SpeedShop User's Guide

Document Number 007-3311-001

CONTRIBUTORS

Written and Illustrated by Janet Home-Lorenzin, Wendy Ferguson and Dany Galgani.
Production by Ruth Christian.
Engineering contributions by Marty Itzkowitz, Pete Orelup, Jun Yu and Marco Zagha.

SpeedShop User's Guide
Document Number 007-3311-001

Contents

List of Figures

List of Tables

About This Guide

This manual is a user's guide for the SpeedShop performance tools, Release 1.0. 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.

Chapter 1

Introduction to Performance Analysis

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

- “Sources of Performance Problems”
- “Fixing Performance Problems”
- “SpeedShop Tools”
- “Using SpeedShop Tools for Performance Analysis”
- “Collecting Data for Part of a Program”

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 might 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 large amounts 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, it is useful to be able to collect data for just that part of the program.

Since programs exhibit different behavior during different phases of operation, you need to identify the limiting resource during 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. And 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, you may need to streamline your code. This can entail modifying algorithms, reordering code to avoid interlocks, removing nonessential steps, blocking to keep data in cache and registers, or 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, you can try one of the following techniques:

- Improve overlap of I/O with computation
- Optimize data usage to minimize disk access
- Use data compression

Memory-bound Processes

A *memory-bound* program that continuously needs to swap out pages of memory. Page thrashing is often due to accessing virtual memory on a haphazard rather than strategic basis. One telltale indication of a page-thrashing condition on a system with paging to a local disk, is noise due to disk accesses. To fix a memory-bound process, you can try to improve the memory reference patterns or, if possible, decrease the memory used by the program.

Bugs

You may find that a bug is causing the performance problem. For example, you might find that you are unnecessarily reading from the same file twice in different parts of the

program, that floating point exceptions are slowing down your program, that old code has not been completely removed, or that you are 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 comprise only a part of performance tuning. Other areas that you can tune, but 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 system, it's a bad idea to do so. Any such "features" may break in future releases of the compiler system. 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.

Main Commands

- *ssusage*

  The *ssusage* command allows you to collect information about your program's use of machine resources. Output from *ssusage* can be used to determine where most resources are being spent.

- *ssrun*

  The *ssrun* command 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*

  The *prof* command 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

- *pixie*

  The *pixie* command 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.

  pixie reads an executable program, partitions it into basic blocks, and writes an equivalent program (with a *.pixie* extension by default) containing additional code that counts the execution of each basic block.

- *fbdump*

  The *fbdump* command prints out the formatted contents of compiler feedback files generated by *prof*.

- *squeeze*

  The *squeeze* command is used to allocate a region of virtual memory and lock the virtual memory down into real memory, making it unavailable to other processes.

- *thrash*

  The *thrash* command is used to explore paging behavior by allowing you to allocate a block of memory, then accessing the allocated memory to explore paging behavior.

- *ssdump*

  The *ssdump* program prints out formatted performance data that was collected while running *ssrun*. This program is included for SpeedShop debugging purposes. You will not 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 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 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 callstack data is used to attribute exclusive user time to the function at the bottom of each callstack (*i.e.*, 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 callstack at the time of the exception. *prof* generates a report showing inclusive and exclusive floating-point exception counts.

Libraries

Versions of the SpeedShop libraries *libss.so* and *libssrt.so* are available to support applications built using shared libraries (DSOs) only and 32-bit, n32 or 64-bit Bit ABIs.

- *libss.so*

  libss.so is a shared library (DSO) that supports *libssrt.so*. *libss.so* data normally appears in experiment results generated with *prof*.

- *libssrt.so*

  libssrt.so is a shared library (DSO) that gets linked in to the program you specify when you run an experiment. All the collection of performance data 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*

  Replaces the standard *libfpe.so* for the purposes of collecting floating point exception data. Click fpe_ss to view the reference page.

- *libmalloc_ss.so*

  Inserts versions of **malloc** routines from libc.so.1 that allow tracing all calls to **malloc**, **free**, **realloc**, **memalign**, and **valloc**. Click malloc_ss to view the reference page.

API

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

Supported Programs 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 MP-runtime, and displays the data accordingly.
- *pthreads*: Currently supported only with data on a per-process basis, not per-thread. The behavior of the *pthreads* library itself is monitored just like any other user-level code. Future releases of the SpeedShop tools will provide per thread support for `pthreads`.
- 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, and 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 then 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 the above 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 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 program_name
```

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 click ssusage to view the reference page.

Using *ssrun* and *prof*

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".

1. Build the executable. In most circumstances, you can build the executable as you would normally. See Chapter 6, "Building your Executable" for all the information you need 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 on running experiments over part of a program.

3. To collect performance data, call *ssrun* with the parameters below. Use the information in Table 1-1 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, click ssrun to view the reference page.
exp_type	Experiment name.
prog_name	Executable name.
prog_args	Arguments to the executable

Table 1-1 Choosing an Experiment Type

Performance Problem	Experiment(s) to Run
High user CPU time	**usertime** **pcsamp** ***_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 run of the executable, the experiment data is stored in a file with a name of the format *prog_name.exp_type.pid*

   For more information on the *ssrun* command, see Chapter 6, "Setting Up and Running Experiments: ssrun," or click ssrun to view the 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, click prof to view the 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 click prof to view the reference page.

Collecting Data for Part of a Program

If you have a performance problem in only one part of your program, you may want to collect performance data for just that part.You can do this by setting caliper points around the problem area when running an experiment, then using *prof -calipers* option to generate a report for the problem area.

You can set caliper points using one of the following approaches:

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

For more information on using calipers, see Chapter 6, "Setting Up and Running Experiments: ssrun."

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”
- “Tutorial Setup”
- “Analyzing Performance Data”

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 is based on a sample program called *generic*. There are three versions of the program: the *generic* directory contains files for the 32-bit ABI, the *generic64* directory contains files for the 64-bit ABI and *genericn32* directory contains files for the N 32-bit ABI. When you do the tutorial, choose the version of *generic* 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 version of *generic* in the *genericn32* directory.

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* that you will be using 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 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.
        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.
```

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

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

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

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

   `cd <your_dir>/generic`
4. Compile the program by typing

   `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`

Analyzing Performance Data

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

- “A usertime Experiment”
- “A pcsamp Experiment”
- “A Hardware Counter Experiment”
- “An ideal Experiment”
- “An fpe Trace”

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

A 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."

1. From the command line, type

   ```
   ssrun -usertime generic
   ```

   This starts the experiment. Output from *generic* and from *ssrun* will be printed to *stdout* as shown in the example below. A data file with a name generated by concatenating the process name, *generic*, the experiment type, *usertime*, and the process ID is also generated. In this example, the filename is *generic.usertime.24479*.

```
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.
        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.
```

