SpeedShop User's Guide

Document Number 007-3311-002

CONTRIBUTORS

Written by Janet Home-Lorenzin and Wendy Ferguson. Updated by Renate Kempf and Sandra Motroni
Illustrated by Dany Galgani
Production by Kirsten Johnson
Engineering contributions by Marty Itzkowitz, Pete Orelup, Alexandros Poulos, Jun Yu, Marco Zagha, Chris Hull, Zaineb Asaf, Aaron Schuman, and Brond Larson.

SpeedShop User's Guide
Document Number 007-3311-002

Contents

List of Figures

List of Tables

About This Guide

This manual is a user's guide for the SpeedShop performance tools, Release 1.2. It contains the following chapters:

- Chapter 1, "Introduction to Performance Analysis," provides a general introduction to performance analysis concepts and techniques, plus an overview of the SpeedShop tools.
- Chapter 2, "Tutorial for C Users," provides a tutorial on how to collect performance data and generate reports for a C program.
- Chapter 3, "Tutorial for Fortran Users," provides a tutorial on how to collect performance data and generate reports for Fortran programs running on single-processor machines.
- Chapter 4, "Experiment Types," describes the types of experiments that can be performed using SpeedShop tools.
- Chapter 5, "Collecting Data on Machine Resource Usage," describes how to use the *ssusage* command to collect information about a program's machine resource usage.
- Chapter 6, "Setting Up and Running Experiments: ssrun," explains in detail how to set up and run experiments using *ssrun*, and explains how to use caliper points to generate reports for part of a program.
- Chapter 7, "Analyzing Experiment Results: prof," explains how to generate reports from performance data using *prof*.
- Chapter 8, "Using SpeedShop in Expert Mode: pixie," explains how to use *pixie* and *prof* directly, without invoking *ssrun*.
- Chapter 9, "Miscellaneous Commands," explains how to use the *thrash* and *squeeze* commands to determine the memory usage, or working set, of your application. It also covers commands to print performance data files.

About This Revision

This revision of the manual was prepared in the spring of 1997. It includes bug fixes and updates for the Speedshop 1.2 release.

Chapter 1

Introduction to Performance Analysis

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

- "Sources of Performance Problems" provides a general overview of potential performance problems.
- "Fixing Performance Problems" discusses how you can use SpeedShop to determine what the problems are.
- "SpeedShop Tools" lists SpeedShop commands, experiment types, and libraries.
- "Using SpeedShop Tools for Performance Analysis" steps you through the general steps to take when using SpeedShop.

Sources of Performance Problems

To tune a program's performance, you need to determine its consumption of machine resources. At any point (or phase) 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

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've 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.

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

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

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. One telltale indication of page-thrashing with paging to a local disk is noise during disk accesses. To fix a memory-bound process, try to improve the memory reference patterns or, if possible, decrease the memory used by the program.

Bugs

Certain bugs can cause performance problems. Examples include:

- The program is unnecessarily reading from the same file twice in different parts.
- 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()**).

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's a bad idea to do so. Any such "features" may break in future compiler releases. The best way to produce efficient code that can be trusted to remain efficient is to follow good programming practices. In particular, choose good algorithms and leave the details to the compiler.

SpeedShop Tools

The SpeedShop tools allow you to run experiments and generate reports to track down the sources of performance problems. SpeedShop consists of a set of commands that can be run in a shell, an API, 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, and supported programs and languages.

Main Commands

SpeedShop provides the commands listed in Table 1-1.

Table 1-1 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.

Additional Commands

SpeedShop provides the additional commands shown in Table 1-2.

Table 1-2 SpeedShop Additional Commands

Command	Description
pixie	Instruments an executable to enable basic block counting experiments to be performed. If you use *ssrun*, you will not normally need to call this program directly.
fbdump	Prints out the formatted contents of compiler feedback files generated by *prof*.
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 paging behavior.
ssdump	Prints out formatted performance data that was collected while running *ssrun*. This program is included for SpeedShop debugging purposes. You don't normally need to use it.

Experiment Types

You can conduct the following types of experiments using the *ssrun* command:

- Statistical PC sampling with **pcsamp** experiments**.**

  Data is measured by periodically sampling the Program Counter (PC) of the target executable when it is executing in the CPU. The PC 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.

- Statistical hardware counter sampling with **_hwc** experiments.

  Hardware counter experiments are available on R10000™ systems that have built-in hardware counters. Data is measured by collecting information each time the specified hardware counter overflows. You can specify the hardware counter and the overflow interval you want to use.

- 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.

- Basic block counting with **ideal**.

  Data is measured by counting basic blocks and calculating an ideal CPU time for each function. This involves instrumenting the program to divide 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. Instrumentation also permits a count of all dynamic (function-pointer) calls to be recorded.

- Floating point exception trace with **fpe**.

  A floating point exception trace collects each floating point exception with the exception type and the call stack at the time of the exception. *prof* generates a report showing inclusive and exclusive floating point exception counts.

SpeedShop Libraries

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

Table 1-3 provides information about the different SpeedShop libraries.

Table 1-3 SpeedShop Libraries

Library	Description
libss.so	A shared library (DSO) that supports *libssrt.so*. *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 process(es), 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 reference 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 reference page for more information.
libpixrt.so	A shared library (DSO) used by pixified programs.

API

The SpeedShop API is primarily available to allow you to use **ssrt_caliper_point** to set caliper points in your source code. See “Using Calipers” in Chapter 6 for information on using caliper points. For information on other API functions, see the ssapi reference page.

Supported Programming Models and Languages

The SpeedShop tools support programs with the following characteristics:

- Shared libraries (DSOs.)
- Non-stripped executables.
- Executables containing *fork*, *sproc*, *system*, or *exec* commands.
- Executables using supported techniques for opening, closing, and/or delay-loading DSOs.
- C, C++, Fortran (Fortran-77, Fortran-90, and High-Performance Fortran), or Ada® 95 source code.
- Power Fortran and Power C source code.

  prof understands the syntax and semantics of the multi-processing runtime 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.

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:

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:

- 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 or WorkShop debuggers.
- Use *prof* to analyze the data and generate reports.

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.

```
ssusage prog_name prog_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, "Collecting Data on Machine Resource Usage," or see the ssusage reference page.

Using ssrun and prof to Gather and Analyze Performance Data

This section describes the steps involved in a performance analysis cycle when using the main interface to the SpeedShop tools: the *ssrun* command.

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 either do this by calling *ssrun* with the appropriate experiment type, or you can set up your environment to call *pixie* directly to instrument your executable. Information on setting up your environment and running *pixie* directly can be found in Chapter 8, "Using SpeedShop in Expert Mode: pixie."

To perform a performance analysis, follow these general steps:

1. Build the executable.

   You can usually build the executable as you would normally. See "Building Your Executable" in Chapter 6 for information on how to build the executable.

2. Specify caliper points if you want to collect data for only a portion of your program.

   See "Collecting Data for Part of a Program" for more information.

3. To collect performance data, call *ssrun* with the parameters below. Use the information in Table 1-4 to determine which experiments to run:

   `ssrun` *flags exp_type prog_name prog_args*

flags	One or more valid flags. For a complete list of flags, see the ssrun reference page.
exp_type	Experiment name.
prog_name	Executable name.
prog_args	Arguments to the executable

Table 1-4 Choosing an Experiment Type

Performance Problem	Experiment(s) to Run
High user CPU time	**usertime** **pcsamp** (four variants) ***_hwc** experiments **ideal**
High system CPU time	If floating point exceptions are suspected: **fpe**
High I/O time	**ideal**, then examine counts of I/O routines
High paging (majf)	**ideal**, then **prof -feedback** and **cord** to rearrange procedures. If inefficient heap usage is suspected, use WorkShop's Performance Analyzer to gather information.

For each process of the executable, the experiment data is stored in a file with a name of the format *prog_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 1-5 for letter codes and descriptions.

Table 1-5 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, "Setting Up and Running Experiments: ssrun," or view the ssrun reference page.

4. To generate a report of the experiment, call *prof* with the following parameters:

   `prof` *flags data_file*

   flags One or more valid flags. For a complete list of flags, see the prof reference page.

   data_file The name of the file in which the experiment data was recorded.

   For more information on using *prof*, see Chapter 7, "Analyzing Experiment Results: prof," or see the prof reference page.

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.

You can set caliper points using one of the following:

- the SpeedShop API
- the caliper signal environment
- a debugger such as the ProDev WorkShop debugger

For more information on using calipers, see "Using Calipers" in Chapter 6.

Chapter 2

Tutorial for C Users

This chapter presents a tutorial for using the SpeedShop tools to gather and analyze performance data in a C program, and covers these topics:

- "Tutorial Overview" introduces the sample program and explains the different scenarios in which it will be used.
- "Tutorial Setup" steps you through the necessary setup for running the experiment.
- "Analyzing Performance Data" 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 form of the results should be the same.

Tutorial Overview

This tutorial uses a sample program called *generic*. There are three versions of the program:

generic directory	Contains files for the n32-bit ABI
generic64 directory	Contains files for the 64-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.

Contents of the generic Program

generic 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 usertime
- scan a directory and run the *stat* command on each file
- perform file I/O
- generate a number of floating point exceptions
- link and call a routine in a DSO

Output from the generic Program

Output from the program looks like the following:

```
0:00:00.000 ======== (24479) Begin script Fri 03 May 96  10:17:13.
begin script `ll.u.d.i.f.dso'
0:00:00.032 ======== (24479) start of linklist Sun 03 May 96  10:17:13.
linklist completed.
0:00:00.002 ======== (24479) start of usrtime Fri 03 May 96  10:17:13.
usertime completed.
0:00:10.824 ======== (24479) start of dirstat Fri 03 May 96  10:17:24.
dirstat of /usr/include completed, 242 files.
0:00:10.844 ======== (24479) start of iofile Fri 03 May 96  10:17:24.
iofile on /unix completed, 4221656 chars.
0:00:11.036 ======== (24479) start of fpetraps Fri 03 May 96  10:17:24.
fpetraps completed.
0:00:11.038 ======== (24479) start of libdso Fri  03 May 96  10:17:24.
dlslave_init executed
dlslave_routine executed
        slaveusertime completed, x = 5000000.000000.
        libdso: dynamic routine returned 13
        end of script `u.d.i.f.dso'
0:00:11.930 ======== (24479) End script Fri  03 May 96  10:17:25.
```

Use of the generic Program

The tutorial shows you how to perform the following experiments using *ssrun*, and how to interpret experiment-specific reports generated by *prof*:

- **usertime**
- **pcsamp**
- **dsc_hwc**
- **ideal**
- **fpe**

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_`*version your_dir*
3. Change to the directory you just created:

   `cd` *your_dir*`/generic_`*version*
4. Compile the program by entering

   ```
   make all
   ```

   This provides an executable for the experiment.
5. Set the library path so that the program can find shared libraries in the *generic* directory:

   `setenv LD_LIBRARY_PATH` *your_dir*`/generic_`*version*

Analyzing Performance Data

This section explains how to run the following experiments on the *generic* program and generate and interpret the results:

- "usertime Experiment"
- "pcsamp Experiment"
- "Hardware Counter Experiment"
- "ideal Experiment"
- "fpe Trace"

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

usertime Experiment

This section explains how to perform a **usertime** experiment. The **usertime** experiment allows you to gather data on the amount of user time spent in each function in your program. For more information on **usertime**, see the "usertime Experiment" section in Chapter 4, "Experiment Types."

Performing a usertime Experiment

From the command line, enter

```
ssrun -usertime generic
```

This 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 filename is *generic.usertime.m24479*.

```
0:00:00.000 ======== (24479) Begin script Fri 03 May 96  10:17:13.
begin script `ll.u.d.i.f.dso'
0:00:00.032 ======== (24479) start of linklist Sun 03 May 96  10:17:13.
linklist completed.
0:00:00.002 ======== (24479) start of usrtime Fri 03 May 96  10:17:13.
usertime completed.
0:00:10.824 ======== (24479) start of dirstat Fri 03 May 96  10:17:24.
dirstat of /usr/include completed, 242 files.
0:00:10.844 ======== (24479) start of iofile Fri 03 May 96  10:17:24.
iofile on /unix completed, 4221656 chars.
0:00:11.036 ======== (24479) start of fpetraps Fri 03 May 96  10:17:24.
fpetraps completed.
0:00:11.038 ======== (24479) start of libdso Fri  03 May 96  10:17:24.
dlslave_init executed
dlslave_routine executed
        slaveusertime completed, x = 5000000.000000.
        libdso: dynamic routine returned 13
        end of script `u.d.i.f.dso'
0:00:11.930 ======== (24479) End script Fri  03 May 96  10:17:25.
```

Generating a Report

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

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

prof prints results to *stdout*. Note that the *prof* output below is a partial listing.

```
------------------------------------------------------------------------
Profile listing generated Mon Nov 18 11:43:45 1996
    with: prof generic.usertime.m24479
------------------------------------------------------------------------
Total Time (secs)     : 43.98
Total Samples         : 1466
Stack backtrace failed: 1
Sample interval (ms)  : 30
CPU                   : R4600
FPU                   : R4600
Clock                 : 100.0MHz
Number of CPUs        : 1
------------------------------------------------------------------------
index  %Samples  self descendents  total          name
[1]     99.9%    0.00       43.95    1465         __start
[2]     99.9%    0.00       43.95    1465         main
[3]     99.9%    0.00       43.95    1465         Scriptstring
[4]     94.5%    0.00       41.55    1385          usrtime
[5]     94.5%   41.52        0.03    1385          anneal
[6]      3.0%    0.00        1.32      44          libdso
[7]      3.0%    0.00        1.32      44          dlslave_routine
[8]      3.0%    1.32        0.00      44          slaveusrtime
[9]      2.2%    0.00        0.96      32          iofile
[10]     2.2%    0.00        0.96      32          fread
[11]     2.1%    0.00        0.93      31          __filbuf
[12]     2.1%    0.93        0.00      31          _read
[13]     0.2%    0.00        0.09       3          dirstat
[14]     0.1%    0.00        0.06       2          _stat
[15]     0.1%    0.06        0.00       2          _xstat
[16]     0.1%    0.00        0.03       1          linklist
[17]     0.1%    0.00        0.03       1          _malloc
[18]     0.1%    0.00        0.03       1          _smalloc
[19]     0.1%    0.00        0.03       1          __malloc
[20]     0.1%    0.00        0.03       1          init2da
[21]     0.1%    0.03        0.00       1          __sinf
[22]     0.1%    0.00        0.03       1          _readdir
[23]     0.1%    0.03        0.00       1          _ngetdents
[24]     0.1%    0.03        0.00       1          memcpy
```

Analyzing the Report

The report shows information for each function. The remaining columns are described below:

- The `index` column provides an index number for reference.
- The `%Samples` column shows the cumulative percentage of inclusive time spent in each function and its descendents. For example, 99.9% of the time was spent in **Scriptstring()** and all functions listed below it.
- The `self` 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 **memcpy()**.
- The `descendents` columns shows how much time, in seconds, was spent in callees of the function. For example, 43.95 seconds were spent in the callees of **main()**.
- The `total` column provides information on the number of samples of the function and all of its descendants.

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 source code line, machine instruction, and function in your program. For more information on **pcsamp**, see the "pcsamp Experiment" section in Chapter 4, "Experiment Types."

From the command line, enter

```
ssrun -fpcsamp generic
```

This starts the experiment. The **f** option is used with **pcsamp** for this program because the program runs quickly and does not gather much data using the default **pcsamp** experiment. 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 filename is *generic.fpcsamp.m14480*.

