function, with inclusive time attributed to each. For `ideal` experiments, the attribution is based on a heuristic. For the various callstack sampling and tracing experiments, the attribution is precise, although `usertime`, `totaltime`, and some `_hwctime` experiments are statistical in nature. This option is ignored for experiments in which the data does not support inclusive calculations. It delivers the same display as `-gprof`.
`-calipers` [*n1*] *n2*	Restricts analysis to a segment of program execution. This option works only for SpeedShop experiments. Causes `prof` to compute the data between caliper points *n1* and *n2*, rather than for the entire experiment. If *n1* >= *n2*, an error is reported. If *n1* is negative, it is set to the beginning of the experiment. If *n2* is greater than the maximum number of caliper points recorded, it is set to the maximum. If *n1* is omitted, zero (the beginning of the program) is assumed.

Name	Result
`-calls`	Sorts the function list by the number of procedure calls rather than by time. This option can only be used when generating reports for `ideal` experiments or for basic block counting data obtained with `pixie`.
`-c[lock]` [*n*]	Sets the CPU clock speed to (*n*), expressed in megahertz. This option is useful when generating reports for `ideal` experiments or for basic block counting data obtained with `pixie`. The default is the clock speed of the machine on which the performance data was collected.
`-[no]cordfb`	Disables or enables cord feedback file generation for the executable only. Cord feedback is used to arrange procedures in the binary in an optimal ordering. This improves both paging and instruction cache performance. Users can use `cord`(1) or `ld`(1) to actually do the procedure ordering. For more information on how to reorder code regions, see the *MIPSpro Compiling and Performance Tuning Guide*.
`-cordfball`	Enables cord feedback for the executable and all DSOs.
`-cycle` *n*	Sets the cycle time to *n* nanoseconds.

`-debug`: *dbg_flags*	Sets *dbg_flags* to combinations of the following:
	`GPROF_FLAG 0x00000001`
	`COUNTS_FLAG 0x00000002`
	`SAMPLE_FLAG 0x00000004`
	`MISS_FLAG 0x00000008`
	`FEEDBACK_FLAG 0x00000010`
	`CORD_FLAG 0x00000020`
	`USERPC_FLAG 0x00000040`
	`MDEBUG_FLAG 0x00000080`
	`BEAD_FLAG 0x00000100`
	`LIBSSRT_FLAG 0x00000200`

`-dis[assemble]`	Disassembles and annotates the analyzed object code with cycle times if you have run an `ideal` experiment, collected data using `pixie`, or have run a `pcsamp` or `prof_hwc` experiment.
`-dislimit` *n*	Disassembles only those basic blocks with a frequency >= *n*.
`-dso` [*dso_name*]	Generates a report only for the named DSO. If you do not specify a value for *dso_name*, `prof` prints a list of applicable DSO names. Only the base name, not the full path name, of the DSO needs to be specified.
`-dsolist`	List all the DSOs in the program and their start and end text addresses.

Name	Result
`-e[xclude]` *procs*	Excludes information on the specified procedures. If you specify uppercase `-E`, `prof` also omits the specified procedures from the base upon which it calculates percentages.
`-feedback`	Produces files with information that can be used to (a) arrange procedures in the binary in an optimal ordering using `cord`, and (b) tell the compiler how to optimize compilation of the program using `cc -fb filename.cfb`. This option can be used when generating reports for `ideal` experiments or for basic block counting data obtained with `pixie`. `cord` feedback files are named *program.fb* or *libso.fb*. Compiler feedback files are named *program.cfb* or *libso.cfb*. These are binary files and may be dumped using the `fbdump` command. Procedures are normally ordered by their measured invocation counts; if `-gprof` is also specified, procedures are ordered using call graph counts, rather than invocation counts.
`-h[eavy]`	Lists the most heavily used lines of source code in descending order of use, sorting lines by their frequency of use. This option can be used when generating reports for `ideal`, `pcsamp`, or `prof_hwc` experiments or for basic block counting data obtained with `pixie`.
`-inclusive`	Sorts function list by inclusive data rather than by exclusive data. This option can only be used when generating reports for those experiments that have inclusive data; it is ignored for others.
`-l[ines]`	Lists the most heavily used lines of source code in descending order of use, but lists lines grouped by procedure, sorted by cycles executed per procedure. This option can be used when generating reports for `ideal`, `pcsamp`, or `prof_hwc` experiments, or for basic block counting data obtained with `pixie`.
`-nh`	Supresses various header blocks from the output.
`-o[nly]` *procs*	Reports information on only the procedures specified. If you specify uppercase `-O`, `prof` uses only the procedures, rather than the entire program, as the base upon which it calculates percentages.
`-pthreads` *pthread id*	Analyzes data only for the specified pthread identifier (for `usertime` and `fpe` experiments on applications that use pthreads on IRIX 6.5 or later systems).
`-q[uit]` *n*	Condenses output listings by truncating `-p[rocedures]`, `-h[eavy]`, `-l[ines]`, and `-gprof` listings. You can specify *n* in three ways: *n*, an integer, truncates everything after *n* lines;

Name	Result
	n%, an integer followed by a percent sign, does not print any procedure or line with less than *n* in the % column;
	*n*cum%, an integer followed by cum%, does not print any procedure or line with more than *n* in the cum% column. That is, it truncates the listing after the last procedure or line which brings the cumulative total to *n*%. If -gprof is also specified, it behaves the same as -q *n*%.
	For example, -q 15 truncates each part of the report after 15 lines of text. -q 15% truncates each part of the report that represents less than 15% of the whole, and -q 15cum% truncates each part of the report that has a cumulative percentage above 15%.
-rel[ative]	Shows percentage attribution in a butterfly report relative to the central function. The default is to show percentages as absolute percentages over the whole run.
-r12000|-r10000|-r8000 |-r5000|-r4000 |-r3000	Overrides the default processor scheduling model that prof uses to generate a report. If this option is not specified, prof uses the scheduling model for the processor on which the experiment is being run.
-showss	Enables the display of functions from the SpeedShop run–time DSO. Usually those functions are suppressed from the reports and computations. In addition, some statistics for the prof command's own memory usage will be printed.
-S (-source)	Disassembles and annotates the analyzed object code with cycle times, or PC samples, and source code, if you have run an ideal, pcsamp, or prof_hwc experiment, or collected data using pixie.
-u[sage]	Prints a report on system statistics and timers.
-ws	Generates, for the executable only, a working-set file for the current caliper setting.
-wsall	Generates, for the executable and all the non-ignored DSOs, a working-set file for the current caliper setting.
-xdso *dso_name*	Excludes the named DSO from any reports. Only the base name, not the full path name, of the DSO need be specified; the .so suffix is required. Multiple instances of the -xdso flag can be specified.

7.1.3 `prof` Output

The `prof` command generates a performance report that is printed to `stdout`. Warning and fatal errors are printed to `stderr`.

Note: Fortran alternate entry point times are attributed to the `main` function or subroutine, since there is no general way for `prof` to separate the times for the alternate entries.

7.2 Using `prof` with `ssrun`

When you call `prof` with one or more SpeedShop performance data files, it collects the data from all the output files and produces a listing. The `prof` command is able to detect which experiment was run and generate an appropriate report. It provides reports for all experiment types.

In cases where `prof` accepts more than one data file as input, it sums up the results. The multiple input data files must be generated from the same executable, using the same experiment type.

The `prof` command may report times for procedures named with a prefix of `*DF*`, for example `*DF*_hello.init_2`. `DF` stands for *Dummy Function* and indicates cycles spent in parts of text which are not in any function: `init`, `fini`, and `MIPS.stubs` sections, for example.

The most frequently used reports that `prof` generates are described in the following sections:

- `usertime` Experiment Reports, see Section 7.2.1, page 98.
- `pcsamp` Experiment Reports, see Section 7.2.2, page 100.
- Hardware Counter Experiment Reports, see Section 7.2.3, page 101.
- `ideal` Experiment Reports, see Section 7.2.4, page 102.
- `fpe` Trace Reports, see Section 7.2.5, page 105.

7.2.1 `usertime` Experiment Reports

For `usertime` experiments, `prof` generates CPU times for individual routines and shows how those times compare with the rest of the program. The column heading are as follows:

- The `index` column provides an index number for reference.

- The excl.secs column shows how much time, in seconds, was spent in the function itself (exclusive time). For example, less than one hundredth of a second was spent in __start(), but 0.03 of a second was spent in fread.
- The excl.% column shows the percentage of a program's total time that was spent in the function.
- The cum.% column shows the percentage of the complete program time that has been spent in the functions that have been listed so far.
- The incl.secs column shows how much time, in seconds, was spent in the function and descendents of the function.
- The incl.% column shows the cumulative percentage of inclusive time spent in each function and its descendents.
- The samples column provides the number of samples of the function and all of its descendants.
- The procedure (dso:file,line) columns list the function name, its DSO name, its file name, and its line number

The following example is an abbreviated version of the full report. For a complete report, see Section 2.3.1.2, page 17.

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Mon Feb  2 11:07:15 1998
   prof generic.usertime.m10981
                 generic (n32): Target program
                      usertime: Experiment name
                         ut:cu: Marching orders
                 R4400 / R4000: CPU / FPU
                             1: Number of CPUs
                           175: Clock frequency (MHz.)
  Experiment notes--
          From file generic.usertime.m10981:
        Caliper point 0 at target begin, PID 10981
                        /usr/demos/SpeedShop/linpack.demos/c/generic
        Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of statistical callstack sampling data (usertime)--
                           809: Total Samples
                             0: Samples with incomplete traceback
                        24.270: Accumulated Time (secs.)
                          30.0: Sample interval (msecs.)
```

```
-------------------------------------------------------------------------
Function list, in descending order by exclusive time
-------------------------------------------------------------------------
 [index]  excl.secs excl.%   cum.%  incl.secs incl.%    samples  procedure
(dso: file, line)

     [4]     22.770  93.8%   93.8%     22.770  93.8%        759  anneal
(generic: generic.c, 1573)
```

7.2.2 pcsamp Experiment Reports

For `[f]pcsamp[x]` experiments, `prof` generates a function list annotated with the number of samples taken for the function and the estimated time spent in the function. The column headings are as follows:

- The `secs` column shows the amount of CPU time that was spent in the function.
- The `%` column shows the percentage of the total program time that was spent in the function.
- The `cum.%` column shows the percentage of the complete program time that has been spent in the functions that have been listed so far.
- The `samples` column shows how many samples were taken when the process was executing in the function.
- The `function (dso:file, line)` columns list the function, its DSO name, its file name, and its line number.

The following is output from an `fpcsamp` experiment:

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Mon Feb  2 11:01:36 1998
   prof generic.fpcsamp.m11140
                 generic (n32): Target program
                       fpcsamp: Experiment name
                pc,2,1000,0:cu: Marching orders
                 R4400 / R4000: CPU / FPU
                             1: Number of CPUs
                           175: Clock frequency (MHz.)
  Experiment notes--
          From file generic.fpcsamp.m11140:
        Caliper point 0 at target begin, PID 11140
                        /usr/demos/SpeedShop/linpack.demos/c/generic
        Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of statistical PC sampling data (fpcsamp)--
                         23828: Total samples
                        23.828: Accumulated time (secs.)
                           1.0: Time per sample (msecs.)
                             2: Sample bin width (bytes)
-------------------------------------------------------------------------
Function list, in descending order by time
-------------------------------------------------------------------------
 [index]      secs    %    cum.%   samples  function (dso: file, line)

     [1]    22.279  93.5%  93.5%     22279  anneal (generic: generic.c,1573)
```

7.2.3 Hardware Counter Experiment Reports

For the various hwc experiments, prof generates a function list annotated with the number of overflows of hardware counters generated by the function. The column headings are as follows:

- The counts column shows the extrapolated event count based on the number of samples and the overflow value for the particular counter.
- The % column shows the percentage of the program's overflows that occurred in the function.
- The cum.% column shows the percentage of the program's overflows that occurred in the functions that have been listed so far.
- The samples column shows the number of times the program counter was sampled during execution of the function.
- The function (dso: file, line) columns show the name, the DSO, the file name, and line number of the function.