2. To generate a report on the data collected, type the following at the command line:

   `prof` *<your_output_file_name>*

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

```
------------------------------------------------------------------------------
Profile listing generated Sun May 19 16:32:23 1996
    with:       prof generic.usertime.14427
------------------------------------------------------------------------------
        Total Time (secs)     : 21.51
```

```
        Total Samples        : 717
        Stack backtrace failed: 0
        Sample interval (ms)  : 30
        CPU                   : R4000
        FPU                   : R4010
        Clock                 : 150.0MHz
        Number of CPUs        : 1
-----------------------------------------------------------------------------
                                    caller/total       parents
index  %time   self descendents  total (self)     name
                                    callee/descend     children
-----------------------------------------------------------------------------
[1]    100.0%  0.00    21.51      717 (0)     __start [1]
               0.00    21.48      716/717         0x10001b44 main [2]
               0.03     0.00        1/717         0x10001b04 __readenv_sigfpe[20]
-----------------------------------------------------------------------------
               0.00    21.48      716/716        0x10001b44 __start [1]
[2]     99.9%  0.00    21.48      716 (0)    main [2]
               0.00    21.48      716/716        0x10001bd0 Scriptstring [3]
-----------------------------------------------------------------------------
               0.00    21.48      716/716        0x10001bd0 main [2]
[3]     99.9%  0.00    21.48      716 (0)    Scriptstring [3]
               0.00    18.69      623/716        0x10001f64 usrtime [4]
               0.00     1.50       50/716        0x10001f64 iofile [6]
               0.00     1.08       36/716        0x10001f64 libdso [10]
               0.00     0.15        5/716        0x10001e78 genLog [13]
               0.00     0.03        1/716        0x10001f64 dirstat [30]
               0.00     0.03        1/716        0x1000202c genLog [13]
-----------------------------------------------------------------------------
               0.00    18.69       623/623       0x10001f64 Scriptstring [3]
[4]     86.9%  0.00    18.69       623 (0)   usrtime [4]
              18.66     0.03       623/623       0x10004edc anneal [5]
-----------------------------------------------------------------------------
              18.66     0.03        623/623      0x10004edc usrtime [4]
[5]     86.9% 18.66     0.03        623 (622) anneal [5]
               0.00     0.03          1/1        0x10005ad0 init2da [27]
-----------------------------------------------------------------------------
               0.00     1.50         50/50       0x10001f64 Scriptstring [3]
[6]      7.0%  0.00     1.50         50 (0)   iofile [6]
               0.00     1.50         50/50       0x100025f4 fread [7]
-----------------------------------------------------------------------------
               0.00     1.50         50/50       0x100025f4 iofile [6]
[7]      7.0%  0.00     1.50         50 (0)   fread [7]
               0.00     1.44         48/50       0xfac80ec __filbuf [8]
               0.00     0.03          1/50       0xfac8060 _findbuf [33]
```

```
                0.03     0.00          1/50       0xfac8188 memcpy [18]
...
```

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 %time 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 **__readenv_sigfpe**.
- The descendents columns shows how much time, in seconds, was spent in callees of the function. For example, 21.48 seconds were spent in **main**.
- The caller / total, total (self), callee / descend column provides information on the number of cycles out of the total spent on the function, its callers and its callees. For example, the **anneal** function (index number 5) shows 623 / 623 for its caller (**usrtime**), 623(622) for itself, and 1 / 1 for its callee (**init2da**.)

A 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 instruction and function in your program. For more information on **pcsamp**, see the “pcsamp Experiment” section in Chapter 4, “Experiment Types.”

1. From the command line, type

   ```
   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* will be printed to *stdout* as shown in the example below. A data file with a name generated by concatenating the process name, *generic*, the experiment type, *pcsamp*, and the process ID is also generated. In this example, the filename is *generic.pcsamp.2277*.

```
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.
        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.
```

2. To generate a report on the data collected and redirect the output to a file, type 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.14480
-------------------------------------------------------------------------------
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/SShop/genericn32/generic:/usr/demos/SShop/genericn32/generic.c)
   1046     1s(  5.5)   19s( 98.8)   slaveusrtime
(/usr/demos/SShop/genericn32/dlslave.so:/usr/demos/SShop/genericn32/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/SShop/genericn32/generic:/usr/demos/SShop/genericn32/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/SShop/genericn32/generic:/usr/demos/SShop/genericn32/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
```

- 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.

A 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 "Hardware Counter Experiments" section in Chapter 4, "Experiment Types."

1. From the command line, type

   ```
   ssrun -dsc_hwc generic
   ```

   This starts the experiment. Output from *generic* and from *ssrun* will be printed to *stdout* as shown in the example below. A data file with a name generated by concatenating the process name, *generic*, the experiment type, *dsc_hwc*, and the process ID is also generated. In this example, the filename is *generic.dsc_hwc.5999*.

2. To generate a report on the data collected and redirect the output to a file, type 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 Sun May 19 17:35:21 1996
    with:       prof generic.dsc_hwc.5999
------------------------------------------------------------------------------
        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 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)
      2( 20.0)          6( 60.0)         anneal
(/usr/people/larry/generic64/generic:/usr/people/larry/generic64/generic.c)
      1( 10.0)          7( 70.0)       _findiop
(/usr/lib64/libc.so.1:/work/irix/lib/libc/libc_64_M4/stdio/flush.c)
```

```
      1( 10.0)          8( 80.0)        init2da
(/usr/people/larry/generic64/generic:/usr/people/larry/generic64/generic.c)
      1( 10.0)          9( 90.0) libss_timer_unset
(/usr/lib64/libssrt.so:../../ssrt/lib/sstimer.c)
      1( 10.0)         10(100.0)        _doprnt
(/usr/lib64/libc.so.1:/work/irix/lib/libc/libc_64_M4/print/doprnt.c)

     10(100.0)                      TOTAL
```

- 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 6: 2 from **anneal** and 4 from **memcpy**.
- The procedure (dso:file) column shows the procedure name and the DSO and filename that contain the procedure.

An 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.”

1. From the command line, type

   ```
   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, which are given a *.pixie* extension, and inserting information into the copies. Output from *generic* and from *ssrun* will be printed to *stdout* as shown in the example below. A data file with a name generated by concatenating the process name, *generic*, the experiment type, *ideal*, and the process ID is also generated. In this example, the filename is *generic.ideal.14517*

```
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 libraries, then the *generic* executable, and any DSOs that *generic* calls.

2. To generate a report on the data collected, type 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.14517

         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/SShop/genericn32/generic.c)
 135001332( 5.07)  99.88     0.90   75000822     1
slaveusrtime(./dlslave.so:/usr/demos/SShop/genericn32/dlslave.c)
   1593422( 0.06)  99.94     0.01    1378737  4382
memcpy(./libc.so.1:/work/irix/lib/libc/libc_n32_M3/strings/bcopy.s)
    735797( 0.03)  99.97     0.00     506627  4123
fread(./libc.so.1:/work/irix/lib/libc/libc_n32_M3/stdio/fread.c)
    187200( 0.01)  99.98     0.00     124800  1600
next(./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/SShop/genericn32/generic.c)
     91200( 0.00)  99.98     0.00      62400  1600
_drand48(./libc.so.1:/work/irix/lib/libc/libc_n32_M3/math/drand48.c)
```

```
     85497( 0.00)  99.99     0.00      56518     1
init2da(generic:/usr/demos/SShop/genericn32/generic.c)
     74095( 0.00)  99.99     0.00      28063   628
__sinf(./libm.so:/work/cmplrs/libm/fsin.c)
     56192( 0.00)  99.99     0.00       9360    16
offtime(./libc.so.1:/work/irix/lib/libc/libc_n32_M3/gen/time_comm.c)
     51431( 0.00)  99.99     0.00      36405    35
_doprnt(./libc.so.1:/work/irix/lib/libc/libc_n32_M3/print/doprnt.c)
     27951( 0.00) 100.00     0.00      19670   259
__filbuf(./libc.so.1:/work/irix/lib/libc/libc_n32_M3/stdio/_filbuf.c)
     21392( 0.00) 100.00     0.00      10136    58
fwrite(./libc.so.1:/work/irix/lib/libc/libc_n32_M3/stdio/fwrite.c)
     12744( 0.00) 100.00     0.00       9497   231
_readdir(./libc.so.1:/work/irix/lib/libc/libc_n32_M3/gen/readdir.c)
      9960( 0.00) 100.00     0.00       7536    96
_xflsbuf(./libc.so.1:/work/irix/lib/libc/libc_n32_M3/stdio/flush.c)
      7211( 0.00) 100.00     0.00       3959     1
dirstat(generic:/usr/demos/SShop/genericn32/generic.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 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 file name. For example, the first line reports statistics for the procedure **anneal** in the file *generic.c* in the generic executable.

An 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."

1. From the command line, type

   ```
   ssrun -fpe generic
   ```

   This starts the experiment. Output from *generic* and from *ssrun* will be printed to *stdout* as shown in the example below. A data file with a name generated by concatenating the process name, *generic*, the experiment type, *fpe*, and the process ID is also generated. In this example, the filename is *generic.fpe.2334*.

2. To generate a report on the data collected, type the following at the command line:

   `prof` *<your_output_file_name>*

   The report should look similar to the following partial listing.