```
0:00:00.000 ======== (14480)             Begin script Sun  19 May 96  17:18:33.
        begin script `ll.u.d.i.f.dso'
0:00:00.074 ======== (14480)        start of linklist Sun  19 May 96  17:18:33.
        linklist completed.
0:00:00.085 ======== (14480)         start of usrtime Sun  19 May 96  17:18:33.
        usertime completed.
0:00:17.985 ======== (14480)         start of dirstat Sun  19 May 96  17:18:51.
        dirstat of /usr/include completed, 230 files.
0:00:18.008 ======== (14480)          start of iofile Sun  19 May 96  17:18:51.
        iofile on /unix completed, 4221656 chars.
0:00:20.321 ======== (14480)        start of fpetraps Sun  19 May 96  17:18:54.
        fpetraps completed.
0:00:20.323 ======== (14480)          start of libdso Sun  19 May 96  17:18:54.
dlslave_init executed
dlslave_routine executed
        slaveusertime completed, x = 5000000.000000.
        libdso: dynamic routine returned 13
        end of script `ll.u.d.i.f.dso'
0:00:21.394 ======== (14480)               End script Sun  19 May 96  17:18:55.
```

Generating a Report

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

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

Output similar to the following is generated:

```
------------------------------------------------------------------------------
Profile listing generated Sun May 19 17:21:27 1996
    with:       prof generic.fpcsamp.m14480
------------------------------------------------------------------------------
samples   time    CPU    FPU   Clock   N-cpu  S-interval Countsize
  19077    19s  R4000  R4010 150.0MHz   1      1.0ms     2(bytes)

Each sample covers 4 bytes for every  1.0ms ( 0.01% of 19.0770s)
------------------------------------------------------------------------------
  -p[rocedures] using pc-sampling.
  Sorted in descending order by the number of samples in each procedure.
  Unexecuted procedures are excluded.
------------------------------------------------------------------------------
samples   time(%)      cum time(%)      procedure (dso:file)

  17794    18s( 93.3)   18s( 93.3)         anneal (/usr/demos/SpeedShop/
          generic/generic:/usr/demos/SpeedShop/generic/generic.c)
   1046     1s(  5.5)   19s( 98.8)   slaveusrtime (/usr/demos/SpeedShop/generic/
          dlslave.so:/usr/demos/SpeedShop/generic/dlslave.c)
    163  0.16s(  0.9)   19s( 99.6)          _read (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/sys/read.s)
     34 0.034s(  0.2)   19s( 99.8)         memcpy (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/strings/bcopy.s)
     20  0.02s(  0.1)   19s( 99.9)         _xstat (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/sys/xstat.s)
      8 0.008s(  0.0)   19s( 99.9)          fread (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/stdio/fread.c)
      3 0.003s(  0.0)   19s(100.0)         iofile (/usr/demos/SpeedShop/generic/
          generic:/usr/demos/SpeedShop/generic/generic.c)
      3 0.003s(  0.0)   19s(100.0)        _doprnt (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/print/doprnt.c)
      1 0.001s(  0.0)   19s(100.0)         __sinf (/usr/lib32/libm.so:
          /work/cmplrs/libm/fsin.c)
      1 0.001s(  0.0)   19s(100.0)        init2da (/usr/demos/SpeedShop/generic/
          generic:/usr/demos/SpeedShop/generic/generic.c)
      1 0.001s(  0.0)   19s(100.0)         _write (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/sys/write.s)
      1 0.001s(  0.0)   19s(100.0)       _drand48 (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/math/drand48.c)
      1 0.001s(  0.0)   19s(100.0)      _morecore (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/gen/malloc.c)
      1 0.001s(  0.0)   19s(100.0)         fwrite (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/stdio/fwrite.c)

  19077    19s(100.0)   19s(100.0)          TOTAL
```

Analyzing the Report

The report has the following columns:

- The `samples` columns shows how many samples were taken when the process was executing in the function.
- The `time(%)` column shows the amount of time, and the percentage of that time over the total time that was spent in the function.
- The `cum time(%)` column shows how much time has been spent in the function up to and including the procedure in the list.
- The `procedure (dso:file)` column lists the procedure, its DSO name and file name. For example, the first line reports statistics for the procedure **anneal()** in the file *generic.c* in the generic executable.

Hardware Counter Experiment

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

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 allows you to capture information about secondary data cache misses. For more information on hardware counter experiments, see "ideal Experiment" in Chapter 4, "Experiment Types."

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 filename is *generic.dsc_hwc.m5999*.

Generating a Report

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

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

The report should look similar to the following partial listing:

```
--------------------------------------------------------------------------------
Profile listing generated Thu Jun  5 13:23:14 1997
    with:     prof generic.dsc_hwc.m5999
--------------------------------------------------------------------------------

     Counter               : Sec cache D misses
     Counter overflow value: 131
     Total number of ovfls : 10
     CPU                   : R10000
     FPU                   : R10010
     Clock                 : 196.0MHz
     Number of CPUs        : 1

--------------------------------------------------------------------------------
  -p[rocedures] using counter overflow.
  Sorted in descending order by the number of overflows in each procedure.
  Unexecuted or inlined procedures are excluded.
--------------------------------------------------------------------------------

  overflows(%)  cum overflows(%)      procedure (dso:file)

      4( 40.0)          4( 40.0)         memcpy (/usr/lib32/libc.so.1:bcopy.s)
      2( 20.0)          6( 60.0)         anneal (generic:generic.c)
      2( 20.0)          8( 80.0)        init2da (generic:generic.c)
1( 10.0)          9( 90.0)       _smalloc (/usr/lib32/libc.so.1:malloc.c)
      1( 10.0)         10(100.0)        _doprnt (/usr/lib32/libc.so.1:doprnt.c)

     10(100.0)                     TOTAL
```

Analyzing the Report

The columns in the report provide the following information:

- The `overflows(%)` column shows the number of overflows caused by the function, and the percentage of that number over the total number of overflows in the program.
- The `cum overflows(%)` column shows a cumulative number and percentage of overflows. For example, the **anneal()** function shows two overflows, but the cumulative number of overflows is six. Two overflows come from **anneal()** and four come from **memcpy()**.
- The `procedure (dso:file)` column shows the procedure name and the DSO and filename that contain the procedure.

ideal Experiment

This section takes you through the steps to perform an **ideal** experiment. For more information on ideal, see the "ideal Experiment" section in Chapter 4, "Experiment Types."

Performing an ideal Experiment

From the command line, enter

```
ssrun -ideal generic
```

This starts the experiment. First the executable and libraries are instrumented using *pixie*. 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 32 libraries, `.pixn32` for all n32 libraries, and `.pix64` for all 64 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 filename is *generic.ideal.m14517*, and the output to *stdout* looks like the following:

```
Beginning libraries
        /usr/lib32/libssrt.so
        /usr/lib32/libss.so
        /usr/lib32/libm.so
        /usr/lib32/libc.so.1
Ending libraries, beginning "generic"
...
Beginning libraries
Ending libraries, beginning "dlslave.so"
...
```

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, then the *generic* executable itself. The section that starts `Beginning libraries` and ends with `Ending libraries, beginning 'dlslave.so'` is added when the DSO *dlslave.so* is dlopen'd.

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:

```
Prof run at: Sun May 19 17:49:10 1996
 Command line: prof generic.ideal.m14517

        2662778531: Total number of cycles
         17.75186s: Total execution time
        1875323907: Total number of instructions executed
             1.420: Ratio of cycles / instruction
               150: Clock rate in MHz
            R4000: Target processor modelled
------------------------------------------------------------
```

```
Procedures sorted in descending order of cycles executed.
Unexecuted procedures are not listed. Procedures
beginning with *DF* are dummy functions and represent
init, fini and stub sections.
------------------------------------------------------------
        cycles(%)  cum %     secs    instrns    calls procedure(dso:file)

2524610038(94.81)  94.81    16.83 1797940023     1 anneal(generic.pixie:/usr
          /demos/SpeedShop/generic/generic.c)
 135001332( 5.07)  99.88     0.90   75000822     1 slaveusrtime
          (./dlslave.so.pixn32:/usr/demos/SpeedShop/generic/dlslave.c)
   1593422( 0.06)  99.94     0.01    1378737  4382 memcpy(./libc.so.1.pixn32:
          /work/irix/lib/libc/libc_n32_M3/strings/bcopy.s)
    735797( 0.03)  99.97     0.00     506627  4123 fread(./libc.so.1.pixn32:
          /work/irix/lib/libc/libc_n32_M3/stdio/fread.c)
    187200( 0.01)  99.98     0.00     124800  1600 next(./libc.so.1.pixn32:
          /work/irix/lib/libc/libc_n32_M3/math/drand48.c)
    136116( 0.01)  99.98     0.00      82498     1 iofile(generic.pixie:
          /usr/demos/SpeedShop/generic/generic.c)
     91200( 0.00)  99.98     0.00      62400  1600 _drand48(./libc.so.1.pixn32:
          /work/irix/lib/libc/libc_n32_M3/math/drand48.c)
     85497( 0.00)  99.99     0.00      56518     1 init2da(generic.pixie:
          /usr/demos/SpeedShop/generic/generic.c)
     74095( 0.00)  99.99     0.00      28063   628 __sinf(./libm.so.pixn32:
          /work/cmplrs/libm/fsin.c)
     56192( 0.00)  99.99     0.00       9360    16 offtime(./libc.so.1.pixn32:
          /work/irix/lib/libc/libc_n32_M3/gen/time_comm.c)
     51431( 0.00)  99.99     0.00      36405    35 _doprnt(./libc.so.1.pixn32:
          /work/irix/lib/libc/libc_n32_M3/print/doprnt.c)
     27951( 0.00) 100.00     0.00      19670   259 __filbuf(./libc.so.1.pixn32:
          /work/irix/lib/libc/libc_n32_M3/stdio/_filbuf.c)
     21392( 0.00) 100.00     0.00      10136    58 fwrite(./libc.so.1.pixn32:
          /work/irix/lib/libc/libc_n32_M3/stdio/fwrite.c)
     12744( 0.00) 100.00     0.00       9497   231 _readdir(./libc.so.1.pixn32:
          /work/irix/lib/libc/libc_n32_M3/gen/readdir.c)
      9960( 0.00) 100.00     0.00       7536    96 _xflsbuf(./libc.so.1.pixn32:
          /work/irix/lib/libc/libc_n32_M3/stdio/flush.c)
      7211( 0.00) 100.00     0.00       3959     1 dirstat(generic.pixie:
          /usr/demos/SpeedShop/generic/generic.c)
```

Analyzing the Report

The columns in the report provide the following information

- The `cycles(%)` column reports the number and percentage of machine cycles used for the procedure. For example, 2524610038 cycles, or 94.81% of cycles were spent in the **anneal()** procedure.
- The `cum%` column shows the cumulative percentage of cycles. For example, `99.88%` of all cycles were spent between the top two functions in the listing: **anneal()** and **slaveusrtime()**.
- The `secs` column shows the number of seconds spent in the procedure. For example, 16.83 seconds were spent in the **anneal()** procedure. The time represents an idealized computation based on modelling the machine. It ignores potential floating point interlocks and memory latency time (cache misses and memory bus contention).
- The `instrns` column shows the number of instructions executed for a procedure. For example, there were `1797940023` instructions devoted to the **anneal()** procedure.
- The `calls` column reports the number of calls to the procedure. For example, there was just one call to the **anneal()** procedure.
- The `procedure (dso:file)` column lists the procedure, its DSO name and filename. For example, the first line reports statistics for the procedure **anneal()** in the file *generic.c* in the generic executable.

fpe Trace

This section takes you through the steps to perform an fpe trace. For more information on the fpe trace, see the "fpe Trace" section in Chapter 4, "Experiment Types."

Performing an fpe Trace

From the command line, enter

```
ssrun -fpe generic
```

This starts the experiment. Output from *generic* and from *ssrun* is printed to *stdout*. A data file with a name generated by concatenating the process name, *generic*, the experiment type, *fpe*, and the experiment ID is also generated. In this example, the filename is *generic.fpe.m18823*.

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:

```
-------------------------------------------------------------------------
Profile listing generated Mon Nov 18 11:46:33 1996
    with:     prof generic.fpe.m18823
-------------------------------------------------------------------------
Total FPEs            : 4
Stack backtrace failed: 0
CPU                   : R4600
FPU                   : R4600
Clock                 : 100.0MHz
Number of CPUs        : 1
-------------------------------------------------------------------------
index  %FPEs     self descendents  total    name
[1]    100.0%       0           4       4   __start
[2]    100.0%       0           4       4   main
[3]    100.0%       0           4       4   Scriptstring
[4]    100.0%       4           0       4   fpetraps
```

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 `index` column provides an index number for reference.
- The `%FPEs` column shows the percentage of the total number of floating point exceptions that were found in the function.
- The `self` column shows how many floating point exceptions were found in the function. For example, four floating point exceptions were found in **fpetraps**.
- The `descendents` columns shows how many floating point exceptions were found in the descendents of the function. For example, four floating point exceptions were found in the descendents of **main()**.
- The `total` column provides information on the number of floating point exceptions out of the total that were found.

This concludes the tutorial.

Chapter 3

Tutorial for Fortran Users

This chapter presents a tutorial for using the SpeedShop tools to gather and analyze performance data in a Fortran program, and covers these topics:

- "Tutorial Overview" introduces the sample program and explains the different scenarios in which it will be used.
- "Tutorial Setup" steps you through the necessary setup for running the experiment.
- "Analyzing Performance Data" 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 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 form of the results should be the same.

Tutorial Overview

This tutorial is based on a standard benchmark program called *linpackup*. There are three versions of the program: the *linpack* directory contains files for the n32-bit ABI, the *linpack64* directory contains files for the 64-bit ABI and the *linpacko32* directory contains files for the 32-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

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
  1.180E+00  1.000E-02  1.190E+00  1.528E+01  1.309E-01  2.125E+01
  1.180E+00  1.000E-02  1.190E+00  1.528E+01  1.309E-01  2.125E+01
  1.180E+00  1.000E-02  1.190E+00  1.528E+01  1.309E-01  2.125E+01
  1.180E+00  1.000E-02  1.190E+00  1.528E+01  1.309E-01  2.125E+01

 times for array with leading dimension of 300
  1.180E+00  1.000E-02  1.190E+00  1.528E+01  1.309E-01  2.125E+01
  1.180E+00  2.000E-02  1.200E+00  1.515E+01  1.320E-01  2.143E+01
  1.180E+00  1.000E-02  1.190E+00  1.528E+01  1.309E-01  2.125E+01
  1.181E+00  1.200E-02  1.193E+00  1.524E+01  1.312E-01  2.130E+01
```

Experiments Performed in This Tutorial

The tutorial shows you how to perform the following experiments using *ssrun*, and how to interpret experiment-specific reports generated by *prof*:

- **usertime**
- **pcsamp**
- **dsc_hwc**
- **ideal**

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 where you have write permission:

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

   `cd` *your_dir/linpack_version*
4. Compile the program by entering

   `make all`

   This provides an executable for the experiment.