The following is output from a dsc_hwc hardware counter experiment:

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Mon Feb  2 11:11:44 1998
   prof generic.dsc_hwc.m294398
                 generic (n32): Target program
                       dsc_hwc: Experiment name
                 hwc,26,131:cu: Marching orders
               R10000 / R10010: CPU / FPU
                            16: Number of CPUs
```

```
                          195: Clock frequency (MHz.)
     Experiment notes--
             From file generic.dsc_hwc.m294398:
           Caliper point 0 at target begin, PID 294398
                           /usr/demos/SpeedShop/linpack.demos/c/generic
           Caliper point 1 at exit(0)
   -------------------------------------------------------------------------
   Summary of R10K perf. counter overflow PC sampling data (dsc_hwc)--
                            6: Total samples
          Sec cache D misses (26): Counter name (number)
                          131: Counter overflow value
                          786: Total counts
   -------------------------------------------------------------------------
   Function list, in descending order by counts
   -------------------------------------------------------------------------
    [index]        counts     %   cum.%   samples  function (dso: file, line)

        [1]           131  16.7%  16.7%         1  init2da (generic: generic.c, 1430)
        [2]           131  16.7%  33.3%         1  genLog (generic: generic.c, 1686)
        [3]           131  16.7%  50.0%         1  _write (libc.so.1: writeSCI.c, 27)
                      393  50.0% 100.0%         3  **OTHER** (includes excluded DSOs, rld, etc.)

                      786 100.0% 100.0%         6  TOTAL
```

7.2.4 `ideal` Experiment Reports

For `ideal` experiments, `prof` generates a function list annotated with the number of cycles and instructions attributed to the function and the estimated time spent in the function.

The `prof` command does not take into account interactions between basic blocks. Within a single basic block, `prof` computes cycles for one execution and multiplies it with the number of times that basic block is executed.

If any of the object files linked into the application have been stripped of line number information (with `ld -x`, for example), `prof` warns about the affected procedures. The instruction counts for such procedures are shown as a procedure total, not on a per-basic-block basis. Where a line number would normally appear in a report on a function without line numbers, question marks appear instead. The column headings are as follows:

- The `excl.secs` column shows the minimum number of seconds that might be spent in the function under ideal conditions.

- The `excl.%` column represents how much of the program's total time was spent in the function.
- The `cum.%` column shows the cumulative percentage of time spent in the functions that have been listed so far.
- The `cycles` column reports the number of machine cycles used by the function.
- The `instructions` column shows the number of instructions executed by a function.
- The `calls` column reports the number of calls to the function.
- The `procedure (dso:  file, line)` column lists the procedure, its DSO name, its file name, and the line number.

The following is output from an `ideal` experiment:

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Mon Feb  2 13:23:25 1998
   prof generic.ideal.m10966
                 generic (n32): Target program
                         ideal: Experiment name
                         it:cu: Marching orders
                 R4400 / R4000: CPU / FPU
                             1: Number of CPUs
                           175: Clock frequency (MHz.)
  Experiment notes--
          From file generic.ideal.m10966:
        Caliper point 0 at target begin, PID 10966
               /usr/demos/SpeedShop/linpack.demos/c/generic.pixie
        Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of ideal time data (ideal)--
                    2062563179: Total number of instructions executed
                    3929944273: Total computed cycles
                        22.457: Total computed execution time (secs.)
                         1.905: Average cycles / instruction
-------------------------------------------------------------------------
Function list, in descending order by exclusive ideal time
-------------------------------------------------------------------------
 [index]   excl.secs   excl.%     cum.%        cycles  instructions
calls  function  (dso: file, line)
```

```
     [1]      21.453    95.5%     95.5%    3754320037    1971220024
   1  anneal (generic: generic.c, 1573)
```

If the `-butterfly` flag is added to `prof`, a list of callers and callees of each function is provided:

```
-------------------------------------------------------------------------
Butterfly function list, in descending order by inclusive ideal time
-------------------------------------------------------------------------
         attrib.% attrib.time                         incl.time  caller [index]
 [index]   incl.%   incl.time    self%   self-time         procedure [index]
                              attrib.% attrib.time    incl.time  callee [index]
-------------------------------------------------------------------------
     [1]   100.0%      22.456     0.0%       0.000         __start [1]
                                100.0%      22.456       22.456  main [2]
                                  0.0%       0.000        0.000  __readenv_sigfpe [131]
                                  0.0%       0.000        0.000  __istart [132]
-------------------------------------------------------------------------
           100.0%      22.456                            22.456  __start [1]
     [2]   100.0%      22.456     0.0%       0.000         main [2]
                                100.0%      22.456       22.456  Scriptstring [3]
-------------------------------------------------------------------------
           100.0%      22.456                            22.456  main [2]
     [3]   100.0%      22.456     0.0%       0.000         Scriptstring [3]
                                 95.5%      21.454       21.454  usrtime [4]
                                  3.7%       0.829        0.829  libdso [6]
                                  0.8%       0.172        0.172  cvttrap [9]
                                  0.0%       0.001        0.001  iofile [11]
                                  0.0%       0.000        0.000  dirstat [23]
                                  0.0%       0.000        0.001  genLog [12]
                                  0.0%       0.000        0.000  linklist [26]
                                  0.0%       0.000        0.000  fpetraps [27]
                                  0.0%       0.000        0.000  fprintf [21]
                                  0.0%       0.000        0.000  sprintf [17]
                                  0.0%       0.000        0.000  strcmp [60]
-------------------------------------------------------------------------
            95.5%      21.454                            22.456  Scriptstring [3]
     [4]    95.5%      21.454     0.0%       0.000         usrtime [4]
                                 95.5%      21.454       21.454  anneal [5]
                                  0.0%       0.000        0.001  genLog [12]
                                  0.0%       0.000        0.000  fprintf [21]
------------------------------------------------------------------------
```

7.2.5 `fpe` Trace Reports

The `fpe` trace report shows information for each function. The function name is shown in the right column of the report. The remaining columns are described below.

- The `excl.FPEs` column shows how many floating point exceptions were found in the function.
- The `excl.%` column shows the percentage of the total number of floating-point exceptions that were found in the function.
- The `cum.%` column shows the percentage of floating-point exceptions in the program that have been encountered so far in the list.
- The `incl.FPEs` column shows how many floating-point exceptions were attributed to the function and all of the functions it called.
- The `incl.%` column provides information on the percentage of the program's total number of floating-point exceptions.

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Mon Feb  2 13:26:33 1998
   prof generic.fpe.m12213
                generic (n32): Target program
                          fpe: Experiment name
                       fpe:cu: Marching orders
                R4400 / R4000: CPU / FPU
                            1: Number of CPUs
                          175: Clock frequency (MHz.)
  Experiment notes--
          From file generic.fpe.m12213:
        Caliper point 0 at target begin, PID 12213
                       /usr/demos/SpeedShop/linpack.demos/c/generic
        Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of FPE callstack tracing data (fpe)--
                            4: Total FPEs
                            0: Samples with incomplete traceback
-------------------------------------------------------------------------
Function list, in descending order by exclusive FPEs
-------------------------------------------------------------------------
 [index]   excl.FPEs excl.%   cum.%   incl.FPEs incl.%  function  (dso:file)