```
------------------------------------------------------------------------------
Profile listing generated Tue May  7 19:21:30 1996
    with:prof generic.fpe.2334
------------------------------------------------------------------------------
Total FPEs           : 4
Stack backtrace failed: 0
CPU                  : R4000
FPU                  : R4010
Clock                : 150.0MHz
Number of CPUs       : 1
------------------------------------------------------------------------------
                                caller/total       parents
index  %FPEs     self descendents  total (self)    name
                                callee/descend      children

------------------------------------------------------------------------------
[1]    100.0%       0           4       4 (0)      __start [1]
                    0           4       4/4            0x10001aa0 main [2]
------------------------------------------------------------------------------
                    0           4       4/4            0x10001aa0 __start [1]
[2]    100.0%       0           4       4 (0)      main [2]
                    0           4       4/4            0x10001b2c Scriptstring
[3]
------------------------------------------------------------------------------
                    0           4       4/4            0x10001b2c main [2]
[3]    100.0%       0           4       4 (0)      Scriptstring [3]
                    4           0       4/4            0x10001ec0 fpetraps [4]
```

```
-----------------------------------------------------------------------------
                    4           0       4/4            0x10001ec0 Scriptstring
[3]
[4]    100.0%       4           0       4 (4)      fpetraps [4]
```

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 one callee. 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. For example, 4 floating point exceptions were found in the descendents of **main**.
- The caller / total, total (self), callee / descend column provides information on the number of floating point exceptions out of the total that were found.
- The parents, name, children column shows the function names, as described above.

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"
- "Tutorial Setup"
- "Analyzing Performance Data"

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 32-bit ABI, the *linpack64* directory contains files for the 64-bit ABI and the *linpackn32* directory contains files for the N32-bit ABI. Each *linpack* directory contains versions of the program for a single processor (*linpackup*) and for multiple processors (*linpackd*). When you do 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 *linpackn32* directory.

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

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

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

times are reported for matrices of order   300
      dgefa      dgesl      total     mflops       unit      ratio
 times for array with leading dimension of 301
  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
```

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

You need to 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 typing

   ```
   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:

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

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

A 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."

1. From the command line, type

   ```
   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* will be printed to *stdout* as shown in the example below. A data file with a name generated by concatenating the process name, *linpackup*, the experiment type, *usertime*, and the process ID is also generated. In this example, the filename is *linpackup.usertime.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
```

2. To generate a report on the data collected, type the following at the command line:

   `prof` *<your_output_file_name>*

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

```
------------------------------------------------------------------------------
Profile listing generated Sun May 19 18:31:51 1996
    with:       prof linpackup.usertime.17566
------------------------------------------------------------------------------
        Total Time (secs)     : 55.59
        Total Samples         : 1853
        Stack backtrace failed: 0
        Sample interval (ms)  : 30
        CPU                   : R8000
        FPU                   : R8010
        Clock                 :  75.0MHz
        Number of CPUs        : 1
```

```
-------------------------------------------------------------------------------
                                caller/total        parents
index  %time     self descendents  total (self)    name
                                callee/descend      children
-------------------------------------------------------------------------------
[1]     99.9%    0.00       55.56    1852 (0)      __start [1]
                 0.00       55.56    1852/1852         0x10000ea8 main [2]
-------------------------------------------------------------------------------
                 0.00       55.56    1852/1852         0x10000ea8 __start [1]
[2]     99.9%    0.00       55.56    1852 (0)      main [2]
                 0.00       55.56    1852/1852         0xae9c218 linp [3]
-------------------------------------------------------------------------------
                 0.00       55.56    1852/1852         0xae9c218 main [2]
[3]     99.9%    0.00       55.56    1852 (0)      linp [3]
                 0.44       30.40    1028/1852         0x10002488 dgefa [4]
                 0.04        3.08     104/1852         0x10001020 dgefa [4]
                 0.04        3.05     103/1852         0x1000185c dgefa [4]
                 0.04        3.05     103/1852         0x10001688 dgefa [4]
                 0.04        3.05     103/1852         0x10001a94 dgefa [4]
                 0.04        3.05     103/1852         0x1000207c dgefa [4]
                 0.04        3.05     103/1852         0x10001ea8 dgefa [4]
                 0.04        3.02     102/1852         0x10002250 dgefa [4]
                 1.41        0.00      47/1852         0x10002434 matgen [6]
                 0.00        0.30      10/1852         0x1000255c dgesl [7]
                 0.15        0.00       5/1852         0x10002210 matgen [6]
                 0.15        0.00       5/1852         0x1000181c matgen [6]
                 0.15        0.00       5/1852         0x10001e68 matgen [6]
                 0.12        0.00       4/1852         0x10001194 matgen [6]
                 ...
```

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 one callee. 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 0.44 of a second was spent in the first call to **dgefa**.

- The descendents columns shows how much time, in seconds, was spent in callees of the function. For example, in the third row, 55.56 seconds were spent in callees of **linp**.
- The caller / total, total (self), callee / descend column provides information on the number of cycles out of the total spent on the function, its callers and its callees. For example, the **linp** function (index number 3) shows 1852 / 1852 for its caller (**main**), 1852(0) for itself, and `1028/1852` for its first callee (**dgefa**.)

A 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."

1. From the command line, type

   ```
   ssrun -pcsamp linpackup
   ```

   This starts the experiment. Output from *linpackup* and from *ssrun* will be printed to *stdout* as shown in the example below. A data file with a name generated by concatenating the process name, *linpackup*, the experiment type, *pcsamp*, and the process ID is also generated. In this example, the filename is *linpackup.pcsamp.17576*.

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

2. To generate a report on the data collected, type 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.17576
------------------------------------------------------------------------------
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
```

- 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 **daxpy** in the file *linpackup.f* in the *linpackup* executable.

A 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“Hardware Counter Experiments” section in Chapter 4, “Experiment Types.”

1. From the command line, type

   ```
   ssrun -dsc_hwc linpackup
   ```

   This starts the experiment.

This starts the experiment. Output from *linpackup* and from *ssrun* will be printed to *stdout*. A data file with a name generated by concatenating the process name, *linpackup*, the experiment type, *dsc_hwc*, and the process ID is also generated. In this example, the filename is *linpackup.dsc_hwc.6180*.

