

SpeedShop User's Guide

007-3311-009

COPYRIGHT

© 1998–2002 Silicon Graphics, Inc. All rights reserved; provided portions may be copyright in third parties, as indicated elsewhere herein. No permission is granted to copy, distribute, or create derivative works from the contents of this electronic documentation in any manner, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

LIMITED RIGHTS LEGEND

The electronic (software) version of this document was developed at private expense; if acquired under an agreement with the USA government or any contractor thereto, it is acquired as "commercial computer software" subject to the provisions of its applicable license agreement, as specified in (a) 48 CFR 12.212 of the FAR; or, if acquired for Department of Defense units, (b) 48 CFR 227-7202 of the DoD FAR Supplement; or sections succeeding thereto. Contractor/manufacturer is Silicon Graphics, Inc., 1600 Amphitheatre Pkwy 2E, Mountain View, CA 94043-1351.

TRADEMARKS AND ATTRIBUTIONS

Silicon Graphics, SGI, the SGI logo, IRIX, Onyx2, and Origin are registered trademarks and OpenMP and ProDev are trademarks of Silicon Graphics, Inc.

MIPS, R4000, R4400, R5000, R14000, and R16000 are trademarks or registered trademarks of MIPS Technologies, Inc., R12000 is a trademark of MIPS Technologies, Inc., used under license by Silicon Graphics, Inc. UNIX is a registered trademark of the Open Group in the United States and other countries.

Cover design by Sarah Bolles, Sarah Bolles Design, and Dany Galgani, SGI Technical Publications.

New Features in this Guide

The `ssfilter` command, used to convert SpeedShop MPI experiment files to Vampir format, is no longer supported.

A new command, `sscompare`, is documented; this command is used to create comparison reports of data from experiment files. See "Comparing Experiment Results", page 132, for information about the `sscompare` command.

Information about a new tracing experiment for MPI has been added to Chapter 6, "Setting Up and Running Experiments: `ssrun`", page 71.

Record of Revision

Version	Description
005	August 1998 Brings the manual into conformance with the 1.3.2 version of the SpeedShop software.
006	April 1999 Supports the 1.4 version of the SpeedShop software.
007	October 2000 Supports the 1.4.1 version of the SpeedShop software.
008	November 2001 Supports the 1.4.3 version of the SpeedShop software.
009	September 2002 Supports the 1.4.4 version of the SpeedShop software.

Contents

About This Guide	xvii
Related Publications	xviii
Obtaining Publications	xix
Conventions	xix
Reader Comments	xix
1. Introduction to Performance Analysis	1
Sources of Performance Problems	1
Fixing Performance Problems	2
SpeedShop Tools	3
Commands	3
Experiment Types	4
SpeedShop Libraries	5
API	6
Supported Programming Models and Languages	6
Using SpeedShop Tools for Performance Analysis	7
Using <code>ssusage</code> to Evaluate Machine Resource Use	8
Gathering and Analyzing Performance Data	8
Collecting Data for Part of a Program	11
2. Tutorial for C Users	13
Tutorial Overview	13
Contents of the <code>generic</code> Program	14
Output from the <code>generic</code> Program	14
Tutorial Setup	15
007-3311-009	vii

Analyzing Performance Data	16
A usertime Experiment	16
Performing a usertime Experiment	16
Generating a Report	18
Analyzing the Report	19
A pcsamp Experiment	20
Generating a Report	21
Analyzing the Report	22
A Hardware Counter Experiment	23
Performing a Hardware Counter Experiment	23
Generating a Report	23
Analyzing the Report	24
A Basic Block Experiment	25
Performing a bbcounts Experiment	25
Generating a Report	27
Analyzing the Report	28
An fpe Trace	29
Performing an fpe Trace	29
Generating a Report	29
Analyzing the Report	30
3. Tutorial for Fortran Users	33
Tutorial Overview	34
Tutorial Setup	35
Analyzing Performance Data	35
A usertime Experiment	36
Performing a usertime Experiment	36

Generating a Report	37
Analyzing the Report	38
A <code>pcsamp</code> Experiment	39
Performing a <code>pcsamp</code> Experiment	39
Generating a Report	40
Analyzing the Report	41
A Hardware Counter Experiment	41
Performing a Hardware Counter Experiment	42
Generating a Report	42
Analyzing the Report	43
A <code>bbcounts</code> Experiment	44
Performing a <code>bbcounts</code> Experiment	44
Generating a Report	45
Analyzing the Report	47
MPI Tracing Tutorial	47
4. Experiment Types	51
Selecting an Experiment	51
Floating-Point Exception Trace Experiment (<code>fpe</code>)	53
Heap Trace Experiment (<code>heap</code>)	53
Hardware Counter Experiments (<code>*_hwc, *_hwctime</code>)	54
Two Tools for Hardware Counter Experiments	54
<code>_hwc</code> Hardware Counter Experiments	55
<code>_hwctime</code> Hardware Counter Experiments	57
Hardware Counter Numbers	59
Basic Block Counting Experiment (<code>bbcounts</code>)	62
How SpeedShop Prepares Files	62

How SpeedShop Calculates CPU Time for <code>bbcounts</code> Experiments	63
Inclusive Basic Block Counting	63
Using <code>pcsamp</code> and <code>bbcounts</code> Together	64
I/O Trace Experiment (<code>io</code>)	65
MPI Call Tracing Experiment (<code>mpi/mpi_trace</code>)	66
NUMA Profiling Experiment (<code>numa</code>)	66
PC Sampling Experiment (<code>pcsamp</code>)	67
Call Stack Profiling Experiment (<code>usertime/totaltime</code>)	68
5. Collecting Data on Machine Resource Usage	69
<code>ssusage</code> Syntax	69
<code>ssusage</code> Results	69
6. Setting Up and Running Experiments: <code>ssrun</code>	71
Building Your Executable	71
Special Information for MP Fortran Programs	72
Setting Up Output Directories and Files	73
Using Run-Time Environment Variables	74
User Environment Variables	74
Process Tracking Environment Variables	76
Expert-Mode Environment Variables	77
Using Marching Orders	78
Defining the Base Experiment	79
Running Experiments	81
<code>ssrun</code> Syntax	81
<code>ssrun</code> Examples	83
Example Using the <code>pcsampx</code> Experiment	83

Example Displaying Data in WorkShop	84
Example Using the <code>-v</code> Option	85
Using <code>ssrun</code> with a Debugger	85
Running Experiments on MPI Programs	86
Generating MPI Tracing Experiments	86
Generating Other Experiments for Programs Using MPI	90
Running Experiments on Programs Using Pthreads	91
Running Experiments on Programs That Use OpenMP Directives	91
Using Calipers	92
Setting Calipers with the <code>ssrt_caliper_point</code> Function	93
Setting Time-Oriented Calipers	94
Setting Calipers with Signals	95
Setting Calipers with a Debugger	96
Effects of <code>ssrun</code>	96
7. Analyzing Experiment Results	99
Using <code>prof</code> to Generate Performance Reports	99
<code>prof</code> Arguments	99
<code>prof</code> Options	100
<code>prof</code> Output	105
Using <code>prof</code> with <code>ssrun</code>	105
<code>usertime</code> Experiment Reports	106
<code>pcsamp</code> Experiment Reports	107
Hardware Counter Experiment Reports	108
<code>bbcounts</code> Experiment Reports	110
<code>fpe</code> Trace Reports	113
Using <code>prof</code> Options	114

Using the <code>-dis</code> Option	114
Using the <code>-S</code> Option	120
Using the <code>-calipers</code> Option	123
Using the <code>-butterfly</code> Option	124
Generating Reports for Different Machine Types	128
Generating Reports for Multiprocessed Executables	129
Determining Program Overhead	129
Generating Compiler Feedback Files	132
Comparing Experiment Results	132
8. Miscellaneous Commands	135
Using the <code>thrash</code> Command	135
<code>thrash</code> Syntax	135
Effects of <code>thrash</code>	136
Using the <code>squeeze</code> Command	136
<code>squeeze</code> Syntax	136
Effects of <code>squeeze</code>	137
Calculating the Working Set of a Program	137
Combining Multiple Experiment Files into One	139
Glossary	143
Index	147

Figures

Figure 3-1	An MPI Experiment in <code>cvperf</code>	49
Figure 6-1	MPI Numerical Format	89

Tables

Table 1-1	Letter Codes in Process Experiment ID Numbers	9
Table 4-1	Summary of Experiments	52
Table 4-2	R10000 Hardware Counter Numbers	59
Table 4-3	R12000, R14000, R16000 Hardware Counter Numbers	61
Table 4-4	Basic Block Counts and PC Profile Counts Compared	64
Table 6-1	Setting Caliper Points	93
Table 7-1	Options for prof	100

About This Guide

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

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

This book is intended for experienced programmers and others who are interested in optimizing program performance.

The following chapters are included in this book:

- Chapter 1, "Introduction to Performance Analysis", page 1, provides a general introduction to performance analysis concepts and techniques, plus an overview of the SpeedShop tools.
- Chapter 2, "Tutorial for C Users", page 13, provides a tutorial on how to collect performance data and generate reports for a C program.
- Chapter 3, "Tutorial for Fortran Users", page 33, provides a tutorial on how to collect performance data and generate reports for Fortran programs running on single-processor machines.
- Chapter 4, "Experiment Types", page 51, describes the types of experiments that can be performed using SpeedShop tools.
- Chapter 5, "Collecting Data on Machine Resource Usage", page 69, describes how to use the `ssusage(1)` command to collect information about a program's machine resource usage.
- Chapter 6, "Setting Up and Running Experiments: `ssrun`", page 71, explains in detail how to set up and run experiments using `ssrun(1)`, and explains how to use caliper points to generate reports for part of a program.
- Chapter 7, "Analyzing Experiment Results", page 99, explains how to generate reports from performance data using `prof(1)` and `sscompare(1)`.

- Chapter 8, "Miscellaneous Commands", page 135, explains how to use the `thrash(1)` and `squeeze(1)` commands to determine the memory usage, or working set, of your application. It also includes commands to print performance data files.

Related Publications

The following documents contain additional information that may be helpful:

- *Guide to SGI Compilers and Compiling Tools*
- *C Language Reference Manual*
- *MIPSpro C++ Programmer's Guide*
- *ProDev WorkShop: Debugger User's Guide*
- *ProDev WorkShop: Performance Analyzer User's Guide*
- *ProDev WorkShop: Overview*
- *ProDev WorkShop: Static Analyzer User's Guide*
- *ProDev WorkShop: ProMP User's Guide*
- *MIPSpro Fortran 77 Programmer's Guide*
- *MIPSpro Fortran 77 Language Reference Manual*
- *MIPSpro Fortran Language Reference Manual, Volume 1*
- *MIPSpro Fortran Language Reference Manual, Volume 2*
- *MIPSpro Fortran Language Reference Manual, Volume 3*
- *MIPSpro Fortran 90 Commands and Directives Reference Manual*
- *MIPSpro N32/64 Compiling and Performance and Tuning Guide*
- *Origin 2000 and Onyx2 Performance Tuning and Optimization Guide*
- *MPI Programmer's Manual*

Obtaining Publications

To obtain SGI documentation, go to the SGI Technical Publications Library at <http://techpubs.sgi.com>.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
<code>command</code>	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
<i>variable</i>	Italic typeface denotes variable entries and words or concepts being defined.
user input	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
[]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.
GUI	This font denotes the names of graphical user interface (GUI) elements such as windows, screens, dialog boxes, menus, toolbars, icons, buttons, boxes, fields, and lists.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact us in any of the following ways:

- Send e-mail to the following address:

`techpubs@sgi.com`

- Use the Feedback option on the Technical Publications Library World Wide Web page:

`http://techpubs.sgi.com`

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.
- Send mail to the following address:

Technical Publications
SGI
1600 Amphitheatre Pkwy., M/S 535
Mountain View, California 94043-1351

- Send a fax to the attention of "Technical Publications" at +1 650 932 0801.

We value your comments and will respond to them promptly.

Introduction to Performance Analysis

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

- "Sources of Performance Problems", page 1, provides a general overview of potential performance problems.
- "Fixing Performance Problems", page 2, shows you how to use SpeedShop to isolate and fix performance problems.
- "SpeedShop Tools", page 3, describes SpeedShop commands, experiment types, and libraries.
- "Using SpeedShop Tools for Performance Analysis", page 7, shows you how to analyze your system performance.

Sources of Performance Problems

To tune a program's performance, you must first determine where machine resources are being used. At any point in a process, there is one limiting resource controlling the speed of execution. Processes can be slowed down by:

- CPU speed and availability: a *CPU-bound* process spends its time executing in the CPU and is limited by CPU speed and availability. To improve the performance of 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 processing: an *I/O-bound* process has to wait for input/output (I/O) to complete. I/O 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 size and availability:** a program that continuously needs to swap out pages of memory is called *memory-bound*. Page thrashing is often due to accessing virtual memory on a haphazard rather than strategic basis; cache misses result. Insufficient memory bandwidth could also be the problem.

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 may find that you are reading in 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`).
- **Performance phases:** because 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 phase. Performance analysis is an iterative process.
- **Cache thrashing:** If an application does not access CPU caches efficiently, the application will run slower while the CPU and operating system reload cache entries.

Fixing Performance Problems

The SpeedShop tools described in this manual can help you to identify specific performance problems described later; these techniques are only a part of performance tuning. You can also tune graphics, I/O, the kernel, system parameters, memory, and real-time system calls. For a complete guide to all performance tools and the documentation about those tools, see *Guide to SGI Compilers and Compiling Tools*.

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

SpeedShop Tools

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

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

Commands

SpeedShop provides the following commands to help you analyze your programs:

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

SpeedShop provides the following additional commands:

- `squeeze`: Allocates a region of virtual memory and locks the virtual memory down into real memory, making it unavailable to other processes.
- `thrash`: Allows you to allocate a block of memory, then access the allocated memory to explore system paging behavior.

Experiment Types

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

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

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

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

Hardware counter experiments are available on R10000, R12000, R14000, and R16000 systems that have built-in hardware counters. Data is measured by counting each time the specified hardware counter exceeds its maximum value, or overflows. You can specify the hardware counter and the overflow interval you want to use. For more information on the hardware counter experiments, see "Hardware Counter Experiments (*_hwc, *_hwctime)", page 54.

- `usertime` experiments display a program's CPU time by statistical call-stack profiling.

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

- The `totaltime` experiment returns wall-clock time in a manner identical to that of the `usertime` experiment. It uses statistical callstack profiling, based on wall-clock time, with a time sample interval of 30 milliseconds.
- `bbcounts` experiments display an estimated time based on linear basic blocks counting.

Data is measured by counting the number of executions of each *basic block* and calculating an estimated time for each function. This involves instrumenting the program to divide the code into basic blocks, which are consecutive sequences of instructions with a single entry point, a single exit point, and no branches into or out of the sequence. Instrumentation also records a count of all dynamic (function-pointer) calls.

Because an exact count of every instruction in your program is recorded, you can also use the `bbcounts` experiment to determine the efficiency of your algorithm and identify any code that is not executed.

- `fpe` experiments trace floating-point exceptions.

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

SpeedShop Libraries

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

The following list describes the different SpeedShop libraries.

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

- `libpixrt.so`: A shared library (DSO) used by programs that have been instrumented for basic block counting.

API

The SpeedShop application programming interface (API) allows you to use the `ssrt_caliper_point` function to set caliper points in your source code. See "Using Calipers", page 92, for information on using caliper points. For information on other API functions, see the `ssapi(3)` man page.

Supported Programming Models and Languages

The SpeedShop tools support programs with the following characteristics:

- Shared libraries (DSOs).
- Unstripped executables.
- Executables that call `fork(2)`, `sproc(2)`, `system(3F)`, or `exec(2)`.
- Executables using supported techniques for opening, closing, and delay-loading DSOs.
- C, C++, Fortran (Fortran 77 and Fortran 90), or Ada (1.4.2 and older versions) source code.
- Power Fortran and Power C source code. `prof` understands the syntax and semantics of the multiprocessing run time and displays the data accordingly.
- `pthread`s, supported with data on a per-program basis.
- Message Passing Interface (MPI) or other message-passing paradigms. Currently supported by providing data on the behavior of each process. The behavior of the MPI library itself is monitored just like any other user-level code. See the *MPI Programmer's Manual* for details about the MPI library.
- The OpenMP collection of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism.

Using SpeedShop Tools for Performance Analysis

Performance tuning typically consists of:

1. Examining machine resource usage
2. Breaking down the process into phases
3. Identifying the resource bottleneck within each phase
4. Correcting the cause of the bottleneck

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

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

1. Build the application.
2. Run experiments on the application to collect performance data.
3. Examine the performance data.
4. Generate an improved version of the program.
5. Compare performance of improved version of the program against the previous version. To do this, use the `sscompare` command to compare the new version to the previous version to verify that improvements are being made.
6. Repeat steps 1 through 5 as needed.

To accomplish this using SpeedShop tools, do the following:

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

Using `ssusage` to Evaluate Machine Resource Use

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

```
ssusage executable_name executable_args
```

From the `ssusage` output, you can decide which experiments to run to collect data for further study. For more information on `ssusage`, see Chapter 5, "Collecting Data on Machine Resource Usage", page 69, or see the `ssusage(1)` man page.

Gathering and Analyzing Performance Data

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

To perform a performance analysis, follow these general steps:

1. Build the executable.

You can usually build the executable as you would normally. See "Building Your Executable", page 71, for information on how to build the executable.

2. Specify caliper points if you want to analyze data for only a portion of your program.
3. To collect performance data, issue the `ssrun` command with the following parameters:

```
% ssrun ssrun_options -exp_type executable_name executable_args
```

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

- *ssrun_options*: zero or more valid options. For a complete list of options, see the `ssrun(1)` man page.
- *exp_type*: experiment name.
- *executable_name*: executable name.
- *executable_args*: arguments to the executable.

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

- **High-user CPU time:** `usertime`, `pcsamp` (four variants), `_hwc/_hwctime` (hardware counter experiments), or `bbcounts`.
- **High-system CPU time:** if floating-point exceptions are suspected, run an `fpe` trace.
- **High I/O time:** `bbcounts`, then examine counts of I/O routines.
- **High paging rates:** `bbcounts`, then `prof -cordfb` and `cord` to rearrange procedures.

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

executable_name.exp_type.id

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

`generic.pixbb.m10966`

See the following table for letter codes and descriptions.

Table 1-1 Letter Codes in Process Experiment ID Numbers

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

For more information on the `ssrun` command, see Chapter 6, "Setting Up and Running Experiments: `ssrun`", page 71, or see the `ssrun(1)` man page.