     [1]          4 100.0%  100.0%          4 100.0%  fpetraps (generic: generic.c, 405)
```

7.3 Using `prof` Options

This section shows the output from calling `prof` with some of the options available for `prof`.

7.3.1 Using the `-dis` Option

For `pcsamp` and `ideal` experiments, the `-dis` option to `prof` can be used to obtain machine instruction information. `prof` provides the standard report and then appends the machine instruction information to the end of the report. The following example shows partial output from `prof` for a `pcsamp` experiment.

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Tue Feb  3 10:48:59 1998
   prof -dis generic.pcsamp.m14493
                 generic (n32): Target program
                        pcsamp: Experiment name
               pc,2,10000,0:cu: Marching orders
                 R4400 / R4000: CPU / FPU
                             1: Number of CPUs
                           175: Clock frequency (MHz.)
  Experiment notes--
   From file generic.pcsamp.m14493:
 Caliper point 0 at target begin, PID 14493
   /usr/demos/SpeedShop/c/generic
 Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of statistical PC sampling data (pcsamp)--
                          2707: Total samples
                        27.070: Accumulated time (secs.)
                          10.0: Time per sample (msecs.)
                             2: Sample bin width (bytes)
-------------------------------------------------------------------------
Function list, in descending order by time
-------------------------------------------------------------------------
 [index]      secs    %    cum.%   samples  function (dso: file, line)
     [1]    25.240  93.2%  93.2%      2524  anneal (generic: generic.c, 1573)
     [2]     1.090   4.0%  97.3%       109  slaveusrtime (dlslave.so: dlslave.c, 22)
     [3]     0.390   1.4%  98.7%        39  __read (libc.so.1: read.s, 20)
     [4]     0.230   0.8%  99.6%        23  cvttrap (generic: generic.c, 317)
     [5]     0.090   0.3%  99.9%         9  _xstat (libc.so.1: xstat.s, 12)
     [6]     0.010   0.0%  99.9%         1  __write (libc.so.1: write.s, 20)
     [7]     0.010   0.0% 100.0%         1  _ngetdents (libc.so.1: ngetdents.s, 16)
     [8]     0.010   0.0% 100.0%         1  _doprnt (libc.so.1: doprnt.c, 285)

            27.070 100.0% 100.0%      2707  TOTAL

-------------------------------------------------------------------------
Disassembly listing, annotated with PC sampling overflows
-------------------------------------------------------------------------
.
.
.
/usr/demos/SpeedShop/linpack.demos/c/generic.c
anneal: <0x10006830-0x10006b3c>    2524 total samples(93.24%)
  [1573] 0x10006830 0x27bdffd0 addiu sp,sp,-48  # 1
  [1573] 0x10006834 0xffbc0020 sd gp,32(sp)  # 2
  [1573] 0x10006838 0xffbf0018 sd ra,24(sp)  # 3
  [1573] 0x1000683c 0x3c030002 lui v1,0x2  # 4
  [1573] 0x10006840 0x246397e8 addiu v1,v1,-26648  # 5
  [1573] 0x10006844 0x0323e021 addu gp,t9,v1  # 6
  [1575] 0x10006848 0xd7808370 ldc1 $f0,-31888(gp)  # 7
  <2 cycle stall for following instruction>
  [1575] 0x1000684c 0xf7a00000 sdc1 $f0,0(sp)  # 10
  [1577] 0x10006850 0x24010001 li at,1  # 11
  [1577] 0x10006854 0x8f82816c lw v0,-32404(gp)  # 12
  <2 cycle stall for following instruction>
  [1577] 0x10006858 0xac410000 sw at,0(v0)  # 15
  [1578] 0x1000685c 0x8f998148 lw t9,-32440(gp)  # 16
  [1578] 0x10006860 0x0c00171b jal 0x10005c6c  # 17
  [1578] 0x10006864 0000000000 nop  # 18
  <2 cycle stall for following instruction>
  [1586] 0x10006868 0xafa00008 sw zero,8(sp)  # 21
  [1586] 0x1000686c 0x8fa40008 lw a0,8(sp)  # 22
  <2 cycle stall for following instruction>
  [1586] 0x10006870 0x28842710 slti a0,a0,10000  # 25
  [1586] 0x10006874 0x108000ac beq a0,zero,0x10006b28  # 26
  [1586] 0x10006878 0000000000 nop  # 27
  <2 cycle stall for following instruction>
  [1588] 0x1000687c 0x24070001 li a3,1  # 30
  [1588] 0x10006880 0xafa7000c sw a3,12(sp)  # 31
  [1588] 0x10006884 0x8f868164 lw a2,-32412(gp)  # 32
  <2 cycle stall for following instruction>
  [1588] 0x10006888 0x8cc60000 lw a2,0(a2)  # 35
  <2 cycle stall for following instruction>
  [1588] 0x1000688c 0x24c6ffff addiu a2,a2,-1  # 38
```

```
[1588] 0x10006890 0x8fa5000c lw a1,12(sp)  # 39
<2 cycle stall for following instruction>
[1588] 0x10006894 0x00a6282a slt a1,a1,a2  # 42
[1588] 0x10006898 0x10a0009c beq a1,zero,0x10006b0c  # 43
[1588] 0x1000689c 0000000000 nop  # 44
<2 cycle stall for following instruction>
[1589] 0x100068a0 0x240a0001 li t2,1  # 47
^------    1 samples(0.04%)------^
[1589] 0x100068a4 0xafaa0010 sw t2,16(sp)  # 48
^------    1 samples(0.04%)------^
[1589] 0x100068a8 0x8f898164 lw t1,-32412(gp)  # 49
<2 cycle stall for following instruction>
[1589] 0x100068ac 0x8d290000 lw t1,0(t1)  # 52
<2 cycle stall for following instruction>
[1589] 0x100068b0 0x2529ffff addiu t1,t1,-1  # 55
[1589] 0x100068b4 0x8fa80010 lw t0,16(sp)  # 56
<2 cycle stall for following instruction>
[1589] 0x100068b8 0x0109402a slt t0,t0,t1  # 59
[1589] 0x100068bc 0x11000089 beq t0,zero,0x10006ae4  # 60
[1589] 0x100068c0 0000000000 nop  # 61
<2 cycle stall for following instruction>
[1590] 0x100068c4 0x8faf000c lw t7,12(sp)  # 64
^------   27 samples(1.00%)------^
<2 cycle stall for following instruction>
[1590] 0x100068c8 0x25ef0001 addiu t7,t7,1  # 67
^------    7 samples(0.26%)------^
[1590] 0x100068cc 0x000f7080 sll t6,t7,2  # 68
^------   30 samples(1.11%)------^
[1590] 0x100068d0 0x01cf7021 addu t6,t6,t7  # 69
^------    8 samples(0.30%)------^
[1590] 0x100068d4 0x000e70c0 sll t6,t6,3  # 70
^------    5 samples(0.18%)------^
[1590] 0x100068d8 0x8faf0010 lw t7,16(sp)  # 71
^------    8 samples(0.30%)------^
<2 cycle stall for following instruction>
[1590] 0x100068dc 0x01cf7021 addu t6,t6,t7  # 74
^------    9 samples(0.33%)------^
[1590] 0x100068e0 0x000e70c0 sll t6,t6,3  # 75
^------   27 samples(1.00%)------^
[1590] 0x100068e4 0x8f8f817c lw t7,-32388(gp)  # 76
^------   14 samples(0.52%)------^
<2 cycle stall for following instruction>
[1590] 0x100068e8 0x01cf7021 addu t6,t6,t7  # 79
```

```
^------    9 samples(0.33%)------^
[1590] 0x100068ec 0x25ce0008 addiu t6,t6,8  # 80
^------   28 samples(1.03%)------^
[1590] 0x100068f0 0xd5c10000 ldc1 $f1,0(t6)  # 81
^------    7 samples(0.26%)------^
[1590] 0x100068f4 0x8fad000c lw t5,12(sp)  # 82
^------   10 samples(0.37%)------^
<2 cycle stall for following instruction>
[1590] 0x100068f8 0x25ad0001 addiu t5,t5,1  # 85
^------   21 samples(0.78%)------^
[1590] 0x100068fc 0x000d6080 sll t4,t5,2  # 86
^------   19 samples(0.70%)------^
[1590] 0x10006900 0x018d6021 addu t4,t4,t5  # 87
^------    9 samples(0.33%)------^
[1590] 0x10006904 0x000c60c0 sll t4,t4,3  # 88
^------   14 samples(0.52%)------^
[1590] 0x10006908 0x8fad0010 lw t5,16(sp)  # 89
^------    8 samples(0.30%)------^
<2 cycle stall for following instruction>
[1590] 0x1000690c 0x018d6021 addu t4,t4,t5  # 92
^------    8 samples(0.30%)------^
[1590] 0x10006910 0x000c60c0 sll t4,t4,3  # 93
^------   30 samples(1.11%)------^
[1590] 0x10006914 0x8f8d817c lw t5,-32388(gp)  # 94
^------   10 samples(0.37%)------^
<2 cycle stall for following instruction>
[1590] 0x10006918 0x018d6021 addu t4,t4,t5  # 97
^------    8 samples(0.30%)------^
[1590] 0x1000691c 0xd5820000 ldc1 $f2,0(t4)  # 98
^------   28 samples(1.03%)------^
[1590] 0x10006920 0x8fab000c lw t3,12(sp)  # 99
^------    9 samples(0.33%)------^
<2 cycle stall for following instruction>
[1590] 0x10006924 0x256b0001 addiu t3,t3,1  # 102
^------   11 samples(0.41%)------^
[1590] 0x10006928 0x000b5080 sll t2,t3,2  # 103
^------   25 samples(0.92%)------^
[1590] 0x1000692c 0x014b5021 addu t2,t2,t3  # 104
^------   11 samples(0.41%)------^
[1590] 0x10006930 0x000a50c0 sll t2,t2,3  # 105
^------    8 samples(0.30%)------^
[1590] 0x10006934 0x8fab0010 lw t3,16(sp)  # 106
^------   11 samples(0.41%)------^
```

```
     <2 cycle stall for following instruction>
     [1590] 0x10006938 0x014b5021 addu t2,t2,t3  # 109
     ^------    7 samples(0.26%)------^
     [1590] 0x1000693c 0x000a50c0 sll t2,t2,3  # 110
     ^------   26 samples(0.96%)------^
     [1590] 0x10006940 0x8f8b817c lw t3,-32388(gp)  # 111
     ^------   13 samples(0.48%)------^
     <2 cycle stall for following instruction>
     [1590] 0x10006944 0x014b5021 addu t2,t2,t3  # 114
     ^------    9 samples(0.33%)------^
     [1590] 0x10006948 0x254afff8 addiu t2,t2,-8  # 115
     ^------   26 samples(0.96%)------^
     [1590] 0x1000694c 0xd5430000 ldc1 $f3,0(t2)  # 116
     ^------   11 samples(0.41%)------^
     [1590] 0x10006950 0x8fa9000c lw t1,12(sp)  # 117
     ^------   10 samples(0.37%)------^
     <2 cycle stall for following instruction>
     [1590] 0x10006954 0x00094080 sll t0,t1,2  # 120
     ^------   11 samples(0.41%)------^
   .
   .
   .
```

The listing shows statistics about the procedure `anneal()` in the file `generic.c` and lists the beginning and ending addresses of `anneal()`: <0x100065b8-0x100068c4>. The five columns display the following information:

Column	Displays
1	Line number of the instruction: [1573].
2	Beginning address of the instruction: 0x10006830.
3	Instruction in hexadecimal: 0x27bdffd0.
4	Assembler form (mnemonic) of the instruction: addiu sp,sp,-48.
5	Cycle in which the instruction executed: # 1.

Other information includes:

- The number of times the immediately preceding branch was executed and taken (`ideal` only).
- The total number of cycles in a basic block and the percentage of the total cycles for that basic block, the number of times the branch terminating that

basic block was executed, and the number of cycles for one execution of that basic block (`ideal` only).

- The total number of samples at an instruction (`pcsamp` only).
- Any cycle stalls, that is, cycles that were wasted.

7.3.2 Using the -S Option

For `ideal` experiments, the `-S` option to `prof` can be used to obtain source line information. `prof` provides the standard report and then appends the source line information to the end of the report.

This example shows output from calling `prof` for an `ideal` experiment:

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Tue Feb  3 13:49:07 1998
   prof -S generic.ideal.m15682
                 generic (n32): Target program
                         ideal: Experiment name
                         it:cu: Marching orders
                 R4400 / R4000: CPU / FPU
                             1: Number of CPUs
                           175: Clock frequency (MHz.)
  Experiment notes--
   From file generic.ideal.m15682:
 Caliper point 0 at target begin, PID 15682
   /usr/demos/SpeedShop/c/generic.pixie
 Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of ideal time data (ideal)--
                    2062563562: Total number of instructions executed
                    3929944935: Total computed cycles
                        22.457: Total computed execution time (secs.)
                         1.905: Average cycles / instruction
    .
    .
    .
-----------------------
disassembly listing
-----------------------

*DF*_generic.MIPS.stubs_1
*DF*_dlslave.text_2@0x5ffe40e0-0x5ffe4ec8: <0x10001ad8-0x10001ec4>
```