2. To generate a report on the data collected, type 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 18:20:14 1996
    with:       prof linpackup.dsc_hwc.6180
------------------------------------------------------------------------------
        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/people/larry/linpack64/linpackup.f)
    307( 11.2)       2440( 89.1)         MATGEN
(linpackup:/usr/people/larry/linpack64/linpackup.f)
    275( 10.0)       2715( 99.2)          DGEFA
(linpackup:/usr/people/larry/linpack64/linpackup.f)
     11(  0.4)       2726( 99.6)         IDAMAX
(linpackup:/usr/people/larry/linpack64/linpackup.f)
      3(  0.1)       2729( 99.7)          DMXPY
(linpackup:/usr/people/larry/linpack64/linpackup.f)
      3(  0.1)       2732( 99.8)          DGESL
(linpackup:/usr/people/larry/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
```

- 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.

An 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.”

1. From the command line, type

   ```
   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, which are given a *.pixie* extension, and inserting information into the copies. Output from *linpackup* and from *ssrun* will be printed to *stdout* as shown in the example below. A data file with a name generated by concatenating the process name, *linpackup*, the experiment type, *ideal*, and the process ID is also generated. In this example, the filename is *linpackup.ideal.17580*.

```
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”
...
```

2. To generate a report on the data collected, type 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.17580

        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:linpackup.f)
 207582228( 3.63)  98.06     2.77  157405518    18 matgen(linpackup:linpackup.f)
  67844858( 1.19)  99.25     0.90   72325769    17 dgefa(linpackup:linpackup.f)
  19920277( 0.35)  99.60     0.27   17658342  5083 dscal(linpackup:linpackup.f)
  18115251( 0.32)  99.91     0.24   15675343  5083 idamax(linpackup:linpackup.f)
   4053920( 0.07)  99.98     0.05    3605124     1 dmxpy(linpackup:linpackup.f)
    786709( 0.01) 100.00     0.01     695776    17 dgesl(linpackup:linpackup.f)
     41357( 0.00) 100.00     0.00      83826  1116
__flsbuf(./libc.so.1:/work/irix/lib/libc/libc_n32_M4/stdio/_flsbuf.c)
     30294( 0.00) 100.00     0.00      29094     1 linp(linpackup:linpackup.f)
     17330( 0.00) 100.00     0.00      39823   867
x_putc(./libftn.so:/lv7/mtibuild/nodebug/workarea/mongoose/libI77/wsfe.c)
     12294( 0.00) 100.00     0.00      25795    28
x_wEND(./libftn.so:/lv7/mtibuild/nodebug/workarea/mongoose/libI77/wsfe.c)
     10620( 0.00) 100.00     0.00      14340    53
wrt_E(./libftn.so:/lv7/mtibuild/nodebug/workarea/mongoose/libI77/wrtfmt.c)
      9617( 0.00) 100.00     0.00      14889    71
do_fio64_mp(./libftn.so:/lv7/mtibuild/nodebug/workarea/mongoose/libI77/fmt.c)
      4940( 0.00) 100.00     0.00       7917   380
```

- 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 calls. For example, `98.06%` of all calls 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 `1797940023` instructions devoted to the **anneal** procedure.
- The `calls` column reports the number of calls to the procedure. For example, there were 772633 calls to the **daxpy** procedure.
- The `procedure (dso:file)` column lists the procedure, its DSO name and file name. 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 of the experiment types available within SpeedShop. It contains the following sections:

- “Selecting an Experiment”
- “usertime Experiment”
- “pcsamp Experiment”
- “Hardware Counter Experiments”
- “ideal Experiment”
- “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 below.

Table 4-1 Summary of Experiments

Experiment Type	Clues	Data Collected
usertime	Slow program, nothing else known. Not CPU-bound.	Inclusive and exclusive user time for each function by sampling the callstack at 30 millisecond intervals.
ideal	CPU-bound	Ideal CPU time at the function, source line and machine instruction levels using instrumentation for basic block counting.
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.
pcsamp	High user CPU time.	Actual CPU time at the source line, machine instruction and function levels by sampling the program counter at 10 or 1 millisecond intervals.
hwc	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.

usertime Experiment

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

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 seconds.

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 (*i.e.*,

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.

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 slowdown of execution of the program of no more than 15%. Data from a `usertime` experiment is statistical in nature and will show some variance from run to run.

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.

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. 16-bit bins allow a maximum of 65,000 counts, whereas a 32-bit bin allows 65,000 x 65,000 the number of counts. Using a 32-bit bin provides more accurate information, but requires additional disk space.

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 will exhibit variance inversely proportional to the running time.

This experiment can be used together with the ideal experiment to compare actual and ideal times spent in the CPU. A major discrepancy between PC Sample 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	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: on each processor clock cycle, when events for which there are hardware counters occur, each event is counted by incrementing the appropriate hardware counter. For example, when a floating point instruction is graduated in a cycle, the Graduated Floating-point Instruction counter is incremented by one.

There are two tools that allow you to access hardware counter data: *perfex*, a command-line interface that provides program-level event information, and SpeedShop, which allows you to perform the hardware counter experiments described below. For more information on *perfex*, and on hardware counters, click perfex or r10k_counters to view the reference pages.

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, it is reset to zero, and a count of how many overflows have occurred at the present program instruction address is incremented. 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.

[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 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:

Table 4-3 Hardware Counter Numbers

0	Cycles
1	Issued instructions

Table 4-3 (continued) Hardware Counter Numbers

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

Table 4-3 (continued) Hardware Counter Numbers

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

ideal Experiment

The `ideal` experiment instruments the executables and any DSOs to permit basic block counting and counting of all dynamic (function-pointer) calls. This involves dividing 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. Counter code is inserted at the beginning of each basic block to increment a counter every 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 a *.pixie* extension 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.

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.

The assumption made in calculating ideal CPU time is that each instruction takes exactly one cycle, and ignores potential floating point interlocks and memory latency time (cache misses and memory bus contention). Each system call is also assumed to take one cycle.

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.

Note: The assumption made in propagating times from a callee to a caller is that all calls are equivalent, so that the time attributed is divided equally for all calls. For some functions (**sin()**, for example), this assumption is plausible. For others (matrix multiply, 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*.

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 SpeedShop's *ssusage* command. Finding out the machine resources that your program uses can help you identify performance bottlenecks and the performance experiments you need to run. You can use Table 1-1 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	The 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`	The elapsed time during the command, in seconds.
`user`	The user CPU time in seconds.
`sys`	The 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 which 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"
- "Running Experiments"
- "Running Experiments on MPI Programs"
- "Using Calipers"
- "Effects of ssrun"
- "Customizing Your Environment"

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 will need to make changes to the way you build your executable or set certain environment variables. This section lists the conditions under which you may need to change the way you build your executable program. For information on setting environment variables, see section "Customizing Your Environment."

- If you have used the **ssrt_caliper_point** function provided in the SpeedShop libraries, you need to explicitly link in the SpeedShop libraries *libss.so* and *libssrt.so*. For more information on setting caliper points, see section "Using Calipers."
- If you are planning to build your executable using the **-32** option to *cc*, and you want to run the **usertime** experiment, you must add **-lexc** to the link line. For more information on *cc -32*, click cc to view the 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 since 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 it 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 is generally better to use compiler optimization level 2 or less when gathering an execution profile.
- 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 *linpack*, most of the time spent is in the routine **DAXPY**, which can be parallelized. In a 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_”.
  - In both 32-bit and 64-bit compilations with the **-g** option, for an **ideal** experiment, the source annotations behave differently and incorrectly 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_”) does not 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 **-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, and they are named using the following convention:

prog_name.exp_type.pid

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

You can change the default filename or directory for performance data files using environment variables.

- Set _SPEEDSHOP_OUTPUT_DIRECTORY to the directory you want to use if you want to generate performance data files in a directory other than the current working directory.
- Set _SPEEDSHOP_OUTPUT_FD to the number of the file descriptor to be used for writing the output file if you want to specify a file to which to write the performance data.
- Set _SPEEDSHOP_OUTPUT_FILENAME to the filename you want to use for recording performance data. If _SPEEDSHOP_OUTPUT_DIRECTORY is also specified, it is prepended to the filename you specify.

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 6-1 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_type 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 6-1 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.”