Analyzing Performance Data

This section provides steps on how to run the following experiments on the *linpackup* program and generate and interpret the results:

- "usertime Experiment"
- "Hardware Counter Experiment"
- "pcsamp Experiment"
- "ideal Experiment"

You can follow the tutorial from start to finish, or you can follow steps for just the experiment(s) you want.

usertime Experiment

This section takes you through the steps to perform a **usertime** experiment. The **usertime** experiment allows you to gather data on the amount of user time spent in each function in your program. For more information on **usertime**, see the "usertime Experiment" section in Chapter 4, "Experiment Types."

Performing a usertime Experiment

From the command line, enter

```
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 example below. 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.m17566*.

```
ssrun: target PID 17566
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
  3.050E+00  3.000E-02  3.080E+00  5.903E+00  3.388E-01  5.500E+01
  3.030E+00  3.000E-02  3.060E+00  5.941E+00  3.366E-01  5.464E+01
  3.030E+00  3.000E-02  3.060E+00  5.941E+00  3.366E-01  5.464E+01
  3.030E+00  3.000E-02  3.060E+00  5.941E+00  3.366E-01  5.464E+01

 times for array with leading dimension of 300
  3.030E+00  3.000E-02  3.060E+00  5.941E+00  3.366E-01  5.464E+01
  3.040E+00  3.000E-02  3.070E+00  5.922E+00  3.377E-01  5.482E+01
  3.040E+00  3.000E-02  3.070E+00  5.922E+00  3.377E-01  5.482E+01
  3.034E+00  3.000E-02  3.064E+00  5.933E+00  3.371E-01  5.471E+01
```

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
```

prof interprets the type of experiment you have performed and prints results to *stdout*. The report below shows partial *prof* output:

```
------------------------------------------------------------------------
Profile listing generated Mon Nov 18 11:39:36 1996
    with:  prof linpackup.usertime.m17566
------------------------------------------------------------------------

Total Time (secs)     : 115.11
Total Samples         : 3837
Stack backtrace failed: 2
Sample interval (ms)  : 30
CPU                   : R4600
FPU                   : R4600
Clock                 : 100.0MHz
Number of CPUs        : 1

------------------------------------------------------------------------

index  %Samples  self descendents  total          name
[1]     99.9%    0.00      115.05    3835          __start
[2]     99.9%    0.00      115.05    3835          main
[3]     99.9%    0.00      115.05    3835          linp
[4]     94.9%    3.12      106.17    3643          dgefa
[5]     92.7%  106.71        0.00    3557          daxpy
[6]      3.8%    4.35        0.00     145          matgen
[7]      0.9%    0.00        1.08      36          dgesl
[8]      0.2%    0.27        0.00       9          dscal
[9]      0.2%    0.27        0.00       9          idamax
[10]     0.2%    0.00        0.18       6          s_wsfe64
[11]     0.2%    0.00        0.18       6          s_wsfe_com
[12]     0.2%    0.00        0.18       6          wsfe
[13]     0.1%    0.03        0.09       4          f_init
[14]     0.1%    0.00        0.06       2          f77canseek
[15]     0.1%    0.03        0.03       2          _isatty
[16]     0.1%    0.06        0.00       2          dmxpy
[17]     0.1%    0.03        0.03       2          s_stop
[18]     0.0%    0.03        0.00       1          _mips2_test_and_set
[19]     0.0%    0.00        0.03       1          _ftell64
[20]     0.0%    0.00        0.03       1          memset
```

```
[21]     0.0%    0.03        0.00       1          _blk_init
[22]     0.0%    0.03        0.00       1          __oserror
[23]     0.0%    0.03        0.00       1          c_sfe
[24]     0.0%    0.00        0.03       1          do_ud
[25]     0.0%    0.00        0.03       1          check_buflen
[26]     0.0%    0.00        0.03       1          _malloc
[27]     0.0%    0.00        0.03       1          __malloc
[28]     0.0%    0.03        0.00       1          _morecore
[29]     0.0%    0.00        0.03       1          pars_f
[30]     0.0%    0.00        0.03       1          f_s
[31]     0.0%    0.00        0.03       1          f_list
[32]     0.0%    0.00        0.03       1          i_tem
[33]     0.0%    0.00        0.03       1          ne_d
[34]     0.0%    0.03        0.00       1          op_gen
[35]     0.0%    0.00        0.03       1          do_fioxr8v
[36]     0.0%    0.00        0.03       1          do_fio64_mp
[37]     0.0%    0.03        0.00       1          w_ed
[38]     0.0%    0.03        0.00       1          f_exit
```

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 provides an index number for reference.
- The `%time` column shows the cumulative percentage of inclusive time spent in each function and its descendents. For example, in the third row, 99.9% of the time was spent in **linp** and all functions listed below it.
- The `self` 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 **linp**, but 3.12 seconds were spent in **dgefa**.
- The `descendents` columns shows how much time, in seconds, was spent in callees of the function. For example, in the third row, 115.05 seconds were spent in callees of **linp**.
- The `total` column provides information on the number of cycles out of the total spent on the function.

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

pcsamp Experiment

This section takes you through the steps 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 the "pcsamp Experiment" section in Chapter 4, "Experiment Types."

Performing a pcsamp Experiment

From the command line, enter

```
ssrun -pcsamp linpackup
```

This starts the experiment.

Output from *linpackup* 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 (*linpackup*), the experiment type, *pcsamp*, and the experiment ID. In this example, the filename is *linpackup.pcsamp.m17576*.

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

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
```

```
-------------------------------------------------------------------------------
Profile listing generated Sun May 19 18:38:50 1996
    with:       prof linpackup.pcsamp.m17576
-------------------------------------------------------------------------------
samples   time    CPU    FPU   Clock   N-cpu  S-interval Countsize
   5421    54s  R8000  R8010  75.0MHz   1     10.0ms     2(bytes)

Each sample covers 4 bytes for every 10.0ms ( 0.02% of 54.2100s)
-------------------------------------------------------------------------------
  -p[rocedures] using pc-sampling.
  Sorted in descending order by the number of samples in each procedure.
  Unexecuted procedures are excluded.
-------------------------------------------------------------------------------
samples   time(%)      cum time(%)      procedure (dso:file)

   5064    51s( 93.4)   51s( 93.4)          daxpy (linpackup:linpackup.f)
    240   2.4s(  4.4)   53s( 97.8)         matgen (linpackup:linpackup.f)
     76  0.76s(  1.4)   54s( 99.2)          dgefa (linpackup:linpackup.f)
     19  0.19s(  0.4)   54s( 99.6)          dscal (linpackup:linpackup.f)
     17  0.17s(  0.3)   54s( 99.9)         idamax (linpackup:linpackup.f)
      4  0.04s(  0.1)   54s(100.0)          dmxpy (linpackup:linpackup.f)
      1  0.01s(  0.0)   54s(100.0)          _ioctl (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M4/sys/ioctl.s)

   5421    54s(100.0)   54s(100.0)          TOTAL
```

Analyzing the Report

The report has the following columns:

- The `samples` column shows how many samples were taken when the process was executing in the function.
- The `time(%)` column shows the amount of time, and the percentage of that time over the total time that was spent in the function.
- The `cum time(%)` column shows how much time has been spent in the function up to and including the procedure in the list.
- The `procedure (dso:file)` column lists the procedure, its DSO name and filename. For example, the first line reports statistics for the procedure **daxpy** in the file *linpackup.f* in the *linpackup* executable.

Hardware Counter Experiment

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

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 allows you to capture information about secondary data cache misses. For more information on hardware counter experiments, see the“ideal Experiment” section in Chapter 4, “Experiment Types.”

Performing a hardware counter Experiment

From the command line, enter

```
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.m6180*.

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:

```
-------------------------------------------------------------------------------
Profile listing generated Sun May 19 18:20:14 1996
    with:       prof linpackup.dsc_hwc.m6180
-------------------------------------------------------------------------------
        Counter              : Sec cache D misses
        Counter overflow value: 131
        Total numer of ovfls  : 2737
        CPU                  : R10000
        FPU                  : R10010
        Clock                : 196.0MHz
        Number of CPUs       : 1
-------------------------------------------------------------------------------
  -p[rocedures] using counter overflow.
  Sorted in descending order by the number of overflows in each procedure.
  Unexecuted procedures are excluded.
-------------------------------------------------------------------------------
  overflows(%)  cum overflows(%)      procedure (dso:file)

   2133( 77.9)       2133( 77.9)          DAXPY (linpackup:/usr/demos/
         SpeedShop/linpack64/linpackup.f)
    307( 11.2)       2440( 89.1)         MATGEN (linpackup:/usr/demos/
         SpeedShop/linpack64/linpackup.f)
    275( 10.0)       2715( 99.2)          DGEFA (linpackup:/usr/demos/
         SpeedShop/linpack64/linpackup.f)
     11(  0.4)       2726( 99.6)         IDAMAX (linpackup:/usr/demos/
         SpeedShop/linpack64/linpackup.f)
      3(  0.1)       2729( 99.7)          DMXPY (linpackup:/usr/demos/
         SpeedShop/linpack64/linpackup.f)
      3(  0.1)       2732( 99.8)          DGESL (linpackup:/usr/demos/
         SpeedShop/linpack64/linpackup.f)
      1(  0.0)       2733( 99.9)         memset (/usr/lib64/libc.so.1:
         /work/irix/lib/libc/libc_64_M4/strings/bzero.s)
      1(  0.0)       2734( 99.9)         fflush (/usr/lib64/libc.so.1:
         /work/irix/lib/libc/libc_64_M4/stdio/flush.c)
      1(  0.0)       2735( 99.9)    _mixed_dtoa (/usr/lib64/libc.so.1:
         /work/irix/lib/libc/libc_64_M4/math/mixed_dtoa.c)
```

```
    1(  0.0)       2736(100.0)           wsfe (/usr/lib64/libftn.so:
        /work/cmplrs/libI77/wsfe.c)
    1(  0.0)       2737(100.0)         f_exit (/usr/lib64/libftn.so:
        /work/cmplrs/libI77/close.c)

2737(100.0)                        TOTAL
```

Analyzing the Report

The report has the following columns:

- The `overflows(%)` column shows the number of overflows caused by the function, and the percentage of that number over the total number of overflows in the program.
- The `cum overflows(%)` column shows a cumulative number and percentage of overflows. For example, the **MATGEN** function shows 307 overflows, but the cumulative number of overflows is 2440.
- The `procedure (dso:file)` column shows the procedure name and the DSO and filename that contain the procedure.

ideal Experiment

This section takes you through the steps to perform an **ideal** experiment. For more information on collecting ideal-time data, and basic block counting, see the "ideal Experiment" section in Chapter 4, "Experiment Types."

Performing an ideal Experiment

From the command line, enter

```
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 example below. 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 filename is *linpackup.ideal.n17580*.

```
Beginning libraries
        /usr/lib32/libssrt.so
        /usr/lib32/libss.so
        /usr/lib32/libftn.so
        /usr/lib32/libm.so
        /usr/lib32/libc.so.1
Ending libraries, beginning "linpackup"
...
```

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 should contain results that look something like the following:

```
Prof run at: Sun May 19 18:46:10 1996
 Command line: prof linpackup.ideal.m17580

         5722510379: Total number of cycles
          76.30014s: Total execution time
         4906763725: Total number of instructions executed
              1.166: Ratio of cycles / instruction
                 75: Clock rate in MHz
              R8000: Target processor modelled
------------------------------------------------------------
Procedures sorted in descending order of cycles executed.
Unexecuted procedures are not listed. Procedures
beginning with *DF* are dummy functions and represent
init, fini and stub sections.
------------------------------------------------------------
        cycles(%)  cum %     secs    instrns    calls procedure(dso:file)

5404032607(94.43)  94.43    72.05 4639092022 772633 daxpy(linpackup.pixie:
          linpackup.f)
 207582228( 3.63)  98.06     2.77  157405518    18 matgen(linpackup.pixie:
          linpackup.f)
```