```
  7 total cycles(0.00%) invoked 0 times, average ? cycles/invocation
[1] 0x10001ad8 0x0006000d break 0x6  # 1
^---      0 total cycles(0.00%) executed     0 times, average  1 cycles.---^
[1] 0x10001adc 0x8f998010 lw t9,-32752(gp)  # 1
[1] 0x10001ae0 0x03e07825 move t7,ra  # 2
<1 cycle stall for following instruction>
[1] 0x10001ae4 0x0320f809 jalr ra,t9  # 4
[1] 0x10001ae8 0x3418003a ori t8,zero,0x3a  # 5
<2 cycle stall for following instruction>
^---      7 total cycles(0.00%) executed     1 times, average  7 cycles.---^
[1] 0x10001aec 0000000000 nop  # 1
[1] 0x10001af0 0x8f998010 lw t9,-32752(gp)  # 2
[1] 0x10001af4 0x03e07825 move t7,ra  # 3
<1 cycle stall for following instruction>
[1] 0x10001af8 0x0320f809 jalr ra,t9  # 5
[1] 0x10001afc 0x3418003b ori t8,zero,0x3b  # 6
<2 cycle stall for following instruction>
^---      0 total cycles(0.00%) executed     0 times, average  8 cycles.---^
[1] 0x10001b00 0000000000 nop  # 1
[1] 0x10001b04 0x8f998010 lw t9,-32752(gp)  # 2
[1] 0x10001b08 0x03e07825 move t7,ra  # 3
<1 cycle stall for following instruction>
[1] 0x10001b0c 0x0320f809 jalr ra,t9  # 5
[1] 0x10001b10 0x3418003c ori t8,zero,0x3c  # 6
<2 cycle stall for following instruction>
^---      0 total cycles(0.00%) executed     0 times, average  8 cycles.---^
[1] 0x10001b14 0000000000 nop  # 1
[1] 0x10001b18 0x8f998010 lw t9,-32752(gp)  # 2
[1] 0x10001b1c 0x03e07825 move t7,ra  # 3
<1 cycle stall for following instruction>
[1] 0x10001b20 0x0320f809 jalr ra,t9  # 5
[1] 0x10001b24 0x3418003d ori t8,zero,0x3d  # 6
<2 cycle stall for following instruction>
^---      0 total cycles(0.00%) executed     0 times, average  8 cycles.---^
[1] 0x10001b28 0000000000 nop  # 1
[1] 0x10001b2c 0x8f998010 lw t9,-32752(gp)  # 2
[1] 0x10001b30 0x03e07825 move t7,ra  # 3
<1 cycle stall for following instruction>
[1] 0x10001b34 0x0320f809 jalr ra,t9  # 5
[1] 0x10001b38 0x3418003e ori t8,zero,0x3e  # 6
<2 cycle stall for following instruction>
^---      0 total cycles(0.00%) executed     0 times, average  8 cycles.---^
[1] 0x10001b3c 0000000000 nop  # 1
```

```
[1] 0x10001b40 0x8f998010 lw t9,-32752(gp)  # 2
[1] 0x10001b44 0x03e07825 move t7,ra  # 3
<1 cycle stall for following instruction>
[1] 0x10001b48 0x0320f809 jalr ra,t9  # 5
[1] 0x10001b4c 0x3418003f ori t8,zero,0x3f  # 6
<2 cycle stall for following instruction>
^---      0 total cycles(0.00%) executed     0 times, average  8 cycles.---^
[1] 0x10001b50 0000000000 nop  # 1
[1] 0x10001b54 0x8f998010 lw t9,-32752(gp)  # 2
[1] 0x10001b58 0x03e07825 move t7,ra  # 3
<1 cycle stall for following instruction>
[1] 0x10001b5c 0x0320f809 jalr ra,t9  # 5
[1] 0x10001b60 0x34180040 ori t8,zero,0x40  # 6
<2 cycle stall for following instruction>
^---      0 total cycles(0.00%) executed     0 times, average  8 cycles.---^
[1] 0x10001b64 0000000000 nop  # 1
[1] 0x10001b68 0x8f998010 lw t9,-32752(gp)  # 2
[1] 0x10001b6c 0x03e07825 move t7,ra  # 3
<1 cycle stall for following instruction>
[1] 0x10001b70 0x0320f809 jalr ra,t9  # 5
[1] 0x10001b74 0x34180041 ori t8,zero,0x41  # 6
<2 cycle stall for following instruction>
^---      0 total cycles(0.00%) executed     0 times, average  8 cycles.---^
[1] 0x10001b78 0000000000 nop  # 1
[1] 0x10001b7c 0x8f998010 lw t9,-32752(gp)  # 2
[1] 0x10001b80 0x03e07825 move t7,ra  # 3
<1 cycle stall for following instruction>
[1] 0x10001b84 0x0320f809 jalr ra,t9  # 5
[1] 0x10001b88 0x34180042 ori t8,zero,0x42  # 6
<2 cycle stall for following instruction>
.
.
.
```

7.3.3 Using the `-calipers` Option

When you run `prof` on the output of an experiment in which you have recorded caliper points, you can use the `-calipers` option to specify the area of the program for which you want to generate a performance report. For example, if you record just one caliper point in the middle of your program, `prof` can provide a report from the beginning of the program up to the first caliper point using the following command:

```
prof -calipers 0 1
```

The `prof` command can also provide a report from the caliper point to the end of the program using the following command:

```
prof -calipers 1 2
```

If you record two caliper points (0, 1, 2, 3), `prof` can generate a report from the second to the third caliper point:

```
prof -calipers 1 2
```

7.3.4 Using the `-butterfly` Option

For `ideal`, `usertime`, and `fpe` experiments, the `-butterfly` option to `prof` can be used to obtain inclusive metric information. `prof` provides the standard report and then appends the inclusive function counts information to the end of the report. The following example is partial output from `prof`, showing just the inclusive function counts report.

With inclusive cycle counting, `prof` prints a list of functions at the end, which are called but not defined. It also includes functions from `libss`; they are instrumented, but their data is normally excluded.

`prof` does not list the cycles of a procedure in the inclusive listing for the following reasons:

- `init`, `fini`, and `MIPS.stubs` sections are not part of any procedure.
- Calls to procedures that do not use a “jump and link” are not recognized as procedure calls. (This is not true for `ideal` experiments.)
- When global procedures with the same name are executed in different DSOs, only one of them is listed.

These exceptions are listed at the end of the report.

This example shows output from calling `prof` for a `usertime` experiment:

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Thu Feb 12 13:52:09 1998
   prof -gprof generic.usertime.m10981
                 generic (n32): Target program
                      usertime: Experiment name
                         ut:cu: Marching orders
                 R4400 / R4000: CPU / FPU
```

```
                              1: Number of CPUs
                            175: Clock frequency (MHz.)
   Experiment notes--
           From file generic.usertime.m10981:
         Caliper point 0 at target begin, PID 10981
                         /usr/demos/SpeedShop/linpack.demos/c/generic
         Caliper point 1 at exit(0)
-------------------------------------------------------------------------
Summary of statistical callstack sampling data (usertime)--
                            809: Total Samples
                              0: Samples with incomplete traceback
                         24.270: Accumulated Time (secs.)
                           30.0: Sample interval (msecs.)
-------------------------------------------------------------------------
Function list, in descending order by exclusive time
-------------------------------------------------------------------------
 [index]  excl.secs excl.%   cum.%  incl.secs incl.%    samples  procedure
(dso: file, line)

     [4]     22.770  93.8%   93.8%     22.770  93.8%        759  anneal
(generic: generic.c, 1573)
     [6]      1.020   4.2%   98.0%      1.020   4.2%         34  slaveusrtime
(dlslave.so: dlslave.c, 22)
     [9]      0.210   0.9%   98.9%      0.210   0.9%          7  cvttrap
(generic: generic.c, 317)
    [12]      0.120   0.5%   99.4%      0.120   0.5%          4  _pm_create_special
(libc.so.1: pm.c, 191)
    [14]      0.090   0.4%   99.8%      0.090   0.4%          3  _migr_policy_args_init
(libc.so.1: pm.c, 398)
    [10]      0.030   0.1%   99.9%      0.180   0.7%          6  iofile
(generic: generic.c, 464)
    [11]      0.030   0.1%  100.0%      0.150   0.6%          5  _doscan_f
(libc.so.1: inline_doscan.c, 615)
     [1]      0.000   0.0%  100.0%     24.270 100.0%        809  __start
(generic: crt1text.s, 101)
     [2]      0.000   0.0%  100.0%     24.270 100.0%        809  main
(generic: generic.c, 101)
     [3]      0.000   0.0%  100.0%     24.270 100.0%        809  Scriptstring
(generic: generic.c, 184)
     [5]      0.000   0.0%  100.0%     22.770  93.8%        759  usrtime
(generic: generic.c, 1377)
    [15]      0.000   0.0%  100.0%      0.090   0.4%          3  dirstat
(generic: generic.c, 348)
```

```
    [16]      0.000   0.0%  100.0%      0.090   0.4%          3  _pread
(libc.so.1: preadSCI.c, 33)
    [13]      0.000   0.0%  100.0%      0.120   0.5%          4  _fullocale
(libc.so.1: _locale.c, 77)
     [7]      0.000   0.0%  100.0%      1.020   4.2%         34  libdso
(generic: generic.c, 619)
     [8]      0.000   0.0%  100.0%      1.020   4.2%         34  dlslave_routine
(dlslave.so: dlslave.c, 7)

-------------------------------------------------------------------------
Butterfly function list, in descending order by inclusive time
-------------------------------------------------------------------------
         attrib.% attrib.time                         incl.time  caller
(callsite) [index]
 [index]   incl.%   incl.time    self%   self-time       procedure [index]
                              attrib.% attrib.time    incl.time  callee
(callsite) [index]
-------------------------------------------------------------------------
     [1]   100.0%      24.270     0.0%       0.000         __start [1]
                                100.0%      24.270       24.270  main
(@0x10001fac; generic: crt1text.s, 177) [2]
-------------------------------------------------------------------------
           100.0%      24.270                            24.270  __start
(@0x10001fac; generic: crt1text.s, 177) [1]
     [2]   100.0%      24.270     0.0%       0.000         main [2]
                                100.0%      24.270       24.270  Scriptstring
(@0x10002040; generic: generic.c, 111) [3]
-------------------------------------------------------------------------
           100.0%      24.270                            24.270  main
(@0x10002040; generic: generic.c, 111) [2]
     [3]   100.0%      24.270     0.0%       0.000         Scriptstring
[3]
                                 93.8%      22.770       22.770  usrtime
(@0x10002460; generic: generic.c, 214) [5]
                                  4.2%       1.020        1.020  libdso
(@0x10002460; generic: generic.c, 214) [7]
                                  0.9%       0.210        0.210  cvttrap
(@0x10002460; generic: generic.c, 214) [9]
                                  0.7%       0.180        0.180  iofile
(@0x10002460; generic: generic.c, 214) [10]
                                  0.4%       0.090        0.090  dirstat
(@0x10002460; generic: generic.c, 214) [15]
-------------------------------------------------------------------------
```

```
            93.8%      22.770                            22.770  usrtime
(@0x10005c30; generic: generic.c, 1393) [5]
     [4]    93.8%      22.770    93.8%      22.770         anneal [4]
---------------------------------------------------------------------
            93.8%      22.770                            24.270  Scriptstring
(@0x10002460; generic: generic.c, 214) [3]
     [5]    93.8%      22.770     0.0%       0.000         usrtime [5]
                                 93.8%      22.770       22.770  anneal
(@0x10005c30; generic: generic.c, 1393) [4]
---------------------------------------------------------------------
             4.2%       1.020                             1.020  dlslave_routine
(@0x5ffe0690; dlslave.so: dlslave.c, 9) [8]
     [6]     4.2%       1.020     4.2%       1.020         slaveusrtime
[6]
---------------------------------------------------------------------
             4.2%       1.020                            24.270  Scriptstring
(@0x10002460; generic: generic.c, 214) [3]
     [7]     4.2%       1.020     0.0%       0.000         libdso [7]
                                  4.2%       1.020        1.020  dlslave_routine
(@0x100032a0; generic: generic.c, 650) [8]
---------------------------------------------------------------------
             4.2%       1.020                             1.020  libdso
(@0x100032a0; generic: generic.c, 650) [7]
     [8]     4.2%       1.020     0.0%       0.000         dlslave_routine [8]
                                  4.2%       1.020        1.020  slaveusrtime
(@0x5ffe0690; dlslave.so: dlslave.c, 9) [6]
---------------------------------------------------------------------
             0.9%       0.210                            24.270  Scriptstring
(@0x10002460; generic: generic.c, 214) [3]
     [9]     0.9%       0.210     0.9%       0.210         cvttrap [9]
---------------------------------------------------------------------
             0.7%       0.180                            24.270  Scriptstring
(@0x10002460; generic: generic.c, 214) [3]
    [10]     0.7%       0.180     0.1%       0.030         iofile [10]
                                  0.6%       0.150        0.150  _doscan_f
(@0x10002d48; generic: generic.c, 483) [11]
---------------------------------------------------------------------
             0.6%       0.150                             0.180  iofile
(@0x10002d48; generic: generic.c, 483) [10]
    [11]     0.6%       0.150     0.1%       0.030         _doscan_f [11]
                                  0.5%       0.120        0.120  _fullocale
(@0x0fadebe4; libc.so.1: inline_doscan.c, 808) [13]
---------------------------------------------------------------------
```

```
             0.5%      0.120                            0.120  _fullocale
(@0x0fb0b1b8; libc.so.1: _locale.c, 84) [13]
    [12]     0.5%      0.120     0.5%      0.120         _pm_create_special [12]
-----------------------------------------------------------------------
             0.5%      0.120                            0.150  _doscan_f
(@0x0fadebe4; libc.so.1: inline_doscan.c, 808) [11]
    [13]     0.5%      0.120     0.0%      0.000         _fullocale [13]
                                 0.5%      0.120        0.120  _pm_create_special
(@0x0fb0b1b8; libc.so.1: _locale.c, 84) [12]
-----------------------------------------------------------------------
             0.4%      0.090                            0.090  _pread
(@0x0fb05928; libc.so.1: preadSCI.c, 33) [16]
    [14]     0.4%      0.090     0.4%      0.090         _migr_policy_args_init [14]
-----------------------------------------------------------------------
             0.4%      0.090                           24.270  Scriptstring
(@0x10002460; generic: generic.c, 214) [3]
    [15]     0.4%      0.090     0.0%      0.000         dirstat [15]
                                 0.4%      0.090        0.090  _pread
(@0x10002a5c; generic: generic.c, 381) [16]
-----------------------------------------------------------------------
             0.4%      0.090                            0.090  dirstat
(@0x10002a5c; generic: generic.c, 381) [15]
    [16]     0.4%      0.090     0.0%      0.000         _pread [16]
                                 0.4%      0.090        0.090  _migr_policy_args_init
(@0x0fb05928; libc.so.1: preadSCI.c, 33) [14]
-----------------------------------------------------------------------
```