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

‣ `prof options data_file`

- *options*: one or more valid options. For a complete list of options, see the `prof(1)` man page or "prof Options", page 100.
 - *data_file*: the name of the file in which the experiment data was recorded.
5. The `sscompare` command can be used to analyze the performance data in experiment files that were generated by SpeedShop tools such as `ssrun`, and produce a comparison report. When comparing application performance, make sure to make a copy of the original binary code and a copy of the original experiment file. Then you can compare the original experiment results with the newer (hopefully improved) results.

The following are some useful comparisons:

- application performance before and after optimization
- multiple ranks in an MPI application
- multiple threads in an OpenMP applications
- different experiments for the same application

The comparison report produced by `sscompare` contains a legend and a table of performance data. Each input file and the type of performance data it contains is listed in the legend with a numeric column key. The table contains multiple columns of data; the type of data is dependent on the options used to generate the report.

`sscompare` can be used with the following SpeedShop experiment types:

- `usertime`
- `pcsamp`
- `bbcounts`

See the `sscompare(1)` man page or "Comparing Experiment Results", page 132, for more details.

Collecting Data for Part of a Program

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

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

- Direct calls to the SpeedShop API.
- The caliper signal environment.
- A debugger such as the ProDev WorkShop debugger.
- Periodic caliper points with pollpoint caliper points.

For more information on using calipers, see "Using Calipers", page 92.

Tutorial for C Users

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

- "Tutorial Overview", page 13, introduces the sample program and explains the different scenarios in which it will be used.
- "Tutorial Setup", page 15, steps you through the necessary setup for running the experiment.
- "Analyzing Performance Data", page 16, steps you through five different experiments, discussing first how to do the experiments, then how to interpret the results.

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

The C tutorial demonstrates the following experiments:

- `usertime`
- `pcsamp`
- basic block count (`bbcounts`)
- floating point exception (`fpe`)

For an example of an MPI tracing tutorial and a hardware counter experiment, see Chapter 3, "Tutorial for Fortran Users", page 33.

Tutorial Overview

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

- `generic` directory : contains files for the n32-bit ABI

- `generico32` directory: contains files for the (old) 32-bit ABI
- `generic64` directory: contains files for the 64-bit ABI

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

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

Contents of the `generic` Program

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

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

Output from the `generic` Program

Output from the program looks like the following:

```
0:00:00.000 ===== (27173)          Begin script Fri  06 Feb 1998
15:03:31.
    begin script 'll.u.cvt.d.i.f.dso'
0:00:00.002 ===== (27173)          start of linklist Fri  06 Feb 2002
15:03:31.
    linklist completed.
0:00:00.003 ===== (27173)          start of usertime Fri  06 Feb 2002
15:03:31.
    usertime completed.
0:00:25.572 ===== (27173)          start of cvttrap Fri  06 Feb 2002
```

```
15:03:57.
    cvttrap completed, y = 2.60188e+14, z = 2.60188e+14.
0:00:25.806 ===== (27173)          start of dirstat Fri  06 Feb 2002
15:03:57.
    dirstat of /usr/include completed, 304 files.
0:00:26.618 ===== (27173) start of iofile -- stdio Fri  06 Feb 2002
15:03:58.
    stdio iofile on /unix completed, 7307988 chars.
0:00:26.864 ===== (27173)          start of fpetraps Fri  06 Feb 2002
15:03:58.
    fpetraps completed.
0:00:26.865 ===== (27173)          start of libdso Fri  06 Feb 2002
15:03:58.
dlslave_init executed
dlslave_routine executed
    slaveusertime completed, x = 5000000.000000.
    libdso: dynamic routine returned 13
    end of script 'll.u.cvt.d.i.f.dso'
0:00:27.972 ===== (27173)          End script Fri  06 Feb 2002
15:03:59.
```

Tutorial Setup

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

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

```
% cp -r generic your_dir
```

3. Change to the directory you just created:

```
% cd your_dir/generic
```

4. Compile the program, by entering:

```
% make all
```

This provides an executable for the experiment.

Analyzing Performance Data

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

- `usertime`. As a first cut at optimization, this may be the most useful experiment. It breaks down a program into its functions and returns the CPU time used in each. See "A `usertime` Experiment", page 16.
- `pcsamp`. This experiment uses a different method to return the CPU time. See "A `pcsamp` Experiment", page 20.
- `dsc_hwc`. This experiment counts the number of times a required data item was not in secondary data cache. If the data item is not in secondary data cache, it must be fetched from memory, which requires more time. See "A Hardware Counter Experiment", page 23.
- `bbcounts`. This experiment counts basic block usage and estimates a linear time. It also maps out a complete call graph. See "A Basic Block Experiment", page 25.
- `fpe`. This experiment counts the number of floating-point exceptions in each function. See "An `fpe` Trace", page 29.

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

A `usertime` Experiment

This section explains how to perform a `usertime` experiment. The `usertime` experiment allows you to gather data on the amount of CPU time spent in each function in your program.

Note: Due to statistical sampling of the call stack, not all functions may appear in the experiment output.

For more information on `usertime`, see "Call Stack Profiling Experiment (`usertime/totaltime`)", page 68.

Performing a `usertime` Experiment

From the command line, enter the following:

```
% ssrun -usertime generic
```

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

```
0:00:00.000 ===== (16957)          Begin script Mon  18 Mar 2002
06:56:38.
      begin script 'll.u.cvt.d.i.f.dso'
0:00:00.004 ===== (16957)          start of linklist Mon  18 Mar 2002
06:56:38.
      linklist completed.
0:00:00.005 ===== (16957)          start of usertime Mon  18 Mar 2002
06:56:38.
      usertime completed.
0:00:18.736 ===== (16957)          start of cvttrap Mon  18 Mar 2002
06:56:57.
      cvttrap completed, y = 2.60188e+14, z = 2.60188e+14.
0:00:18.906 ===== (16957)          start of dirstat Mon  18 Mar 2002
06:56:57.
      dirstat of /usr/include completed, 264 files.
0:00:18.941 ===== (16957) start of iofile -- stdio Mon  18 Mar 2002
06:56:57.
      stdio iofile on /unix completed, 7965088 chars.
0:00:20.426 ===== (16957)          start of fpetraps Mon  18 Mar 2002
06:56:59.
      fpetraps completed.
0:00:20.428 ===== (16957)          start of libdso Mon  18 Mar 2002
06:56:59.
dlslave_init executed
dlslave_routine executed
      slaveusertime completed, x = 5000000.000000.
      libdso: dynamic routine returned 13
      end of script 'll.u.cvt.d.i.f.dso'
0:00:21.217 ===== (16957)          End script Mon  18 Mar 2002
06:56:59.
```

Generating a Report

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

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

In this example, *your_output_file_name* would be `generic.usertime.m10981`. The `prof` command prints results to `stdout`.

Note: Because of line width restrictions, the DSO, file name, and line number information at the end of each line is wrapped to the next line in the following sample output.

```
-----  
SpeedShop profile listing generated Mon Mar 18 07:00:30 2002  
  prof generic.usertime.m16957  
    generic (n32): Target program  
      usertime: Experiment name  
      ut:cu: Marching orders  
  R5000 / R5000: CPU / FPU  
      1: Number of CPUs  
      180: Clock frequency (MHZ.)  
  
Experiment notes--  
  From file generic.usertime.m16957:  
Caliper point 0 at target begin, PID 16957  
  /speedtest/generic/generic  
Caliper point 1 at exit(0)  
-----  
Summary of statistical callstack sampling data (usertime)--  
      664: Total Samples  
      0: Samples with incomplete traceback  
  19.920: Accumulated Time (secs.)  
      30.0: Sample interval (msecs.)  
-----  
Function list, in descending order by exclusive time  
-----  
[index]  excl.secs  excl.%   cum.%   incl.secs  incl.%   samples  procedure (dso: file, line)  
  
  [3]    18.570  93.2%   93.2%   18.570  93.2%    619  anneal (generic: generic.c, 1570)  
  [5]     0.750   3.8%   97.0%    0.750   3.8%     25  slaveusertime (dlslave.so: dlslave.c, 22)  
  [8]     0.420   2.1%   99.1%    0.420   2.1%     14  __read (libc.so.1: read.s, 20)
```

[12]	0.150	0.8%	99.8%	0.150	0.8%	5	cvtttrap (generic: generic.c, 318)
[13]	0.030	0.2%	100.0%	0.030	0.2%	1	_ngetdents (libc.so.1: ngetdents.s, 16)
[1]	0.000	0.0%	100.0%	19.920	100.0%	664	main (generic: generic.c, 102)
[2]	0.000	0.0%	100.0%	19.920	100.0%	664	Scriptstring (generic: generic.c, 185)
[4]	0.000	0.0%	100.0%	18.570	93.2%	619	usertime (generic: generic.c, 1385)
[14]	0.000	0.0%	100.0%	0.030	0.2%	1	dirstat (generic: generic.c, 349)
[15]	0.000	0.0%	100.0%	0.030	0.2%	1	_readdir (libc.so.1: readdir.c, 135)
[9]	0.000	0.0%	100.0%	0.420	2.1%	14	iofile (generic: generic.c, 462)
[10]	0.000	0.0%	100.0%	0.420	2.1%	14	fread (libc.so.1: fread.c, 27)
[11]	0.000	0.0%	100.0%	0.420	2.1%	14	_read (libc.so.1: readSCI.c, 27)
[6]	0.000	0.0%	100.0%	0.750	3.8%	25	libdso (generic: generic.c, 623)
[7]	0.000	0.0%	100.0%	0.750	3.8%	25	dlslave_routine (dlslave.so: dlslave.c, 7)

Analyzing the Report

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

- The `index` column assigns a reference number to each function.
- The `excl.secs` column shows how much time, in seconds, was spent in the function itself (exclusive time). Routines that begin with an underscore, such as `__start`, are internal routines that you cannot change.
- The `excl.%` column shows the percentage of a program's total time that was spent in the function.
- The `cum.%` column shows the percentage of the complete program time that has executed in the routines listed so far.
- The `incl.secs` column shows how much time, in seconds, was spent in the function and descendents of the function.
- The `incl.%` column shows the cumulative percentage of inclusive time spent in each function and its descendents.
- The `samples` column shows how many samples were taken when the process was executing in the function and in all of the function's descendents.
- The `procedure (dso:file,line)` columns list the function name, its DSO name, its file name, and its line number. For example, the top line reports statistics for the function `anneal`, the DSO `generic`, in the file `generic.c`, at line 1570.

A pcsamp Experiment

This section explains how to perform a `pcsamp` experiment. The `pcsamp` experiment allows you to gather information on actual CPU time for each function in your program. For more information on `pcsamp`, see "PC Sampling Experiment (`pcsamp`)", page 67.

From the command line, enter the following:

```
% ssrun -fpcsamp generic
```

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

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

```
0:00:00.000 ===== (16969)          Begin script Mon  18 Mar 2002
07:02:19.
      begin script 'll.u.cvt.d.i.f.dso'
0:00:00.005 ===== (16969)          start of linklist Mon  18 Mar 2002
07:02:19.
      linklist completed.
0:00:00.008 ===== (16969)          start of usertime Mon  18 Mar 2002
07:02:19.
      usertime completed.
0:00:18.260 ===== (16969)          start of cvttrap Mon  18 Mar 2002
07:02:37.
      cvttrap completed, y = 2.60188e+14, z = 2.60188e+14.
0:00:18.430 ===== (16969)          start of dirstat Mon  18 Mar 2002
07:02:37.
      dirstat of /usr/include completed, 264 files.
0:00:18.464 ===== (16969) start of iofile -- stdio Mon  18 Mar 2002
07:02:37.
      stdio iofile on /unix completed, 7965088 chars.
0:00:18.813 ===== (16969)          start of fpetraps Mon  18 Mar 2002
07:02:38.
      fpetraps completed.
0:00:18.815 ===== (16969)          start of libdso Mon  18 Mar 2002
07:02:38.
```



```

dlslave_init executed
dlslave_routine executed
    slaveusertime completed, x = 5000000.000000.
    libdso: dynamic routine returned 13
    end of script 'll.u.cvt.d.i.f.dso'
0:00:19.577 ===== (16969)                End script Mon  18 Mar 2002
07:02:39.

```

Generating a Report

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

```
% prof your_output_file_name > pcsamp.results
```

Output similar to the following is generated:

```
-----
SpeedShop profile listing generated Mon Mar 18 07:03:54 2002
```

```

prof generic.fpcsamp.m16969
    generic (n32): Target program
    fpcsamp: Experiment name
pc,2,1000,0:cu: Marching orders
R5000 / R5000: CPU / FPU
    1: Number of CPUs
    180: Clock frequency (MHz.)

```

Experiment notes--

```

From file generic.fpcsamp.m16969:
Caliper point 0 at target begin, PID 16969
  /speedtest/generic/generic
Caliper point 1 at exit(0)

```

```
-----
Summary of statistical PC sampling data (fpcsamp)--
```

```

    19329: Total samples
    19.329: Accumulated time (secs.)
    1.0: Time per sample (msecs.)
    2: Sample bin width (bytes)

```

```
-----
Function list, in descending order by time
-----
```

[index]	secs	%	cum.%	samples	function (dso: file, line)
[1]	18.084	93.6%	93.6%	18084	anneal (generic: generic.c, 1570)
[2]	0.716	3.7%	97.3%	716	slaveusrtime (dlslave.so: dlslave.c, 22)
[3]	0.329	1.7%	99.0%	329	__read (libc.so.1: read.s, 20)
[4]	0.147	0.8%	99.7%	147	cvtttrap (generic: generic.c, 318)
[5]	0.031	0.2%	99.9%	31	_xstat (libc.so.1: xstat.s, 12)
[6]	0.012	0.1%	99.9%	12	__write (libc.so.1: write.s, 20)
[7]	0.004	0.0%	100.0%	4	fread (libc.so.1: fread.c, 27)
[8]	0.002	0.0%	100.0%	2	iofile (generic: generic.c, 462)
[9]	0.001	0.0%	100.0%	1	fprintf (libc.so.1: fprintf.c, 23)
[10]	0.001	0.0%	100.0%	1	_cerror (libc.so.1: cerror.s, 30)
	0.002	0.0%	100.0%	2	**OTHER** (includes excluded DSOs, rld, etc.)
	19.329	100.0%	100.0%	19329	TOTAL

Analyzing the Report

The report has the following columns:

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

A Hardware Counter Experiment

Note: This experiment can be performed only on systems that have built-in hardware counters (machines with the R10000, R12000, R14000, or R16000 class of CPU).

This section takes you through the steps to perform a hardware counter experiment. There are a number of hardware counter experiments, but this tutorial describes the steps involved in performing the `dsc_hwc` experiment. This experiment captures information about secondary data cache misses. For more information on hardware counter experiments, see "Hardware Counter Experiments (`*_hwc`, `*_hwctime`)", page 54.

Performing a Hardware Counter Experiment

From the command line, enter:

```
% ssrun -dsc_hwc generic
```

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

Generating a Report

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

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

The report should look similar to the following listing:

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

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

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

          786 100.0% 100.0%     6  TOTAL
```

Analyzing the Report

The information immediately preceding the function list displays the following:

- The `Total samples` is the number of times the program counter was sampled. It is sampled once for each *overflow*, or once each time the hardware counter exceeds the specified value.
- The `Counter name (number)` indicates the hardware counter used in the experiment. In this case, hardware counter 26 counts the number of times a value required in a calculation was not available in secondary cache. For a complete list of the hardware counters and their numbers, see Table 4-2, page 59.
- The `Counter overflow value` is the number at which the hardware counter overflows or exceeds its preset value. In this case, the value is 131, which is the default. The `fdsc_hwc` experiment runs the same hardware counter experiment with the preset value of 29. You can change the overflow value by setting the `_SPEEDSHOP_HWC_COUNTER_OVERFLOW` environment variable to a value larger than 0, the `_SPEEDSHOP_HWC_COUNTER_NUMBER` environment variable to 26, and

running the `prof_hwc` experiment instead of `dsc_hwc`. See "`_hwc` Hardware Counter Experiments" to learn how to choose a counter overflow value.

- The `Total counts` is the total number of times a value was not in secondary cache when needed. This value is determined by multiplying the total number of samples by the overflow value; extra counts that do not cause an overflow are not recorded.

The function list has the following columns:

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

A Basic Block Experiment

This section takes you through the steps to perform an `bbcounts` experiment. The times returned represent an idealized computation. This experiment ignores potential floating-point interlocks and memory latency time (cache misses and memory bus contention). The times returned will always be lower than the times for an actual run. For more information on the `bbcounts` experiment, see "Basic Block Counting Experiment (`bbcounts`)", page 62.

Performing a `bbcounts` Experiment

From the command line, enter

```
% ssrun -bbcunts generic
```

This starts the experiment. First the executable, `rld`, and the DSOs are instrumented. This entails making copies of the libraries and executables, giving the copies an extension of `.pixie`.

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

```
instrumenting /lib32/rld
instrumenting /usr/lib32/libssrt.so
instrumenting /usr/lib32/libss.so
instrumenting /usr/lib32/libm.so
instrumenting /usr/lib32/libc.so.1
instrumenting /speedtest/generic/generic

0:00:00.001 ===== (16991)          Begin script Mon  18 Mar 2002
07:05:46.
    begin script `ll.u.cvt.d.i.f.dso'
0:00:00.016 ===== (16991)          start of linklist Mon  18 Mar 2002
07:05:46.
    linklist completed.
0:00:00.025 ===== (16991)          start of usertime Mon  18 Mar 2002
07:05:46.
    usertime completed.
0:00:19.943 ===== (16991)          start of cvttrap Mon  18 Mar 2002
07:06:06.
    cvttrap completed, y = 2.60188e+14, z = 2.60188e+14.
0:00:20.201 ===== (16991)          start of dirstat Mon  18 Mar 2002
07:06:06.
    dirstat of /usr/include completed, 264 files.
0:00:20.245 ===== (16991) start of iofile -- stdio Mon  18 Mar 2002
07:06:06.
    stdio iofile on /unix completed, 7965088 chars.
0:00:20.623 ===== (16991)          start of fpetraps Mon  18 Mar 2002
07:06:07.
    fpetraps completed.
0:00:20.631 ===== (16991)          start of libdso Mon  18 Mar 2002
07:06:07.
    instrumenting /speedtest/generic/./dlslave.so
```

```

dlslave_init executed
dlslave_routine executed
    slaveusertime completed, x = 5000000.000000.
    libdso: dynamic routine returned 13
    end of script 'll.u.cvt.d.i.f.dso'
0:00:22.058 ===== (16991)                End script Mon  18 Mar 2002
07:06:08.

```

The output statements beginning with “instrumenting declares that `ssrun` is instrumenting first the libraries and then the `generic` executable itself.

Generating a Report

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

```
% prof your_output_file_name > bbcounts.results
```

This command redirects output to a file called `bbcounts.results`. The file contains results that look similar to the following partial listing. The number of functions and their names may also vary.