```
67844858( 1.19)  99.25     0.90   72325769    17 dgefa(linpackup.pixie:
        linpackup.f)
19920277( 0.35)  99.60     0.27   17658342  5083 dscal(linpackup.pixie:
        linpackup.f)
18115251( 0.32)  99.91     0.24   15675343  5083 idamax(linpackup.pixie:
        linpackup.f)
 4053920( 0.07)  99.98     0.05    3605124     1 dmxpy(linpackup.pixie:
        linpackup.f)
  786709( 0.01) 100.00     0.01     695776    17 dgesl(linpackup.pixie:
        linpackup.f)
   41357( 0.00) 100.00     0.00      83826  1116 __flsbuf(./libc.so.1.pixn32:
        /work/irix/lib/libc/libc_n32_M4/stdio/_flsbuf.c)
   30294( 0.00) 100.00     0.00      29094     1 linp(linpackup.pixie:
        linpackup.f)
   17330( 0.00) 100.00     0.00      39823   867 x_putc(./libftn.so.pixn32:
        /lv7/mtibuild/nodebug/workarea/mongoose/libI77/wsfe.c)
   12294( 0.00) 100.00     0.00      25795    28 x_wEND(./libftn.so.pixn32:
        /lv7/mtibuild/nodebug/workarea/mongoose/libI77/wsfe.c)
   10620( 0.00) 100.00     0.00      14340    53 wrt_E(./libftn.so.pixn32:
        /lv7/mtibuild/nodebug/workarea/mongoose/libI77/wrtfmt.c)
    9617( 0.00) 100.00     0.00      14889    71 do_fio64_mp
        (./libftn.so.pixn32:/lv7/mtibuild/nodebug/workarea/mongoose/libI77/
        fmt.c)
    4940( 0.00) 100.00     0.00       7917   380
```

Analyzing the Report

The report has the following columns:

- The `cycles(%)` column reports the number and percentage of machine cycles used for the procedure. For example, 5404032607 cycles, or 94.43% of cycles were spent in the **daxpy** procedure.
- The `cum%` column shows the cumulative percentage of cycles. For example, `98.06%` of all cycles were spent between the top two functions in the listing: **daxpy** and **matgen**.
- The `secs` column shows the number of seconds spent in the procedure. For example, 72.05 seconds were spent in the **daxpy** procedure. The time represents an idealized computation based on modelling the machine. It ignores potential floating point interlocks and memory latency time (cache misses and memory bus contention.)
- The `instrns` column shows the number of instructions executed for a procedure. For example, there were `157405518` instructions devoted to the **matgen** procedure.
- The `calls` column reports the number of calls to the procedure. For example, there were 18 calls to the **matgen** procedure.
- The `procedure (dso:file)` column lists the procedure, its DSO name and filename. For example, the first line reports statistics for the procedure **daxpy** in the file *linpackup.f* in the *linpackup* executable.

This concludes the tutorial.

Chapter 4

Experiment Types

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

- “Selecting an Experiment”
- “usertime Experiment”
- “pcsamp Experiment”
- “ideal Experiment”
- “Hardware Counter Experiments”
- “fpe Trace”

For information on how to run the experiments described in this chapter, see Chapter 6, “Setting Up and Running Experiments: ssrun.”

Selecting an Experiment

Table 4-1 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 listed, see the sections listed in Table 4-1.

Table 4-1 Summary of Experiments

Experiment	Clues	Data Collected
"usertime Experiment"	Slow program, nothing else known. Not CPU-bound.	Inclusive and exclusive user time for each function by sampling the callstack at 30-millisecond intervals.
"pcsamp Experiment"	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.
"ideal Experiment"	CPU-bound.	Ideal CPU time at the function, source line and machine instruction levels using instrumentation for basic block counting.
"ideal Experiment"	High user CPU time.	On R10000 class machines, exclusive counts at the source line, machine instruction, and function levels for overflows of the following counters: clock cycle, graduated instructions, primary instruction-cache misses, secondary instruction-cache misses, primary data-cache misses, secondary data-cache misses, TLB misses, graduated floating-point instructions.
"fpe Trace"	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.

usertime Experiment

The **usertime** experiment uses statistical call stack profiling, based on wall clock time, to measure inclusive and exclusive user time spent in each function while your program runs. This experiment uses an interval of 30 milliseconds.

Data is measured by periodically sampling the callstack. The program's callstack data is used to

- 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

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 whether the program was running or blocked; hence, the time computed represents the total time, both within and outside the CPU. If the target process was blocked for a long time as a result of an instruction, that instruction will show up as having a high time.

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**.

pcsamp Experiment

The **pcsamp** experiment uses statistical PC sampling to estimate actual CPU time for each source code line, machine 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 done with kernel support. The actual CPU time is calculated by multiplying the number of times an instruction appears in the PC by the interval specified for the experiment (either 1 or 10 milliseconds.)

To collect the data, the kernel regularly stops the process if it is in the CPU, increments a counter for 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 bins, based on user and system time. If the optional **x** suffix is used, a 32-bit bin size will be used. Using a 32-bit bin provides more accurate information, but requires additional disk space.

- 16-bit bins allow a maximum of 65,000 counts.
- 32-bit bins allow approximately 4,000,000 counts.

PC-sampling time runs should incur a slowdown of execution of the program of no more than 5%. The measurements are statistical in nature, and exhibit variance inversely proportional to the running time.

ideal Experiment

The **ideal** experiment instruments the executables and any DSOs to permit basic block counting and counting of all dynamic (function-pointer) calls.

How SpeedShop Prepares Files

To permit block counting, SpeedShop

- 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 and all the DSOs it uses are instrumented, including *libc.so.1*, *libexc.so*, *libm.so*, *libss.so*, *libssrt.so*. Instrumented files with an extension *.pix**, where * depends on the ABI, are written to the current working directory.

After the transformations are complete, the program's symbol table and translation table are updated so that debuggers can map between transformed addresses and the original program's addresses, and reference the measured performance data to the untransformed code.

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.

How SpeedShop Calculates CPU Time

prof uses a machine model to convert the block execution counts into an idealized exclusive 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.

SpeedShop calculates ideal CPU time by using the TDT models. Potential floating-point interlocks are taken into account inside the same basic blocks, but ignored across basic block boundaries. Memory latency time (cache misses and memory bus contention) is ignored. The computed ideal time is therefore always less than the real time that any run would take. See Table 4-2 for a comparison of running a **pcsamp** experiment, which generates estimated actual CPU time, and running an **ideal** experiment.

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 machine line levels.

Inclusive Basic Block Counting

The basic block counting explained in the previous section allows you to measure ideal time spent in each procedure, but doesn't 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 proportionately to all 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()** ends up with all the cycles counted in the program. for example), the assumption can be *very misleading*. If **foo()** calls **matmult()** 99 times for 2X2 matrices, while **bar()** calls it once for 100X100 matrices, the inclusive time report will attribute 99% of **matmult()**'s time to **foo()**, but actually almost all the time derives from **bar()**'s one call.

To generate a report that shows inclusive time, specify the **-gprof** flag to *prof*.

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

- 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 4-2.

Table 4-2 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 instrumenting	Data collection done with the kernel
Slows program down by factor of three	Has minimal impact on program speed
Generates an exact count	Generates statistical counts

Hardware Counter Experiments

The experiments described in this section are available for systems that have hardware counters (R10000 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 each processor clock cycle when events occur for which there are hardware counters. For example, when a floating point instruction is graduated in a cycle, the graduated floating-point instruction counter is incremented by 1.

Two Tools for Hardware Counter Experiments

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

- *perfex* is command-line interface that provides program-level event information, For more information on *perfex*, and on hardware counters, see the perfex or r10k_counters reference pages.
- SpeedShop allows you to perform the hardware counter experiments described in the next section (“SpeedShop Hardware Counter Experiments”).

SpeedShop Hardware Counter Experiments

In 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. The generated reports show exclusive hardware counts, that is, information about where the program counter was, not 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 [f]gi_hwc, [f]cy_hwc, [f]ic_hwc, [f]isc_hwc, [f]dc_hwc, [f]dsc_hwc, [f]tlb_hwc, [f]gfp_hwc, and prof_hwc.

[f]gi_hwc

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 overflows of the counter at an overflow interval of 32771. If the optional **f** prefix is used, the overflow interval is 6553.

[f]cy_hwc

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 overflows of the counter, at an overflow interval of 16411. If the optional **f** prefix is used, the overflow interval is 3779.

[f]ic_hwc

The **[f]ic_hwc** experiment counts overflows of the primary instruction-cache miss counter. The primary instruction-cache miss 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 the overflow of the counter at an overflow interval of 2053. If the optional **f** prefix is used, the overflow interval is 419.

[f]isc_hwc

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 the overflow of the counter at an overflow interval of 131. If the optional **f** prefix is used, the overflow interval is 29.

[f]dc_hwc

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 the overflow of the counter at an overflow interval of 2053. If the optional **f** prefix is used, the overflow interval is 419.

[f]dsc_hwc

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 the overflow of the counter at an overflow interval of 131. If the optional **f** prefix is used, the overflow interval is 29.

[f]tlb_hwc

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

[f]gfp_hwc

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 which graduated on the previous cycle. The experiment uses statistical PC sampling based on overflows of the counter, at an overflow interval of 32771. If the optional **f** prefix is used, the overflow interval is 6553.

prof_hwc

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 may be any number between 0 and 31. Hardware counters are described in the *MIPS R10000 Microprocessor User's Manual*, Chapter 14, and on the r10k_counters reference page. The hardware counter numbers are provided in Table 4-3.

_SPEEDSHOP_HWC_COUNTER_OVERFLOW
: The value of this variable may 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 2053.

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.

Hardware Counter Numbers

The possible numeric values for the _SPEEDSHOP_HWC_COUNTER_NUMBER variable are shown in Table 4-3.

Table 4-3 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

Table 4-3 (continued) Hardware Counter Numbers

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

fpe 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 of execution of the program of no more than 15%. These measurements are exact, not statistical.

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

Chapter 5

Collecting Data on Machine Resource Usage

This chapter describes how to collect machine resource usage data using the SpeedShop *ssusage* 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 Table 1-4 to identify which experiments to run, based on the results of running *ssusage* on your program.

ssusage Syntax

`ssusage` *prog_name* [*prog_args*]

prog_name	Name of the executable for which you want to collect machine resource usage data.
prog_args	Arguments to your executable, if any.

ssusage Results

ssusage prints output to *stderr*. For example, the command

```
ssusage generic
```

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 is the machine resource usage information that *ssusage* provides. Each field in the report is described below.

real	Elapsed time during the command, in seconds.
user	User CPU time in seconds.
sys	System CPU time 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/written. Note that these are attributed to the process that first requests a block, but do not necessarily directly correlate with the process' own I/O operations.
vcx	Voluntary context switches, that is, those caused by the process' own actions.
icx	Involuntary context switches, that is, those caused by the scheduler.

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

Chapter 6

Setting Up and Running Experiments: ssrun

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"
- "Setting Up Output Directories and Files"
- "Using Runtime Environment Variables"
- "Running Experiments"
- "Running Experiments on MPI Programs"
- "Running Experiments on Programs Using Pthreads"
- "Using Calipers"
- "Effects of ssrun"

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 "Using Runtime Environment Variables."

- If you have used the **ssrt_caliper_point()** function provided in the SpeedShop libraries, you have to explicitly link in the SpeedShop libraries *libss.so* and *libssrt.so*. For more information on setting caliper points, see "Using Calipers."
- 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 reference page.
- If you have built a stripped executable, you need to rebuild a non-stripped version to use with SpeedShop. For example, if you are using *ld* to link your C program, do not use the **-s** option because this 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, the procedure inlining the optimization performs can result in extremely misleading profiles since 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's generally better to use compiler optimization level 2 or less when gathering an execution profile.

Special Information for MP Fortran Programs

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

- For all FORTRAN 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.

  – 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 a 32-bit version, the DAXPY routine is named "daxpy_," and the MP routine "_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, spurious annotations will appear on these other 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.

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:

prog_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 6-1 for letter codes and descriptions.

Table 6-1 Letter Codes in Experiment ID Numbers

Letter code	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()**

In a single-process application, *ssrun* generates a single performance data file. In a multi-process application, *ssrun* generates a performance data file for each process.

You can change the default filename or directory for performance data files using environment variables. See _SPEEDSHOP_OUTPUT_DIRECTORY and _SPEEDSHOP_OUTPUT_FILENAME in Table 6-2 for more information.

Using Runtime Environment Variables

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

- "User Environment Variables"
- "Process Tracking Environment Variables"
- "Expert-Mode Environment Variables"

User Environment Variables

A number of environment variables are normally used to control the operation of SpeedShop. Table 6-2 lists these variables.

Table 6-2 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.

Table 6-2 (continued) General Environment Variables

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 may 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_DIRECTORY is also specified, it is prepended to the filename you specify.

Process Tracking Environment Variables

A number of environment variables may be used for controlling the treatment of processes spawned from the original target. Table 6-3 lists these variables.

Table 6-3 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 don't 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. **Note:** In the current release, data are recorded independent of whether the process calls **exec()** or not.
_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.

Expert-Mode Environment Variables

A number of variables may be used for debugging and finer control of the operation of SpeedShop. Table 6-4 lists these variables.

Table 6-4 Expert-Mode Environment Variables

Variable	Description
_SPEEDSHOP_SAMPLING_MODE	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 different from 1, data is generated for the executable and all DSOs it uses.
_SPEEDSHOP_INIT_DEFERRED_SIG *sig_num*	If specified, initialization of the experiment is not performed when the target process starts, but will be 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 will not be performed when the target process exits, but rather will happen when the specified signal is sent to the process. A handler for the given signal will be 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 runtime. It is normally set by *ssrun*, but may be overwritten.
_SPEEDSHOP_MARCHING_ORDERS	Passes the marching orders of the experiment to the runtime. It is normally set by *ssrun* from the experiment type, but may be overwritten.
_SPEEDSHOP_SBRK_BUFFER_LENGTH	Defines the maximum size of the internal malloc arena used. This arena is completely separate from the user's arena, and has a default size of 0x100000.

Table 6-4 (continued) Expert-Mode Environment Variables

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 done in usertime experiments, and as is done at caliper-samples for all experiments. The option is used as a workaround for various unwind bugs in *libexc*.

Running Experiments

This section describes how to use *ssrun* to perform experiments. For information on using *pixie* directly, see Chapter 8, "Using SpeedShop in Expert Mode: pixie."

ssrun Syntax

`ssrun` *flags -exp_type prog_name prog_args*

flags Zero or more of the flags described in Table 6-5 that control the data collection and the treatment of descendent processes or programs, and how the data is to be externalized.

-exp_type The experiment type. Experiments are described in detail in Chapter 4, "Experiment Types."

prog_name The name of the program on which you want to run an experiment.

args Arguments to your program, if any.

ssrun generates a performance data file that is named as described in the section "Building Your Executable."

Table 6-5 Flags for ssrun

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*.
-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.
-v	Prints a log of the operation of *ssrun* to *stderr*. The same behavior occurs if the environment variable _SPEEDSHOP_VERBOSE is set a 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 a to a non-zero-length string. This option can be used to see how to set the various environment variables, and how to invoke instrumentation when necessary.

ssrun Examples

This section provides examples of using *ssrun* with options and experiment types. For additional examples, see Chapter 2, "Tutorial for C Users," or Chapter 3, "Tutorial for Fortran Users."

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 the "pcsamp Experiment" section in Chapter 4, "Experiment Types."

This 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.16064*:

```
prof generic.pcsampx.m16064
```

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, "Analyzing Experiment Results: prof."