7.4 Generating Reports for Different Machine Types

If you need to generate a report for a machine model that is different from the one on which the experiment was performed, you can use several of the `prof` options to specify a machine model.

For example, if you record an `ideal` experiment on an R4000 processor with a clock frequency of 100 megahertz, but you want to generate a report for an R10000 processor, the `prof` command would be the following:

```
prof -r10000 -clock 196 generic.ideal.m4561
```

7.5 Generating Reports for Multiprocessed Executables

You can gather data from executables that use the `sproc`(2) and `sprocsp`(2) system calls, such as those executables generated by POWER Fortran and POWER C. Prepare and run the job using the same method as for uniprocessed executables. For multiprocessed executables, each thread of execution writes its own separate data file. View these data files with `prof`.

The only difference between multiprocessed and regular executables is how the data files are named. The data files are named *prog_name.experiment_type.id*.

The experiment ID, `id`, consists of one or two letters (designating the process type) and the process ID number. See Table 4 for the letter codes and their meanings. This naming convention avoids the potential conflict of multiple threads attempting to write simultaneously to the same file.

7.6 Generating Feedback Files

If you run an `ideal` experiment, run `prof` with the `-feedback` option to generate a feedback file that can be used to arrange procedures more efficiently on the next compilation. You can rearrange procedures using the `-fb` on compiler command lines.

To reorder code regions for the `cord`(1) command, use the `-cordfb` or `-cordfball` option to `prof`.

For more information, see your compiler man page, the `cord`(1) man page, or the *MIPSpro Compiling and Performance Tuning Guide*.

Using SpeedShop in Expert Mode: pixie [8]

This chapter provides information on how to run pixie and prof without invoking ssrun. By calling pixie directly, you can generate the following performance data:

- An exact count of the number of times each basic block in your program is executed. A basic block is a sequence of instructions that is entered only at the beginning of the sequence and is exited only at the end. No jumps into or out of a basic block are permitted.
- Counts for callers of a routine as well as counts for callees. prof can provide inclusive basic block counting by propagating regular counts to callers of a routine.

For more information on basic block counting and inclusive basic block counting, see Section 7.2.4, page 102.

This chapter contains the following sections:

- Using pixie, see Section 8.1, page 121.
- Obtaining Basic Block Counts, see Section 8.2, page 124.
- Obtaining Inclusive Basic Block Counts, see Section 8.3, page 130.

8.1 Using pixie

Your can use pixie to measure the frequency of code execution. pixie reads an executable program, partitions it into basic blocks, and writes (instruments) an equivalent program containing additional code that counts the execution of each basic block.

Note that the execution time of an instrumented program is two to five times longer than that of an uninstrumented one. This timing change may alter the behavior of a program that deals with a graphical user interface (GUI) or depends on events that are based on an external clock, such as SIGALRM.

8.1.1 `pixie` Syntax

The syntax for `pixie` is as follows:

`pixie` *prog_name* [*options*]

prog_name	Name of the input program.
options	Zero or more of the keywords listed in Table 15.

8.1.2 `pixie` Options

Table 15 lists `pixie` options. For a complete list of options, view the `pixie`(1) man page.

Table 15. Options for `pixie`

Name	Result
`-addlibs` *lib1*`.so:`*...libn*`.so`	Adds *lib1*`.so:` ... *libn*`.so` to the library list of the executable. No libraries are added by default.
`-`[no]`autopixie`	Permits or prevents a recursive instrumenting of all dynamic shared libraries used by the input file during run time. `pixie` keeps the timestamp and checksum from the original executable. Thus, before instrumenting a shared library, `pixie` checks any files that it has already processed that match the `lib` it is to instrument. If the fields match, they are not instrumented. `pixie` cannot detect shared libraries opened with `dlopen()`, and hence it does not instrument them. All used DSOs need to be instrumented for the pixified executable to work. The default behavior with shared libraries is `-noautopixie`. The default behavior with an executable is `-autopixie`.
`-copy`	Produces a copy of the target with function list (map) and arc list (graph) sections but does not instrument the target.
`-counts_file` *file*	Specifies the name to be used for the output `.Counts` file. By default, `.Counts` is appended to the original program name.
`-directory` *dir_name*	Writes output files to *dir_name*. Files are written to the current directory by default.
`-dso`	Treats the executable as an o32 DSO. Performs a search of standard o32 library directories. A `.pix32` extension is used.

Name	Result
`-dso32`	Treats the executable as an n32 DSO. Performs a search of standard n32 library directories. A `.pixn32` extension is used.
`-dso64`	Treats the executable as an n64 DSO. Performs a search of standard n64 library directories. A `.pix64` extension is used.
`-fcncounts`	Produces an instrumented executable that counts function calls and arc calls but not basic-block or branch counts.
`-[no]longbranch`	During instrumentation, some transformations can push a branch offset beyond its legal range and `pixie` generates warnings about branch offsets being out of range. This option causes `pixie` to transform these instructions into jumps. The default is `-nolongbranch`.
`-[no]pids`	The `-pids` option appends the process ID number to the end of the *file*`.Counts`. This is useful if you want to run the program instrumented with `pixie` through a variety of tests before generating the statistics with `prof`(1). If specified, the `-nopids` option is overridden by any process that issues a `fork`(2) or `sproc`(2) system call. The default is `-nopids`.
`-pixie_file` *name*	Specifies the name of the executable processed by `pixie`.
`-[no]verbose`	Prints or suppresses messages summarizing the binary-to-binary translation process. The default is `-noverbose`.
`-suffix` *.suffix*	Appends *.suffix* to the executable and DSOs processed by `pixie`. . The default suffix is `.pixie`.

8.1.3 pixie Output

The `pixie` command generates a set of files with a `.pixie` extension. These files are essentially copies of your original executable and any DSOs you specified in the call to `pixie` with code inserted to enable the collection of performance data when the `.pixie` version of your program is run.

If you use the `-verbose` flag with `pixie`, it reports the size of the old and new code. The new code size is the size of the code `pixie` will actually execute. It does not count read-only data (including a copy of the original text and another data block the same size as the original text) put into the text section. Calling `size` on the `.pixie` file reports a much larger text size than `pixie -verbose`, because `size` also counts everything in the text segment.

When you run the `.pixie` version of your program, one or more `.Counts` files are generated. The name of an output `.Counts` file is that of the original program with any leading directory names removed and `.Counts` appended. If the program executes calls to `sproc()`, `sprocsp()`, or `fork()`, multiple `.Counts` files are generated: one for each process in the shared group. In this case, each file will have the process ID appended to its name.

8.2 Obtaining Basic Block Counts

Use this procedure to obtain basic block counts. Also refer to Figure 3, page 126, which illustrates how basic block counting works. Though the preferred method of getting basic block information is using `ssrun -ideal`, you can use pixie directly.

1. Compile and link your program. The following example uses the input file `myprog.c`:

   ```
   % cc -o myprog myprog.c
   ```

   The `cc` compiler compiles `myprog.c` into an executable called `myprog`.

2. Run `pixie` to generate the equivalent program containing basic-block-counting code.

   ```
   % pixie myprog
   ```

   The `pixie` command takes `myprog` and writes an equivalent program, `myprog.pixie`, containing additional code that counts the execution of each basic block. `pixie` also writes an equivalent program for each shared object used by the program (in the form: `libname.so.pix*`), containing additional code that counts the execution of each basic block. For example, if `myprog` uses `libc.so.1`, `pixie` generates `libc.so.1.pix*`. (The value of * depends on the ABI.)

3. Execute the files generated by `pixie` (`myprog.pixie`) in the same way you executed the original program:

   ```
   % myprog.pixie
   ```

   This program generates a list of basic block counts in files named `myprog.Counts`. If the program executes `fork` or `sproc`, a process ID is appended to the end of the file name (for example, `myprog.Counts.34521`) for each process.

Note: Your program may not run as you expect when you invoke it with a `.pixie` extension. Some programs, `uncompress` and `vi`, for example, treat their arguments differently when the name of the program changes. You may need to rename the `.pixie` version of your program back to its original name.

A valid `.Counts` file is generated under most normal and abnormal program terminations. If signal handlers are installed, you must use `exit`(2) to terminate, since the run-time fatal signal handlers will be overwritten.

4. Run the profile formatting program `prof`(1), specifying the name of the `.Counts` file for the program, as shown in the following example:

   ```
   % prof myprog.Counts
   ```

   `prof` extracts information from `myprog.Counts` and prints it in an easily readable format. If multiple `.Counts` files exist, you can use the wildcard character (*) to specify all the files.

   ```
   % prof myprog.Counts*
   ```

You can run the program several times, altering the input data, to create multiple profile data files.