```

-----
SpeedShop profile listing generated Mon Mar 18 07:07:06 2002
  prof generic.bbcounts.m16991
    generic (n32): Target program
      bbcounts: Experiment name
        it:cu: Marching orders
      R5000 / R5000: CPU / FPU
        1: Number of CPUs
        180: Clock frequency (MHz.)
  Experiment notes--
  From file generic.bbcounts.m16991:
  Caliper point 0 at target begin, PID 16991
    /speedtest/generic/generic
  Caliper point 1 at exit(0)
-----
Summary of ideal time data (bbcounts)--
  2048459432: Total number of instructions executed
  3266522347: Total computed cycles
    18.147: Total computed execution time (secs.)
    1.595: Average cycles / instruction
-----

```

Function list, in descending order by exclusive ideal time

```
-----
```

[index]	excl.secs	excl.%	cum.%	cycles	instructions	calls	function (dso: file, line)
[1]	17.304	95.4%	95.4%	3114690027	1956780024	1	anneal (generic.pixbb: generic.c, 1570)
[2]	0.694	3.8%	99.2%	125000842	75000732	1	slaveusrtime (dlslave.so: dlslave.c, 22)
[3]	0.139	0.8%	99.9%	25000068	15000054	1	cvtttrap (generic.pixbb: generic.c, 318)
[4]	0.002	0.0%	100.0%	348146	324941	1286	general_find_symbol (rld: rld.c, 2038)
[5]	0.002	0.0%	100.0%	337349	305518	2874	resolve_relocations (rld: rld.c, 2636)
[6]	0.001	0.0%	100.0%	148338	148338	1301	elfhash (rld: obj.c, 1184)
[7]	0.001	0.0%	100.0%	95076	95076	4138	obj_dynsym_got (rld: objfcn.c, 46)
[8]	0.000	0.0%	100.0%	88459	88459	1663	strcmp (rld: strcmp.s, 34)
[9]	0.000	0.0%	100.0%	78410	69944	1	fix_all_defined (rld: rld.c, 3419)
[10]	0.000	0.0%	100.0%	76859	74309	1289	resolve_symbol (rld: rld.c, 1828)
[11]	0.000	0.0%	100.0%	75590	58123	1	init2da (generic.pixbb: generic.c, 1427)
[12]	0.000	0.0%	100.0%	72817	71565	1256	resolving (rld: rld.c, 1499)
[13]	0.000	0.0%	100.0%	67753	62361	487	fread (libc.so.1: fread.c, 27)
[14]	0.000	0.0%	100.0%	48270	45600	53	_doprnt (libc.so.1: doprnt.c, 227)
[15]	0.000	0.0%	100.0%	48000	35200	1600	_drand48 (libc.so.1: drand48.c, 116)
[16]	0.000	0.0%	100.0%	38783	27864	628	__sinf (libm.so: fsin.c, 97)
[17]	0.000	0.0%	100.0%	34408	34320	6	search_for externals (rld: rld.c, 3987)
.							
.							
.							
.							

Analyzing the Report

The columns in the report provide the following information:

- The `index` column assigns a reference number to each function.
- The `excl.secs` column shows the minimum number of seconds that might be spent in the function under ideal conditions.
- The `excl.%` column shows how much of the program’s total time was spent in the function.
- The `cum.%` column shows the cumulative percentage of time spent in the functions listed so far.
- The `cycles` column shows the total number of machine cycles used by the function.

- The `instructions` column shows the total number of instructions executed by a function.
- The `calls` column shows the total number of calls made to the function.
- The `function (dso:file, line)` columns list the function, its DSO name, its file name, and the line number.

An `fpe` Trace

This section takes you through the steps to perform a floating-point exception (`fpe`) trace, which identifies functions in which floating-point exceptions have occurred. For more information on the `fpe` trace, see "Floating-Point Exception Trace Experiment (`fpe`)", page 53.

Performing an `fpe` Trace

From the command line, enter:

```
% ssrun -fpe generic
```

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

Generating a Report

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

```
% prof your_output_file_name > fpe.results
```

The report should look similar to the following partial listing:

```
-----  
SpeedShop profile listing generated Mon Mar 18 07:09:16 2002  
  prof generic.fpe.m17002  
    generic (n32): Target program  
      fpe: Experiment name  
      fpe:cu: Marching orders  
R5000 / R5000: CPU / FPU  
      1: Number of CPUs  
      180: Clock frequency (MHz.)
```

```
Experiment notes--
  From file generic.fpe.ml7002:
  Caliper point 0 at target begin, PID 17002
    /speedtest/generic/generic
  Caliper point 1 at exit(0)
-----
Summary of FPE callstack tracing data (fpe)--
      4: Total FPEs
      0: Samples with incomplete traceback
-----
Function list, in descending order by exclusive FPEs
-----
[index]  excl.FPEs  excl.%  cum.%  incl.FPEs  incl.%  function (dso: file, line)

      [1]         4 100.0% 100.0%         4 100.0% fpetraps (generic: generic.c, 406)
      [2]         0  0.0% 100.0%         4 100.0% main (generic: generic.c, 102)
      [3]         0  0.0% 100.0%         4 100.0% Scriptstring (generic: generic.c, 185)
```

Analyzing the Report

The report shows information for each function:

- The `index` column assigns a reference number to each function.
- The `excl.FPEs` column shows how many floating-point exceptions were found in the function. .
- The `excl.%` column shows the percentage of the total number of floating-point exceptions that were found in the function.
- The `cum.%` column shows the percentage of exclusive floating-point exceptions in the functions that have been listed so far. The list is sorted by the number of floating-point exceptions, with the most in the top line and the least in the bottom line. Because all of the exceptions are in the first function listed in this example, all entries in this column are 100%.
- The `incl.FPEs` column shows how many floating-point exceptions were generated by the function and the functions it called.

- The `incl.%` column shows the percentage of the program's total number of floating-point exceptions in this function and the functions it called. Because `fpetraps` is called through all of the other functions, they are all listed as 100%.
- The `function (dso:file, line)` columns list the routine name, its DSO name, its file name, and its line number.

Tutorial for Fortran Users

This chapter provides two tutorials for using the SpeedShop tools to gather and analyze performance data in a Fortran program. There are three versions of the first program:

- The `linpack` directory contains files for the n32-bit ABI.
- The `linpack64` directory contains files for the 64-bit ABI.
- The `linpacko32` directory contains files for the o32-bit ABI.

The first tutorial covers the following topics:

- "Tutorial Overview", page 34, introduces the sample program and explains the different scenarios in which it will be used.
- "Tutorial Setup", page 35, leads you through the necessary setup for running the experiment.
- "Analyzing Performance Data", page 35, steps you through different experiments, discussing first how to do the experiments, then how to interpret the results.

The second tutorial creates a Message Passing Interface (MPI) experiment. The experiment file is generated by SpeedShop and displayed by the WorkShop performance analyzer. See "MPI Tracing Tutorial", page 47.

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

The Fortran tutorial demonstrates the following experiments:

- `usertime`
- `pcsamp`
- hardware counters
- basic block count (`bbcounts`)
- MPI trace experiment

For an example of a floating point exception experiment (fpe), see Chapter 2, "Tutorial for C Users", page 13.

Tutorial Overview

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

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

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

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

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

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

times for array with leading dimension of 300
3.800E+00  4.000E-02  3.840E+00  4.734E+00  4.224E-01  6.857E+01
3.810E+00  4.000E-02  3.850E+00  4.722E+00  4.235E-01  6.875E+01
```

```
3.770E+00 4.000E-02 3.810E+00 4.772E+00 4.191E-01 6.804E+01
3.782E+00 4.000E-02 3.822E+00 4.757E+00 4.205E-01 6.825E+01
```

Tutorial Setup

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

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

```
% cp -r linpack your_dir
```

3. Change to the directory you just created:

```
% cd your_dir/linpack
```

4. Compile the program by entering:

```
% make all
```

This provides an executable for the experiment.

Note: You must APO installed in order to build these files. For sales and licensing information, contact your SGI sales representative.

Analyzing Performance Data

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

- The `usertime` experiment. It returns the *CPU time* used by each routine in your program. See "A `usertime` Experiment", page 36.
- The `pcsamp` experiment. It returns CPU time for each routine in your program. See "A `pcsamp` Experiment", page 39.
- The `dsc_hwc` (secondary data cache hardware counter) experiment. In a hardware counter experiment, the program counter is sampled every time a hardware

counter exceeds a specified limit. In the experiment performed in this section, the hardware counter keeps track of the number of times a data item required in a calculation was not present in secondary data cache. When a data item is not in cache, it must be retrieved from memory, which is a more time-consuming process. See "A Hardware Counter Experiment", page 41.

- The `bbcounts` experiment. This experiment calculates the best time achievable. See "A `bbcounts` Experiment", page 44.

A `usertime` Experiment

This section lists the steps you need to perform a `usertime` experiment. The `usertime` experiment allows you to gather data on the amount of CPU time spent in each routine in your program. For more information, see "Call Stack Profiling Experiment (`usertime/totaltime`)", page 68.

Performing a `usertime` Experiment

From the command line, enter the following:

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

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

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

```
ssrun: target PID 18819
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
ssrun: setenv _RLDN32_LIST libss.so:libssrt.so:DEFAULT
ssrun: setenv _RLD64_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
3.010E+00  3.000E-02  3.040E+00  5.980E+00  3.344E-01  5.429E+01
3.010E+00  3.000E-02  3.040E+00  5.980E+00  3.344E-01  5.429E+01
3.010E+00  3.000E-02  3.040E+00  5.980E+00  3.344E-01  5.429E+01
3.010E+00  3.000E-02  3.040E+00  5.980E+00  3.344E-01  5.429E+01

```

```

times for array with leading dimension of 300
3.020E+00  3.000E-02  3.050E+00  5.961E+00  3.355E-01  5.446E+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
3.024E+00  3.000E-02  3.054E+00  5.953E+00  3.360E-01  5.454E+01

```

Generating a Report

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

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

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

Note: Lines have been wrapped because of line width restrictions.

```
-----
SpeedShop profile listing generated Tue Mar 19 06:52:47 2002
```

```

prof linpackup.usertime.ml8819
      linpackup (n32): Target program
      usertime: Experiment name
      ut:cu: Marching orders

```

3: Tutorial for Fortran Users

```

R5000 / R5000: CPU / FPU
      1: Number of CPUs
      180: Clock frequency (MHz.)

Experiment notes--
  From file linpackup.usertime.m18819:
  Caliper point 0 at target begin, PID 18819
    /speedlin/linpack/linpackup
  Caliper point 1 at exit(0)
-----
Summary of statistical callstack sampling data (usertime)--
      1960: Total Samples
          0: Samples with incomplete traceback
      58.800: Accumulated Time (secs.)
          30.0: Sample interval (msecs.)
-----
Function list, in descending order by exclusive time
-----
[ index]  excl.secs  excl.%   cum.%   incl.secs  incl.%   samples  procedure  (dso: file, line)

   [4]    54.600  92.9%   92.9%    54.600  92.9%    1820  daxpy (linpackup: linpackup.f, 495)
   [5]     1.920   3.3%   96.1%     1.920   3.3%     64  matgen (linpackup: linpackup.f, 199)
   [3]     1.800   3.1%   99.2%    56.250  95.7%    1875  dgefa (linpackup: linpackup.f, 221)
   [7]     0.300   0.5%   99.7%     0.300   0.5%     10  idamax (linpackup: linpackup.f, 700)
   [8]     0.120   0.2%   99.9%     0.120   0.2%     4  dscal (linpackup: linpackup.f, 670)
   [9]     0.030   0.1%   99.9%     0.030   0.1%     1  dmxpy (linpackup: linpackup.f, 826)
  [10]     0.030   0.1%  100.0%     0.030   0.1%     1  _type_f (libftn.so: fmt.c, 761)
   [1]     0.000   0.0%  100.0%    58.800 100.0%    1960  main (libftn.so: main.c, 76)
   [2]     0.000   0.0%  100.0%    58.800 100.0%    1960  linp (linpackup: linpackup.f, 3)
   [6]     0.000   0.0%  100.0%     0.570   1.0%     19  dgesl (linpackup: linpackup.f, 324)
  [11]     0.000   0.0%  100.0%     0.030   0.1%     1  do_fioxr8v (libftn.so: fmt.c, 1603)
  [12]     0.000   0.0%  100.0%     0.030   0.1%     1  do_fio64_mp (libftn.so: fmt.c, 626)

```

Analyzing the Report

The report shows information for each function.

- The `index` column, which enumerates the routines in the program, provides an index number for reference.
- The `excl.secs` column shows how much time, in seconds, was spent in the routine itself (exclusive time).

- The `excl.%` column shows the percentage of a program's total time that was spent in the function. For example, the `daxpy` routine consumed 92.9% of the program's time.
- The `cum.%` column shows the percentage of the complete program time that has been spent in the routines that have been listed so far.
- The `incl.secs` column shows how much time, in seconds, was spent in the function and descendents of the function.
- The `incl.%` column shows the cumulative percentage of inclusive time spent in each routine and its descendents.
- The `samples` column provides the number of samples taken from the function and all of its descendents.
- The `function (dso:file, line)` column lists the routine name, its DSO name, its file name, and its line number.

Note: Many functions shown here have only one or two hits. The data for those functions is not statistically significant. (Routines that begin with an underscore, such as `__start`, are internal routines that you cannot change.)

A `pcsamp` Experiment

This section lists the steps you need to perform a `pcsamp` experiment. The `pcsamp` experiment allows you to gather information on actual CPU time for each source code line, machine line, and function in your program. For more information on `pcsamp`, see "PC Sampling Experiment (`pcsamp`)", page 67.

Performing a `pcsamp` Experiment

From the command line, enter the following:

```
% ssrun -pcsamp linpackup
```

This starts the experiment.

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

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

Generating a Report

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

```
% prof your_output_file_name > pcsamp.results
```

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

```
-----  
SpeedShop profile listing generated Tue Mar 19 07:02:22 2002  
  prof linpackup.pcsamp.ml8866  
    linpackup (n32): Target program  
      pcsamp: Experiment name  
    pc,2,10000,0:cu: Marching orders  
      R5000 / R5000: CPU / FPU  
        1: Number of CPUs  
        180: Clock frequency (MHz.)  
  
Experiment notes--  
  From file linpackup.pcsamp.ml8866:  
  Caliper point 0 at target begin, PID 18866  
    /speedlin/linpack/linpackup  
  Caliper point 1 at exit(0)  
  
-----  
Summary of statistical PC sampling data (pcsamp)--  
      5669: Total samples  
    56.690: Accumulated time (secs.)  
     10.0: Time per sample (msecs.)  
       2: Sample bin width (bytes)  
  
-----  
Function list, in descending order by time  
-----  
[index]      secs      %      cum.%      samples  function (dso: file, line)
```

[1]	53.050	93.6%	93.6%	5305	daxpy (linpackup: linpackup.f, 495)
[2]	1.860	3.3%	96.9%	186	matgen (linpackup: linpackup.f, 199)
[3]	1.430	2.5%	99.4%	143	dgefa (linpackup: linpackup.f, 221)
[4]	0.190	0.3%	99.7%	19	idamax (linpackup: linpackup.f, 700)
[5]	0.110	0.2%	99.9%	11	dscal (linpackup: linpackup.f, 670)
[6]	0.050	0.1%	100.0%	5	dmxpy (linpackup: linpackup.f, 826)
	56.690	100.0%	100.0%	5669	TOTAL

Analyzing the Report

The report has the following columns:

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

A Hardware Counter Experiment

Note: This experiment can be performed only on systems that have built-in hardware counters (the R10000, R12000, R14000, and R16000 classes of machines).

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

This section lists the steps you need to perform a hardware counter experiment. The tutorial describes the steps involved in performing the `dsc_hwc` experiment. This experiment allows you to capture information about secondary data cache misses. For

more information on hardware counter experiments, see "Hardware Counter Experiments (*_hwc, *_hwctime)", page 54.

Performing a Hardware Counter Experiment

From the command line, enter the following:

```
% ssrun -dsc_hwc linpackup
```

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

Generating a Report

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

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

Output similar to the following is generated:

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

Function list, in descending order by counts

```
-----
[index]  counts      %   cum.%  samples  function (dso: file, line)

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

        383699 100.0% 100.0%   2929  TOTAL
```

Analyzing the Report

The information immediately above the function list displays the following:

- The `Total samples` is the number of times the program counter was sampled. It is sampled once for each *overflow*, or each time the hardware counter exceeds the specified value.
- The `Counter name (number)` indicates the hardware counter used in the experiment. In this case, hardware counter 26 counts the number of times a value required in a calculation was not available in secondary cache. For a complete list of the hardware counters and their numbers, see Table 4-2, page 59.
- The `Counter overflow value` is the number at which the hardware counter overflows, or exceeds its preset value. In this case, the value is 131, which is the default. You can change the overflow value by setting the `_SPEEDSHOP_HWC_COUNTER_OVERFLOW` environment variable to a value larger than 0, the `_SPEEDSHOP_HWC_COUNTER_NUMBER` environment variable to 26, and running the `prof_hwc` experiment rather than `dsc_hwc`.

See "`_hwctime` Hardware Counter Experiments", page 57 to learn how to choose a counter overflow value.

- The `Total counts` is the total number of times a value was not in secondary cache when needed. This value is determined by multiplying the total number of samples by the overflow value; extra counts that do not cause an overflow are not recorded.

The function list has the following columns:

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

A `bbcounts` Experiment

This section provides the steps you need to perform a `bbcounts` or *basic block counts* experiment. This experiment counts basic block usage and estimates a linear time. It also maps a complete call graph. See "Basic Block Counting Experiment (`bbcounts`)", page 62.

Performing a `bbcounts` Experiment

From the command line, enter the following:

```
% ssrun -bbcounts linpackup
```

This starts the experiment. This entails making copies of the libraries and executables, giving them an extension of `.pixie`.