```
------------------------------------------------------------------------------
Profile listing generated Thu May 23 10:30:40 1996
    with:       prof generic.pcsampx.m16064
------------------------------------------------------------------------------
samples   time    CPU    FPU   Clock   N-cpu  S-interval Countsize
   2058    21s  R4000  R4010 150.0MHz   1     10.0ms     4(bytes)

Each sample covers 4 bytes for every 10.0ms ( 0.05% of 20.5800s)
------------------------------------------------------------------------------
  -p[rocedures] using pc-sampling.
  Sorted in descending order by the number of samples in each procedure.
  Unexecuted procedures are excluded.
------------------------------------------------------------------------------
samples   time(%)      cum time(%)      procedure (dso:file)

1926      19s( 93.6)   19s( 93.6)         anneal (generic:/usr/demos/
          SpeedShop/generic/generic.c)
 111       1.1s(  5.4)   20s( 99.0)   slaveusrtime (/usr/demos/SpeedShop/
          generic/dlslave.so:/usr/demos/SpeedShop/generic/dlslave.c)
  15      0.15s(  0.7)   21s( 99.7)          _read (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/sys/read.s)
   2      0.02s(  0.1)   21s( 99.8)         memcpy (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/strings/bcopy.s)
   1      0.01s(  0.0)   21s( 99.9)         _xstat (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/sys/xstat.s)
   1      0.01s(  0.0)   21s( 99.9)        _ltzset (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/gen/time_comm.c)
   1      0.01s(  0.0)   21s(100.0)         __sinf (/usr/lib32/libm.so:
          /work/cmplrs/libm/fsin.c)
   1      0.01s(  0.0)   21s(100.0)         _write (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/sys/write.s)

2058        21s(100.0)   21s(100.0)          TOTAL
```

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*.

This example performs a **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
...
```

Using ssrun With a Debugger

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

Follow these steps to run the FPE trace experiment on *generic*, and then run *generic* in a debugger.

1. Call *ssrun* as follows:

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

   ssrun 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. Get the process ID of the call to *ssrun* using a command such as *ps*.
3. Start your debugging session.
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 either *dbx* or the WorkShop debugger to set calipers to record performance data for a part of your program. See "Using Calipers" for more information on setting calipers.

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 "mailing" data "messages" to other processes, even those running on distant computers.

If your program uses the MPI, you need to set up SpeedShop experiments 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, a script, named *exp_script*, might look like the following:

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

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

   ```
   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 rather from a master program that spawns the actual MPI application slaves. 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 NFS mounted directory. 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.

Running Experiments on Programs Using Pthreads

Pthreads are the threads 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 (one for each o32, n32 and n64 ABI) 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 sprocs 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.

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 setting caliper points to identify the area or areas 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 "Using the -calipers Option" in Chapter 7, "Analyzing Experiment Results: prof."

Table 6-6 shows how you can set caliper points in three different ways.

Table 6-6 Setting Caliper Points

Use This Approach...	For These Benefits...
Explicitly link with the SpeedShop runtime and call **ssrt_caliper_point** to record a caliper sample.	Allows you to set a caliper point at a specific location in a file.
Define a signal to be used to record 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're 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 recorded 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 recorded 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 a program closes a DSO by calling **dlclose()**.
- When an exit signal is received, such as SIGHUP, SIGINT, SIGPIPE, SIGALRM, SIGTERM, SIGUSR1, SIGUSR2, SIGPOLL, SIGIO, SIGRTMIN or SIGRTMAX.

Setting Calipers With ssrt_caliper_point

To set calipers with *ssrt_caliper_point*, follow these steps:

1. Insert calls to **ssrt_caliper_point()** in your source code. Call the function with the argument 1 (True).

   ```
   ...
   ssrt_caliper_point(1);
   ...
   ```

   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*.

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 doesn't terminate the program. The signal should also not be caught by the target program, because this would interfere with its use for triggering a caliper point.

   The following signals are good choices because they don't have any 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 runtime.

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: function entry or exit, line numbers, execution addresses, watchpoints, pollpoints (timer-based). You can also attach conditions and/or cycle counts.

1. Set a breakpoint in your program at the point at which you want to set a caliper point.
2. When the process stops, evaluate the expression *libss_caliper_point(1)*.

   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.

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 variables, see "Using Runtime Environment Variables."
- 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 runtime library writes the appropriate experiment data to the output file.

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* generates pixified versions of all the libraries that the program uses, as well as the executable.

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

  – *.pixie* for the executable

  – *.pix32* for all 32 libraries

  – *.pixn32* for all n32 libraries

  – *.pix64* for all 64 libraries

  The generated files are written to the current working directory, and include code that allows performance data to be collected for each function and basic block.

  For more information on the *ideal* experiment, see the "ideal Experiment" section in Chapter 4, "Experiment Types."

Chapter 7

Analyzing Experiment Results: prof

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

- “Using prof to Generate Performance Reports”
- “Using prof With ssrun”
- “Using prof Options”
- “Generating Reports for Different Machine Types”
- “Generating Reports for Multiprocessed Executables”
- “Generating Compiler Feedback Files”
- “Interpreting Reports”

Using prof to Generate Performance Reports

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

The *prof* command can be used in two modes:

- To generate a report from performance data gathered during experiments recorded by *ssrun:*

  ```
  prof <options> <perf-data-file> <perf-data-file> ...
  ```

- To generate a report from data files produced by running a program that has been instrumented by *pixie*:

  ```
  prof executable_name [options] [pixie counts file]
  ```

  This chapter focuses on the use of *prof* to generate reports from *ssrun* experiments. For information on *prof* for a *pixie* experiment, see Chapter 8, "Using SpeedShop in Expert Mode: pixie."

prof Syntax

The syntax for *prof* when using it with data files from *ssrun* is:

`prof` *options data_file data_file* `...`

options — Zero or more of the options described in Table 7-2.

data_file — One or more names of performance data files generated by *ssrun*. These files are usually of the format *prog_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 7-1 for letter codes and descriptions.

Table 7-1 lists the letter codes for *id*.

Table 7-1 Letter Codes in Experiment ID Numbers

Letter Code	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()**

prof Options

Table 7-2 lists *prof* options. For more information, see the prof reference page.

Table 7-2 Options for prof

Name	Result
-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.
–c[**lock**] *n*	Lists the number of seconds spent in each routine, based on the CPU clock frequency *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 to use the clock frequency of the machine where the performance data was collected.
-cycle *n*	Sets the cycle time to *n* nanoseconds.

Table 7-2 (continued) Options for prof

Name	Result
-den[sity]	Prints a list of procedures with non-zero instruction cycles sorted by the instruction density, which is the number of cycles per instruction. This option can be used when generating reports for **ideal** experiments, or for basic block counting data obtained with *pixie*.

-debug:*dbg_flags*	*dbg_flags* can be 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.
–dso [*dso_name*]	Generates a report only for the named DSO. If you don't specify *dso_name*, *prof* prints a list of applicable DSO names. Only the basename of the DSO needs to be specified.
-dsolist	List all the DSOs in the program and their start and end text addresses.
–e[xclude] *proc1...procN*	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.

Table 7-2 (continued) Options for prof

Name	Result
–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 *progam.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.
–gprof	Calculates cycles and propagates basic block counting to a procedure's callers proportionately. This option can be used when generating reports for **ideal** experiments, or for basic block counting data obtained with *pixie*. It can also be used for **fpe** and **usertime** experiments.
–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*.
–i[**nvocations**]	Lists the number of times each procedure is invoked. This option can be used when generating reports for **ideal** experiments, or for basic block counting data obtained with *pixie*.
–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*.
-nocounts	Analyzes an executable or a *.o* file using the pixie machine model, and assuming each instruction is executed once. This analysis cannot match any possible real run of any executable which contains one or more conditional branch instructions.
-nofilenames	Removes *a.out*, DSO, and source filenames from the listing; useful for scripted analysis of *prof* output.

Table 7-2 (continued) Options for prof

Name	Result
–o[**nly**] *proc1...procN*	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.
–p[**rocedures**]	Lists the time spent in each procedure.
-pthreads *pthrd1... pthrdN*	Analyzes data only for the specified pthreads (for **usertime** and **fpe** experiments on applications that use pthreads (on Irix 6.5 or later)).
–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; *n***%**, an integer followed by a percent sign, does not print any procedure or line with less than *n* in the % column; *ncum*%, 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%.
-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.
-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.
–z[**ero**]	Lists the procedures that are never invoked. Use this option when generating reports for **ideal** experiments, or for basic block counting data obtained with *pixie*.

prof Output

prof 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/subroutine, since there is no general way for *prof* to separate the times for the alternate entries.

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 depending on the experiment type. 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.

prof 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** and **fini** sections, and **MIPS.stubs** sections, for example.

The types of reports that **prof** generates are described in the following sections:

- "usertime Experiment Reports"
- "pcsamp Experiment Reports"
- "Hardware Counter Experiment Reports"
- "ideal Experiment Reports"
- "FPE Trace Reports"

usertime Experiment Reports

For **usertime** experiments, *prof* generates a list of callers and callees of each function, with information on how much time was spent in the function, its callers and its callees.

The report shows information for each function, its callers and its callees. The function names are show in the right-hand column of the report. The function that is being reported is shown outdented from its caller and callee(s). For example, the first function shown in this report is **__start()**, which has no callers and two callees. The remaining columns are described below:

- The `index` column provides an index number for reference.
- The `%samples` column shows the cumulative percentage of time spent in each function.
- The `self` column shows how much time, in seconds, was spent in the function.
- The `descendents` columns shows how much time, in seconds, was spent in callees of the function.
- The `total` column provides information on the number of samples of the function.

This example is a truncated version of the full report. For a complete report see "Generating a Report" on page 20.

```
------------------------------------------------------------------------
Profile listing generated Mon Nov 18 11:43:45 1996
    with: prof generic.usertime.m24479
------------------------------------------------------------------------

Total Time (secs)     : 43.98
Total Samples         : 1466
Stack backtrace failed: 1
Sample interval (ms)  : 30
CPU                   : R4600
FPU                   : R4600
Clock                 : 100.0MHz
Number of CPUs        : 1

------------------------------------------------------------------------

index  %Samples  self descendents  total     name
[1]     99.9%    0.00       43.95    1465    __start
```

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 `samples` column shows how many samples of the function were taken.
- The `time(%)` column shows the amount of time, and the percentage of that time over the total time that was spent in the function.
- The `cum time(%)` column shows how much time has been spent up to and including the procedure being examined.
- The `procedure (dso:file)` column lists the procedure, its DSO name and file name. For example, the first line reports statistics for the procedure **anneal** in the file *generic.c* in the *generic* executable.

```
------------------------------------------------------------------------------
Profile listing generated Sun May 19 17:21:27 1996
    with:       prof generic.fpcsamp.m14480
------------------------------------------------------------------------------
samples   time    CPU    FPU   Clock   N-cpu  S-interval Countsize
  19077    19s  R4000  R4010 150.0MHz   1      1.0ms     2(bytes)

Each sample covers 4 bytes for every  1.0ms ( 0.01% of 19.0770s)
------------------------------------------------------------------------------
  -p[rocedures] using pc-sampling.
  Sorted in descending order by the number of samples in each procedure.
  Unexecuted procedures are excluded.
------------------------------------------------------------------------------
samples   time(%)      cum time(%)      procedure (dso:file)

  17794    18s( 93.3)   18s( 93.3)         anneal (/usr/demos/SpeedShop/
          generic/generic:/usr/demos/Speedshop/generic/generic.c)
```

Hardware Counter Experiment Reports

For the various **hwc** experiments, *prof* generates a function list annotated with the number of overflows generated by the function.

- The `overflows(%)` column shows the number of overflows caused by the function, and the percentage of that number over the total number of overflows in the program.
- The `cum overflows(%)` column shows a cumulative number and percentage of overflows.
- The `procedure (dso:file)` column shows the procedure name and the DSO and filename that contain the procedure.

```
-------------------------------------------------------------------------------
Profile listing generated Sun May 19 17:35:21 1996
    with:       prof generic.dsc_hwc.m5999
-------------------------------------------------------------------------------
        Counter               : Sec cache D misses
        Counter overflow value: 131
        Total numer of ovfls  : 10
        CPU                   : R10000
        FPU                   : R10010
        Clock                 : 196.0MHz
        Number of CPUs        : 1
-------------------------------------------------------------------------------
  -p[rocedures] using counter overflow.
  Sorted in descending order by the number of overflows in each procedure.
  Unexecuted procedures are excluded.
-------------------------------------------------------------------------------
  overflows(%)  cum overflows(%)      procedure (dso:file)

      4( 40.0)          4( 40.0)         memcpy (/usr/lib64/libc.so.1:
          /work/irix/lib/libc/libc_64_M4/strings/bcopy.s)
```

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.

prof 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 `cycles(%)` column reports the number and percentage of machine cycles used for the procedure. For example, 2524610038 cycles, or 94.81% of cycles were spent in the **anneal()** procedure.
- The `cum%` column shows the cumulative percentage of calls. For example, 99.88% of all calls were spent between the top two functions in the listing: **anneal()** and **slaveusrtime()**.
- The `secs` column shows the number of seconds spent in the procedure. For example, 16.83 seconds were spent in the **anneal()** procedure. The time represents an idealized computation based on modelling the machine. Potential floating-point interlocks and memory latency time (cache misses and memory bus contention) are ignored.
- The `instrns` column shows the number of instructions executed for a procedure. For example, there were 1797940023 instructions devoted to the **anneal()** procedure.
- The `calls` column reports the number of calls to the procedure. For example, there was just one call to the **anneal()** procedure.
- The `procedure (dso:file)` column lists the procedure, its DSO name and filename. For example, the first line reports statistics for the procedure **anneal()** in the file *generic.c* in the *generic* executable.