The time computation assumes a best case execution; actual execution takes longer. This is because the time includes predicted stalls within a basic block, but not actual stalls that may occur entering a basic block. It also assumes that all instructions and data are in cache, that is, it excludes the delays due to cache misses, memory fetches and stores, translation lookaside buffer and page faults, and other operating system overhead.

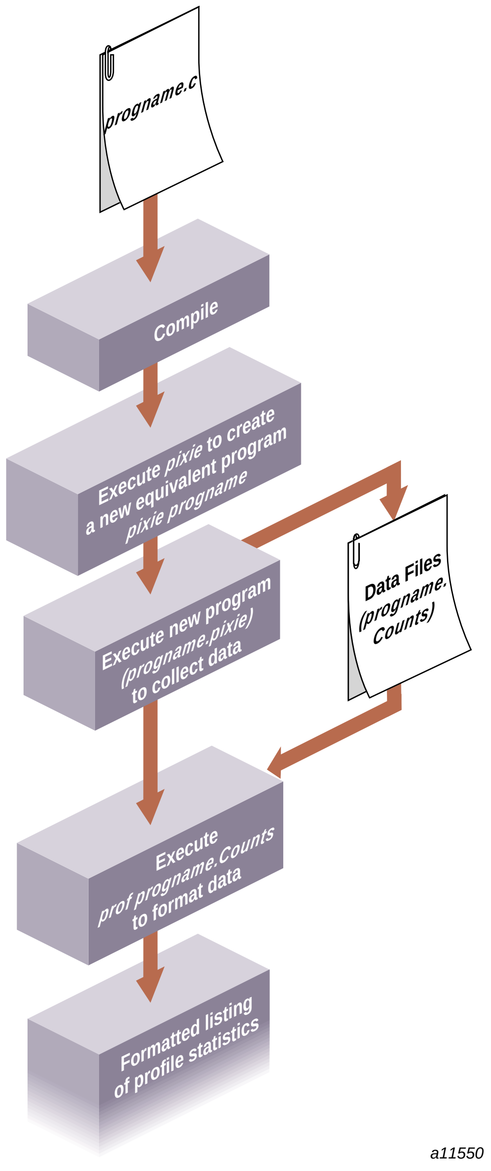

progname.c
Compile
Execute pixie to create a new equivalent program
pixie progname
Execute new program (progname.pixie) to collect data
Data Files (progname.Counts)
Execute prof progname.Counts to format data
Formatted listing of profile statistics
a11550

Figure 3. How Basic Block Counting Works

8.2.1 Examples of Basic Block Counting

The examples in this section illustrate how to use `prof` to obtain basic block counting information from a C program, `generic`.

8.2.1.1 Example Using `prof` with No Options

The partial listing that follows illustrates the report generated for basic block counts `generic`. The `prof` command first provides a standard report of basic block counts, then provides a report reflecting any options provided to `prof`.

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Tue Feb  3 14:25:43 1998
   prof generic generic.Counts
                 generic (n32): Target program
                  pixie-counts: Experiment name
                  pixie-counts: Marching orders
                 R4400 / R4000: CPU / FPU
                             1: Number of CPUs
                           175: Clock frequency (MHz.)
-------------------------------------------------------------------------
Summary of ideal time data (pixie-counts)--
                    2062563311: Total number of instructions executed
                    3929944454: Total computed cycles
                        22.457: Total computed execution time (secs.)
                         1.905: Average cycles / instruction
-------------------------------------------------------------------------
Function list, in descending order by exclusive ideal time
-------------------------------------------------------------------------
 [index]   excl.secs   excl.%     cum.%       cycles  instructions    calls
function  (dso: file, line)
     [1]      21.453    95.5%     95.5%   3754320037    1971220024        1
anneal (generic: generic.c, 1573)
     [2]       0.829     3.7%     99.2%    145001152      75000732        1
slaveusrtime (dlslave.so: dlslave.c, 22)
     [3]       0.171     0.8%    100.0%     30000081      16000054        1
cvttrap (generic: generic.c, 317)
     [4]       0.001     0.0%    100.0%       101504         58124        1
init2da (generic: generic.c, 1430)
     [5]       0.001     0.0%    100.0%        91200         38400     1600
_drand48 (libc.so.1: drand48.c, 116)
     [6]       0.001     0.0%    100.0%        89072         55011      447
fread (libc.so.1: fread.c, 34)
```

```
     [7]       0.000     0.0%    100.0%         74859         47364       53
   _doprnt (libc.so.1: doprnt.c, 285)
     [8]       0.000     0.0%    100.0%         64035         29479      628
   __sinf (libm.so: fsin.c, 93)
     [9]       0.000     0.0%    100.0%         32355          7182        9
   offtime (libc.so.1: time_comm.c, 180)
    [10]       0.000     0.0%    100.0%         17112         11916      305
   _readdir (libc.so.1: readdir.c, 135)
```

- The `excl.secs` column shows the number of seconds spent in each procedure. For example, 21.453 seconds were spent in the `anneal` function. The time represents an idealized computation based on modeling the machine. It ignores potential floating-point interlocks and memory latency time (cache misses and memory bus contention).
- The `excl.%` column lists the percentage of the program's total time spent in each function. The `anneal` function takes 95.5% of the total time.
- The `cum%` column shows the cumulative percentage of calls. For example, 99.2% of the total program time was spent in the top two functions in the listing: `anneal` and `slaveusrtime`.
- The `cycles` column reports the number and percentage of machine cycles used for the procedure. For example, 3,754,320,037 cycles were spent in the `anneal` function.
- The `instructions` column shows the number of instructions executed by a function. For example, the `anneal` function executed 1,971,220,024 instructions.
- The `calls` column reports the number of calls to each function. For example, there was just one call to the `anneal` function.
- The `procedure (dso:  file, line)` columns list the function name, its DSO name, its file name, and its line number. For example, the first line reports statistics for the function `anneal`, in the file `generic`, the DSO `generic.c`, and the line number 1573.

8.2.1.2 Example Using `prof -heavy`

The partial listing that follows shows the source code lines responsible for the largest portion of execution time produced using the `-heavy` option.

```
% prof -heavy generic generic.Counts
```

The following partial listing shows basic block counts sorted in descending order of cycles used:

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Tue Feb  3 15:02:11 1998
   prof -heavy generic generic.Counts
                 generic (n32): Target program
                  pixie-counts: Experiment name
                  pixie-counts: Marching orders
                 R4400 / R4000: CPU / FPU
                             1: Number of CPUs
                           175: Clock frequency (MHz.)
-------------------------------------------------------------------------
Summary of ideal time data (pixie-counts)--
                    2062563311: Total number of instructions executed
                    3929944454: Total computed cycles
                        22.457: Total computed execution time (secs.)
                         1.905: Average cycles / instruction
    .
    .
    .
-------------------------------------------------------------------------
Line list, in descending order by time
-------------------------------------------------------------------------
excl.secs   %    cum.%       cycles   invocations  function (dso: file, line)

19.800   88.2%   88.2%    3464962830      14440000  anneal (generic: generic.c, 1590)
 1.608    7.2%   95.3%     281457170      14440000  anneal (generic: generic.c, 1589)
 0.497    2.2%   97.5%      87000454       5000000  slaveusrtime (dlslave.so: dlslave.c, 29)
 0.331    1.5%   99.0%      57999996       5000000  slaveusrtime (dlslave.so: dlslave.c, 30)
 0.048    0.2%   99.2%       8437511        500000  cvttrap (generic: generic.c, 327)
 0.048    0.2%   99.4%       8437511        500000  cvttrap (generic: generic.c, 334)
 0.044    0.2%   99.6%       7770000        380000  anneal (generic: generic.c, 1588)
 0.037    0.2%   99.8%       6562500        500000  cvttrap (generic: generic.c, 328)
 0.037    0.2%  100.0%       6562500        500000  cvttrap (generic: generic.c, 335)
 0.001    0.0%  100.0%        130009         10000  anneal (generic: generic.c, 1586)
 0.000    0.0%  100.0%         43919          1600  init2da (generic: generic.c, 1443)
```

8.2.1.3 Example Using `prof -quit`

You can limit the output of `prof` to collect information on only the most time-consuming parts of the program by specifying the `-quit` option. You can instruct `prof` to quit after a particular number of lines of output, after listing the elements consuming more than a certain percentage of the total, or after the portion of each listing whose cumulative use is a certain amount.

Consider the following sample listing, which displays only the first four entries:

```
% prof -quit 4 generic generic.Counts
```

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Wed Feb  4 10:18:58 1998
   prof -quit 4 generic generic.Counts
                 generic (n32): Target program
                  pixie-counts: Experiment name
                  pixie-counts: Marching orders
                 R4400 / R4000: CPU / FPU
                             1: Number of CPUs
                           175: Clock frequency (MHz.)
-------------------------------------------------------------------------
Summary of ideal time data (pixie-counts)--
                    2062563311: Total number of instructions executed
                    3929944454: Total computed cycles
                        22.457: Total computed execution time (secs.)
                         1.905: Average cycles / instruction
-------------------------------------------------------------------------
Function list, in descending order by exclusive ideal time
-------------------------------------------------------------------------
 [index]   excl.secs   excl.%     cum.%        cycles instructions    calls  function
(dso: file, line)
     [1]      21.453    95.5%     95.5%    3754320037    1971220024        1  anneal
(generic: generic.c, 1573)
     [2]       0.829     3.7%     99.2%     145001152      75000732        1  slaveusrtime
(dlslave.so: dlslave.c, 22)
     [3]       0.171     0.8%    100.0%      30000081      16000054        1  cvttrap
(generic: generic.c, 317)
     [4]       0.001     0.0%    100.0%        101504         58124        1  init2da
(generic: generic.c, 1430)
```

8.3 Obtaining Inclusive Basic Block Counts

Inclusive basic block counting counts basic blocks and generates a call graph. By propagating regular counts to callers of a routine, `prof` provides inclusive basic block counting. For more information on inclusive basic block counting, see Section 4.4.3, page 53.

To see inclusive data, run the profile formatting program `prof`, specifying the name of the original program, the `-butterfly` flag, and the `.Counts` file for the program, as follows:

```
% prof -butterfly myprog myprog.Counts
```

In the following example, `prof` extracts information from `myprog.Counts` and prints it in an easily readable format. If multiple `.Counts` files exist, you can use the wildcard character (*) to specify all of the files.

```
% prof -butterfly myprog myprog.Counts*
```

8.3.1 Example of prof -butterfly

This section contains part of a sample output obtained by using the `-butterfly` option. For more information on the `-butterfly` option, see Section 7.3.4, page 114. The following command generated the output:

```
% prof -butterfly generic generic.Counts
```

The following output, which has been adjusted slightly in this example, concentrates on the butterfly function list part of the display. The first line in the header applies to the function that called the function under consideration. The second line in the header applies to the function under consideration. The third line applies to the functions it called.