```
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.16064
```

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.16064
------------------------------------------------------------------------------
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/genericn32/generic.c)
111       1.1s(  5.4)   20s( 99.0)   slaveusrtime
(/usr/demos/SS/genericn32/dlslave.so:/usr/demos/SS/genericn32/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 your program.

The following steps assume you want 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 the section “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:

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.

   ```
   mpirun -np 6 exp_script
   ```

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."

You can set caliper points in three different ways:

- You can explicitly link with the SpeedShop runtime and call **ssrt_caliper_point** to record a caliper sample. This is useful when you want to set a caliper point at a specific location in a file.
- You can 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. This is useful if you want to be able to set a caliper point as your program is running.
- You can set a caliper sample trap in *dbx* or the WorkShop debugger. This is done by setting a breakpoint and when the process stops and evaluating the expression **libss_caliper_point(1)**. This is useful if you are working with a debugger in conjunction with SpeedShop.

An implicit caliper point is always present at the start of execution of the process. A final caliper-point is 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 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*

1. Insert calls to **ssrt_caliper_point** in your source code. You can insert one or more calls at any point in your code. The function call should be given the argument 1 (True.)

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

2. Link in the SpeedShop libraries *libss.so* and *libssrt.so* into your application. The libraries can be placed in any order 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

1. Set the_SPEEDSHOP_CALIPER_POINT_SIG variable to the signal number you want to use.

   The following are good choices because they do not have any semantics already associated with them. You must choose a signal that does not terminate the program. The signal should also not be caught by the target program, as this would interfere with its use for triggering a caliper point.

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

2. Run *ssrun* with your program.
3. Run 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 following operations are executed for all experiments:

- The following environment variables are set: _SPEEDSHOP_MARCHING_ORDERS, _SPEEDSHOP_EXPERIMENT_TYPE, and _SPEEDSHOP_TARGET_FILE. For more information on these variables, see "Customizing Your Environment."
- The SpeedShop libraries *libss.so* and *libssrt.so* are inserted as part of your executable using the environment variable _RLD_LIST.
- The target process is invoked 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:

- *libssrt.so* is inserted immediately in front of *libc.so.1* in the executable's library list.
- *ssrun* generates *.pixie* versions of all the libraries that the program uses, as well as the executable. The *.pixie* 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."

Customizing Your Environment

This section provides information on environment variables that you can use to customize your environment for *ssrun*.

- "Setting Environment Variables for Spawned Processes"
- "Setting General Environment Variables"

Setting Environment Variables for Spawned Processes

If your program spawns processes using *fork*, *exec*, or *sproc*, you can use the environment variables described in Table 6-2 to control whether performance data is collected for the spawned processes.

Table 6-2 Environment variables for Spawned Processes

Variable Name	Description
_SPEEDSHOP_TRACE_FORK	When set to True, specifies that processes spawned by calls to *fork* are monitored. It is set to True by default.
_SPEEDSHOP_TRACE_EXEC	When set to True, specifies that processes spawned by calls to *exec* are monitored. It is set to True by default.
_SPEEDSHOP_TRACE_SPROC	When set to True, specifies that processes spawned by calls to *sproc* are monitored. It is set to True by default.

Setting General Environment Variables

You can set any of following environment variables to customize the way that *ssrun* works. For a complete list of SpeedShop environment variables, click speedshop to view the reference page.

Table 6-3 Environment Variables for *ssrun*

Variable Name	Description
_SPEEDSHOP_VERBOSE	Writes a log of each program's operation to *stderr*. If it 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	When set to an empty or non-empty string, suppresses all output other than fatal error messages from SpeedShop. Overrides _SPEEDSHOP_VERBOSE.

Table 6-3 (continued) Environment Variables for *ssrun*

Variable Name	Description
_SPEEDSHOP_SAMPLING_MODE	Used for PC-sampling and hardware-counter profiling. If set to 1, it generates data for the base executable only. If it is not set, or set to anything other than 1, it generates data for the executable and all DSOs.
_SPEEDSHOP_INIT_DEFERRED_SIGNAL	When set to a signal number, the experiment is not initialized when the target process starts, but is delayed until the specified signal is sent to the process. A handler for the signal is installed when the process starts. You must ensure that the handler is not overwritten by the target code.
_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 0x400000.
_SPEEDSHOP_FILE_BUFFER_LENGTH	Defines the size of the buffer used for writing experiment files. The default length is 8kb. The buffer is used for writing small records to the file; large records are written directly, to avoid the buffering overhead.

There are two additional environment variables: _SPEEDSHOP_EXPERIMENT_TYPEE and _SPEEDSHOP_MARCHING_ORDERS. These variables are set by *ssrun* when you run an experiment. If you are using *ssrun*, you do not need to set the variables.

_SPEEDSHOP_EXPERIMENT_TYPE passes the name of the experiment to the runtime. ***ssrun*** parses the command line and generates marching orders for the experiment. The _SPEEDSHOP_MARCHING_ORDERS variable passes the marching orders of the experiment to the runtime. For examples of these variables, see "Example Using the -v Option."

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> ...
  ```

  This chapter focuses on the use of *prof* to generate reports from *ssrun* experiments.

- 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]
  ```

  You can find information on this use of *prof* in 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 7-1.

data_file One or more names of performance data files generated by *ssrun*. These files are usually of the format *prog_name.exp_type.pid*.

prof Options

Table 7-1 lists *prof* options. For more information, click prof to view the reference page.

Table 7-1 Options for *prof*

Name	Result
-calipers *n1 n2*	Restricts analysis to a segment of program execution. This option only works 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.
-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.
-dis[assemble]	Disassembles and annotates the analyzed object code with cycle times if you have run an **ideal** experiment collected data using *pixie*, or the number of PC samples if you have run a **pcsamp** 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.

Table 7-1 (continued) Options for *prof*

Name	Result
-dsolist	List all the DSOs in the program and their start and end text addresses.
–e[**xclude**] *proc1...procN*	Excludes information on the procedures specified. If you specify uppercase **–E**, *prof* also omits the specified procedures from the base upon which it calculates percentages.
–feedback	Produces files with information that can be used to (a) arrange procedures in the binary in an optimal ordering using *cord*, and (b) tell the compiler how to optimize compilation of the program using *cc -fb filename.cfb*. This option can be used when generating reports for **ideal** experiments, or for basic block counting data obtained with *pixie*. *cord* feedback files are named *program.fb* or *libso.fb*. Compiler feedback files are named *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*.
–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** experiments, or for basic block counting data obtained with *pixie*.
–i[**nvocations**]	Lists the number of times each procedure is invoked.
–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.
-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.
–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.

Table 7-1 (continued) Options for *prof*

Name	Result
–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, truncates everything after the line containing *n%* calls in the **%**calls column; *n*, an integer, followed by cum%, truncates everything after the line containing *n*cum% calls in the cum% column. That is, it truncates the listing after the last procedure 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 after the first line that represents less than 15% of the whole, and **-q 15cum%** truncates each part of the report after the line that brought the 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.
–z[**ero**]	Lists the procedures that are never invoked. This option can be used 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 and 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 %time column shows the cumulative percentage of time spent in each function. 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. For example, less than one hundredth of a second was spent in **__start**, but 0.03 of a second was spent in **__readenv_sigfpe**.
- The descendents columns shows how much time, in seconds, was spent in callees of the function. For example, 21.48 seconds were spent in **main**.
- The caller / total, total (self), callee / descend column provides information on the number of cycles out of the total spent on the function, its callers and its callees. For example, the **anneal** function (index number 5) shows 623 / 623 for its caller (**usrtime**), 623(622) for itself, and 1 / 1 for its callee (**init2da**.)