Output from `linpackup` and from `ssrun` is printed to `stdout`, as shown in the following example. A data file is also generated. The name consists of the process name (`linpackup`), the experiment type (`bbcounts`), and the experiment ID. In this example, the file name is `linpackup.bbcounts.m77549`.


```

instrumenting /lib32/rld
instrumenting /usr/lib32/libssrt.so
instrumenting /usr/lib32/libss.so
instrumenting /usr/lib32/libmp.so
instrumenting /usr/lib32/libftn.so
instrumenting /usr/lib32/libm.so
instrumenting /usr/lib32/libc.so.1

```

Generating a Report

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

```
% prof your_output_file_name > bbcounts.results
```

The `prof` command redirects output to a file called `bbcounts.results`. The file should contain results that look something like the following.

```

-----
SpeedShop profile listing generated Tue Mar 19 07:05:08 2002
  prof linpackup.bbcounts.m18879
    linpackup (n32): Target program
      bbcounts: Experiment name
        it:cu: Marching orders
          R5000 / R5000: CPU / FPU
            1: Number of CPUs
              180: Clock frequency (MHz.)

Experiment notes--
  From file linpackup.bbcounts.m18879:
  Caliper point 0 at target begin, PID 18879
    /speedlin/linpack/linpackup
  Caliper point 1 at exit(0)

-----
Summary of ideal time data (bbcounts)--
  4947867081: Total number of instructions executed
  6648387101: Total computed cycles
    36.935: Total computed execution time (secs.)
    1.344: Average cycles / instruction
-----

```

3: Tutorial for Fortran Users

Function list, in descending order by exclusive ideal time

```
-----
```

[index]	excl.secs	excl.%	cum.%	cycles	instructions	calls	function (dso: file, line)
[1]	34.620	93.7%	93.7%	6231556465	4669997342	772633	daxpy (linpackup.pixbb: linpackup.f, 495)
[2]	1.325	3.6%	97.3%	238494366	155792196	18	matgen (linpackup.pixbb: linpackup.f, 199)
[3]	0.689	1.9%	99.2%	123962402	80774803	17	dgefa (linpackup.pixbb: linpackup.f, 221)
[4]	0.138	0.4%	99.6%	24871119	18629195	5083	dscal (linpackup.pixbb: linpackup.f, 670)
[5]	0.113	0.3%	99.9%	20362634	15660094	5083	idamax (linpackup.pixbb: linpackup.f, 700)
[6]	0.029	0.1%	99.9%	5226705	3695170	1	dmxpy (linpackup.pixbb: linpackup.f, 826)
[7]	0.007	0.0%	100.0%	1204552	761974	17	dgesl (linpackup.pixbb: linpackup.f, 324)
[8]	0.004	0.0%	100.0%	710271	659913	2166	general_find_symbol (rld: rld.c, 2038)
[9]	0.003	0.0%	100.0%	490666	447076	4700	resolve_relocations (rld: rld.c, 2636)
[10]	0.001	0.0%	100.0%	239219	239219	2180	elfhash (rld: obj.c, 1184)
[11]	0.001	0.0%	100.0%	163768	163768	7126	obj_dynsym_got (rld: objfcn.c, 46)
[12]	0.001	0.0%	100.0%	157763	157763	3095	strcmp (rld: strcmp.s, 34)
[13]	0.001	0.0%	100.0%	129442	125136	2168	resolve_symbol (rld: rld.c, 1828)
[14]	0.001	0.0%	100.0%	124086	121952	2139	resolving (rld: rld.c, 1499)
[15]	0.001	0.0%	100.0%	114658	102484	2	fix_all_defined (rld: rld.c, 3419)
[16]	0.000	0.0%	100.0%	64956	64615	7	search_for externals (rld: rld.c, 3987)
[17]	0.000	0.0%	100.0%	61565	59304	1116	__flsbuf (libc.so.1: flsbuf.c, 25)
[18]	0.000	0.0%	100.0%	55557	42737	4274	obj_set_dynsym_got (rld: objfcn.c, 82)
[19]	0.000	0.0%	100.0%	42595	42594	867	x_putc (libftn.so: wsfe.c, 177)
[20]	0.000	0.0%	100.0%	42261	37264	1	linp (linpackup.pixbb: linpackup.f, 3)
[21]	0.000	0.0%	100.0%	24161	21845	28	x_wEND (libftn.so: wsfe.c, 225)
[22]	0.000	0.0%	100.0%	17014	17000	8	memset (libc.so.1: bzero.s, 98)
[23]	0.000	0.0%	100.0%	14671	13537	71	do_fio64_mp (libftn.so: fmt.c, 626)
[24]	0.000	0.0%	100.0%	14575	11501	53	wrt_E (libftn.so: wrtfmt.c, 353)
.							
.							
.							
[331]	0.000	0.0%	100.0%	1	1	1	__istart (linpackup.pixbb: crtltinit.s, 14)

Analyzing the Report

The report has the following columns:

- The `index` column assigns a reference number to each function.
- The `excl.secs` column shows the minimum number of seconds that might be spent in the routine under ideal conditions.
- The `excl.%` column represents how much of the program's total time was spent in the routine.
- The `cum.%` column shows the cumulative percentage of time spent in the routines listed so far.
- The `cycles` column shows the total number of machine cycles used by the routine.
- The `instructions` column shows the total number of instructions executed by a routine.
- The `calls` column shows the total number of calls to the routine. For example, there was just one call to the `dmxpy` routine.
- The `function (dso:file, line)` column lists the name, the DSO name, the file name, and the line number for the routine.

MPI Tracing Tutorial

The `mpi` experiment traces and times calls to MPI routines; the results are viewable with `prof`. The `mpi_trace` experiment produces the same results but the results are viewable only in `cvperf`.

The following steps generate tracing data for an MPI program. Before running this tutorial, you must first obtain a copy of the `matmul.f` file. You can perform a web search to find a downloadable copy, or go to any of the following URLs to obtain a copy of the file:

```
http://scv.bu.edu/SCV/Tutorials/F90/intrinsics/MATMUL.html  
http://www.dartmouth.edu/~rc/classes/intro_mpi/matmult.html
```

Save the copy in the `/usr/demos/SpeedShop` directory.

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

```
% setenv MPI_RLD_HACK_OFF 1
```

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

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

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

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

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

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

4. Finally, display an experiment file with the WorkShop `cvperf(1)` command. You can use `prof` to display an `mpi` experiment and you can use `cvperf` to view an `mpi_trace` experiment.

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

To display the output, select either **MPI Stats View (Graphs)** or **MPI Stats View (Numerical)** from the **Views** menu. See Figure 3-1, page 49, for an illustration of the **MPI Stats View (Graphs)**.

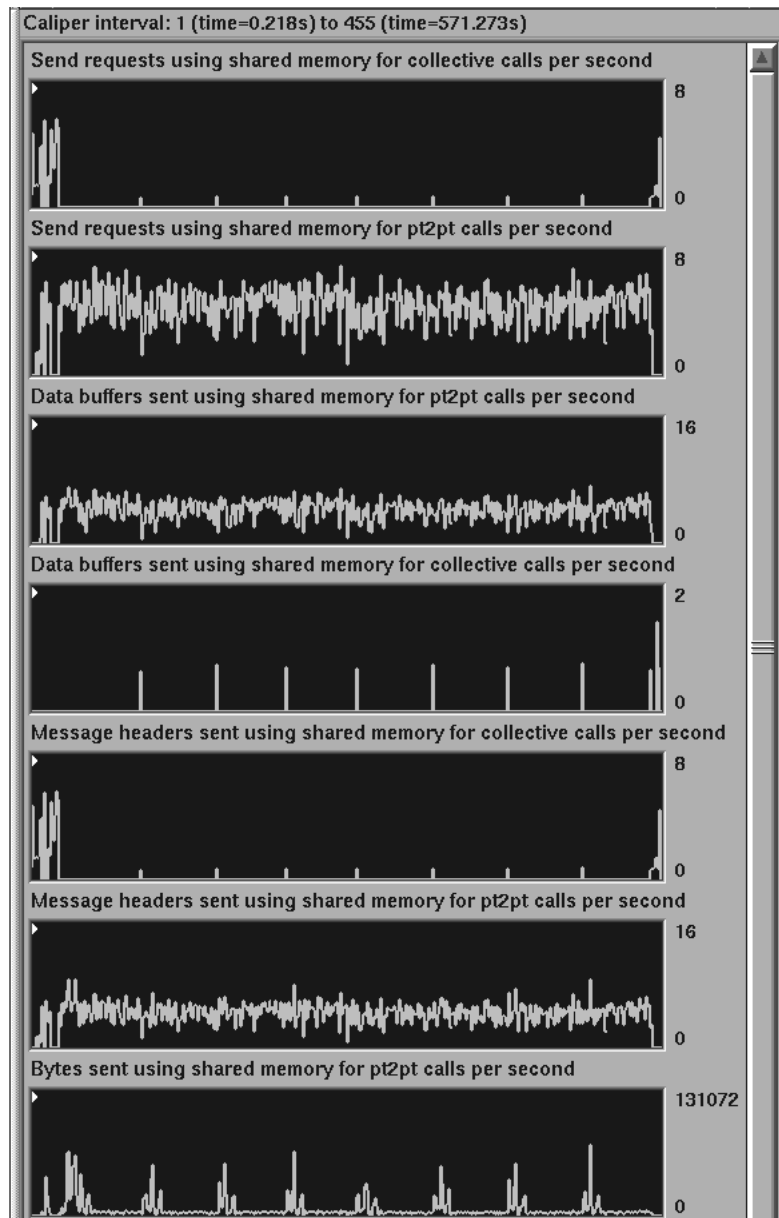


Figure 3-1 An MPI Experiment in `cvperf`

Experiment Types

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

- "Selecting an Experiment", page 51.
- "Floating-Point Exception Trace Experiment (fpe)", page 53.
- "Heap Trace Experiment (heap)", page 53.
- "Hardware Counter Experiments (*_hwc, *_hwctime)", page 54.
- "Basic Block Counting Experiment (bbcounts)", page 62.
- "I/O Trace Experiment (io)", page 65.
- "NUMA Profiling Experiment (numa)", page 66
- "PC Sampling Experiment (pcsamp)", page 67.
- "Call Stack Profiling Experiment (usertime/totaltime)", page 68.

For information on how to run the experiments described in this chapter, see Chapter 6, "Setting Up and Running Experiments: `ssrun`", page 71.

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, see the relevant section in the remainder of this chapter.

Table 4-1 Summary of Experiments

Experiment	Clues	Data Collected
fpe	High system time. Presence of floating-point operations.	All floating-point exceptions, with the exception type and the call stack at the time of the exception. See "Floating-Point Exception Trace Experiment (fpe)", page 53.
heap	Memory utilization.	Heap trace data from each processor in a multiprocessor system. See "Heap Trace Experiment (heap)", page 53.
_hwc	High user CPU time.	On R10000, R12000, R14000, and R16000 class machines, counts at the source line, machine instruction, and function levels of various hardware events, including: clock cycles, graduated instructions, primary instruction cache misses, secondary instruction cache misses, primary data cache misses, secondary data cache misses, translation lookaside buffer (TLB) misses, and graduated floating-point instructions. PC sampling is used. See "Hardware Counter Experiments (*_hwc, *_hwctime)", page 54.
_hwctime	High user CPU time.	Similar to _hwc experiment, except that callstack sampling is used. See "Hardware Counter Experiments (*_hwc, *_hwctime)", page 54.
bbcounts	CPU-bound.	CPU time at the function, source line, and machine instruction levels using basic block counting. See "Basic Block Counting Experiment (bbcounts)", page 62.
io	I/O-bound.	Traces the following I/O system calls: read, readv, write, writev, open, close, dup, pipe, creat. See "I/O Trace Experiment (io)", page 65.
mpi	mpi performance is poor.	Traces and times calls to various MPI routines. See "MPI Call Tracing Experiment (mpi/mpi_trace)", page 66
mpi_trace	mpi performance is poor.	Traces calls to various MPI routines and generates a file viewable in the cvperf performance analyzer window. This is a deprecated experiment and will be removed in a future release. See "MPI Call Tracing Experiment (mpi/mpi_trace)", page 66.

Experiment	Clues	Data Collected
numa	Slow shared-memory program.	On ccNUMA architecture machines, randomly samples memory accesses and reports: instruction performing the access, memory being accessed, ccNUMA node where memory access originates, ccNUMA node containing the memory, ccNUMA routing distance between these two nodes. See "NUMA Profiling Experiment (numa)", page 66.
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. See "PC Sampling Experiment (pcsamp)", page 67.
totaltime	Slow program, nothing else known. Not CPU-bound.	Inclusive and exclusive real time for each function by sampling the callstack at 30-millisecond intervals. See "Call Stack Profiling Experiment (usertime/totaltime)", page 68.
usertime	Slow program, nothing else known. Not CPU-bound.	Inclusive and exclusive CPU time for each function by sampling the callstack at 30-millisecond intervals. See "Call Stack Profiling Experiment (usertime/totaltime)", page 68.

Floating-Point Exception Trace Experiment (`fpe`)

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

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

Heap Trace Experiment (`heap`)

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

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

Hardware Counter Experiments (`*_hwc`, `*_hwctime`)

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

The experiments described in this section are available for systems that have hardware counters (R10000, R12000, R14000, and R16000 class machines). Hardware counters allow you to count various types of events, such as cache misses and counts of issued and graduated instructions.

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

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

Two Tools for Hardware Counter Experiments

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

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

- SpeedShop allows you to perform the hardware counter experiments described in the next sections ("`_hwc` Hardware Counter Experiments" and "`_hwctime` Hardware Counter Experiments", page 57).

`_hwc` Hardware Counter Experiments

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

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

By preceding the experiment name with an `f` (for example, `cy_hwc` becomes `fcy_hwc`), you reduce the value of an overflow to approximately one-fifth of the original value. By preceding the experiment name with an `s`, you increase the value of an overflow over the original value.

The following list describes the hardware counter experiments:

- `[f|s]gi_hwc` experiment: this experiment counts overflows of the graduated instruction counter. The graduated instruction counter is incremented by the number of instructions that were graduated on the previous cycle. The experiment uses statistical PC sampling based on an overflow interval of 32,771. If the optional `f` prefix is used, the overflow interval is 6,553. If the `s` prefix is used, the overflow interval is 3,999,971.
- `[f|s]cy_hwc` experiment: this experiment counts overflows of the cycle counter. The cycle counter is incremented on each clock cycle. The experiment uses statistical PC sampling based on an overflow interval of 16,411. If the optional `f` prefix is used, the overflow interval is 3,779. If the optional `s` prefix is used, the overflow interval is 1,999,993.
- `[f|s]ic_hwc` experiment: this experiment counts overflows of the primary instruction cache miss counter. The counter is incremented one cycle after an instruction fetch request is entered into the miss handling table. The experiment uses statistical PC sampling based on an overflow interval of 2,053. If the optional

`f` prefix is used, the overflow interval is 419. If the optional `s` prefix is used, the overflow interval is 524,309.

- `[f|s]isc_hwc` experiment: this experiment counts overflows of the secondary instruction cache miss counter. The secondary instruction cache miss counter is incremented after the last 16-byte block of a 64-byte primary instruction cache line is written into the instruction cache. The experiment uses statistical PC sampling based on an overflow interval of 131. If the optional `f` prefix is used, the overflow interval is 29. If the optional `s` prefix is used, the overflow interval is 65,537.
- `[f|s]dc_hwc` experiment: this experiment counts overflows of the primary data cache miss counter. The primary data cache miss counter is incremented on the cycle after a primary cache data refill is begun. The experiment uses statistical PC sampling based on an overflow interval of 2,053. If the optional `f` prefix is used, the overflow interval is 419. If the optional `fs` prefix is used, the overflow interval is 524,309.
- `[f|s]dsc_hwc` experiment: this experiment counts overflows of the secondary data cache miss counter. The secondary data cache miss counter is incremented on the cycle after the second 16-byte block of a primary data cache line is written into the data cache. The experiment uses statistical PC sampling, based on an overflow interval of 131. If the optional `f` prefix is used, the overflow interval is 29. If the optional `s` prefix is used, the overflow interval is 65,537.
- `[f|s]tlb_hwc` experiment: this experiment counts overflows of the translation lookaside buffer (TLB) counter. The TLB counter is incremented on the cycle after the TLB miss handler is invoked. The experiment uses statistical PC sampling based on an overflow interval of 257. If the optional `f` prefix is used, the overflow interval is 53. If the optional `s` prefix is used, the overflow interval is 19,997.
- `[f|s]gfp_hwc` experiment: this experiment counts overflows of the graduated floating-point instruction counter. The graduated floating-point instruction counter is incremented by the number of floating-point instructions that graduated on the previous cycle. If the optional `f` prefix is used, the overflow interval is 6,553. If the optional `s` prefix is used, the overflow interval is 3,999,971.
- `[f|s]fsc_hwc` experiment: this experiment uses statistical PC sampling based on overflows of the failed store conditionals counter. If the optional `f` prefix is used, the overflow interval is 401. If the optional `s` prefix is used, the overflow interval is 19,997.
- `prof_hwc` experiment: for any hardware counter not otherwise covered in "_hwc Hardware Counter Experiments", or to choose different overflow intervals for

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

- `_SPEEDSHOP_HWC_COUNTER_NUMBER`: the value of this variable can be between 0 and 31. Hardware counters are described in the *MIPS R10000 User's Guide* and on the `r10k_counters(1)` man page. The hardware counter numbers are provided in the tables in "Hardware Counter Numbers", page 59.
- `_SPEEDSHOP_HWC_COUNTER_OVERFLOW`: The value of this variable can be any number greater than 0. Some numbers may produce data that is not statistically random, but rather reflects a correlation between the overflow interval and a cyclic behavior in the application. You may want to do two or more runs with different overflow values.

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

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

`_hwctime` Hardware Counter Experiments

The following sections describe `_hwctime` hardware counter experiments, which run on R10000, R12000, R14000 and R16000 machines only. The `_hwctime` hardware counter experiments also show where the overflows are being triggered in the program. These experiments are similar to the `_hwc` experiments, but record the callstack information rather than showing where the program counter was when the overflow occurred.

See the `perfex(1)` and `r10k_counters(5)` man pages for other methods of returning hardware-level information.

The following list describes these experiments:

- `gi_hwctime` experiment: `hwct,17,1000003,0,SIGPROF:cu`. Profiles the cycle counter using the statistical call stack sampling, based on overflows of the graduated-instruction counter, at an overflow interval of 1000003.

- `cy_hwctime` experiment: `hwct,0,10000019,0,SIGPROF:cu`. Profiles the cycle counter using statistical call-stack sampling based on overflows of the cycle counter, at an overflow interval of 10000019.
- `ic_hwctime` experiment: `hwct,9,8009,0,SIGPROF:cu`. Profiles the cycle counter using statistical call-stack sampling, based on overflows of the primary instruction-cache-miss counter, at an overflow interval of 8009.
- `isc_hwctime` experiment: `hwct,10,2003,0,SIGPROF:cu`. Profiles the cycle counter using statistical call-stack sampling, based on overflows of the secondary instruction-cache-miss counter, at an overflow interval of 2003.
- `dc_hwctime` experiment: `hwct,25,8009,0,SIGPROF:cu`. Profiles the cycle counter using statistical call-stack sampling, based on overflows of the primary data-cache-miss counter, at an overflow interval of 8009.
- `dsc_hwctime` experiment: `hwct,26,2003,0,SIGPROF:cu`. Profiles the cycle counter using statistical call-stack sampling, based on overflows of the secondary data-cache-miss counter, at an overflow interval of 2003.
- `tlb_hwctime` experiment: `hwct,23,2521,0,SIGPROF:cu`. Profiles the cycle counter using statistical call-stack sampling, based on overflows of the TLB miss counter, at an overflow interval of 2521.
- `gfp_hwctime` experiment: `hwct,21,10007,0,SIGPROF:cu`. Profiles the cycle counter using statistical call-stack sampling, based on overflows of the graduated floating-point instruction counter, at an overflow interval of 10007.
- `fsc_hwctime` experiment: `hwct,5,5003,0,SIGPROF:cu`. Profiles the cycle counter using statistical call-stack sampling, based on overflows of the failed store conditionals counter, at an overflow interval of 5003.
- `prof_hwctime` experiment: For any hardware counter not otherwise covered in "`_hwc` Hardware Counter Experiments", or to choose different sampling counter/overflow interval for any hardware counter time experiment, the `prof_hwctime` experiment is available. Here, profiling is done for the counter specified by the environment variable `_SPEEDSHOP_HWC_COUNTER_PROF_NUMBER` using statistical call-stack sampling, based on overflows of the counter specified by the environment variable `_SPEEDSHOP_HWC_COUNTER_NUMBER`, at an interval given by the environment variable `_SPEEDSHOP_HWC_COUNTER_OVERFLOW`.