```
-------------------------------------------------------------------------
SpeedShop profile listing generated Wed Feb  4 10:22:01 1998
   prof -butterfly generic generic.Counts
                 generic (n32): Target program
                  pixie-counts: Experiment name
                  pixie-counts: Marching orders
                 R4400 / R4000: CPU / FPU
                             1: Number of CPUs
                           175: Clock frequency (MHz.)
-------------------------------------------------------------------------
Summary of ideal time data (pixie-counts)--
                    2062563311: Total number of instructions executed
                    3929944454: Total computed cycles
                        22.457: Total computed execution time (secs.)
                         1.905: Average cycles / instruction
-------------------------------------------------------------------------
 .
 .
 .
-------------------------------------------------------------------------
Butterfly function list, in descending order by inclusive ideal time
-------------------------------------------------------------------------
         attrib.% attrib.time                         incl.time  caller (callsite) [index]
 [index]   incl.%   incl.time    self%   self-time       procedure [index]
                              attrib.% attrib.time    incl.time  callee (callsite) [index]
 .
 .
 .
-------------------------------------------------------------------------
           100.0%      22.456                            22.456  main [2]
     [3]   100.0%      22.456     0.0%       0.000         Scriptstring [3]
                                 95.5%      21.454       21.454  usrtime [4]
                                  3.7%       0.829        0.829  libdso [6]
                                  0.8%       0.172        0.172  cvttrap [9]
                                  0.0%       0.001        0.001  iofile [11]
                                  0.0%       0.000        0.000  dirstat [23]
                                  0.0%       0.000        0.001  genLog [12]
                                  0.0%       0.000        0.000  linklist [26]
                                  0.0%       0.000        0.000  fpetraps [27]
                                  0.0%       0.000        0.000  fprintf [21]
                                  0.0%       0.000        0.000  sprintf [17]
                                  0.0%       0.000        0.000  strcmp [60]
-------------------------------------------------------------------------
            95.5%      21.454                            22.456  Scriptstring [3]
     [4]    95.5%      21.454     0.0%       0.000         usrtime [4]
                                 95.5%      21.454       21.454  anneal [5]
                                  0.0%       0.000        0.001  genLog [12]
                                  0.0%       0.000        0.000  fprintf [21]
-------------------------------------------------------------------------
```

Miscellaneous Commands [9]

This chapter describes SpeedShop commands for exploring memory usage and paging, and for printing data files generated by SpeedShop tools. It contains the following sections:

- Using the `thrash` Command, see Section 9.1, page 133.
- Using the `squeeze` Command, see Section 9.2, page 134.
- Calculating the Working Set of a Program, see Section 9.3, page 135.
- Dumping Performance Data Files, see Section 9.4, page 137.
- Dumping Compiler Feedback Files, see Section 9.5, page 143.
- Filtering an MPI experiment file to vampir format with `ssfilter`(1), see Section 9.6, page 144.

9.1 Using the `thrash` Command

The `thrash` command allows you to explore paging behavior by allocating a region of virtual memory and accessing that memory either randomly or sequentially.

9.1.1 `thrash` Syntax

The syntax for the `thrash`(1) command is as follows:

`thrash` [*args*]

args	One or more of the following flags:	
	`-k` *n*	The amount of memory to access in kilobytes, where *n* is the number of kilobytes.
	`-m` *n*	The amount of memory to access in megabytes, where *n* is the number of megabytes.
	`-p` *n*	The amount of memory to access in pages, where *n* is the number of pages.
	`-n` [*count*]	The number of references to make before exiting. The default is 10,000.

`-s`	Sequential thrashing. The default is random.
`-w` *time*	The amount of time, in seconds, `thrash` should sleep after thrashing but before exiting.

9.1.2 Effects of `thrash`

Once the memory is allocated, `thrash` prints a message on `stdout`, saying how much memory it is using and then proceeds to access it. The following is an example:

```
% thrash -m 4

thrashing randomly: 4.00 MB (= 0x00400000 = 4194304 bytes = 1024 pages)

        10000 iterations
```

You can use `thrash` in conjunction with `ssusage`(1) and `squeeze`(1) to determine the approximate available working memory on a system, as described in Section 9.3, page 135.

9.2 Using the squeeze Command

The `squeeze` command lets you specify an amount of virtual memory to lock down into real memory, thus making it unavailable to other processes. This command can be used only in superuser mode.

9.2.1 squeeze Syntax

The syntax for the `squeeze`(1) command is as follows:

`squeeze [`*unit*`]` *amount*

unit One of the following options indicating the unit of measure. If no option is specified, the default is megabytes.

`-k`	Kilobytes
`-m`	Megabytes
`-p`	Pages
`-%`	A percentage of the installed memory

amount	The amount of memory to be locked.

9.2.2 Effects of squeeze

The `squeeze`(1) command performs the following operations:

- Locks down the amount of virtual memory you supply as an argument to the command.
- Prints a message to `stdout` that provides information on how much memory has been locked and how much working memory is available.
- Sleeps indefinitely, or until interrupted by `SIGINT` or `SIGTERM`. At that time, it frees up the memory and exits with an exit message.

Wait until after the exit message is printed before doing any experiments.

Here is an example:

```
% squeeze 4
squeeze: leaving 60.00 MB ( = 0x03c01000 = 62918656 ) available memory;
         pinned 4.00 MB ( = 0x00400000 = 4194304 ) at address 0x1000e000;
         from 64.00 MB ( = 0x04001000 = 67112960 ) installed memory.
```

Use `Ctrl-C` to exit `squeeze`. The following message is printed:

```
squeeze exiting
```

9.3 Calculating the Working Set of a Program

You can use the `thrash`, `squeeze`, and `ssusage` commands together to determine the approximate working set of a program. For all practical purposes, the working set of your program is the size of memory allocated.

The process involves three steps. First you determine the working set of the kernel and other applications:

1. Choose a machine that has a large amount of physical memory (enough to allow your target application to run without any paging other than at startup).
2. Make sure that the machine is running a minimal number of applications that will remain fairly consistent for the duration of these steps.

3. Run `thrash` with `ssusage` to determine the working set of the kernel and any other applications you have running.

   In this example, the `thrash` command uses 4 MB of memory:

   ```
   % ssusage thrash -m 4
   ```

   When the `thrash` command completes, `ssusage` prints the resource usage of `thrash`. The value labeled `majf` gives the number of major page faults (that is, the number of faults that required a physical read). When you run on a machine with a large amount of physical memory, this value is the number of faults needed to start the program, which is the minimum number for any run. For more information on `ssusage`, see Chapter 5, page 65.

4. As super user in a separate window, run the `squeeze` command to lock down an amount of memory.

5. Rerun `thrash` with `ssusage`, as shown here:

   ```
   % ssusage thrash -m 4
   ```

6. Repeat steps 1 and 2, increasing the amount of memory for `squeeze`, until the `majf` number begins to rise.

   The amount of working memory available reported by `squeeze` at the point at which page faults begin to rise for `thrash` tells you the combined working set of `thrash` (approximately 4 MB), the kernel, and any other applications you have running.

7. Deduct the 4 MB that `thrash` uses from the amount of working memory reported by `squeeze` at the point the page faults began to rise.

   This computation helps you find the approximate working set of the kernel and any other applications that are running on the machine. You will need this number when you reach the next steps.

8. Determine the working set of the program you are interested in. Make sure the applications that the machine is running remain consistent with the setup from step 2.

9. Run `ssusage` with your program to ensure that the machine has the amount of memory your program needs.

   `ssusage` *prog_name*

When your program exits, `ssusage` prints the application's resource usage. The `majf` field gives the number of major page faults. When run on a machine with a large amount of physical memory, this value is the number of faults needed to start the program, which is the minimum number for any run.

10. Switch to super user.
11. Run `squeeze` to lock down an amount of memory. The following example locks down 15 megabytes of memory:

    ```
    squeeze 15
    ```

12. Rerun your program with `ssusage`.
13. Repeat steps 11 and 12 until the `majf` number begins to rise.
14. Deduct the amount squeezed at the point at which the application begins to page fault from the total amount of physical memory in the system. This computation determines the combined working set of your program, the kernel, and any other applications you have running.
15. Deduct the amount of working memory calculated in step 7 from the total amount of physical memory in the system. This computation determines the approximate working set of your program.

9.4 Dumping Performance Data Files

All the performance data for a single process is in one file. The file begins with a prologue and continues with a mixture of performance data, sample records, and control records.

The `ssdump` command can be used for printing performance data files. It provides a formatted ASCII dump of one or more performance experiment data files. This command is most likely to be useful in verifying performance data that does not seem accurate when reported through `prof`.

9.4.1 ssdump Syntax

The syntax of the `ssdump`(1) command is as follows:

`ssdump [`*options*`]` *files*

options Zero or more of the following print options:

	`-d`	Prints detailed information for each record in the experiment file. For compressed records, the compressed form will be dumped.
	`-D`	Prints detailed information for each record in the experiment file. For compressed records, the uncompressed form will be dumped.
	`-h`	Prints the hexadecimal contents of the body of each record in the experiment file.
	`-i` *index*	Prints only one record at *index* in the file.
	`-q`	Suppresses the printing of those fields that will normally change from run to run, such as process IDs and time stamps. This option is useful for quality assurance work to enable automatic comparisons of recorded experiments.
	`-s` *offset*	Prints only one record of the experiment file at *offset* into the file.
files	One or more SpeedShop experiment files.	

9.4.2 Experiment File Format

The experiment file is written as a string of beads, or experiment records, each of which has the following characteristics:

- A 32-bit type
- A 32-bit byte count
- A body whose length is given by the byte count, rounded up to a doubleword boundary

The file prologue consists of the following beads:

- File identifier bead, which acts as a magic number, indicating that the file is a SpeedShop data file
- Machine and executable name
- Hardware inventory describing the machine
- Machine page size
- O/S revision, date, and checksum information about the executable

- Target name (the target is the executable after instrumentation)
- Arguments with which the target was invoked
- Instrumentation performed
- Types of performance data that are to be recorded in the remainder of the file

The following example instructs `ssdump` to display the performance data of a `pcsamp` experiment:

```
% ssdump generic.pcsamp.m847
```

The following is partial output from `ssdump`. The format has been adjusted slightly to meet presentation needs.