```
Prof run at: Sun May 19 17:49:10 1996
 Command line: prof generic.ideal.m14517

         2662778531: Total number of cycles
          17.75186s: Total execution time
         1875323907: Total number of instructions executed
              1.420: Ratio of cycles / instruction
                150: Clock rate in MHz
             R4000: Target processor modelled
-------------------------------------------------------
Procedures sorted in descending order of cycles executed.
Unexecuted procedures are not listed. Procedures
beginning with *DF* are dummy functions and represent
init, fini and stub sections.
-------------------------------------------------------

        cycles(%)  cum %     secs    instrns    calls procedure(dso:file)

2524610038(94.81)  94.81    16.83 1797940023     1 anneal(generic:/usr/demos/
          SpeedShop/generic/generic.c)
```

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

```
                        self             kids        called/total    parents
index     cycles(%)     self(%)          kids(%)  called+self     name     index
                        self             kids        called/total    children

[1] 2661528037(99.95%)    71( 0.00%) 2661527966(100.00%)   0     __start [1]
                          44         2661527913          1/1       main [2]
                           5                  0          1/1       __istart [107]
                           4                  0         1/1
__readenv_sigfpe [108]


-------------------------------------------------------------------------------
                          44         2661527913         1/1       __start [1][2]
2661527957(99.95%)        44( 0.00%) 2661527913(100.00%)  1     main [2]
                        2152         2661524760         1/1       Scriptstring[3]
                          67                934         1/1       exit [55]
-------------------------------------------------------------------------------
                        2152         2661524760         1/1       main [2]
[3] 2661526912(99.95%)  2152( 0.00%) 2661524760(100.00%)  1      Scriptstring [3]
                          40         2525080081         1/1         usrtime [4]
                          82          135044460         1/1         libdso [6]
                       68058            1148856         1/2         iofile [10]
                         124              52933         2/8         genLog [16]
                        7211              45001         1/1         dirstat [27]
                        1438              32051         1/1         linklist [31]
                         632              32051         1/1         fpetraps [32]
                         124              10922         2/19        fprintf [20]
                         696                  0        45/45        strcmp [61]
-------------------------------------------------------------------------------
                          40         2525080081         1/1       Scriptstring[3]
[4] 2525080121(94.83%)    40( 0.00%) 2525080081(100.00%)  1     usrtime [4]
                    2524610038            437992         1/1         anneal [5]
                          62              26466         1/8         genLog [16]
                          62               5461         1/19        fprintf [20]
```

FPE Trace Reports

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

- The `index` column provides an index number for reference.
- The `%FPEs` column shows the percentage of the total number of floating point exceptions that were found in the function.
- The `self` column shows how many floating point exceptions were found in the function. For example, 0 floating point exceptions were found in **__start()**.
- The `descendents` columns shows how many floating point exceptions were found in the descendents of the function.
- The `totals` column provides information on the number of floating point exceptions out of the total that were found.

```
-------------------------------------------------------------------------
Profile listing generated Mon Nov 18 11:46:33 1996
    with:      prof generic.fpe.m18823
-------------------------------------------------------------------------
Total FPEs            : 4
Stack backtrace failed: 0
CPU                   : R4600
FPU                   : R4600
Clock                 : 100.0MHz
Number of CPUs        : 1

-------------------------------------------------------------------------
index  %FPEs     self descendents  total         name
[1]    100.0%       0           4       4   __start
```

Using prof Options

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

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 example below shows partial output from *prof*, for a **pcsamp** experiment.

```
------------------------------------------------------------------------------
Profile listing generated Tue May 27 18:04:10 1997
    with:     prof -dis generic.pcsamp.m875
------------------------------------------------------------------------------

samples   time    CPU    FPU   Clock   N-cpu  S-interval Countsize
   4142    41s  R4600  R4600 100.0MHz   1     10.0ms     2(bytes)

Each sample covers 4 bytes for every 10.0ms ( 0.02% of 41.4200s)

------------------------------------------------------------------------------
  -p[rocedures] using pc-sampling.
  Sorted in descending order by the number of samples in each procedure.
  Unexecuted or inlined procedures are excluded.
------------------------------------------------------------------------------

samples   time(%)      cum time(%)      procedure (dso:file)

   3975    40s( 96.0)   40s( 96.0)         anneal (generic:generic.c)
    124   1.2s(  3.0)   41s( 99.0)   slaveusrtime (./dlslave.so:dlslave.c)
     32  0.32s(  0.8)   41s( 99.7)          _read (/usr/lib32/libc.so.1:/xlv1/
         bonsai-sep09/work/irix/lib/libc/libc_n32_M3/sys/read.s)
      4  0.04s(  0.1)   41s( 99.8)         _xstat (/usr/lib32/libc.so.1:/xlv1/
         bonsai-sep09/work/irix/lib/libc/libc_n32_M3/sys/xstat.s)
      2  0.02s(  0.0)   41s( 99.9)          fread (/usr/lib32/libc.so.1:/xlv1/
         bonsai-sep09/work/irix/lib/libc/libc_n32_M3/stdio/fread.c)
      1  0.01s(  0.0)   41s( 99.9)         iofile (generic:generic.c)
      1  0.01s(  0.0)   41s( 99.9)        usrtime (generic:generic.c)
      1  0.01s(  0.0)   41s(100.0)         _write (/usr/lib32/libc.so.1:/xlv1/
         bonsai-sep09/work/irix/lib/libc/libc_n32_M3/sys/write.s
```

```
     1  0.01s(  0.0)   41s(100.0)      _morecore (/usr/lib32/libc.so.1:/xlv1/
         bonsai-sep09/work/irix/lib/libc/libc_n32_M3/gen/malloc.c)
     1  0.01s(  0.0)   41s(100.0)          next (/usr/lib32/libc.so.1:/xlv1
         /bonsai-sep09/work/irix/lib/libc/libc_n32_M3/math/drand48.c)

   4142    41s(100.0)   41s(100.0)          TOTAL
-------------------------------------------------------------------------------
* -dis[assemble] listing annotated pc-samples                                  *
*     Procedures with zero samples are excluded.                               *
-------------------------------------------------------------------------------

...

generic.c
anneal: <0x100065b8-0x100068c4>    3975 total samples(95.97%)
  [1514] 0x100065b8     0x27bdffd0     addiu     sp,sp,-48           # 1
  [1514] 0x100065bc     0xffbf0018     sd        ra,24(sp)           # 2
  [1514] 0x100065c0     0xffbc0020     sd        gp,32(sp)           # 3
  [1514] 0x100065c4     0x3c030002     lui       v1,0x2              # 4
  [1514] 0x100065c8     0x246399e0     addiu     v1,v1,-26144        # 5
  [1514] 0x100065cc     0x0323e021     addu      gp,t9,v1            # 6
  [1516] 0x100065d0     0xd7808040     ldc1      $f0,-32704(gp)      # 7
         <2 cycle stall for following instruction>
  [1516] 0x100065d4     0xf7a00000     sdc1      $f0,0(sp)           # 10
  [1518] 0x100065d8     0x24010001     li        at,1                # 11
  [1518] 0x100065dc     0x8f8281c0     lw        v0,-32320(gp)       # 12
         <2 cycle stall for following instruction>
  [1518] 0x100065e0     0xac410000     sw        at,0(v0)            # 15
  [1519] 0x100065e4     0x8f99819c     lw        t9,-32356(gp)       # 16
         <2 cycle stall for following instruction>
  [1519] 0x100065e8     0x0320f809     jalr      ra,t9               # 19
  [1519] 0x100065ec     0000000000     nop       # 20
         <2 cycle stall for following instruction>
  [1527] 0x100065f0     0xafa00008     sw        zero,8(sp)          # 23
  [1527] 0x100065f4     0x8fa40008     lw        a0,8(sp)            # 24
         <2 cycle stall for following instruction>
  [1527] 0x100065f8     0x28842710     slti      a0,a0,10000         # 27
  [1527] 0x100065fc     0x108000ac     beq       a0,zero,0x100068b0  # 28
  [1527] 0x10006600     0000000000     nop       # 29
         <2 cycle stall for following instruction>
  [1529] 0x10006604     0x24070001     li        a3,1                # 32
  ^------    1 samples(0.02%)------^
  [1529] 0x10006608     0xafa7000c     sw        a3,12(sp)           # 33
  [1529] 0x1000660c     0x8f8681b8     lw        a2,-32328(gp)       # 34
         <2 cycle stall for following instruction>
```

```
[1529] 0x10006610     0x8cc60000     lw        a2,0(a2)            # 37
       <2 cycle stall for following instruction>
[1529] 0x10006614     0x24c6ffff     addiu     a2,a2,-1            # 40
[1529] 0x10006618     0x8fa5000c     lw        a1,12(sp)           # 41
       <2 cycle stall for following instruction>
[1529] 0x1000661c     0x00a6282a     slt       a1,a1,a2            # 44
^------    1 samples(0.02%)------^
[1529] 0x10006620     0x10a0009c     beq       a1,zero,0x10006894  # 45
[1529] 0x10006624     0000000000     nop       # 46
       <2 cycle stall for following instruction>
[1530] 0x10006628     0x240a0001     li        t2,1                # 49
^------    2 samples(0.05%)------^
[1530] 0x1000662c     0xafaa0010     sw        t2,16(sp)           # 50
[1530] 0x10006630     0x8f8981b8     lw        t1,-32328(gp)       # 51
^------    1 samples(0.02%)------^
       <2 cycle stall for following instruction>
[1530] 0x10006634     0x8d290000     lw        t1,0(t1)            # 54
       <2 cycle stall for following instruction>
[1530] 0x10006638     0x2529ffff     addiu     t1,t1,-1            # 57
[1530] 0x1000663c     0x8fa80010     lw        t0,16(sp)           # 58
       <2 cycle stall for following instruction>
[1530] 0x10006640     0x0109402a     slt       t0,t0,t1            # 61
^------    3 samples(0.07%)------^
[1530] 0x10006644     0x11000089     beq       t0,zero,0x1000686c  # 62
^------    1 samples(0.02%)------^
[1530] 0x10006648     0000000000     nop       # 63
       <2 cycle stall for following instruction>
[1531] 0x1000664c     0x8fa90010     lw        t1,16(sp)           # 66
^------   17 samples(0.41%)------^
       <2 cycle stall for following instruction>
[1531] 0x10006650     0x25290001     addiu     t1,t1,1             # 69
^------   18 samples(0.43%)------^
[1531] 0x10006654     0x8fab000c     lw        t3,12(sp)           # 70
^------   15 samples(0.36%)------^
       <2 cycle stall for following instruction>
[1531] 0x10006658     0x256b0001     addiu     t3,t3,1             # 73
^------   33 samples(0.80%)------^
[1531] 0x1000665c     0x000b5080     sll       t2,t3,2             # 74
^------   21 samples(0.51%)------^
[1531] 0x10006660     0x014b5021     addu      t2,t2,t3            # 75
^------    9 samples(0.22%)------^
[1531] 0x10006664     0x000a50c0     sll       t2,t2,3             # 76
^------   15 samples(0.36%)------^
```

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: `[1514]`.
2	Beginning address of the instruction: `0x100065b8`.
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 in an instruction (**pcsamp** only).
- Any cycle stalls, that is, cycles that were wasted.

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:

```
    Prof run at: Tue May 27 18:04:51 1997
 Command line: prof -S -dis generic.ideal.m876

         3900085040: Total number of cycles
          39.00085s: Total execution time
         2046668286: Total number of instructions executed
              1.906: Ratio of cycles / instruction
                100: Clock rate in MHz
             R4600: Target processor modelled
------------------------------------------------------------
Procedures sorted in descending order of cycles executed.
Unexecuted or inlined procedures are not listed. Procedures
beginning with *DF* are dummy functions and represent
init, fini and stub sections.
------------------------------------------------------------

        cycles(%)  cum %  secs  instrns   calls   procedure(dso:file)

3754320037(96.26)  96.26  37.54 1971220024   1 anneal(generic.pixie:generic.c)
 145001146( 3.72)  99.98   1.45   75000728   1 slaveusrtime(./dlslave.so.pixn32:
          dlslave.c)
    187200( 0.00)  99.99   0.00     124800  1600 next(./libc.so.1.pixn32:
          /xlv1/bonsai-sep09/work/irix/lib/libc/libc_n32_M3/math/drand48.c)
    101504( 0.00)  99.99   0.00      58124     1 init2da(generic.pixie:generic.c)
     91200( 0.00)  99.99   0.00      62400  1600 _drand48(./libc.so.1.pixn32:
          /xlv1/bonsai-sep09/work/irix/lib/libc/libc_n32_M3/math/drand48.c)
     78574( 0.00)  99.99   0.00      30063   628 __sinf(./libm.so.pixn32:
          ../../libm/fsin.c)
     64442( 0.00)  99.99   0.00      45661    48 _doprnt(./libc.so.1.pixn32:
          /xlv1/bonsai-sep09/work/irix/lib/libc/libc_n32_M3/print/doprnt.c)
     57888( 0.00) 100.00   0.00       9648    16 offtime(./libc.so.1.pixn32:
          /xlv1/bonsai-sep09/work/irix/lib/libc/libc_n32_M3/gen/time_comm.c)
     43767( 0.00) 100.00   0.00      29215   263 fread(./libc.so.1.pixn32:
          /xlv1/bonsai-sep09/work/irix/lib/libc/libc_n32_M3/stdio/fread.c)
     16484( 0.00) 100.00   0.00      12285   299 _readdir(./libc.so.1.pixn32:
          /xlv1/bonsai-sep09/work/irix/lib/libc/libc_n32_M3/gen/readdir.c)
     12376( 0.00) 100.00   0.00       7224   281 memcpy(./libc.so.1.pixn32:
          /xlv1/bonsai-sep09/work/irix/lib/libc/libc_n32_M3/strings/bcopy.s)
     10526( 0.00) 100.00   0.00       6321     1 dirstat(generic.pixie:generic.c)
      9545( 0.00) 100.00   0.00       6103     1 iofile(generic.pixie:generic.c)
      6258( 0.00) 100.00   0.00       5066   298 _stat(./libc.so.1.pixn32:
          /xlv1/bonsai-sep09/work/irix/lib/libc/libc_n32_M3/sys/stat.c)
```

```
-----------------------
disassembly listing
-----------------------

*DF*_generic.MIPS.stubs_1
*DF*_generic.MIPS.stubs_1: <0x10001a90-0x10001db4>
    154 total cycles(0.00%) invoked 0 times, average ? cycles/invocation
  [1] 0x10001a90     0x0006000d     break     0x6                # 1
  ^---      0 total cycles(0.00%) executed     0 times, average  1 cycles.---^
  [1] 0x10001a94     0x8f998050     lw        t9,-32688(gp)      # 1
  [1] 0x10001a98     0x03e07825     move      t7,ra              # 2
         <1 cycle stall for following instruction>
  [1] 0x10001a9c     0x0320f809     jalr      ra,t9              # 4
  [1] 0x10001aa0     0x34180029     ori       t8,zero,0x29       # 5
         <2 cycle stall for following instruction>
  ^---      7 total cycles(0.00%) executed     1 times, average  7 cycles.---^
  [1] 0x10001aa4     0x8f998050     lw        t9,-32688(gp)      # 1
  [1] 0x10001aa8     0x03e07825     move      t7,ra              # 2
         <1 cycle stall for following instruction>
  [1] 0x10001aac     0x0320f809     jalr      ra,t9              # 4
  [1] 0x10001ab0     0x3418002a     ori       t8,zero,0x2a       # 5

...