Note: These environment variables cannot be used to override the counter numbers or interval for other defined experiments. They are examined only when the `prof_hwctime` or `prof_hwc` experiment is specified.

The default overflow and profiling counter is the cycle counter; the default overflow interval is 10000019.

Hardware Counter Numbers

The possible numeric values for the `_SPEEDSHOP_HWC_COUNTER_NUMBER` and `_SPEEDSHOP_HWC_COUNTER_PROF_NUMBER` variables are shown in the following tables. Table 4-2, page 59, gives the hardware counter numbers for systems with R10000 processors, and Table 4-3, page 61, gives them for systems with R12000/R14000/R16000 processors. For the R10000 processors, if two counter numbers need to be specified, one counter number must be chosen from a group including numbers 0–15 and the other counter number must be chosen from a group including numbers 16–31 due to hardware restrictions. See the `r10k_counters(5)` man page for further details.

Table 4-2 R10000 Hardware Counter Numbers

Number	Indication
0	Cycles
1	Issued instructions
2	Issued loads
3	Issued stores
4	Issued store conditionals
5	Failed store conditionals
6	Decoded branches (rev 2.x processors) or resolved branches (rev 3.x processors)
7	Quadwords written back from secondary cache
8	Correctable secondary cache data array ECC errors

Number	Indication
9	Primary instruction-cache misses
10	Secondary instruction-cache misses
11	Instruction misprediction from secondary cache way prediction table
12	External interventions
13	External invalidations
14	Virtual coherency conditions (or functional unit completions, depending on hardware version)
15	Graduated instructions
16	Cycles
17	Graduated instructions
18	Graduated loads
19	Graduated stores
20	Graduated store conditionals
21	Graduated floating-point instructions
22	Quadwords written back from primary data cache
23	TLB misses
24	Mispredicted branches
25	Primary data cache misses
26	Secondary data cache misses
27	Data misprediction from secondary cache way prediction table
28	External intervention hits in secondary cache
29	External invalidation hits in secondary cache
30	Store/prefetch exclusive to clean block in secondary cache
31	Store/prefetch exclusive to shared block in secondary cache

Table 4-3 R12000, R14000, R16000 Hardware Counter Numbers

Number	Indication
0	Cycles
1	Decoded instructions
2	Decoded loads
3	Decoded stores
4	Miss Handling Table occupancy
5	Failed store conditionals
6	Resolved conditional branches
7	Quadwords written back from secondary cache
8	Correctable secondary cache data array ECC errors
9	Primary instruction-cache misses
10	Secondary instruction-cache misses
11	Instruction misprediction from secondary cache way prediction table
12	External interventions
13	External invalidations
14	Not implemented
15	Graduated instructions
16	Executed prefetch instructions
17	Prefetch primary data cache misses
18	Graduated loads
19	Graduated stores
20	Graduated store conditionals
21	Graduated floating-point instructions
22	Quadwords written back from primary data cache
23	TLB misses
24	Mispredicted branches

Number	Indication
25	Primary data cache misses
26	Secondary data cache misses
27	Data misprediction from secondary cache way prediction table
28	State of intervention hits in secondary cache
29	State of invalidation hits in secondary cache
30	Miss Handling Table (MHT) entries accessing memory
31	Store/prefetch exclusive to shared block in secondary cache

Basic Block Counting Experiment (`bbcounts`)

The `bbcounts` experiment displays an estimated time based on linear basic block counting.

Data is measured by counting the number of executions of each basic block and calculating an estimated time for each function. This involves instrumenting the program to divide the code into basic blocks, which are consecutive sequences of instructions with a single entry point, a single exit point, and no branches into or out of the sequence. Instrumentation also records a count of all dynamic (function-pointer) calls.

Because an exact count of every instruction in your program is recorded, you can also use the `bbcounts` experiment to determine the efficiency of your algorithm and identify any code that is not executed.

How SpeedShop Prepares Files

To permit block counting, SpeedShop does the following:

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

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

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

How SpeedShop Calculates CPU Time for `bbcounts` Experiments

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

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

Basic block counts are translated to an estimated CPU time displayed at the function, source line, and assembly instruction 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 it does not propagate the time up to the caller of that procedure. For example, basic block counting may tell you that procedure `sin(x)` took the most time, but significant performance improvement can only be obtained by optimizing the callers of `sin(x)`. Inclusive basic block counting solves this problem.

Inclusive basic block counting calculates cycles just like regular basic block counting and then propagates it in proportion to its callers. The cycles of procedures obtained using regular basic block counting (called exclusive cycles) are divided up among its callers in proportion to the number of times they called this procedure. For example, if `sin(x)` takes 1000 cycles, and its callers, procedures `foo()` and `bar()`, call

`sin(x)` 25 and 75 times respectively, 250 cycles are attributed to `foo()` and 750 to `bar()`. By propagating cycles this way, `__start()` usually ends up with all the cycles counted in the program. (It is possible to write code that makes determining the complete call graph impossible, in which case you may end up with parts of the call graph disconnected.)

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

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

Using `pcsamp` and `bbcounts` Together

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

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

A comparison between basic block counts (`bbcounts` experiment) and PC profile counts (`pcsamp` experiment) is shown in Table 4-4.

Table 4-4 Basic Block Counts and PC Profile Counts Compared

Basic Block Counts	PC Profile Counts
Used to compute <code>bbcounts</code> CPU time.	Used to estimate actual CPU time.
Data collection by instrumentation.	Data collection by the kernel.

Basic Block Counts	PC Profile Counts
Slows program down by factor of three or more.	Has minimal impact on program speed.
Generates an exact count of every instruction.	Generates statistical, inexact counts.

I/O Trace Experiment (`io`)

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

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

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

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

```
% ssrun -io generic
```

MPI Call Tracing Experiment (`mpi/mpi_trace`)

The `mpi_trace` experiment traces calls to various MPI routines and generates a file that is viewable in `prof`. For a list of the routines that are traced, see the `ssrun` man page.

The `mpi_trace` experiment traces calls to various MPI routines and generates a file that is viewable in the `cvperf` Performance Analyzer window. For a list of the routines that are traced, see the `ssrun` man page (this experiment will be removed in a future release).

For more details about MPI experiments, see "Running Experiments on MPI Programs", page 86.

NUMA Profiling Experiment (`numa`)

The NUMA profiler operates on a statistical basis by periodically interrupting the running application. During each interrupt the application's memory accesses are examined. Interrupts are triggered periodically by waiting for a particular number of CPU hardware performance counter events to occur. For example, the default setting is to interrupt the running application after approximately 100 secondary data cache misses have occurred.

During each interrupt, the profiler begins at the interrupted program counter location and finds the nearest memory access for which it can accurately calculate a target address. This address is then used to determine the ccNUMA node that contains the memory being accessed. The profiler also determines which ccNUMA node is executing the interrupted application thread.

Each time a sample is taken, the following information is stored in the experiments data file:

- ID of the thread performing the memory access
- Program counter for the instruction performing the memory access
- Memory address being accessed
- ccNUMA node executing the memory access
- ccNUMA node containing the memory being accessed
- ccNUMA routing distance (in "hops") between these two ccNUMA nodes

This sampling process is repeated continuously until the application terminates.

The generated data can then be used to locate those lines of application code that generate the largest number of remote memory accesses ('remote' refers to the situation in which a node is performing an access to memory that does not lie on that node). By using facilities already present in IRIX (namely CPU sets and memory locality domains) the application engineer can attempt to minimize these remote accesses.

Applications running on a ccNUMA system do not see the same memory latency for every memory access - even after accounting for all cache effects. Accessing memory located on the same node as that on which you run your code is faster than accessing memory on other nodes. By reducing the number of remote memory accesses the application's performance is enhanced.

PC Sampling Experiment (`pcsamp`)

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

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

To collect the data, the operating system regularly stops the process, increments a counter corresponding to the current value of the PC, and resumes the process. The default sample interval is 10 milliseconds. If you specify the optional `f` prefix to the experiment, a sample interval of 1 millisecond is used. (See "A `pcsamp` Experiment", page 20, for an example.)

By default, the experiment uses 16-bit counters. If the optional `x` suffix is used, a 32-bit counter size will be used. Using a 32-bit bin provides more accurate information, but requires additional memory and disk space. See "Example Using the `pcsampx` Experiment", page 83, for an example.

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

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

Call Stack Profiling Experiment (`usertime/totaltime`)

The `usertime` and the `totaltime` experiments are useful experiments to start your performance analysis. The `usertime` experiment returns CPU time for each function while your program runs and the `totaltime` experiment returns real time for each function.

These experiments use statistical call stack profiling to measure inclusive and exclusive user time. They take a sample every 30 milliseconds. Data is measured by periodically sampling the callstack. The program's callstack data is used to do the following:

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

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

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

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

Collecting Data on Machine Resource Usage

This chapter describes how to collect machine resource usage data using the SpeedShop `ssusage(1)` command. Finding out the machine resources that your program uses can help you identify performance bottlenecks and determine which performance experiments you need to run. You can use the list in "Gathering and Analyzing Performance Data", page 8, to identify which experiments to run, based on the results of running `ssusage` on your program.

ssusage Syntax

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

```
ssusage executable_name [executable_args]
```

- *executable_name*: name of the executable for which you want to collect machine resource usage data.
- *executable_args*: arguments to your executable, if any

ssusage Results

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

```
...  
22.03 real, 18.18 user, 0.21 sys, 7 majf, 120 minf, 0 sw, 241 rb, 0  
wb, 135 vcx, 648 icx, 976 mxrss
```

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

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

- `sys`: system CPU time, during which the system was performing services for the executable, in seconds.
- `majf`: major page faults that cause physical I/O.
- `minf`: minor page faults that require mapping only.
- `sw`: process swaps.
- `rb/wb`: physical blocks read or written. These are attributed to the process that first requests a block, but they do not necessarily directly correlate with the process's own I/O operations.
- `vcx`: voluntary context switches; those caused by the process's own actions.
- `icx`: involuntary context switches; those caused by the scheduler.
- `mrxrss`: maximum resident set size of the program, including any shared pages, in kilobytes.

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

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 has the following sections:

- "Building Your Executable", page 71
- "Setting Up Output Directories and Files", page 73
- "Using Run-Time Environment Variables", page 74
- "Using Marching Orders", page 78
- "Running Experiments", page 81
- "Running Experiments on MPI Programs", page 86
- "Running Experiments on Programs Using Pthreads", page 91
- "Running Experiments on Programs That Use OpenMP Directives", page 91
- "Using Calipers", page 92
- "Effects of `ssrun`", page 96

Building Your Executable

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

This section explains when to change the way you build your executable program. For information on setting environment variables, see "Using Run-Time Environment Variables", page 74.

- If you have used the `ssrt_caliper_point(3)` function provided in the SpeedShop libraries, you have to explicitly link in the SpeedShop libraries file, `libss.so`. For more information on setting caliper points, see "Using Calipers", page 92.

- If you are planning to build your executable using the `-o32` option to the `cc` command, and you want to run the `usertime` experiment, you must add `-lexc` to the link line. For more information on `cc -o32`, see the `cc(1)` man 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. Using the `-s` option strips debugging information from the program object and makes the program unusable for performance analysis.
- If you have used compiler optimization level 3 (`-O3`) and you are performing experiments that report function-level information, inlining can result in extremely misleading profiles. The time spent in the inlined procedure will show up in the profile as time spent in the procedure into which it was inlined. It is generally better to use compiler optimization level 2 (`-O2`) or less when gathering an execution profile.

Special Information for MP Fortran Programs

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

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

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

- In an n32 or 64-bit MP version, the routine has the name `DAXPY`, but most of that work is done in the MP routine named `DAXPY.PREGION1`.
 - In an o32-bit version, the `DAXPY` routine is named `daxpy_`, and the MP routine is `_daxpy_519_aaab_`.
- If you perform a `bbcounts` experiment, the source annotations for 32-bit and 64-bit compilations with the `-g` option differ and are not correct in most cases.
 - In 64-bit source annotations, the exclusive time is correctly shown for each line, but the inclusive time for the first line of the loop (`do` statement) includes the time spent in the loop body. This same time appears on the lines comprising the loop's body, in effect representing a double-counting.

- In 32-bit source annotations, the exclusive time is incorrectly shown for the line comprising the loop's body. The line-level data for the loop-body routine (`_daxpy_519_aaab_`) 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 the `-mp_keep` flag, the line-level data for the loop-body routine is incorrect.

Most lines refer to line 0 of the file and the rest to other lines at seemingly random places in the file. Consequently, false annotations will appear on some lines. Disassembly correctly shows the instructions and their data, but the line numbers are wrong. This reflects essentially the same double-counting problem as seen in 64-bit compilations, but the extra counts go to other places in the file, rather than to the first line of the loop.

Setting Up Output Directories and Files

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

executable_name.exp_type.id

The *id* consists of one or two letters (designating the process type) and the process ID number. The following list describes the letter codes:

- *m*: master process created by `ssrun`.
- *p*: process created by a call to `sproc()`.
- *f*: process created by a call to `fork()`.
- *s*: process created by a call to `system()`.
- *e*: process created by a call to `exec()`.
- *fe*: process created by a call to `fork()` and `exec()`.
- *Rn*: rank number of the MPI process that generated the experiment file.
- *Tn*: OpenMP thread that generated the experiment file.

The following are examples of data file names:

```
stat.bbcounts.m10966
engines.pcsamp.m14493
```

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

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

Using Run-Time Environment Variables

Several environment variables have been defined for use specifically with SpeedShop to provide additional information to SpeedShop commands or SpeedShop library routines at run time. This section provides information about available environment variables, grouped by functionality:

- "User Environment Variables", page 74
- "Process Tracking Environment Variables", page 76
- "Expert-Mode Environment Variables", page 77

User Environment Variables

The following list describes a number of environment variables that are normally used to control the operation of SpeedShop.

- `_SPEEDSHOP_CALIPER_POINT_SIG sig_num`: causes the specified signal number to be used for recording a caliper point in the experiment.
- `_SPEEDSHOP_HWC_COUNTER_NUMBER num`: specifies the counter to be used for `prof_hwc` experiments. Counters are numbered between 0 and 31, and are described in the *MIPS R10000 User's Guide*.
- `_SPEEDSHOP_HWC_COUNTER_PROF_NUMBER num`: specifies the counter that will be profiled for `prof_hwctime` experiments.
- `_SPEEDSHOP_HWC_COUNTER_OVERFLOW num`: specifies the overflow value for the counter to be used in `prof_hwc` experiments. The value for `num` must be $0 < num \leq 2147483647$. Some choices may produce data that is not statistically

random but reflects a correlation between the overflow interval and a cyclic behavior in the application. Users may want to do two or more runs with different overflow values.

- `_SPEEDSHOP_INSTR_ARGS`: defines additional instrumentation arguments.
- `_SPEEDSHOP_OUTPUT_DIRECTORY` *dir*: causes the output data files to be placed in the specified directory rather than the current working directory.
- `_SPEEDSHOP_OUTPUT_FILENAME` *filename*: causes the output file to be saved under the specified name. If `_SPEEDSHOP_OUTPUT_FILENAME` is set to `myfile`, the experiment file is named `myfile.suffix` (for example, `myfile.m12345`). If `_SPEEDSHOP_OUTPUT_DIRECTORY` is also specified, the directory is prepended to the file name you specify.
- `_SPEEDSHOP_OUTPUT_NOCOMPRESS`: Disables the compression of performance data.
- `_SPEEDSHOP_POLLPOINT_CALIPER_POINT` *timer_type, timer_interval*: used to add caliper points at regular time intervals into your experiment file (during program execution). Caliper points set with this variable are recorded in the performance data file generated by `ssrun`.
- `_SPEEDSHOP_REUSE_FILE_DESCRIPTOR`: opens and closes the file descriptors for the output files every time performance data is to be written.
- `_SPEEDSHOP_RLD`: defines the full path name to `rld`, and enables `rld` profiling (for `pcsamp` and `_hwc` experiments only). If the path name does not lead to `rld`, SpeedShop determines the correct path name automatically. For example, if you set `_SPEEDSHOP_RLD` to 1, SpeedShop will locate `rld`.
- `_SPEEDSHOP_SBRK_BUFFER_ADDR` *address*: defines the preferred starting address to be used for the internal `malloc` arena. This option has to be used with extreme care since it might result in memory region overlap.
- `_SPEEDSHOP_SBRK_BUFFER_LENGTH`: defines the segment grow size for the internal `malloc` arena used. This arena is completely separate from the user's arena, and it usually grows in default segments of the size `0x100000`.
- `_SPEEDSHOP_VERBOSE` or `_SPEEDSHOP_VERBOSE` *non_empty_string*: causes a log of each program's operation to be written to `stderr`. If this variable is set to an empty string, only major events are logged; if it is set to a non-empty string, more detailed events are logged.

- `_SPEEDSHOP_SILENT`: suppresses all SpeedShop output other than fatal error messages. If both `_SPEEDSHOP_VERBOSE` and `_SPEEDSHOP_SILENT` are set, `_SPEEDSHOP_VERBOSE` is ignored.
- `_SSMALLOC_NO_BUFFERING`: if this environment variable is set, the experiment file for each process will contain only its own heap trace data. Otherwise, the experiment file for each process will contain data from all processes.

To set an environment variable that requires no arguments (for example, `_SPEEDSHOP_SILENT`), use the following:

```
% setenv _SPEEDSHOP_SILENT
```

To set an environment variable that requires a number between 0 and 31 (for example, `_SPEEDSHOP_HWC_COUNTER_NUMBER`), use the following:

```
% setenv _SPEEDSHOP_HWC_COUNTER_NUMBER 15
```

Process Tracking Environment Variables

A number of environment variables may be used for controlling the treatment of processes spawned from the original target, as shown in the following list:

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

- `_SPEEDSHOP_TRACE_MPI_RANKS` [True|False]: if True, specifies that performance data should only be collected for the MPI ranks. It is False by default.