```
Printing experiment record file ‘‘generic.pcsamp.m847’’ (2688 bytes), last written
on Tue  15 Apr 1997  15:27:02
SpeedShop File Preface                1, offset 0 = 0x00000000 (size 32)
          file type 1 (SSRUN); version 4
          process control flags: 0xd
                    _SPEEDSHOP_TRACE_FORK=True
                    _SPEEDSHOP_TRACE_FORK_TO_EXEC=False
                    _SPEEDSHOP_TRACE_SPROC=True
                    _SPEEDSHOP_TRACE_EXEC=True
                    _SPEEDSHOP_TRACE_SYSTEM=False
          ancestor exp file name:
          created: Tue  15 Apr 1997  15:26:10.719
Hardware Inventory                    2, offset 40 = 0x00000028 (size 280)
          hardware inventory: 17 items
          class   1, type   1, contrlr 100, unit 255, state 12
          class   1, type   3, contrlr   0, unit   0, state 8192
          class   1, type   2, contrlr   0, unit   0, state 8208
          class   4, type   8, contrlr   0, unit   0, state 2
          class   5, type   5, contrlr   0, unit   0, state 1
          class   3, type   3, contrlr   0, unit   0, state 16384
          class   3, type   4, contrlr   0, unit   0, state 16384
          class   3, type   9, contrlr   0, unit   0, state 64
          class   3, type   1, contrlr   0, unit   0, state 67108864
          class  12, type   3, contrlr   0, unit   0, state 16
          class   8, type   7, contrlr  17, unit   0, state 16777472
          class  10, type   3, contrlr   0, unit   0, state 16400
          class   8, type   0, contrlr   0, unit   0, state 1
          class   2, type   1, contrlr   0, unit  13, state 2
          class   2, type   2, contrlr   0, unit   2, state 0
          class   2, type   2, contrlr   0, unit   1, state 0
          class   7, type  14, contrlr   0, unit   0, state 0

Experiment name                      3, offset 328 = 0x00000148 (size 8)
          pcsamp

Experiment marching orders           4, offset 344 = 0x00000158 (size 16)
          pc,2,10000,0:cu

Capture module symbol                5, offset 368 = 0x00000170 (size 16)
          pc,2,10000,0

Capture module symbol                6, offset 392 = 0x00000188 (size 8)
          cu

Executable file                      7, offset 408 = 0x00000198 (size 8)
          generic

Target file                          8, offset 424 = 0x000001a8 (size 8)
          generic

Target arguments                     9, offset 440 = 0x000001b8 (size 32)
          Time: Tue  15 Apr 1997  15:26:10.719,  process pid = 847
          arguments: ""
Target begin                        10, offset 480 = 0x000001e0 (size 40)
          process # -1, pid = 847, event # 0
          event type = 0,0
                    at time = Tue  15 Apr 1997  15:26:10.719
Program Object List                 11, offset 528 = 0x00000210 (size 312)
          process # -1, pid = 847, event # 0, -- 5 DSOs
          Program Object 0, Named`generic’
              Link Time Address: 0x0000000010000000
               Run Time Address: 0x0000000010000000
                           Size: 0x0000000000007000 (28672)
                   Base Pointer: 0x0000000000000000

          Program Object 1, Named`/usr/lib32/libss.so’
              Link Time Address: 0x0000000009e50000
               Run Time Address: 0x0000000009e50000
                           Size: 0x0000000000002000 (8192)
                   Base Pointer: 0x0000000000000000

          Program Object 2, Named`/usr/lib32/libssrt.so’
```

```
                     Link Time Address: 0x0000000009da0000
                      Run Time Address: 0x0000000009da0000
                                  Size: 0x000000000008b000 (569344)
                          Base Pointer: 0x0000000000000000

           Program Object 3, Named`/usr/lib32/libm.so'
                    Link Time Address: 0x000000000f840000
                     Run Time Address: 0x000000000f840000
                                 Size: 0x0000000000028000 (163840)
                         Base Pointer: 0x0000000000000000

           Program Object 4, Named`/usr/lib32/libc.so.1'
                    Link Time Address: 0x000000000fa00000
                     Run Time Address: 0x000000000fa00000
                                 Size: 0x0000000000108000 (1081344)
                         Base Pointer: 0x0000000000000000


Target DSO open                        12, offset 848 = 0x00000350 (size 56)
          process # -1, pid = 847, event # 0
                    at time = Tue  15 Apr 1997  15:27:00.716
          fname = ./dlslave.so
Program Object List                    13, offset 912 = 0x00000390 (size 360)
          process # -1, pid = 847, event # 0, -- 6 DSOs
          Program Object 0, Named`generic'
              Link Time Address: 0x0000000010000000
               Run Time Address: 0x0000000010000000
                           Size: 0x0000000000007000 (28672)
                   Base Pointer: 0x0000000000000000

          Program Object 1, Named`/usr/lib32/libss.so'
              Link Time Address: 0x0000000009e50000
               Run Time Address: 0x0000000009e50000
                           Size: 0x0000000000002000 (8192)
                   Base Pointer: 0x0000000000000000

          Program Object 2, Named`/usr/lib32/libssrt.so'
              Link Time Address: 0x0000000009da0000
               Run Time Address: 0x0000000009da0000
                           Size: 0x000000000008b000 (569344)
                   Base Pointer: 0x0000000000000000

          Program Object 3, Named`/usr/lib32/libm.so'
```

```
                 Link Time Address: 0x000000000f840000
                  Run Time Address: 0x000000000f840000
                              Size: 0x0000000000028000 (163840)
                      Base Pointer: 0x0000000000000000

             Program Object 4, Named`/usr/lib32/libc.so.1’
                 Link Time Address: 0x000000000fa00000
                  Run Time Address: 0x000000000fa00000
                              Size: 0x0000000000108000 (1081344)
                      Base Pointer: 0x0000000000000000

             Program Object 5, Named`./dlslave.so’
                 Link Time Address: 0x000000005ffe0000
                  Run Time Address: 0x000000005ffe0000
                              Size: 0x0000000000001000 (4096)
                      Base Pointer: 0x0000000000000000

   Sample event trigger                  14, offset 1280 = 0x00000500 (size 40)
             process # -1, trap index # -1
                       at time = Tue  15 Apr 1997  15:27:01.989, #-1

   Compressed PC sampling array (16-bit)   15, offset 1328 = 0x00000530 (size 320)
             compressed short array, dso index = 0, array size = 7168, 156
             compressed

   Compressed PC sampling array (16-bit)   16, offset 1656 = 0x00000678 (size 16)
             compressed short array, dso index = 1, array size = 2048, 4 compressed

   Compressed PC sampling array (16-bit)   17, offset 1680 = 0x00000690 (size 40)
             compressed short array, dso index = 2, array size = 142336, 16
             compressed

   Compressed PC sampling array (16-bit)   18, offset 1728 = 0x000006c0 (size 16)
             compressed short array, dso index = 3, array size = 40960, 4 compressed

   Compressed PC sampling array (16-bit)   19, offset 1752 = 0x000006d8 (size 64)
             compressed short array, dso index = 4, array size = 270336, 28
             compressed

   Compressed PC sampling array (16-bit)   20, offset 1824 = 0x00000720 (size 48)
             compressed short array, dso index = 5, array size = 1024, 20 compressed

   PC sampling array (16-bit)            21, offset 1880 = 0x00000758 (size 16)
```

```
          short array, dso index = -1, array size = 1

Resource usage                        22, offset 1904 = 0x00000770 (size 680)

Sample data end marker                23, offset 2592 = 0x00000a20 (size 40)

Target termination                    24, offset 2640 = 0x00000a50 (size 40)
          process # -1, pid = 847, event # 0
          event type = 0,0 (normal termination, exit status 0)
                    at time = Tue  15 Apr 1997  15:27:02.231


    ** End-of-File                     25, offset 2688 = 0x00000a80 (size 0)

**** End of experiment record file ‘‘generic.pcsamp.m847’’
```

9.5 Dumping Compiler Feedback Files

The `fbdump` command prints the compiler feedback files generated by running `prof -feedback`. For more information on using compiler feedback files, view the `cord`(1) or `cc`(1) man pages.

9.5.1 fbdump Syntax

The syntax for the `fbdump`(1) command is as follows:

`fbdump` [*`options`*] *file*

options	Zero or more of the options described in table Table 16.
file	The feedback file name. This file has a `.fb` extension.

Table 16. Options for `fbdump`

Option	Prints
`-all`	Feedback using all options. This is the default.
`-ascii`	Feedback in the same style as an earlier version of the feedback dump program.

Option	Prints
`-bb`	Feedback per the basic block table, as described in the `cmplrs/fb.h` file. If `-verbose` is specified, all basic blocks are printed, even those with zero execution counts. If `-verbose` is not specified, `fbdump` prints only the basic blocks that have nonzero execution counts.
`-call`	Feedback call table as described in the `cmplrs/fb.h` file. If `-verbose` is specified, all the points of call are printed, even if they have not been called. If `-verbose` is not specified, `fbdump` prints only the relevant information on the calls.
`-header`	Feedback file header as described in the `cmplrs/fb.h` file.
`-proc`	Feedback procedure table as described in the `cmplrs/fb.h` file. If `-verbose` is specified, all procedures will be printed, even if they are not invoked. If `-verbose` is not specified, `fbdump` prints only the relevant information on the procedures that have been invoked.
`-sections`	Feedback file section headers table as described in the `cmplrs/fb.h` file.
`-str`	Feedback string table.
`-verbose`	All the information in verbose mode, including a table with all zero entries.

9.6 Converting an MPI Experiment File to Vampir Format

The vampir software product displays and analyzes Message Passing Interface (MPI) experiment files. It is a product of Pallas, a software company specializing in high performance computing. For more information on the company and the vampir software, see the following web site:

`http://www.pallas.com`

The `ssfilter`(1) command converts a SpeedShop MPI experiment file into a form in which it can be viewed using vampir. For information on generating MPI experiment files, see Section 6.6, page 82.

The following commands generate a `pcsampx` experiment file on each of the four processors involved in the MPI program and converts them to a single file in vampir format:

```
mpirun -np 4 ssrun -pcsampx verge
ssfilter a.out.mpi.f* -o verge.vampir
```

By default, `ssfilter` periodically returns status information while it is processing. The status tells you what percentage of its job is complete and what percentage remains to be done. You can turn the status messages off by specifying the `-noverbose` option. For information on all of the options, see the `ssfilter`(1) man page.

Glossary [10]

This glossary defines terms used in this document.

basic block	A set of instructions with a single entry point, a single exit point, and no branches into or out of the set.
bead	A record in an experiment.
call stack	A software stack of functions and routines that represent the state of the program at any time. The functions and routines are listed in the reverse order, from top to bottom, in which they were called. If function `a` is immediately below function `b` in the stack, then `a` was called by `b`. The function at the bottom of the stack is the one currently executing.
context switch	The act of saving the state of one process and replacing it with that of another when both processes time-share a single processor.
counts	The number of times an event takes place during data gathering. For example, a count may be kept of the number of times a function executes.
CPU time	*Process virtual time* (see the glossary entry) plus time spent when the system is running on behalf of the process, performing such tasks as executing a system call. This is the time returned in `pcsamp` and `usertime` experiments. It can be specified in an experiment by using the `ut,30000,2` marching orders.
exclusive time	The execution time of a given function but not of any functions called by that function. See *inclusive time*.
graduated instruction	As a performance enhancement, when an R10000 system comes to a point in the execution of a program at which either of two paths might be taken, it begins to execute both paths until it knows for sure which path is correct. Graduated instructions are those on the path it will

	eventually follow. *Issued instructions* are those on the path it does not follow.
inclusive time	The execution time both of a given function and of any functions called by that function. See *exclusive time*.
issued instruction	See the definition of *graduated instruction*.
overflow interval	As used by the hardware counter experiments, it is the number at which a hardware counter exceeds a preset value. See the `speedshop`(1) man page, `dsc_hwc` experiment.
PC	Program counter. A register that contains the address of the instruction that is currently executing.
process virtual time	Time spent when a program is actually running. This does not include either 1) the time spent when the program is swapped out and waiting for a CPU or 2) the time when the operating system is in control, such as executing a system call for the program. The marching orders `ut,30000,1` return process virtual time.
statistical data	Sampling. The results from this method of data gathering vary from run to run.
system time	The time the operating system spends performing services for a program, such as executing system calls and I/O.
TLB	Translation lookaside buffer. This is hardware used by the CPU to quickly translate a virtual address (such as the name of a variable) to a physical memory address.
TDT model	Target Description Table model. A CPU model used to calculate ideal time.
user time	The same as *CPU time*.
wall-clock time	Total time a program takes to execute, including the time it takes waiting for a CPU. This is real time, not computer time. The marching orders `ut,30000,0` return wall-clock time.

Index