```
-------------------------------------------------------------------------------
Profile listing generated Sun May 19 16:32:23 1996
    with:       prof generic.usertime.14427
-------------------------------------------------------------------------------
        Total Time (secs)     : 21.51
        Total Samples         : 717
        Stack backtrace failed: 0
        Sample interval (ms)  : 30
        CPU                   : R4000
        FPU                   : R4010
        Clock                 : 150.0MHz
        Number of CPUs        : 1
-------------------------------------------------------------------------------
                                  caller/total       parents
index  %time   self descendents  total (self)     name
                                  callee/descend     children
-------------------------------------------------------------------------------
[1]    100.0%  0.00    21.51      717 (0)     __start [1]
               0.00    21.48      716/717         0x10001b44 main [2]
               0.03     0.00        1/717         0x10001b04 __readenv_sigfpe[20]
```

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 columns 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.14480
------------------------------------------------------------------------------
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/SShop/genericn32/generic:/usr/demos/SShop/genericn32/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. For example, the **anneal** function shows two overflows, but the cumulative number of overflows is 6: 2 from **anneal** and 4 from **memcpy**.
- 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.5999
-------------------------------------------------------------------------------
        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. 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 file name. 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.14517

        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/SShop/genericn32/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, 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 one callee. 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. For example, 4 floating point exceptions were found in the descendents of **main**.
- The caller / total, total (self), callee / descend column provides information on the number of floating point exceptions out of the total that were found.
- The parents, name, children column shows the function names, as described above.

```
-------------------------------------------------------------------------------
Profile listing generated Tue May  7 19:21:30 1996
    with:prof generic.fpe.2334
-------------------------------------------------------------------------------
Total FPEs            : 4
Stack backtrace failed: 0
CPU                   : R4000
FPU                   : R4010
Clock                 : 150.0MHz
Number of CPUs        : 1
-------------------------------------------------------------------------------
                                caller/total        parents
index  %FPEs     self descendents  total (self)    name
                                callee/descend      children
```

```
-------------------------------------------------------------------------------
[1]    100.0%       0           4       4 (0)      __start [1]
                    0           4       4/4            0x10001aa0 main [2]
-------------------------------------------------------------------------------
```

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 examples below show partial output from *prof*, showing just the machine instruction report.

This example shows output from calling *prof* in the following way:

`prof -dis generic.pcsamp.`*PID*

```
-------------------------------------------------------------------------------
* -dis[assemble] listing annotated pc-samples                                  *
*     Procedures with zero samples are excluded.                               *
-------------------------------------------------------------------------------

/usr/demos/SpeedShop/genericn32/generic.c
anneal: <0x10005aa0-0x10005d78>    1752 total samples(93.19%)
  [1448] 0x10005aa0     0x27bdff90      addiu   sp,sp,-112      # 1
  [1448] 0x10005aa4     0xffbf0020      sd      ra,32(sp)       # 2
  [1448] 0x10005aa8     0xffbc0018      sd      gp,24(sp)       # 3
  [1448] 0x10005aac     0x3c010002      lui     at,0x2  # 4
  [1448] 0x10005ab0     0x2421acb8      addiu   at,at,-21320    # 5
  [1448] 0x10005ab4     0x0039e021      addu    gp,at,t9        # 6
  [1450] 0x10005ab8     0xd7808040      ldc1    $f0,-32704(gp)  # 7
                <2 cycle stall for following instruction>
  [1450] 0x10005abc     0xf7a00010      sdc1    $f0,16(sp)      # 10
  [1452] 0x10005ac0     0x9f8281b0      lwu     v0,-32336(gp)   # 11
  [1452] 0x10005ac4     0x24010001      li      at,1    # 12
                <1 cycle stall for following instruction>
```

```
[1452] 0x10005ac8     0xac410000      sw      at,0(v0)        # 14
[1453] 0x10005acc     0x9f998184      lwu     t9,-32380(gp)   # 15
            <2 cycle stall for following instruction>
[1453] 0x10005ad0     0x0320f809      jalr    ra,t9   # 18
[1453] 0x10005ad4     0000000000      nop     # 19
            <2 cycle stall for following instruction>
[1461] 0x10005ad8     0xafa00008      sw      zero,8(sp)      # 22
[1461] 0x10005adc     0x8fa10008      lw      at,8(sp)        # 23
...
```

The listing shows statistics about the procedure **anneal** in the file *generic.c* and lists the beginning and ending addresses of **anneal**: `<0x1001c90c-0x1001e238>`

- The first column lists the line number of the instruction: `[1448]`
- The second column lists the beginning address of the instruction: `0x10005aa0`.
- The third column shows the instruction in hexadecimal: `0x27bdff90`.
- The next column reports the assembler form (mnemonic) of the instruction: `addiu sp,sp,-112`.
- The last column reports the cycle in which the instruction executed: `# 1`

Other information includes:

- The number of times an above branch was executed and taken:
  `Preceding branch executed 1 times, taken 0 times`
- 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:
  `2152 total cycles(0.00%) invoked 1 times, average 2152 cycles/invocation.`
- Any cycle stalls (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. The examples below show partial output from *prof*, showing just the source line report.

This example shows output from calling *prof* in the following way:

```
prof -S generic.ideal.PID
```

```
...
----------------------
disassembly listing
----------------------
...
                <2 cycle stall for following instruction>
  ^---      7 total cycles(0.00%) executed     1 times, average  7 cycles.---^
/usr/demos/SpeedShop/genericn32/generic.c
dirstat: <0x100022a8-0x100023a8>
    7211 total cycles(0.00%) invoked 1 times, average 7211 cycles/invocation
225: ^L
226: /* Simple routines to execute various types of behaviors */
227:
228: /*=====================================================*/
229: /* adddso -- add a DSO using sgidladd function */
230: #ifndef NONSHARED
231:
232: static     void    *dl_object = NULL;
233:
234: int
235: adddso()
236: {
237:            int i;
238:
239:            /* see if already linked */
240:            if(dl_object != NULL) {
241:                            fprintf(stderr, “libdso: dl_object already
linked\n”);
242:                            return 0;
243:            }
244:
245:            /* Log the event */
246:            genLog(“start of adddso”);
247:
248:            /* open the dynamic shared object */
249:            dl_object = sgidladd(DYNSONAME, RTLD_LAZY);
250:            if(dl_object == NULL) {
251:                            fprintf(stderr, “adddso: sgidladd of %s
failed--%s\n”,
252:                                            DYNSONAME, dlerror());
253:                            return 0;
254:            }
```

```
255:
256:            /* invoke the routine */
257:            i = dlslave_routine();
258:            fprintf(stderr, "\tadddso: dynamic routine returned %d\n", i);
259:
260:    return 0;
261: }
262: #endif
...
```

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 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.

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 execution of global procedures with the same name occurs 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* in the following way:

```
prof -gprof generic.ideal.14641

...
call graph profile:
          The sum of self and descendents is the major sort
          for this listing.

          function entries:

index     the index of the function in the call graph
          listing, as an aid to locating it.
cycles(%cycles)
          the total cycles (percentage of total) of the program
          accounted for by this function and its
          descendents.
self(%)
          cycles (percent of total) spent in this function
          itself.
kids(%)
          cycles (percent of total) spent in the descendents of
          this function on behalf of this function.
called    the number of times this function is called (other
          than recursive calls).
self      the number of times this function calls itself
          recursively.
name      the name of the function, with an indication of
          its membership in a cycle, if any.
index     the index of the function in the call graph
          listing, as an aid to locating it.