Expert-Mode Environment Variables

A number of variables may be used for debugging and finer control of the operation of SpeedShop, as shown in the following list:

- `_SPEEDSHOP_SAMPLING_MODE` or `_SPEEDSHOP_SAMPLING_MODE num`: used for PC sampling and hardware counter profiling. If set to 1, generates data for the base executable only. If not set or set to a value other than 1, data is generated for the executable and all the DSOs it uses.
- `_SPEEDSHOP_INIT_DEFERRED_SIG sig_num`: if specified, initialization of the experiment is not performed when the target process starts. Initialization is delayed until the specified signal is sent to the process. A handler for the given signal is installed when the process starts. It is the user's responsibility to ensure that it is not overridden by the target code.
- `_SPEEDSHOP_SHUTDOWN_SIG sig_num`: if specified, termination of the experiment is not performed when the target process exits. Termination happens when the specified signal is sent to the process. A handler for the given signal is installed when the process starts, and it is the user's responsibility to ensure that it is not overridden by the target code.
- `_SPEEDSHOP_EXPERIMENT_TYPE exp_type`: passes the experiment type to the run-time DSO. The `ssrun` command's `-exp_type` option, which usually specifies the experiment type, overrides this variable. Values for `exp_type` can be found in Table 4-1, page 52.
- `_SPEEDSHOP_EXTRA_MARCHING_ORDERS mo_syntax`: This environment variable may be used to add marching orders to a predefined experiment. See the "Using Marching Orders" section in this Chapter for more information.
- `_SPEEDSHOP_MARCHING_ORDERS mo_syntax`: passes the marching orders of the experiment to the run-time DSO. The `ssrun` command's `-mo`, marching orders, option overrides this environment variable. If this variable is specified, it overrides `_SPEEDSHOP_EXPERIMENT_TYPE`, as well as the `ssrun` command's `-exp_type` option. The `mo_syntax` is discussed in "Using Marching Orders", page 78.
- `_SPEEDSHOP_SBRK_BUFFER_LENGTH size`: defines the maximum size of the internal malloc (memory allocation) area used. This area is completely separate from the user's area and has a default size of 0x100000.

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

Using Marching Orders

Using marching orders is another method of specifying what experiment type you want to run. One of the benefits of using marching orders is that it lets you customize experiments. Any specification of explicit marching orders overrides the environment variable `_SPEEDSHOP_EXPERIMENT_TYPE` or the `-exp_type` option on the `ssrun` command, since these experiment type specifications are translated into possible orders by the command.

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

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

```
% setenv _SPEEDSHOP_MARCHING_ORDERS ut:cu
```

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

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

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

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

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

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

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

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

```
ut,10000,2
```

Defining the Base Experiment

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

- `ut`: a time experiment that returns real time, virtual time, or user time. The default arguments are `30000, 2`. The argument should be specified in multiples of 10,000. The first argument is the interval between call stack samples in microseconds. The second argument is the timer type used to measure the intervals; the supported values are 0, 1, and 2, with the same meanings as for the second argument of `hb` (described later). The argument value `-1` is not valid for `ut`.
- `pc`: a 16-bit or 32-bit PC sampling (`pcsamp`) experiment. The default arguments are `2, 10000, 0`. The first argument is the size of the sample count bins in bytes. The supported values are 2 (16 bits) and 4 (32 bits). The second argument is the sampling rate in microseconds. Supported values are 10,000 (10-millisecond sample interval) and 1000 (1-millisecond sample interval). The third argument is the sampling mode:
 - 0: selects the user executable and all its dynamic shared objects
 - 1: selects only the user executable (without any dynamic shared objects)
- `it`: a 32-bit `bbcounts` experiment. Only 4-byte (32-bit) counters are supported. No additional arguments are needed.
- `mf`: a memory allocation and deallocation experiment that traces calls to `malloc`, `realloc`, `free`, `memalign`, and `valloc` routines. There are no arguments to this marching order. The arguments to these routines and bad calls are recorded. Bad

calls include `malloc` calls of 0 bytes, freeing invalid memory blocks, reallocating invalid memory pointers, and calling `memalign` with invalid arguments. (For descriptions of these routines, see the `malloc(3)` man page.)

- `fpe`: a floating-point exceptions (`fpe`) experiment. There are no arguments. The call stack is sampled whenever a floating-point exception occurs.
- `io`: an I/O trace experiment. There are no arguments. The start time and end time for each of the following I/O system calls are recorded: `creat(2)`, `open(2)`, `read(2)`, `pread(2)`, `write(2)`, `pwrite(2)`, `close(2)`, `pipe(2)`, `dup(2)`, `lseek(2)`, `readv(2)`, and `writev(2)`.
- `mpit`: MPI experiment. There are no arguments. The beginning time, ending time, return value, and arguments are recorded. For a list of the routines traced, see "Generating MPI Tracing Experiments", page 86.

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

- `hwct`: a hardware counter call stack profiling experiment (`_hwctime`). The default arguments are `xx,xxx,0,SIGPROF`. The first argument is the hardware counter number of the counter to be profiled. The second argument is the overflow interval for the counter (a prime number should be specified). The third argument is the hardware counter number of the counter whose overflow will trigger the sampling.
- `hwc`: a hardware counter PC profiling experiment (`-hwc`). The default arguments are `xx,xxx`. The first argument is the hardware counter number. The second argument is the overflow interval for the counter.
- `hb`: heart beat data collection. System-wide, per-process, and MPI resource usage data is collected at regular time intervals. If the program creates multiple processes, data is collected for each process. If the process is using the MPI library, MPI library statistics are also recorded.

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

- 0: real (wall-clock) time.
- 1: virtual time. The timer runs while the user program is executing.

- 2: user time. The timer runs while the user program is executing or the system is processing system calls made by the program.
- `cu`: caliper point usage data collection. It usually appears at the end of a marching order, and there are no arguments. Usage data is recorded at caliper points. As with the `hb` marching order, system-wide, per-process, and MPI resource usage data is or can be collected at these points. But, the `hb` marching order collects data based on time, and the `cu` marching order is based on caliper points that you can set anywhere in your source code. For more information on setting caliper points, see "Using Calipers", page 92.
- `mpi`: traces calls to MPI functions and collects data (such as the time taken by the call, which thread made the call, etc.)
- `nm`: used to profile an application's memory access patterns on ccNUMA architectures. The profiler periodically interrupts the running application, and during each interrupt, the application's memory accesses are examined.

Running Experiments

This section describes how to use `ssrun` to perform experiments.

`ssrun` Syntax

The `ssrun` command takes the following form:

```
ssrun ssrun_options exp_type executable_name executable_args
```

The arguments are as follows:

- *ssrun_options*: zero or more of the options described in the following list. These options control the data collection and the treatment of descendent processes or programs, and they specify how the data is to be externalized.
- *exp_type* | *exp* *exp_type*: the experiment type. Experiments are described in detail in Chapter 4, "Experiment Types", page 51.
- *executable_name*: the name of the program on which you want to run an experiment.

- *executable_args*: arguments to your program, if any.

The `ssrun` command generates a performance data file that is named as described in "Setting Up Output Directories and Files", page 73.

The following list describes the options to the `ssrun` command:

- `-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 `_SPEEDSHOP_MARCHING_ORDERS` is not examined. If both `-exp_type` and `-mo` are specified, the `-mo` option will override the value given by `-exp_type`.
- `-name argv0-value`: specifies that the executable, or its appropriately instrumented version, should be run with `argv[0]` set to *argv0-value*. Normally, both instrumented and uninstrumented executables are run with `argv[0]` set to the original *executable_name* name. *argv0-value* is also used in the *executable_name* portion of the name of the performance data file.
- `-port hostname portno`: specifies that the process is to be left waiting, and notifications of status are to be sent to the socket on the host named by *hostname* and the port specified by *portno*. When the process is ready, a message of the form "running *pid host*" will be sent to inform the requester of the PID of the executing process and the host, which may be remote. A debugger can then attach to it and take control of its execution.
- `-quiet`: suppresses all output other than error messages. If `-quiet` is specified, the `_SPEEDSHOP_SILENT` environment variable is also set for the duration of the `ssrun` command.
- `-ranks mip-ranks`: specifies that performance data should only be collected for the MPI ranks in the comma-separated list of *mip-ranks*.
- `-v`: prints a log of the operation of `ssrun` to `stderr`. The same behavior occurs if the environment variable `_SPEEDSHOP_VERBOSE` is set to a null string.
- `-V`: prints a detailed log of the operation of `ssrun` to `stderr`. The same behavior occurs if the environment variable `_SPEEDSHOP_VERBOSE` is set to a nonzero-length string. This option can be used to see how to set the various environment variables, and how to invoke instrumentation when necessary.
- `-workshop`: specifies special instrumentation so that the experiment files can be read by WorkShop's `cvperf` analyzer.

- `-x display-id window-id`: specifies that the process is to be left waiting and that the window of the WorkShop debugger requesting the creation (as specified by the `display-id` and `window-id` arguments on the command line) be informed of the PID of the target process. A debugger can then attach to it and take control of its execution.

ssrun Examples

This section provides examples of using `ssrun` with options and experiment types. For additional examples, see Chapter 2, "Tutorial for C Users", page 13, or Chapter 3, "Tutorial for Fortran Users", page 33.

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 "PC Sampling Experiment (`pcsamp`)", page 67.

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

```
% ssrun -pcsampx generic
```

To see the performance data that has been generated, run `prof` on the performance data file, `generic.pcsampx.m12185`, as shown in the following example:

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

The report is printed to `stdout`. (This layout of this report has been altered slightly to accommodate presentation needs.) For more information on `prof` and the reports generated by `prof`, see Chapter 7, "Analyzing Experiment Results", page 99.

```
-----
SpeedShop profile listing generated Mon Feb  2 15:08:14 1998
```

```
  prof generic.pcsampx.m12185
    generic (n32): Target program
      pcsampx: Experiment name
pc,4,10000,0:cu: Marching orders
R4400 / R4000: CPU / FPU
      1: Number of CPUs
      175: Clock frequency (MHz.)
```

Experiment notes--

```

    From file generic.pcsampx.m12185:
    Caliper point 0 at target begin, PID 12185
        /usr/demos/SpeedShop/linpack.demos/c/generic
    Caliper point 1 at exit(0)

```

 Summary of statistical PC sampling data (pcsampx)--

```

        2729: Total samples
        27.290: Accumulated time (secs.)
        10.0: Time per sample (msecs.)
        4: Sample bin width (bytes)

```

 Function list, in descending order by time

[index]	secs	%	cum.%	samples	function (dso: file, line)
[1]	25.470	93.3%	93.3%	2547	anneal (generic: generic.c, 1573)
[2]	1.100	4.0%	97.4%	110	slaveusertime (dlslave.so: dlslave.c, 22)
[3]	0.310	1.1%	98.5%	31	__read (libc.so.1: read.s, 20)
[4]	0.240	0.9%	99.4%	24	cvtttrap (generic: generic.c, 317)
[5]	0.150	0.5%	99.9%	15	_xstat (libc.so.1: xstat.s, 12)
[6]	0.010	0.0%	100.0%	1	__write (libc.so.1: write.s, 20)
[7]	0.010	0.0%	100.0%	1	_morecore (libc.so.1: malloc.c, 632)
	27.290	100.0%	100.0%	2729	TOTAL

Example Displaying Data in WorkShop

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

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

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

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


Example Using the `-v` Option

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

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

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

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

```
fraser 75% ssrun -v -pcsampx generic  
  
ssrun: target PID 12345  
ssrun: setenv _SPEEDSHOP_MARCHING_ORDERS pc,4,10000,0:cu  
ssrun: setenv _SPEEDSHOP_EXPERIMENT_TYPE pcsampx  
ssrun: setenv _SPEEDSHOP_TARGET_FILE generic  
ssrun: setenv _RLD_LIST libss.so:libssrt.so:DEFAULT  
...
```

The `_RLD32_LIST` environment variable is used with programs compiled with the `-n32` compiler option. The `_RLD64_LIST` environment variable is used with programs compiled with the `-64` compiler option. If neither is set, the value of `_RLD_LIST` is the default. See the `rld(1)` man page for more information.

Using `ssrun` with a Debugger

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

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

1. Call `ssrun` as follows:

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

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

2. Note the process ID returned by `ssrun`.
3. In another window, start your debugging session as follows:

```
% cvd -pid process_id_number
```

4. Attach the process to the debugger.
5. Run the process from the debugger.

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

You can also use a debugger to set calipers for the purpose of recording performance data for a part of your program. See "Using Calipers", page 92, 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 passing data messages to other processes, even those running on distant computers.

SpeedShop offers two types of experiments for MPI programs; see "Generating MPI Tracing Experiments" which follows for more information.

- **MPI tracing experiments:** traces the use of MPI send, receive, and synchronization routines and a few other routines.
- **Other SpeedShop experiments:** generates other SpeedShop experiments, such as `usertime` and `pcsamp`.

See the *MPI Programmer's Manual* for details about MPI use.

Generating MPI Tracing Experiments

Two different MPI experiments are available to help you trace calls to MPI routines. The main difference in the two is how the results can be viewed:

- **MPI_trace experiments** tell you how many times, and at what locations within the application, various routines from the MPI library are called. This is run by using

the `-mpi_trace` option to `ssrun` and it produces a file that is viewable in the Performance Analyzer (`cvperf(1)`) window.

- **MPI experiments** trace calls to MPI functions and collect data such as the time used by the call, which thread and MPI rank made the call, and so on. This type of experiment is generated using the `-mpi` option to `ssrun`. The generated data can then be analyzed using `prof(1)`.

The `ranks` option to `ssrun` specifies that performance data should only be collected for the MPI ranks in the comma-separated list used with `ranks`. See the `ssrun(1)` man page for a list of the functions traced by each option and for more information about the `ranks` option.

The following example demonstrates the `mpi_trace` option. You can use either of the following versions of the `ssrun` command on an executable named `a.out`:

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

If you are running the application on four processors, you will see five output files: one for each processor and one for the master process. The identifier portions of the file names will start either with `m` for the master process or `f` (forked) for a process running on one of the processors. If the first version of the `ssrun` command, illustrated above, is used with an executable named `myprog`, file names similar to the following will be assigned to the output:

```
myprog.mpi.m12345
myprog.mpi.f12346
myprog.mpi.R0.f12346
myprog.mpi.R1.f12347
myprog.mpi.R2.f12348
myprog.mpi.R3.f12349
```

The `Rx` identifier does not correspond to a processor number but it does correspond to the MPI rank of the process for which the file was generated.

Depending on which option is used, output from the `ssrun` command can be viewed in the **WorkShop Performance Analyzer** window or by using the `prof(1)` command. You can bring up the Performance Analyzer with the `cvperf(1)` command. You can view the information in either graphical or numerical format. Graphs that do not contain data are not displayed. For an example of a portion of a numerical display, see Figure 6-1, page 89.

Note: The MPI tracing experiment does not track down communicators, and it does not trace all collective operations.

Caliper interval: 1 (time=0.218s) to 455 (time=571.273s)

Retries allocating mpi headers:	
per proc for collective calls	0
per host for collective calls	0
per proc for pt2pt calls	0
per host for pt2pt calls	0
Retries allocating mpi buffers:	
per proc for collective calls	0
per host for collective calls	0
per proc for pt2pt calls	0
per host for pt2pt calls	0
Send requests using:	
shared memory for collective calls	97
shared memory for pt2pt calls	2742
hippi bypass for collective calls	0
hippi bypass for pt2pt calls	0
tcp/ip for collective calls	0
tcp/ip for pt2pt calls	0
Data buffers sent using:	
shared memory for pt2pt calls	2695
shared memory for collective calls	11
hippi bypass for pt2pt calls	0
hippi bypass for collective calls	0
tcp/ip for pt2pt calls	0
tcp/ip for collective calls	0
Message headers sent using:	
shared memory for collective calls	97
shared memory for pt2pt calls	2804
hippi bypass for collective calls	0
hippi bypass for pt2pt calls	0
tcp/ip for collective calls	0
tcp/ip for pt2pt calls	0
Bytes sent using:	
shared memory for pt2pt calls	4779840
shared memory for collective calls	16056
hippi bypass for pt2pt calls	0
hippi bypass for collective calls	0
tcp/ip for pt2pt calls	0
tcp/ip for collective calls	0

Figure 6-1 MPI Numerical Format

For a description of the use of the `prof` command, see "Running Experiments", page 81, for examples of the use of `ssrun` and `prof`.

Generating Other Experiments for Programs Using MPI

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

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

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

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

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

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

The second method is to use one of the following:

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

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

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

Running Experiments on Programs Using Pthreads

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

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

Running Experiments on Programs That Use OpenMP Directives

The OpenMP Fortran API and the OpenMP C/C++ API specify a collection of compiler directives, library functions, and environment variables that can be used to specify shared memory parallelism in Fortran, C, or C++ programs. The `-mp` compiler option causes OpenMP directives to be used in creating an executable that may be run using one or more processors.

Performance data collection is done on a per-processor basis. If an executable named `test1` is run under the `ssrun` command using n processors for a `usertime` experiment, then files similar to the following are created for the performance data:

```
test1.usertime.m109327
test1.usertime.T0.p109331
test1.usertime.T1.p109345
test1.usertime.T2.p109353
```

The `Tx` identifier is the number of the OpenMP thread that generated the file. The number of processors may be specified internally in the program using a call to an OpenMP subroutine variable or function `omp_set_num_threads`, or externally via the environment variable `OMP_NUM_THREADS`. The experiment output may be examined via `prof` using the file for each process, or `ssaggregate` may be used to

create an aggregated file from all of the experiment files. Then the results for the entire experiment could be analyzed at once.

Using Calipers

In some cases, you may want to generate performance data reports for only a part of your program. You can do this by selecting caliper points to identify the area of your program or the time interval during execution for which you want to see performance data. When you run `prof`, you can specify a region for which to generate a report by supplying the `-calipers` option and the appropriate caliper numbers. For more information on `prof -calipers`, see "Using the `-calipers` Option", page 123.

Table 6-1, page 93, shows the different ways you can set caliper points.

Table 6-1 Setting Caliper Points

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

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

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

- When a fatal signal is received, such as `SIGQUIT`, `SIGILL`, `SIGTRAP`, `SIGABRT`, `SIGEMT`, `SIGFPE`, `SIGBUS`, `SIGSEGV`, `SIGSYS`, `SIGXCPU`, or `SIGXFSZ`. Note that this list does not and cannot include `SIGKILL`.
- When the program calls an `exec` function, such as `execve()` or `execvp()`.
- When an exit signal is received, such as `SIGHUP`, `SIGINT`, `SIGPIPE`, `SIGALRM`, `SIGTERM`, `SIGUSR1`, `SIGUSR2`, `SIGPOLL`, `SIGIO`, `SIGRTMIN`, or `SIGRTMAX`.