generic.c
main: <0x10001ecc-0x10002000>
    44 total cycles(0.00%) invoked 1 times, average 44 cycles/invocation
File 'generic.c':
Skipping source listing to line 87
88: void    sproctestgrandchild(void *);    /* sproc grandchild code */
89:
90: static  struct timeval   starttime;     /* starting time--first timestamp */
91: static  struct timeval   ttime;         /* last-recorded timestamp */
92: static  struct timeval   deltatime;
93:
94: int     pagesize;
95:
96: main(unsigned argc, char **argv)
97: {
  [97] 0x10001ecc     0x27bdffd0     addiu     sp,sp,-48          # 1
  [97] 0x10001ed0     0xffbf0008     sd        ra,8(sp)           # 2
  [97] 0x10001ed4     0xffbc0010     sd        gp,16(sp)          # 3
  [97] 0x10001ed8     0x3c010002     lui       at,0x2             # 4
  [97] 0x10001edc     0x2421e0cc     addiu     at,at,-7988        # 5
  [97] 0x10001ee0     0x0321e021     addu      gp,t9,at           # 6
```

```
  [97] 0x10001ee4     0xafa40024     sw        a0,36(sp)           # 7
  [97] 0x10001ee8     0xafa5002c     sw        a1,44(sp)           # 8
98:      int     i;
99:
100:     /* initialize the timestamp */
101:     (void) gettimeofday(&starttime, NULL);
  [101] 0x10001eec     0x27848360     addiu     a0,gp,-31904        # 9
  [101] 0x10001ef0     0x00002825     move      a1,zero             # 10
  [101] 0x10001ef4     0x8f99807c     lw        t9,-32644(gp)       # 11
  ^---     11 total cycles(0.00%) executed     1 times, average 11 cycles.---^
  [101] 0x10001ef8     0x0320f809     jalr      ra,t9               # 1
  [101] 0x10001efc     0000000000     nop       # 2
        <2 cycle stall for following instruction>
  ^---      4 total cycles(0.00%) executed     1 times, average  4 cycles.---^
102:
103:     /* set up to reap any children */
104:     (void) sigset(SIGCHLD, (SIG_PF)reapSig);
  [104] 0x10001f00     0x24040012     li        a0,18               # 1
  [104] 0x10001f04     0x8f858144     lw        a1,-32444(gp)       # 2
  [104] 0x10001f08     0x8f998080     lw        t9,-32640(gp)       # 3
  ^---      3 total cycles(0.00%) executed     1 times, average  3 cycles.---^
  [104] 0x10001f0c     0x0320f809     jalr      ra,t9               # 1
  [104] 0x10001f10     0000000000     nop       # 2
        <2 cycle stall for following instruction>
  ^---      4 total cycles(0.00%) executed     1 times, average  4 cycles.---^
105:
106:     if(argc == 1) {
  [106] 0x10001f14     0x8fa20024     lw        v0,36(sp)           # 1
  [106] 0x10001f18     0x24030001     li        v1,1                # 2
        <1 cycle stall for following instruction>
  [106] 0x10001f1c     0x1443000c     bne       v0,v1,0x10001f50    # 4
  [106] 0x10001f20     0000000000     nop       # 5
        <2 cycle stall for following instruction>
    Preceding branch executed 1 times, taken 0 times.
  ^---      7 total cycles(0.00%) executed     1 times, average  7 cycles.---^
107:        Scriptstring(DEFAULT_SCRIPT);
  [107] 0x10001f24     0x8f84805c     lw        a0,-32676(gp)       # 1
        <2 cycle stall for following instruction>
  [107] 0x10001f28     0x24847038     addiu     a0,a0,28728         # 4
  [107] 0x10001f2c     0x8f99814c     lw        t9,-32436(gp)       # 5
  ^---      5 total cycles(0.00%) executed     1 times, average  5 cycles.---^
  [107] 0x10001f30     0x0320f809     jalr      ra,t9               # 1
```

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 set 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
```

prof 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 set two caliper points, *prof* can generate a report from the first to the second caliper point:

```
prof -calipers 1 2
```

Using the -gprof Option

For **ideal**, **usertime**, and **fpe** experiments, the **-gprof** option to prof can be used to obtain inclusive basic block counting information. *prof* provides the standard report and then appends the inclusive function counts information to the end of the report. The example below shows 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. This list includes functions starting with **_rld** because **rld** is not instrumented. It also includes functions from *libss*; they are instrumented, but their data is normally excluded.

prof fails to list cycles of a procedure in the inclusive listing for the following reasons:

- init & fini sections and MIPS stubs are not part of any procedure.
- Calls to procedures that don't use a "jump and link" are not recognized as procedure calls.
- 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:

```
-------------------------------------------------------------------------------
Profile listing generated Tue May 27 18:18:21 1997
    with:     prof -gprof generic.usertime.m1019
-------------------------------------------------------------------------------

     Total Time (secs)     : 41.01
     Total Samples         : 1367
     Stack backtrace failed: 1
     Sample interval (ms)  : 30
     CPU                   : R4600
     FPU                   : R4600
     Clock                 : 100.0MHz
     Number of CPUs        : 1


-------------------------------------------------------------------------------

index  %Samples     self descendents  total        name
[1]      99.9%    0.00       40.98    1366         __start
[2]      99.9%    0.00       40.98    1366         main
[3]      99.9%    0.00       40.98    1366         Scriptstring
[4]      96.0%    0.00       39.36    1312         usrtime
[5]      96.0%   39.36        0.00    1312         anneal
[6]       3.1%    0.00        1.26      42         libdso
[7]       3.1%    0.00        1.26      42         dlslave_routine
[8]       3.1%    1.26        0.00      42         slaveusrtime
[9]       0.8%    0.00        0.33      11         iofile
[10]      0.7%    0.00        0.30      10         fread
[11]      0.7%    0.30        0.00      10         _read
[12]      0.1%    0.00        0.03       1         dirstat
[13]      0.1%    0.00        0.03       1         _stat
[14]      0.1%    0.03        0.00       1         _xstat
[15]      0.1%    0.00        0.03       1         genLog
[16]      0.1%    0.00        0.03       1         fprintf
[17]      0.1%    0.00        0.03       1         _doprnt
[18]      0.1%    0.00        0.03       1         _dowrite
[19]      0.1%    0.00        0.03       1         fwrite
[20]      0.1%    0.03        0.00       1         _write


-------------------------------------------------------------------------------
     Gprof Listing
-------------------------------------------------------------------------------
```

```
                                     caller/total        parents
index  %time     self descendents  total (self)    name
                                     callee/descend      children

-------------------------------------------------------------------------------

[1]     99.9%    0.00       40.98    1366 (0)       __start [1]
                 0.00       40.98    1366/1366          0x10001e9c main [2]

-------------------------------------------------------------------------------

                 0.00       40.98    1366/1366          0x10001e9c __start [1]
[2]     99.9%    0.00       40.98    1366 (0)       main [2]
                 0.00       40.98    1366/1366          0x10001f30 Scriptstring
[3]

-------------------------------------------------------------------------------

                 0.00       40.98    1366/1366          0x10001f30 main [2]
[3]     99.9%    0.00       40.98    1366 (0)       Scriptstring [3]
                 0.00       39.36    1312/1366          0x10002378 usrtime [4]
                 0.00        1.26      42/1366          0x10002378 libdso [6]
                 0.00        0.33      11/1366          0x10002378 iofile [9]
                 0.00        0.03       1/1366          0x10002378 dirstat [12]

-------------------------------------------------------------------------------

                 0.00       39.36    1312/1312          0x10002378 Scriptstring
[3]
[4]     96.0%    0.00       39.36    1312 (0)       usrtime [4]
                39.36        0.00    1312/1312          0x100059b8 anneal [5]

-------------------------------------------------------------------------------

                39.36        0.00    1312/1312          0x100059b8 usrtime [4]
[5]     96.0%   39.36        0.00    1312 (1312)    anneal [5]

-------------------------------------------------------------------------------

                 0.00        1.26      42/42            0x10002378 Scriptstring
[3]
[6]      3.1%    0.00        1.26      42 (0)       libdso [6]
```

```
                0.00        1.26      42/42           0x10003028
dlslave_routine [7]

-------------------------------------------------------------------------------

                0.00        1.26      42/42           0x10003028 libdso [6]
[7]      3.1%   0.00        1.26      42 (0)      dlslave_routine [7]
                1.26        0.00      42/42           0x5ffe0650 slaveusrtime
[8]

-------------------------------------------------------------------------------

                1.26        0.00      42/42           0x5ffe0650
dlslave_routine [7]
[8]      3.1%   1.26        0.00      42 (42)     slaveusrtime [8]

-------------------------------------------------------------------------------

                0.00        0.33      11/11           0x10002378 Scriptstring
[3]
[9]      0.8%   0.00        0.33      11 (0)      iofile [9]
                0.00        0.30      10/11           0x10002ab8 fread [10]
                0.00        0.03       1/11           0x10002a5c genLog [15]

-------------------------------------------------------------------------------

                0.00        0.30      10/10           0x10002ab8 iofile [9]
[10]     0.7%   0.00        0.30      10 (0)      fread [10]
                0.30        0.00      10/10           0xfad26e0 _read [11]

-------------------------------------------------------------------------------

                0.30        0.00      10/10           0xfad26e0 fread [10]
[11]     0.7%   0.30        0.00      10 (10)     _read [11]

-------------------------------------------------------------------------------

                0.00        0.03       1/1            0x10002378 Scriptstring
[3]
[12]     0.1%   0.00        0.03       1 (0)      dirstat [12]
                0.00        0.03       1/1            0x10002820 _stat [13]

-------------------------------------------------------------------------------

                0.00        0.03       1/1            0x10002820 dirstat [12]
```

```
[13]     0.1%    0.00        0.03       1 (0)      _stat [13]
                 0.03        0.00       1/1            0xfaf8c10 _xstat [14]

-------------------------------------------------------------------------------

                 0.03        0.00       1/1            0xfaf8c10 _stat [13]
[14]     0.1%    0.03        0.00       1 (1)      _xstat [14]

-------------------------------------------------------------------------------

                 0.00        0.03       1/1            0x10002a5c iofile [9]
[15]     0.1%    0.00        0.03       1 (0)      genLog [15]
                 0.00        0.03       1/1            0x10006bc4 fprintf [16]

-------------------------------------------------------------------------------

                 0.00        0.03       1/1            0x10006bc4 genLog [15]
[16]     0.1%    0.00        0.03       1 (0)      fprintf [16]
                 0.00        0.03       1/1            0xfab55ec _doprnt [17]

-------------------------------------------------------------------------------

                 0.00        0.03       1/1            0xfab55ec fprintf [16]
[17]     0.1%    0.00        0.03       1 (0)      _doprnt [17]
                 0.00        0.03       1/1            0xfab215c _dowrite [18]

-------------------------------------------------------------------------------

                 0.00        0.03       1/1            0xfab215c _doprnt [17]
[18]     0.1%    0.00        0.03       1 (0)      _dowrite [18]
                 0.00        0.03       1/1            0xfab1ddc fwrite [19]

-------------------------------------------------------------------------------

                 0.00        0.03       1/1            0xfab1ddc _dowrite [18]
[19]     0.1%    0.00        0.03       1 (0)      fwrite [19]
                 0.03        0.00       1/1            0xfad30f8 _write [20]

-------------------------------------------------------------------------------

                 0.03        0.00       1/1            0xfad30f8 fwrite [19]
[20]     0.1%    0.03        0.00       1 (1)      _write [20]
```

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

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

Generating Reports for Multiprocessed Executables

You can gather data from executables that use the **sproc()** and **sprocsp()** 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* like any other data files.

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 7-3 for letter codes and descriptions.This naming convention avoids the potential conflict of multiple threads attempting to write simultaneously to the same file.

Table 7-3 Letter Codes in Experiment ID Numbers

Letter Code	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()**

Generating Compiler 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 recompile. You can rearrange procedures using the **-fb** flag to *cc*, or using the *cord* command. For more information, view the cc or cord reference page.

Interpreting Reports

If the target process was blocked for a long time as a result of an instruction, that instruction will show up as having a low or zero CPU time. On the other hand, CPU-intensive instructions will show up as having a high CPU time.

One way to sanity-check inclusive cycle counts is to look at the percentage cycles for **__start()**. If the value is anything less than 98 -99%, the inclusive report is suspect. Look for other warnings that *prof* didn't take into account certain procedures.

Chapter 8

Using SpeedShop in Expert Mode: pixie

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 and exits only at the end.
- 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 Chapter 7, "Analyzing Experiment Results: prof."

This chapter contains the following sections:

- "Using pixie"
- "Obtaining Basic Block Counts"
- "Obtaining Inclusive Basic Block Counts"

Using pixie

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 such as SIGALRM that are based on an external clock.

pixie Syntax

The syntax for *pixie* is

```
pixie prog_name [options]
```

prog_name Name of the input program.

options Zero or more of the keywords listed in Table 8-1.

pixie Options

Table 8-1 lists *pixie* options. For a complete list of options, view the pixie reference page.

Table 8-1 Options for pixie

Name	Result
-addlibs *lib1.s0:...libN.so*	Adds *lib1.s0:...libN.so* to the library list of the executable. No libraries are added by default.
-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.
-dso	Treats executable as an o32 DSO. Performs a search of standard o32 library directories. A *.pix32* extension is used.
-dso32	Treats executable as an n32 DSO. Performs a search of standard n32 library directories. A *.pixn32* extension is used.

Table 8-1 (continued) Options for pixie

Name	Result
-dso64	Treats executable as an n64 DSO. Performs a search of standard n64 library directories. A *.pix64* extension is used.
-directory *dir_name*	Writes output files to *dir_name*. Files are written to the current directory by default.
-fcncounts	Produces an instrumented executable that counts function calls and arc calls, but not basic-block or branch counts.
-idtrace_file *number*	Specifies a UNIX® file descriptor number for the trace output file. Default is 19.
–[no]autopixieo32	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 pixified files that it finds matching the *lib* it is to instrument. If the fields match, they are not instrumented. *pixie* cannot detect shared libraries opened with **dlopen()** (and hence 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**.
-[no]idtrace	[Disables] or enables tracing of both instruction and data memory. Default is -noidtrace.
-[no]ittrace	[Disables] or enables tracing of instruction memory references. Default is -noittrace.
-[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]verbose	Prints or suppresses messages summarizing the binary-to-binary translation process. The default is **–noverbose**.
-pixie_file *name*	Specify the name of the pixiefied executable.
-suffix *.suffix*	Appends *.suffix* to the pixified executable and DSOs. The default suffix is *.pixie*.

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 share group. In this case, each file will have the process ID appended to its name.

Obtaining Basic Block Counts

Use this procedure to obtain basic block counts. Also refer to Figure 8-1, which illustrates how basic block counting works.

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
   ```

   pixie 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. Set the path for your *.pixie* files. *pixie* uses the *rld* search path for libraries (see *rld*(1) for the default paths). If the *.pixie* files are in your local directory, set the path as

   ```
   % setenv LD_LIBRARY_PATH .
   ```

4. Execute the file(s) 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 filename (for example, *myprog.Counts.345*) 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.

   To generate a valid *.Counts* file, your program must terminate normally or with a call to **exit()**. If it terminates with a signal such as SIGINT, the program must use a signal handler and leave the program through **exit()**.

5. Run the profile formatting program *prof* specifying the name of the original program and the *.Counts* file for the program:

   ```
   % prof myprog 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 of the files.