          parent listings:
```

```
self*
          cycles of this function's self time
          which is due to calls from this parent.
kids*
          cycles of this function's
          descendent time which is due to calls from this
          parent.
called**  the number of times this function is called by
          this parent.  This is the numerator of the
          fraction which divides up the function's time to
          its parents.
total*    the number of times this function was called by
          all of its parents.  This is the denominator of
          the propagation fraction.
parents   the name of this parent, with an indication of the
          parent's membership in a cycle, if any.
index     the index of this parent in the call graph
          listing, as an aid in locating it.

          children listings:

self*
          cycles of this child's self time
          which is due to being called by this function.
kids*
          cycles of this child's descendent's
          time which is due to being called by this
          function.
called**  the number of times this child is called by this
          function.  This is the numerator of the
          propagation fraction for this child.
total*    the number of times this child is called by all
          functions.  This is the denominator of the
          propagation fraction.
children  the name of this child, and an indication of its
          membership in a cycle, if any.
index     the index of this child in the call graph listing,
          as an aid to locating it.

          * these fields are omitted for parents (or
          children) in the same cycle as the function.  If
          the function (or child) is a member of a cycle,
          the propagated times and propagation denominator
          represent the self time and descendent time of the
          cycle as a whole.
```

```
          ** static-only parents and children are indicated
          by a call count of 0.

          cycle listings:
          the cycle as a whole is listed with the same
          fields as a function entry.  Below it are listed
          the members of the cycle, and their contributions
          to the time and call counts of the cycle.

          All times are in milliseconds.
-------------------------------------------------------------------------------
          NOTE: any functions which are not part of the call
                graph are listed at the end of the gprof listing
-------------------------------------------------------------------------------
                        self             kids        called/total    parents
index   cycles(%)       self(%)          kids(%)  called+self     name     index
                        self             kids        called/total    children
[1]  2661528080(99.95%)  71( 0.00%) 2661528009(100.00%) 0       __start [1]
                          44         2661527956         1/1         main [2]
                           5                  0         1/1         __istart [107]
                           4                  0         1/1
                                                            __readenv_sigfpe [108]
-------------------------------------------------------------------------------
                          44         2661527956         1/1         __start [1]
[2] 2661528000(99.95%)   44( 0.00%) 2661527956(100.00%) 1       main [2]
                        2152         2661524803         1/1         Scriptstring
[3]                       67                934         1/1         exit [55]
```

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 a 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.4561
```

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 the way in which the data files are named. The data files are named *prog_name.Counts.process_id*. This naming convention avoids the potential conflict of multiple threads attempting to write simultaneously to the same file.

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, click cc or cord to view the reference pages.

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 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 SIGALARM 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, click the word pixie to view the 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.
-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.
–[**no**]**autopixie**	Permits or prevents a recursive instrumenting of all dynamic shared libraries used by the input file during run time. *pixie* keeps the timestamp and checksum from the original executable. Thus, before instrumenting a shared library, *pixie* checks any *lib.pixie* 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 *.pixie* executable to work. The default behavior with shared libraries is **–noautopixie**. The default behavior with an executable is **–autopixie**.

Table 8-1 (continued) Options for *pixie*

Name	Result
-[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**.
-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.pixie*), containing additional code that counts the execution of each basic block. For example, if *myprog* uses *libc.so.1*, pixie generates *libc.so.1.pixie*.

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.

   Note: 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. Also it assumes that all instructions and data are in cache (for example, 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.

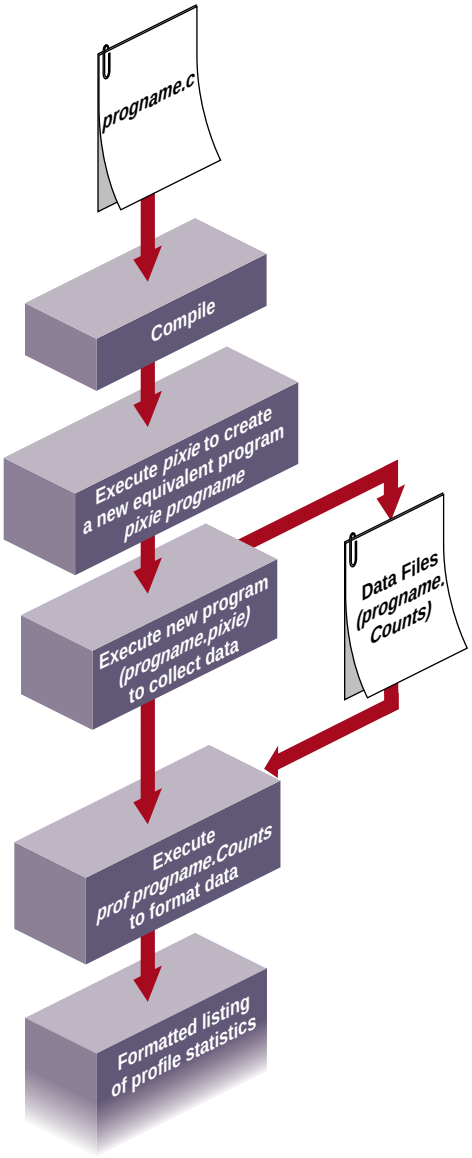

Figure 8-1 How Basic Block Counting Works

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/SS/genericn32/generic.c)
135001332( 5.07)   99.88    0.90  75000822     1
slaveusrtime(./dlslave.so:/usr/demos/SS/genericn32/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/SS/genericn32/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 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 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 file name. 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/genericn32/generic.c)
207945880( 7.81%)   94.56% 14440000  1464  anneal
(generic:/usr/demos/SpeedShop/genericn32/generic.c)
```

```
81000506( 3.04%)  97.60% 5000000       29  slaveusrtime
(dlslave.so:/usr/demos/SpeedShop/generic32/dlslave.c)
54000000( 2.03%)  99.63% 5000000       30  slaveusrtime
(dlslave.so:/usr/demos/SpeedShop/generic32/dlslave.c)
6600000( 0.25%)  99.88% 380000       1463  anneal
(generic:/usr/demos/SpeedShop/generic32/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/generic32/generic.c)
```

Example Using prof –quit

You can limit the output of *prof* to 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/genericn32/generic.c)
135001332( 5.07)   99.88     0.90    75000822       1
slaveusrtime(./dlslave.so:/usr/demos/SpeedShop/genericn32/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.” Use the following procedure to obtain inclusive basic clock counts. Also refer to Figure 8-1 which illustrates how inclusive 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.pixie*), containing additional code that counts the execution of each basic block. For example, if *myprog* uses *libc.so.1*, pixie generates *libc.so.1.pixie*.

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. The **–gprof** information is bundled in the *.Counts* file.

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.

Note: 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, 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*
   ```