Setting Calipers with the `ssrt_caliper_point` Function

To set caliper points using the `ssrt_caliper_point(3)` function, follow these steps:

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

Example for C:

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

Example for Fortran:

```
. . .
INTEGER SSRT_CALIPER_POINT
. . .
i = SSRT_CALIPER_POINT (1, 'bgn_calc')
. . .
```

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

2. Link the SpeedShop library `libss.so` into your application. Place the `-lss` option at the end of your compile or link command so that the library is the last to be referenced.
3. Run your program with `ssrun` and the desired experiment type. For example, if you want to run the `bbcounts` experiment on `generic`:

```
% ssrun -bbcounts generic
```

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

Setting Time-Oriented Calipers

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

```
_SPEEDSHOP_POLLPOINT_CALIPER_POINT timer_type, timer_interval
```

The arguments are as follows:

- *timer_type*: can have one of the following values:

- 0: Real time. This is the total time a program spent while executing. It includes both time spent when a program is swapped out waiting for a CPU and the time the operating system is in control, performing some task for the program such as I/O or executing a system call.
 - 1: process virtual time. This is the time spent when the program is actually running. This does not include either the time spent when a program is swapped out waiting for a CPU or the time the operating system is in control, performing some task for the program such as I/O or executing another system call.
 - 2: CPU time. This is process virtual time plus the time the system is running on behalf of the process. The system time could include performing I/O or executing other system calls.
- *timer_interval*: the integer interval, in seconds, at which a new caliper will be set.

The caliper points you have set with the `_SPEEDSHOP_POLLPOINT_CALIPER_POINT` environment variable are recorded in the performance data file that is generated by `ssrun`. For the usertime experiment, *timer_type* must be 2.

Setting Calipers with Signals

To set calipers with signals, follow these steps:

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

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

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

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

2. Execute your program with `ssrun`.
3. In another window, enter a command such as `ps` or `top` to determine the process ID of `ssrun`. This is also the process ID of the program you are working on.

4. In this window, send the signal you used in step 1 to the process using the `kill` command:

```
% kill -sig_num pid
```

Caliper point data is recorded at the point in the program where the signal sent by the `kill` command interrupts the executing `ssrun` process.

Setting Calipers with a Debugger

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

Use the following procedure:

1. Set a breakpoint in your program where you want a caliper point.
2. When the process stops, evaluate the expression `ssrt_caliper_point(3)`. The evaluation of the expression always returns zero, but a side effect of the evaluation is the recording of the appropriate data.
3. Resume execution of the process.

Effects of `ssrun`

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

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

For more information on these environment variables, see "Using Run-Time Environment Variables", page 74.

- Inserts the SpeedShop libraries `libss.so` and `libssrt.so` as part of your executable using the environment variable `_RLD_LIST`.
- Invokes the file `executable_name` by calling `exec()`.

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

Analyzing Experiment Results

This chapter provides information on how to view and analyze experiment results by using the `prof(1)` report generator. This chapter has the following sections:

- "Using `prof` to Generate Performance Reports", page 99
- "Using `prof` with `ssrun`", page 105
- "Using `prof` Options", page 114
- "Generating Reports for Different Machine Types", page 128
- "Generating Reports for Multiprocessed Executables", page 129
- "Determining Program Overhead", page 129
- "Generating Compiler Feedback Files", page 132
- "Comparing Experiment Results", page 132

Using `prof` to Generate Performance Reports

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

Use either of the following syntaxes to generate a report from performance data gathered during experiments recorded by `ssrun(1)`:

```
prof [options] [speedshop_data_file] . . .
```

or

```
prof [options] executable_name [speedshop_data_file] . . .
```

`prof` Arguments

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

- *options*: zero or more of the options described in Table 7-1, page 100.
- *executable_name*: the name of the executable file (including its path) created by the compiler. This argument is needed if `prof` is unable to locate the executable relative to the location of the data files being analyzed because the data or the executable were moved after the files were created.
- *speedshop_data_file*: one or more names of performance data files generated by `ssrun`. The file names may differ only in the ID portion of their names. The *exp_type* portion of the names must be identical.

prof Options

The following table lists `prof` options that are current for this release. For more information and for a list of any newly added options since this printing, see the `prof(1)` man page.

Table 7-1 Options for `prof`

Name	Result
<code>-archinfo</code>	Reports the number of times each register was used as a destination, base (integer registers only), or source; how many times each instruction opcode was used; and some detailed statistics concerning branches jumps, and how many delay slots were filled with no-op instructions. Works only with <code>bbcounts</code> experiments.
<code>-basicblocks</code>	Prints a list of all the basic blocks executed, ordered by the number of cycles spent in each basic block. Works only with <code>bbcounts</code> experiments.
<code>-b[utterfly]</code>	Causes <code>prof</code> to print a report showing the callers and callees of each function, with inclusive time attributed to each. For <code>bbcounts</code> experiments, the attribution is based on a heuristic. For the various callstack sampling and tracing experiments, the attribution is precise. The <code>usertime</code> , <code>totaltime</code> , and some <code>_hwctime</code> experiments are statistical in nature and so are not exact. This option is ignored for experiments in which the data does not support inclusive calculations. It delivers the same display as <code>-gprof</code> .
<code>-calipers [n1],[n2]</code>	Restricts analysis to a segment of program execution. This option works only for SpeedShop experiments.

Name	Result																				
	Causes <code>prof</code> to compute the data between caliper points <i>n1</i> and <i>n2</i> , rather than for the entire experiment. If <i>n1</i> >= <i>n2</i> , an error is reported. If <i>n2</i> is greater than the maximum number of caliper points recorded, it is set to the maximum. If <i>n1</i> is omitted, zero (the beginning of the program) is assumed.																				
<code>-calls</code>	Sorts the function list by the number of procedure calls rather than by time. This option can only be used when generating reports for <code>bbcounts</code> experiments.																				
<code>-clock n</code>	Sets the CPU clock speed to (<i>n</i>), expressed in megahertz. This option is useful when generating reports for <code>bbcounts</code> experiments. The default is the clock speed of the machine on which the performance data was collected.																				
<code>-[no]cordfb</code>	Enables or disables (<code>-nocordfb</code>) cord feedback file generation for the executable only. Cord feedback is used to arrange procedures in the binary in an optimal ordering. This improves both paging and instruction cache performance. Users can use <code>cord(1)</code> or <code>ld(1)</code> to actually do the procedure ordering.																				
<code>-cordfball</code>	Enables cord feedback for the executable and all DSOs.																				
<code>-cycle n</code>	Sets the cycle time to <i>n</i> nanoseconds. This parameter may be used as another way of setting the clock speed. See also the description for <code>-clock n</code> .																				
<code>-debug:dbg_flags</code>	Sets <i>dbg_flags</i> . <i>dbg_flag</i> should be specified as a hexadecimal value made by adding up combinations of the hexadecimal values listed below (Example: <code>-debug:0x00000102</code>): <table border="0"> <tbody> <tr> <td><code>GPROF_FLAG</code></td> <td><code>0x00000001</code></td> </tr> <tr> <td><code>COUNTS_FLAG</code></td> <td><code>0x00000002</code></td> </tr> <tr> <td><code>SAMPLE_FLAG</code></td> <td><code>0x00000004</code></td> </tr> <tr> <td><code>MISS_FLAG</code></td> <td><code>0x00000008</code></td> </tr> <tr> <td><code>FEEDBACK_FLAG</code></td> <td><code>0x00000010</code></td> </tr> <tr> <td><code>CORD_FLAG</code></td> <td><code>0x00000020</code></td> </tr> <tr> <td><code>USERPC_FLAG</code></td> <td><code>0x00000040</code></td> </tr> <tr> <td><code>MDEBUG_FLAG</code></td> <td><code>0x00000080</code></td> </tr> <tr> <td><code>BEAD_FLAG</code></td> <td><code>0x00000100</code></td> </tr> <tr> <td><code>LIBSSRT_FLAG</code></td> <td><code>0x00000200</code></td> </tr> </tbody> </table>	<code>GPROF_FLAG</code>	<code>0x00000001</code>	<code>COUNTS_FLAG</code>	<code>0x00000002</code>	<code>SAMPLE_FLAG</code>	<code>0x00000004</code>	<code>MISS_FLAG</code>	<code>0x00000008</code>	<code>FEEDBACK_FLAG</code>	<code>0x00000010</code>	<code>CORD_FLAG</code>	<code>0x00000020</code>	<code>USERPC_FLAG</code>	<code>0x00000040</code>	<code>MDEBUG_FLAG</code>	<code>0x00000080</code>	<code>BEAD_FLAG</code>	<code>0x00000100</code>	<code>LIBSSRT_FLAG</code>	<code>0x00000200</code>
<code>GPROF_FLAG</code>	<code>0x00000001</code>																				
<code>COUNTS_FLAG</code>	<code>0x00000002</code>																				
<code>SAMPLE_FLAG</code>	<code>0x00000004</code>																				
<code>MISS_FLAG</code>	<code>0x00000008</code>																				
<code>FEEDBACK_FLAG</code>	<code>0x00000010</code>																				
<code>CORD_FLAG</code>	<code>0x00000020</code>																				
<code>USERPC_FLAG</code>	<code>0x00000040</code>																				
<code>MDEBUG_FLAG</code>	<code>0x00000080</code>																				
<code>BEAD_FLAG</code>	<code>0x00000100</code>																				
<code>LIBSSRT_FLAG</code>	<code>0x00000200</code>																				

Name	Result
-dis[assemble]	Disassembles and annotates the analyzed object code with cycle times if you have run an <code>bbcounts</code> experiment, collected data using <code>pixie</code> , or have run <code>apcsamp</code> or <code>_hwc/_hwctime</code> experiment.
-dislimit <i>n</i>	Disassembles only those basic blocks with a frequency $\geq n\%$. This option applies to the same experiments as the <code>-disassemble</code> option.
-dso <i>dsoname</i>	Generates a report only for the named DSO. Only the base name, not the full path name, of the DSO needs to be specified; the <code>.so</code> suffix is required. Multiple instances of the <code>-dso</code> flag can be given.
-dsolist	List all the DSOs in the program and their start and end text addresses.
-e[xclude] <i>procedure_name</i>	If you use one or more <code>-e[xclude]</code> options, the profiler omits the specified procedure from the listing. If any option uses an upper-case E for <code>-E[xclude]</code> , <code>prof</code> also omits that procedure from the base upon which it calculates percentages.
-feedback	Produces files with information that can be used to arrange procedures in the binary in an optimal ordering using <code>cord</code> . The <code>cord</code> feedback files are named <i>program.fb</i> or <i>libso.fb</i> . Procedures are normally ordered by their measured invocation counts; if <code>-gprof</code> is also specified, procedures are ordered using call graph counts, rather than invocation counts.
-fpe_counts	For FPE experiments, this option restricts the output to only show data from FPE events of type <i>n</i> . The default is to display the combined results from all FPE events.
-fpe_type <i>n</i>	Used with FPE experiments. Restricts the output to only show data from FPE events of type <i>n</i> . The default is to display the combined results from all FPE events.
-gprof	(See <code>-b[utterfly]</code> .)
-h[eavy]	Lists the most heavily used lines of source code in descending order of use, sorting lines by their execution time. This option can be used when generating reports for <code>bbcounts</code> , <code>pcsamp</code> , or <code>_hwc</code> experiments.
-inclusive	Sorts function list by inclusive data rather than by exclusive data. This option can only be used when generating reports for those experiments that have inclusive data; it is ignored for others.

Name	Result
-l[ines]	Lists the most heavily used lines of source code in descending order of use, but lists lines grouped by procedure, sorted by cycles executed per procedure. This option can be used when generating reports for <code>bbcounts</code> , <code>pcsamp</code> , or <code>_hwc</code> experiments.
-nh	Suppresses various header blocks from the output.
-o[nly] <i>procedure_name</i>	If you use one or more <code>-o[nly]</code> options, the profile listing includes only the named procedures, rather than the entire program. If any option uses an uppercase <code>-O[nly]</code> , <code>prof</code> uses only the named procedures, rather than the entire program, as the base upon which it calculates percentages.
-overhead	Generates overhead data for a parallel program. Overhead data includes how much time was spent when the program had no parallel work to do, how much time was lost when work was not spread evenly among the processors, and so on.
-pthreads <i>pthread_id</i>	Analyzes data only for the specified pthread identifier (for <code>usertime</code> , <code>totaltime</code> , <code>_hwctime</code> , <code>io</code> , and <code>fpe</code> experiments on applications that use pthreads on IRIX 6.5 or later systems). <i>pthread_id</i> may be a list of pthread identifiers separated by commas.
-q[uit] <i>n</i>	Condenses output listings by truncating <code>-h[eavy]</code> , <code>-l[ines]</code> , and <code>-gprof</code> listings. You can specify <i>n</i> in three ways: <i>n</i> , an integer, truncates everything after <i>n</i> functions are listed; <i>n%</i> , an integer followed by a percent sign, truncates the listing after the first entry that represents less than <i>n</i> percent of the total; <i>ncum%</i> , an integer followed by <code>cum%</code> , truncates the listing after enough entries have been printed to account for <i>n</i> percent of the cumulative total. If <code>-b[utterfly]</code> is also specified, it behaves the same as <code>-q n%</code> . For example, <code>-q 15</code> truncates each part of the report after 15 lines of text, <code>-q 15%</code> truncates each part after the first entry that represents less than 15 percent of the whole, and <code>-q 15cum%</code> truncates each part after the entry that brought the cumulative percentage above 15%.
-rel[ative]	Shows percentage attribution in a butterfly report relative to the central function. The default is to show percentages as absolute percentages over the whole run.

Name	Result
<code>-repository <i>directory</i></code>	Uses the SpeedShop DSO information from <i>directory</i> instead of processing the DSO in memory. This reduces the time it takes to retrieve source file information, procedure name, and address from the DSO. If you have multiple SpeedShop DSO repositories, you can use this option multiple times to define all the needed repositories. It can also be used to save the experiments because saving the DSO information will allow <code>prof</code> to not use the original DSO (which may have been modified). The <i>directory</i> is used in Read-Only mode.
<code>-r16000 r14000 -r12000 -r10000</code>	Overrides the default processor scheduling model that <code>prof</code> uses to generate a report. If this option is not specified, <code>prof</code> uses the scheduling model for the processor on which the experiment is being run. These options are only meaningful for an <code>bbcounts</code> time experiment or <code>pixie</code> count data.
<code>-showss</code>	Enables the display of functions from the SpeedShop run-time DSO. Usually those functions are suppressed from the reports and computations. In addition, some statistics for the <code>prof</code> command's own memory usage will be printed.
<code>-S (-source)</code>	Disassembles and annotates the analyzed object code with cycle times, or PC samples, and interleaves and lists the source code, if you have run a <code>bbcounts</code> , <code>pcsamp</code> , or <code>_hwc</code> experiment.
<code>-update_repository <i>directory</i></code>	Processes the DSO and stores the information in <i>directory</i> . This option can only be specified once, and is required to save the DSO information. SpeedShop DSO information files have a <code>_ssInfoabi</code> extension, depending on the <i>abi</i> (32, n32, or 64). The <i>directory</i> is used in Read-Write mode.
<code>-u[sage]</code>	Prints a report on system statistics and timers.
<code>-ws</code>	Generates, for the executable only, a working-set file for the current caliper setting. This option is only meaningful for a <code>bbcounts</code> time experiment or <code>pixie</code> count data. The file suffix is <code>.ws</code> .
<code>-wsall</code>	Generates, for the executable and all the non-ignored DSOs, a working-set file for the current caliper setting. This option is only meaningful for a <code>bbcounts</code> time experiment. The file suffix is <code>.ws</code> .
<code>-xdso <i>dso_name</i></code>	Excludes the named DSO from any reports. Only the base name, not the full path name, of the DSO need be specified; the <code>.so</code> suffix is required. Multiple instances of the <code>-xdso</code> flag can be specified.

prof Output

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

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

Using prof with ssrun

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

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

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

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

- "usertime Experiment Reports", page 106
- "pcsamp Experiment Reports", page 107
- "Hardware Counter Experiment Reports", page 108
- "bbcounts Experiment Reports", page 110
- "fpe Trace Reports", page 113

usertime Experiment Reports

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

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

The following example is an abbreviated version of the full report. For a complete report, see "Generating a Report", page 18.

```
-----  
SpeedShop profile listing generated Mon Feb  2 11:07:15 1998  
  prof generic.usertime.m10981  
    generic (n32): Target program  
      usertime: Experiment name  
      ut:cu: Marching orders  
R4400 / R4000: CPU / FPU  
      1: Number of CPUs  
      175: Clock frequency (MHz.)  
  
Experiment notes--  
  From file generic.usertime.m10981:  
  Caliper point 0 at target begin, PID 10981
```

```

                /usr/demos/SpeedShop/linpack.demos/c/generic
Caliper point 1 at exit(0)
-----
Summary of statistical callstack sampling data (usertime)--
      809: Total Samples
        0: Samples with incomplete traceback
    24.270: Accumulated Time (secs.)
     30.0: Sample interval (msecs.)
-----
Function list, in descending order by exclusive time
-----
[index]  excl.secs  excl.%   cum.%  incl.secs  incl.%   samples  function (dso: file, line)
-----
      [4]    22.770   93.8%   93.8%    22.770   93.8%       759  anneal (generic: generic.c, 1573)

```

pcsamp Experiment Reports

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

- The [*index*] column assigns a reference number to each function.
- The *secs* column shows the amount of CPU time that was spent in the function.
- The % column shows the percentage of the total program time that was spent in the function.
- The *cum.%* column shows the percentage of the complete program time that has been spent in the functions that have been listed so far.
- The *samples* column shows how many samples were taken when the process was executing in the function.
- The *function (dso:file, line)* column lists the function, its DSO name, its file name, and its line number.

The following is output from an *fpcsamp* experiment:

```

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

```

7: Analyzing Experiment Results

```

          fpcsamp: Experiment name
pc,2,1000,0:cu: Marching orders
          R4400 / R4000: CPU / FPU
                  1: Number of CPUs
                  175: Clock frequency (MHz.)

Experiment notes--
          From file generic.fpcsamp.m11140:
          Caliper point 0 at target begin, PID 11140
                  /usr/demos/SpeedShop/linpack.demos/c/generic
          Caliper point 1 at exit(0)

-----
Summary of statistical PC sampling data (fpcsamp)--
          23828: Total samples
          23.828: Accumulated time (secs.)
          1.0: Time per sample (msecs.)
          2: Sample bin width (bytes)

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

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

```

Hardware Counter Experiment Reports

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

- The [index] column assigns a reference number to each function.
- The counts column shows the extrapolated event count based on the number of samples and the overflow value for the particular counter.
- The % column shows the percentage of the program's overflows that occurred in the function.
- The cum.% column shows the percentage of the program's overflows that occurred in the functions that have been listed so far.
- The samples column shows the number of times the program counter was sampled during execution of the function.