   ```
   % prof myprog 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 and memory fetches and stores.

The complete output of the **–pixie** option is often extremely large. Use the **–quit** option with *prof* to restrict the size of the report. Refer to Chapter 7, "Analyzing Experiment Results: prof," for details about *prof* options.

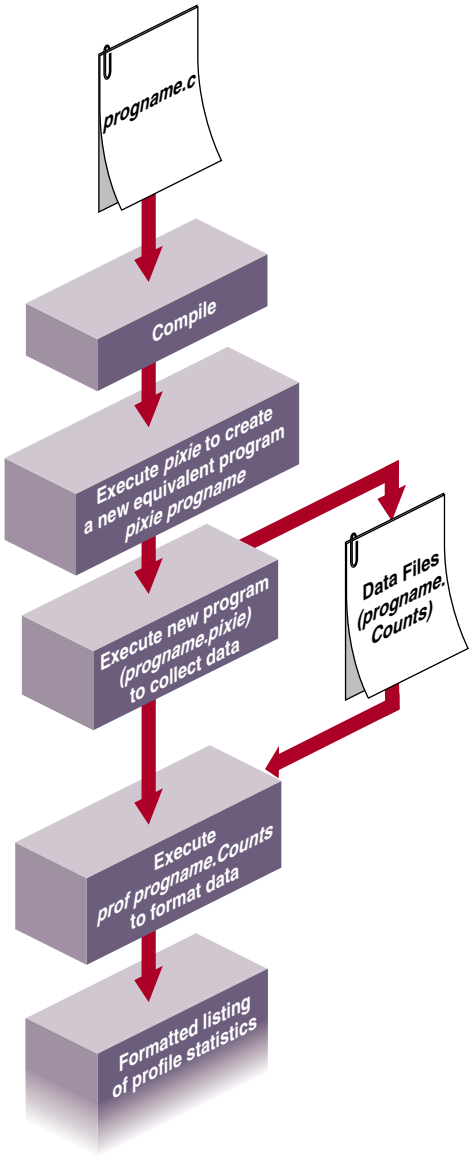

Figure 8-1 How Basic Block Counting Works

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*.

Example Using prof –invocations

The partial listing below illustrates the report generated for basic block counts in *generic*. *prof* first provides a standard report of basic block counts, then provides a report reflecting any options provided to *prof*.

```
% prof -i generic generic.Counts

Prof run at: Fri May 17 12:39:22 1996
 Command line: prof -i generic generic.Counts

         2662778530: Total number of cycles
          17.75186s: Total execution time
         1875323864: Total number of instructions executed
              1.420: Ratio of cycles / instruction
                150: Clock rate in MHz
             R4000: Target processor modelled

------------------------------------------------------------
Procedures sorted in descending order of cycles executed.
Unexecuted procedures are not listed. Procedures
beginning with *DF* are dummy functions and represent
init, fini and stub sections.
------------------------------------------------------------

cycles(%)          cum %   secs   instrns    calls     procedure(dso:file)

2524610038(94.81)  94.81   16.83  1797940023   1 anneal(generic:/usr/demos/
          SpeedShop/generic/generic.c)
135001332( 5.07)   99.88    0.90  75000822     1
slaveusrtime(./dlslave.so:/usr/demos/
          SpeedShop/generic/dlslave.c)
1593518( 0.06)     99.94    0.01  1378788      4385 memcpy(/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/strings/bcopy.s)
735797( 0.03)      99.97    0.00  506627       4123 fread(/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/stdio/fread.c)
187200( 0.01)      99.98    0.00  124800       1600 next(/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/math/drand48.c)
```

```
136116( 0.01)      99.98    0.00  82498        1 iofile(generic:
          /usr/demos/SpeedShop/generic/generic.c)
91200( 0.00)      99.98     0.00  62400  1600 _drand48(/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/math/drand48.c)
...
```

- The `cycles(%)` column reports the number and percentage of machine cycles used for the procedure. For example, 2524610038 cycles, or 94.81% of cycles were spent in the **anneal()** procedure.
- The `cum%` column shows the cumulative percentage of calls. For example, 99.88% of all calls were spent between the top two functions in the listing: **anneal()** and **slaveusrtime()**.
- The `secs` column shows the number of seconds spent in each procedure. For example, 16.83 seconds were spent in the **anneal()** procedure. The time represents an idealized computation based on modelling the machine. It ignores potential floating point interlocks and memory latency time (cache misses and memory bus contention).
- The `instrns` column shows the number of instructions executed for a procedure. For example, there were 1797940023 instructions devoted to the **anneal()** procedure.
- The `calls` column reports the number of calls to each procedure. For example, there was just one call to the **anneal()** procedure.
- The `procedure (dso:file)` column lists the procedure, its DSO name and filename. For example, the first line reports statistics for the procedure **anneal()** in the file *generic.c* in the generic executable.

The partial listing below illustrates the use of the **–i[nvocations]** option. For each procedure, *prof* reports the number of times it was invoked from each of its possible callers and lists the procedure(s) that called it.

```
------------------------------------------------------
Procedures sorted in descending order of times invoked.
Unexecuted procedures are not listed.
------------------------------------------------------

Total number of procedure invocations: 15114
calls(%)      cum%   size(bytes) procedure (dso:file)

4385(29.01)   29.01  3416        memcpy (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/strings/bcopy.s)

4123(27.28)   56.29  1304        fread (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/stdio/fread.c)
1600(10.59)   66.88  312         next (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/math/drand48.c)
1600(10.59)   77.46  180         _drand48 (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/math/drand48.c)
628( 4.16)    81.62  368         __sinf (/usr/lib32/libm.so:
          /work/cmplrs/libm/fsin.c)
259( 1.71)    83.33  524         __filbuf (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/stdio/_filbuf.c)
```

The above listing shows the total procedure invocations (calls) during the run: 12113082.

- The `calls(%)` column reports the number of calls (and the percentage of total calls) per procedure. For example, there were 4385 calls (or 29.01% of the total) spent in the procedure **memcpy()**.
- The `cum%` column shows the cumulative percentage of calls. For example, 56.29% of all calls were spent between **memcpy()** and **fread()**.
- The `size(bytes)` column shows the total byte size of a procedure. For example, the procedure **memcpy()** is 3416 bytes.
- The `procedure (dso:file)` column lists the procedure, its DSO name and its filename. For example, the first line reports statistics for the procedure **memcpy()** in the file *bcopy.s* in *libc.so*.

Example Using prof –heavy

The following partial listing shows the source code lines responsible for the largest portion of execution time produced with the **–heavy** option.

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

The partial listing below shows basic block counts sorted in descending order of cycles used. The fields in the report are described in section “ideal Experiment Reports” section in Chapter 7, “Analyzing Experiment Results: prof.”

```
-------------------------------------------------------
Lines listed in descending order of cycle counts.
-------------------------------------------------------
cycles(%)           cum %  times     line procedure (dso:file)
2309934120(86.75%)  86.75% 14440000  1465  anneal (generic:/usr/demos/
          SpeedShop/generic/generic.c)
207945880( 7.81%)   94.56% 14440000  1464  anneal (generic:/usr/demos/
          SpeedShop/generic/generic.c)
81000506( 3.04%)  97.60% 5000000       29  slaveusrtime (dlslave.so:/usr/demos/
          SpeedShop/generic/dlslave.c)
54000000( 2.03%)  99.63% 5000000       30  slaveusrtime (dlslave.so:/usr/demos/
          SpeedShop/generic/dlslave.c)
6600000( 0.25%)  99.88% 380000       1463  anneal (generic:/usr/demos/
          SpeedShop/generic/generic.c)
418380( 0.02%)   99.89%  32981        493  memcpy (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/strings/bcopy.s)
418380( 0.02%)  99.91%   32981        494  memcpy (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/strings/bcopy.s)
139482( 0.01%)  99.91%   32981        496  memcpy (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/strings/bcopy.s)
139460( 0.01%)  99.92%   32981        495  memcpy (/usr/lib32/libc.so.1:
          /work/irix/lib/libc/libc_n32_M3/strings/bcopy.s)
130009( 0.00%)  99.92%   10000       1461  anneal (generic:/usr/demos/
          SpeedShop/generic/generic.c)
```

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:

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

 Prof run at: Fri May 17 14:09:12 1996
 Command line: prof -quit 4 generic generic.Counts

         2662778530: Total number of cycles
          17.75186s: Total execution time
         1875323864: Total number of instructions executed
              1.420: Ratio of cycles / instruction
                150: Clock rate in MHz
             R4000: Target processor modelled
------------------------------------------------------------
Procedures sorted in descending order of cycles executed.
Unexecuted procedures are not listed. Procedures
beginning with *DF* are dummy functions and represent
init, fini and stub sections.
------------------------------------------------------------
cycles(%)          cum %     secs    instrns        calls     procedure(dso:file)

2524610038(94.81)  94.81     16.83   1797940023     1 anneal(generic:/usr/demos/
          SpeedShop/generic/generic.c)
135001332( 5.07)   99.88     0.90    75000822       1 slaveusrtime(./dlslave.so:
          /usr/demos/SpeedShop/generic/dlslave.c)
1593518( 0.06)     99.94     0.01    1378788        4385 memcpy(/usr/lib32/
          libc.so.1:/work/irix/lib/libc/libc_n32_M3/strings/bcopy.s)
735797( 0.03)      99.97     0.00     506627        4123 fread(/usr/lib32/
          libc.so.1:/work/irix/lib/libc/libc_n32_M3/stdio/fread.c)
```

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 the "ideal Experiment" section in Chapter 4, "Experiment Types."

To see inclusive data, run the profile formatting program *prof* specifying the name of the original program, the **-gprof** flag, and the *.Counts* file for the program.

```
% prof -gprof myprog 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 of the files.

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

Example of prof –gprof

This section contains part of a sample output obtained by using the **–gprof** option. The fields in the report are explained in detail in the report, but are not provided in this example. For more information on the **-gprof** option, see Chapter 7, "Analyzing Experiment Results: prof." (The format of the output has been adjusted slightly.)

```
% prof -gprof generic generic.Counts

...
Prof run at: Fri May 17 14:42:25 1996
 Command line: prof -gprof generic generic.Counts
...
                          self           kids        called/total    parents
index           cycles(%) self(%)        kids(%)  called+self   name     index
                          self           kids        called/total    children

[1] 2662767961 (100.00%)  71( 0.00%) 2662767890(100.00%) 0  __start [1]
                          44         2662767837       1/1      main [2]
                          5                  0        1/1      __istart[111]
                          4                  0        1/1   __readenv_sigfpe[112]
-------------------------------------------------------------------------------
```

```
                          44          2662767837          1/1  __start [1]
[2] 2662767881(100.00%)   44( 0.00%) 2662767837(100.00%) 1      main [2]
                          2152        2662764245          1/1      Scriptstring[3]
                          67          926                 1/1      exit [58]
                          96          309                 1/1      _sigset [67]
                          32          10                  1/9    _gettimeofday[68]
-------------------------------------------------------------------------------
...
```

Chapter 9

Miscellaneous Commands

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"
- "Using the squeeze Command"
- "Calculating the Working Set of a Program"
- "Dumping Performance Data Files"
- "Dumping Compiler Feedback Files"

Using the thrash Command

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

thrash Syntax

```
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.
- **-n** *count* The number of references to make before exiting. The default is 10,000.
- **-p** *N* The amount of memory to access in pages, where *N* is the number of pages.
- **-s** Sequential thrashing. The default is random.
- **-w** *time* The amount of time thrash should sleep after thrashing but before exiting.

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 thrash over it. Here's an example:

```
fraser 82% thrash -m 4
thrashing randomly: 4.00 MB (= 0x00400000 = 4194304 bytes = 1024 pages)
        10000 iterations
```

You can use *thrash* in conjunction with *ssusage* and *squeeze* to determine the approximate available working memory on a system, as described in the section "Calculating the Working Set of a Program".

Using the squeeze Command

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

squeeze Syntax

```
squeeze [flag] amount
```

flag One of the following flags. If no flag 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.

Effects of squeeze

squeeze 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 provide 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's an example:

```
fraser 1# 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
```

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 as follows. 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 start-up).
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 labelled majf gives the number of major page faults (i.e. 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, "Collecting Data on Machine Resource Usage."
4. As superuser in a separate window, run the *squeeze* command to lock down an amount of memory.
5. Rerun *thrash* with *ssusage*:

   ```
   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 out the approximate working set of the kernel and any other applications that are running on the machine. You'll need this number when you reach the next steps.

8. Determine the working set of the program you're 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 superuser.

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.

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*.

ssdump Syntax

```
ssdump [options] {datafile1 ... datafileN} ...
```

options Zero or more of the following print options:

- **-d** Prints detailed information for each bead. For compressed beads, the compressed form will be dumped.
- **-D** Prints detailed information for each bead. For compressed beads, the uncompressed form will be dumped.
- **-h** Prints the hex contents of the body of each bead.
- **-i** *index* Prints only one bead at *index* in the file.
- **-q** Suppresses printing of those fields that will normally change from run to run such as process IDs and time stamps. This option is useful for QA work, to enable automatic comparisons of recorded experiments.
- **-s** *offset* Prints only one bead at *offset* into the file.

Experiment File Format

The file is written as a string of "beads," each of which is a record with

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

The file prologue consists of these 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 calls *ssdump* on performance data for a **pcsamp** experiment:

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

Below is some 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"
```

Dumping Compiler Feedback Files

The *fbdump* command can be used to print out the compiler feedback files generated by running *prof -feedback*. For more information on using compiler feedback files, view the cord or cc reference pages.

fbdump Syntax

```
fbdump options filename
```

options Zero or more of the options described in table Table 9-1.

filename The feedback filename. This file has a *.fb* extension.

Table 9-1 Options for fbdump

Option	Prints.
-all	Feedback using all options. This is the default.
-ascii	Feedback in the same style as earlier version of the feedback dump program.
-bb	Feedback per basic block table as described in "cmplrs/fb.h". 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 non-zero execution counts.
-call	Feedback call table as described in "cmplrs/fb.h". 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 "cmplrs/fb.h".
-proc	Feedback procedure table as described in "cmplrs/fb.h". 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 "cmplrs/fb.h".
-str	Feedback string table.
-verbose	All the information in verbose mode including a table with all zero entries.

Index

D

E

F

G

H

I

L

M

P

R

S

T

U

W

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-3311-002.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-965-0964
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389