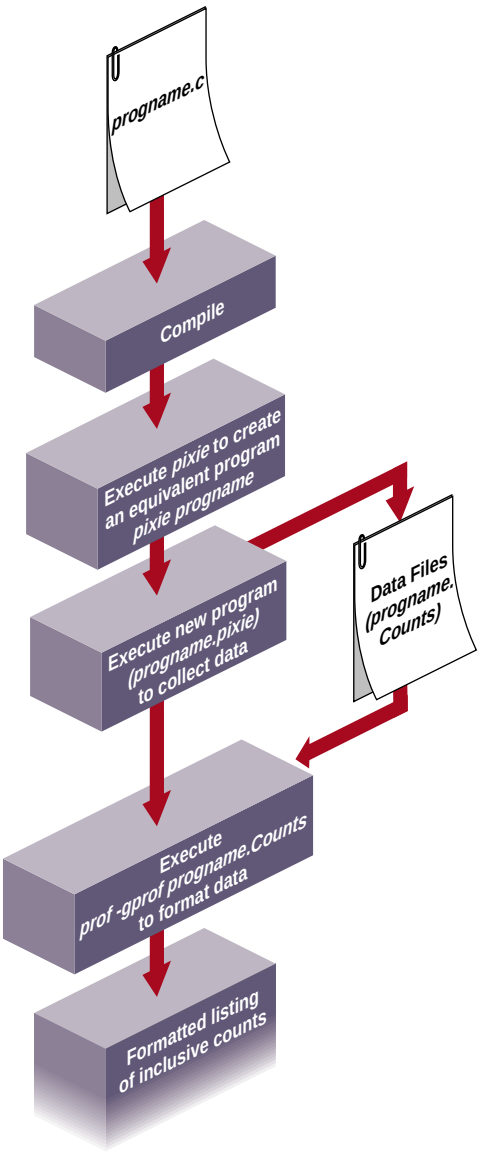

Figure 8-2 How Inclusive Basic Block Counting Works

Example of *prof –gprof*

Here's 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 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 section "Calculating the Working Set of a Program" below.

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 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* saying how much memory has been locked, and how much working memory is available.
- Sleeps indefinitely, or until interrupted by SIGINT or SIGTERM, at which time it frees up the memory and exits with an exit message.

You should 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.

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 4MB 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 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 4 and 5, 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* (~4MB), the kernel and any other applications you have running.

7. Deduct the 4MB that *thrash* uses from the amount of working memory reported by *squeeze* at the point the page faults began to rise to find out the approximate working set of the kernel and any other applications that are running on the machine. This number will be needed when you reach step 15.
8. 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 to find 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 to find 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.

-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.

`datafile1 ... datafileN`
If any of the named files contain beads indicating additional data files, from descendant processes of the original run, those files will also be dumped.

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, and a body whose length is given by the byte-count, rounded up to a double-word boundary.

The file prologue starts with a file-identifier bead, which acts as a magic-number indicating that the file is an SpeedShop data file. Other beads in the prologue give the machine and executable name, the hardware inventory describing the machine, the machine page size, O/S revision, date and checksum information about the executable, the target name (the target is the executable after instrumentation), the arguments with which the target was invoked, the instrumentation performed, and the types of performance data that are to be recorded in the remainder of the file. The following example calls *ssdump* on performance data for an **ideal** experiment.

```
ssdump generic.ideal.847
```

Here’s some partial output from *ssdump*. The format has been adjusted slightly to meet presentation needs.

```
Printing experiment record file “generic.ideal.847” (394024 bytes), last written
on Thu  16 May 96  23:09:55
SpeedShop File Preface                              1
        version 3
        created: Thu  16 May 96  23:09:28.520

Hardware Inventory                                  2, offset 32 = 0x00000020
        hardware inventory: 26 items
        class   1, type   1, contrlr  75, unit 255, state 13
        class   1, type   2, contrlr   0, unit   0, state 4130
...
Experiment name                                     3, offset 464 = 0x000001d0
        ideal

Experiment marching orders                          4, offset 480 = 0x000001e0
        it

Capture module symbol                               5, offset 496 = 0x000001f0
        it,4

Executable file                                     6, offset 512 = 0x00000200
        generic

Target file                                         7, offset 528 = 0x00000210
```

```
        generic.pixie

Target arguments                                    8, offset 552 = 0x00000228
        Time: Thu  16 May 96  23:09:28.520,  process pid = 847
        arguments: ""

Target begin                                        9, offset 592 = 0x00000250
        process # 76283912, pid = 847, event # 0
Program Object List                                10, offset 632 = 0x00000278
        process # 0, pid = 847, event # 0, -- 5 DSOs
        Program Object 0, Named `generic'
                      Link Time Address: 0x0000000000405a98
                       Run Time Address: 0x0000000000405a98
                                   Size: 0x000000000000c6a0 (50848)
                           Base Pointer: 0x0000000000000000

        Program Object 1, Named `./libss.so.pixie'
                      Link Time Address: 0x000000003f854b90
                       Run Time Address: 0x000000003f854b90
                                   Size: 0x00000000000012a0 (4768)
                           Base Pointer: 0x0000000000000000

        Program Object 2, Named `./libssrt.so.pixie'
                      Link Time Address: 0x000000003f564b94
                       Run Time Address: 0x000000003f564b94
                                   Size: 0x0000000000136010 (1269776)
                           Base Pointer: 0x0000000000000000
...

Target DSO open                                    11, offset 936 = 0x000003a8
        process # 0, pid = 847, event # 0
                at time = Thu  16 May 96  23:09:52.888
        fname = dlslave.so

Program Object List                                12, offset 992 = 0x000003e0
        process # 0, pid = 847, event # 0, -- 6 DSOs
        Program Object 0, Named `generic'
                      Link Time Address: 0x0000000000405a98
                       Run Time Address: 0x0000000000405a98
                                   Size: 0x000000000000c6a0 (50848)
                           Base Pointer: 0x0000000000000000
...
Sample event trigger                               13, offset 1360 = 0x00000550
    process 00007, trap index #0
                at time = Thu  16 May 96  23:09:54.476, #0
```

```
Event count array (32-bit)                          14, offset 1408 = 0x00000580
        integer array, dso index = 0, array size = 921

...
Function pointer trace                              20, offset 393592 = 0x00060178
        dso index = 0, 7 function pointer arcs

Function pointer trace                              21, offset 393776 = 0x00060230
        dso index = 2, 6 function pointer arcs

Sample data end marker                              22, offset 393936 = 0x000602d0

Target termination                                  23, offset 393984 = 0x00060300
        process # 0, pid = 847, event # 0
                at time = Thu  16 May 96  23:09:55.084


  ** End-of-File                                    24, offset 394024 = 0x00060328

**** End of experiment record file “generic.ideal.847”
at time = Thu  16 May 96  23:09:28.520
```

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, click cord or cc to view the 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	Result
-all	Prints feedback using all options. This is the default.
-ascii	Prints feedback information in the same style as earlier version of the feedback dump program.
-bb	Prints 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	Prints the 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	Prints the feedback file header as described in "cmplrs/fb.h".
-proc	Prints the 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	Prints the feedback file section headers table as described in "cmplrs/fb.h".
-str	Prints feedback string table.
-verbose	Prints all the information in verbose mode including a table with all zero entries.

Index

D

E

F

G

H

I

L

M

P

T

U

W

Tell Us About This Manual

As a user of Silicon Graphics documentation, your comments are important to us. They 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 to comment on:

- 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

Important Note

Please include the title and part number of the document you are commenting on. The part number for this document is
007-3311-001.

Thank you!

Three Ways to Reach Us

The **postcard** opposite this page has space for your comments. Write your comments on the postage-paid card for your country, then detach and mail it. If your country is not listed, either use the international card and apply the necessary postage or use electronic mail or FAX for your reply.

If **electronic mail** is available to you, write your comments in an e-mail message and mail it to either of these addresses:

- If you are on the Internet, use this address: techpubs@sgi.com
- For UUCP mail, use this address through any backbone site:
  [your_site]!sgi!techpubs

You can forward your comments (or annotated copies of pages from the manual) to Technical Publications at this **FAX** number:
415 965-0964