- The `function (dso:file, line)` column lists the name, the DSO, the file name, and line number of the function.

The following is output from a `dsc_hwc` hardware counter experiment:

7: Analyzing Experiment Results

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

Experiment notes--
  From file generic.dsc_hwc.m294398:
  Caliper point 0 at target begin, PID 294398
    /usr/demos/SpeedShop/linpack.demos/c/generic
  Caliper point 1 at exit(0)

-----
Summary of R10K perf. counter overflow PC sampling data (dsc_hwc)--
      6: Total samples
  Sec cache D misses (26): Counter name (number)
      131: Counter overflow value
      786: Total counts

-----
Function list, in descending order by counts
-----
[index]      counts      %   cum.%   samples  function (dso: file, line)

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

          786 100.0% 100.0%         6  TOTAL
```

bbcounts Experiment Reports

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

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

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

- The `[index]` column assigns a reference number to each function.
- The `excl.secs` column shows the minimum number of seconds that might be spent in the function under `bbcounts` conditions.
- The `excl.%` column represents how much of the program's total time was spent in the function.
- The `cum.%` column shows the cumulative percentage of time spent in the functions that have been listed so far.
- The `cycles` column reports the number of machine cycles used by the function.
- The `instructions` column shows the number of instructions executed by a function.
- The `calls` column reports the number of calls to the function.
- The `function (dso:file, line)` column lists the function, its DSO name, its file name, and the line number.

The following is output from a `bbcounts` experiment.

```
-----
SpeedShop profile listing generated Mon Aug 14 13:51:00 2000
prof -butterfly generic.bbcounts.m46372
    generic (n32): Target program
        bbcounts: Experiment name
        it:cu: Marching orders
R12000 / R12010: CPU / FPU
        127: Number of CPUs
        400: Clock frequency (MHz.)

Experiment notes--
    From file generic.bbcounts.m46372:
    Caliper point 0 at target begin, PID 46372
        generic
    Caliper point 1 at exit(0)
```

7: Analyzing Experiment Results

```
-----
Summary of bccounts time data (bccounts)--
      2048835049: Total number of instructions executed
      2552056463: Total computed cycles
           6.380: Total computed execution time (secs.)
           1.246: Average cycles / instruction
-----
```

Function list, in descending order by exclusive bccounts time

```
-----
```

[index]	excl.secs	excl.%	cum.%	cycles	instructions	incl.secs	incl.%
	calls	function	(dso: file, line)				
[5]	6.088	95.4%	95.4%	2435240026	1956780024	6.088	95.4%
	1	anneal	(generic: generic.c, 1559)				

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

```
-----
Butterfly function list, in descending order by inclusive bccounts time
-----
```

[index]	incl.%	incl.time	self%	self-time	procedure [index]	incl.time	caller (callsite) [index]
[1]	99.9%	6.376	0.0%	0.000	__start [1]		
			99.9%	6.376(0000001)		6.376	main [2]
			0.0%	0.000(0000001)		0.000	__readenv_sigfpe [314]
			0.0%	0.000(0000001)		0.000	__istart [315]
	99.9%	6.376(0000001)				6.376	__start [1]
[2]	99.9%	6.376	0.0%	0.000	main [2]		
			99.9%	6.376(0000001)		6.376	Scriptstring [3]
	99.9%	6.376(0000001)				6.376	main [2]
[3]	99.9%	6.376	0.0%	0.000	Scriptstring [3]		
			95.4%	6.088(0000001)		6.088	usertime [4]
			3.7%	0.238(0000001)		0.238	libdso [6]
			0.8%	0.050(0000001)		0.050	cvtrap [9]
			0.0%	0.000(0000001)		0.000	iofile [31]
			0.0%	0.000(0000002)		0.000	genLog [36]
			0.0%	0.000(0000001)		0.000	dirstat [56]

		0.0%	0.000(0000001)	0.000	linklist [63]	
		0.0%	0.000(0000001)	0.000	fpetraps [65]	
		0.0%	0.000(0000002)	0.000	fprintf [54]	
		0.0%	0.000(0000002)	0.000	sprintf [49]	
		0.0%	0.000(0000061)	0.000	strcmp [47]	

	95.4%		6.088(0000001)	6.376	Scriptstring [3]	
[4]	95.4%	6.088	0.0%	0.000	usertime [4]	
			95.4%	6.088(0000001)	6.088	anneal [5]
			0.0%	0.000(0000001)	0.000	genLog [36]
			0.0%	0.000(0000001)	0.000	fprintf [54]

fpe Trace Reports

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

- The [index] column assigns a reference number to each function.
- The excl.FPEs column shows how many floating point exceptions were found in the function.
- The excl.% column shows the percentage of the total number of floating-point exceptions that were found in the function.
- The cum.% column shows the percentage of floating-point exceptions in the program that have been encountered so far in the list.
- The incl.FPEs column shows how many floating-point exceptions were attributed to the function and descendents of the function.
- The incl.% column shows the cumulative percentage of floating-point exceptions attributed to the function and its descendents.
- The function (dso:file, line) column lists the function name, its DSO name, its file name, and its line number.

```
-----
SpeedShop profile listing generated Mon Feb  2 13:26:33 1998
  prof generic.fpe.m12213
    generic (n32): Target program
      fpe: Experiment name
```

7: Analyzing Experiment Results

```

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

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

```

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

```
-----
SpeedShop profile listing generated Tue Feb  3 10:48:59 1998
prof -dis generic.pcsamp.m14493
    generic (n32): Target program
        pcsamp: Experiment name
pc,2,10000,0:cu: Marching orders
R4400 / R4000: CPU / FPU

```

1: Number of CPUs
175: Clock frequency (MHz.)

Experiment notes--

From file generic.pcsamp.m14493:
Caliper point 0 at target begin, PID 14493
/usr/demos/SpeedShop/c/generic
Caliper point 1 at exit(0)

Summary of statistical PC sampling data (pcsamp)--

2707: Total samples
27.070: Accumulated time (secs.)
10.0: Time per sample (msecs.)
2: Sample bin width (bytes)

Function list, in descending order by time

[index]	secs	%	cum.%	samples	function (dso: file, line)
[1]	25.240	93.2%	93.2%	2524	anneal (generic: generic.c, 1573)
[2]	1.090	4.0%	97.3%	109	slaveusrtime (dlslave.so: dlslave.c, 22)
[3]	0.390	1.4%	98.7%	39	__read (libc.so.1: read.s, 20)
[4]	0.230	0.8%	99.6%	23	cvtttrap (generic: generic.c, 317)
[5]	0.090	0.3%	99.9%	9	__xstat (libc.so.1: xstat.s, 12)
[6]	0.010	0.0%	99.9%	1	__write (libc.so.1: write.s, 20)
[7]	0.010	0.0%	100.0%	1	__ngetdents (libc.so.1: ngetdents.s, 16)
[8]	0.010	0.0%	100.0%	1	__doprnt (libc.so.1: doprnt.c, 285)
	27.070	100.0%	100.0%	2707	TOTAL

Disassembly listing, annotated with PC sampling overflows

.
.
.
/usr/demos/SpeedShop/linpack.demos/c/generic.c
anneal: <0x10006830-0x10006b3c> 2524 total samples(93.24%)
[1573] 0x10006830 0x27bdffd0 addiu sp,sp,-48 # 1
[1573] 0x10006834 0xffbc0020 sd gp,32(sp) # 2
[1573] 0x10006838 0xffbf0018 sd ra,24(sp) # 3
[1573] 0x1000683c 0x3c030002 lui v1,0x2 # 4
[1573] 0x10006840 0x246397e8 addiu v1,v1,-26648 # 5

7: Analyzing Experiment Results

```
[1573] 0x10006844 0x0323e021 addu gp,t9,v1 # 6
[1575] 0x10006848 0xd7808370 ldc1 $f0,-31888(gp) # 7
<2 cycle stall for following instruction>
[1575] 0x1000684c 0xf7a00000 sdc1 $f0,0(sp) # 10
[1577] 0x10006850 0x24010001 li at,1 # 11
[1577] 0x10006854 0x8f82816c lw v0,-32404(gp) # 12
<2 cycle stall for following instruction>
[1577] 0x10006858 0xac410000 sw at,0(v0) # 15
[1578] 0x1000685c 0x8f998148 lw t9,-32440(gp) # 16
[1578] 0x10006860 0x0c00171b jal 0x10005c6c # 17
[1578] 0x10006864 0000000000 nop # 18
<2 cycle stall for following instruction>
[1586] 0x10006868 0xafa00008 sw zero,8(sp) # 21
[1586] 0x1000686c 0x8fa40008 lw a0,8(sp) # 22
<2 cycle stall for following instruction>
[1586] 0x10006870 0x28842710 slti a0,a0,10000 # 25
[1586] 0x10006874 0x108000ac beq a0,zero,0x10006b28 # 26
[1586] 0x10006878 0000000000 nop # 27
<2 cycle stall for following instruction>
[1588] 0x1000687c 0x24070001 li a3,1 # 30
[1588] 0x10006880 0xafa7000c sw a3,12(sp) # 31
[1588] 0x10006884 0x8f868164 lw a2,-32412(gp) # 32
<2 cycle stall for following instruction>
[1588] 0x10006888 0x8cc60000 lw a2,0(a2) # 35
<2 cycle stall for following instruction>
[1588] 0x1000688c 0x24c6ffff addiu a2,a2,-1 # 38
[1588] 0x10006890 0x8fa5000c lw a1,12(sp) # 39
<2 cycle stall for following instruction>
[1588] 0x10006894 0x00a6282a slt a1,a1,a2 # 42
[1588] 0x10006898 0x10a0009c beq a1,zero,0x10006b0c # 43
[1588] 0x1000689c 0000000000 nop # 44
<2 cycle stall for following instruction>
[1589] 0x100068a0 0x240a0001 li t2,1 # 47
^----- 1 samples(0.04%)-----^
[1589] 0x100068a4 0xafaa0010 sw t2,16(sp) # 48
^----- 1 samples(0.04%)-----^
[1589] 0x100068a8 0x8f898164 lw t1,-32412(gp) # 49
<2 cycle stall for following instruction>
[1589] 0x100068ac 0x8d290000 lw t1,0(t1) # 52
<2 cycle stall for following instruction>
[1589] 0x100068b0 0x2529ffff addiu t1,t1,-1 # 55
```



```
[1589] 0x100068b4 0x8fa80010 lw t0,16(sp) # 56
<2 cycle stall for following instruction>
[1589] 0x100068b8 0x0109402a slt t0,t0,t1 # 59
[1589] 0x100068bc 0x11000089 beq t0,zero,0x10006ae4 # 60
[1589] 0x100068c0 0000000000 nop # 61
<2 cycle stall for following instruction>
[1590] 0x100068c4 0x8faf000c lw t7,12(sp) # 64
^----- 27 samples(1.00%)-----^
<2 cycle stall for following instruction>
[1590] 0x100068c8 0x25ef0001 addiu t7,t7,1 # 67
^----- 7 samples(0.26%)-----^
[1590] 0x100068cc 0x000f7080 sll t6,t7,2 # 68
^----- 30 samples(1.11%)-----^
[1590] 0x100068d0 0x01cf7021 addu t6,t6,t7 # 69
^----- 8 samples(0.30%)-----^
[1590] 0x100068d4 0x000e70c0 sll t6,t6,3 # 70
^----- 5 samples(0.18%)-----^
[1590] 0x100068d8 0x8faf0010 lw t7,16(sp) # 71
^----- 8 samples(0.30%)-----^
<2 cycle stall for following instruction>
[1590] 0x100068dc 0x01cf7021 addu t6,t6,t7 # 74
^----- 9 samples(0.33%)-----^
[1590] 0x100068e0 0x000e70c0 sll t6,t6,3 # 75
^----- 27 samples(1.00%)-----^
[1590] 0x100068e4 0x8f8f817c lw t7,-32388(gp) # 76
^----- 14 samples(0.52%)-----^
<2 cycle stall for following instruction>
[1590] 0x100068e8 0x01cf7021 addu t6,t6,t7 # 79
^----- 9 samples(0.33%)-----^
[1590] 0x100068ec 0x25ce0008 addiu t6,t6,8 # 80
^----- 28 samples(1.03%)-----^
[1590] 0x100068f0 0xd5c10000 ldc1 $f1,0(t6) # 81
^----- 7 samples(0.26%)-----^
[1590] 0x100068f4 0x8fad000c lw t5,12(sp) # 82
^----- 10 samples(0.37%)-----^
<2 cycle stall for following instruction>
[1590] 0x100068f8 0x25ad0001 addiu t5,t5,1 # 85
^----- 21 samples(0.78%)-----^
[1590] 0x100068fc 0x000d6080 sll t4,t5,2 # 86
^----- 19 samples(0.70%)-----^
[1590] 0x10006900 0x018d6021 addu t4,t4,t5 # 87
```

7: Analyzing Experiment Results

```
^----- 9 samples(0.33%)-----^
[1590] 0x10006904 0x000c60c0 sll t4,t4,3 # 88
^----- 14 samples(0.52%)-----^
[1590] 0x10006908 0x8fad0010 lw t5,16(sp) # 89
^----- 8 samples(0.30%)-----^
<2 cycle stall for following instruction>
[1590] 0x1000690c 0x018d6021 addu t4,t4,t5 # 92
^----- 8 samples(0.30%)-----^
[1590] 0x10006910 0x000c60c0 sll t4,t4,3 # 93
^----- 30 samples(1.11%)-----^
[1590] 0x10006914 0x8f8d817c lw t5,-32388(gp) # 94
^----- 10 samples(0.37%)-----^
<2 cycle stall for following instruction>
[1590] 0x10006918 0x018d6021 addu t4,t4,t5 # 97
^----- 8 samples(0.30%)-----^
[1590] 0x1000691c 0xd5820000 ldc1 $f2,0(t4) # 98
^----- 28 samples(1.03%)-----^
[1590] 0x10006920 0x8fab000c lw t3,12(sp) # 99
^----- 9 samples(0.33%)-----^
<2 cycle stall for following instruction>
[1590] 0x10006924 0x256b0001 addiu t3,t3,1 # 102
^----- 11 samples(0.41%)-----^
[1590] 0x10006928 0x000b5080 sll t2,t3,2 # 103
^----- 25 samples(0.92%)-----^
[1590] 0x1000692c 0x014b5021 addu t2,t2,t3 # 104
^----- 11 samples(0.41%)-----^
[1590] 0x10006930 0x000a50c0 sll t2,t2,3 # 105
^----- 8 samples(0.30%)-----^
[1590] 0x10006934 0x8fab0010 lw t3,16(sp) # 106
^----- 11 samples(0.41%)-----^
<2 cycle stall for following instruction>
[1590] 0x10006938 0x014b5021 addu t2,t2,t3 # 109
^----- 7 samples(0.26%)-----^
[1590] 0x1000693c 0x000a50c0 sll t2,t2,3 # 110
^----- 26 samples(0.96%)-----^
[1590] 0x10006940 0x8f8b817c lw t3,-32388(gp) # 111
^----- 13 samples(0.48%)-----^
<2 cycle stall for following instruction>
[1590] 0x10006944 0x014b5021 addu t2,t2,t3 # 114
^----- 9 samples(0.33%)-----^
[1590] 0x10006948 0x254afff8 addiu t2,t2,-8 # 115
```

```

^----- 26 samples(0.96%)-----^
[1590] 0x1000694c 0xd5430000 ldc1 $f3,0(t2) # 116
^----- 11 samples(0.41%)-----^
[1590] 0x10006950 0x8fa9000c lw t1,12(sp) # 117
^----- 10 samples(0.37%)-----^
<2 cycle stall for following instruction>
[1590] 0x10006954 0x00094080 sll t0,t1,2 # 120
^----- 11 samples(0.41%)-----^
.
.
.

```

The listing shows statistics about the procedure `anneal()` in the file `generic.c` and lists the beginning and ending addresses of `anneal()`: `<0x10006830-0x10006b3c>`. The five columns display the following information:

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

Other information includes:

- The number of times the immediately preceding branch was executed and taken (`bbcounts` only).
- The total number of cycles in a basic block and the percentage of the total cycles for that basic block; the number of times the branch terminating that basic block was executed; and the number of cycles for one execution of that basic block (`bbcounts` only).
- The total number of samples at an instruction (`pcsamp` only).
- Any cycle stalls, that is, cycles that were wasted.

Using the -s Option

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

This example shows output from calling `prof` for a `bbcounts` experiment. Note that some lines are wrapped here to accommodate page width:

```
-----
SpeedShop profile listing generated Mon Jul 17 14:45:28 2000
  prof -S generic.bbcounts.m190404
        generic (n32): Target program
            bbcounts: Experiment name
            it:cu: Marching orders
R12000 / R12010: CPU / FPU
            128: Number of CPUs
            400: Clock frequency (MHz.)

Experiment notes--
  From file generic.bbcounts.m190404:
  Caliper point 0 at target begin, PID 190404
        generic
  Caliper point 1 bgn_calc
  Caliper point 2 at exit(0)

-----
Summary of bbcounts time data (bbcounts)--
        2048886059: Total number of instructions executed
        2552098900: Total computed cycles
            6.380: Total computed execution time (secs.)
            1.246: Average cycles / instruction

-----
Function list, in descending order by exclusive bbcounts time
-----
[index]  excl.secs  excl.%   cum.%      cycles  instructions  calls
function (dso: file, line)

    [1]      6.088   95.4%   95.4%    2435240026  1956780024     1
anneal (generic: generic.c, 1560)
    [2]      0.238    3.7%   99.1%     95000839   75000732     1
slaveusrtime (dlslave.so: dlslave.c, 22)
    [3]      0.050    0.8%   99.9%     20000056   15000054     1
cvtrap (generic: generic.c, 317)
```

```

    [4]      0.001    0.0%   99.9%      503138      559313      5212
resolve_relocations (rld: rld.c, 2636)
    [5]      0.001    0.0%  100.0%      274847      282220      1255
general_find_symbol (rld: rld.c, 2038)
    [6]      0.000    0.0%  100.0%      116756      120371         3
fix_all_defined (rld: rld.c, 3419)
    [7]      0.000    0.0%  100.0%      115819      145585      1270
elfhash (rld: obj.c, 1184)
    [8]      0.000    0.0%  100.0%      102496      146324      6406
obj_dynsym_got (rld: objfcn.c, 46)
    [9]      0.000    0.0%  100.0%       89123      116619       948
fread (libc.so.1: fread.c, 27)
    [10]     0.000    0.0%  100.0%       74339       58123         1
init2da (generic: generic.c, 1417)
.
.
.

```

```

-----
disassembly listing
-----

```

```

*DF*_generic.MIPS.stubs_1
*DF*_generic.MIPS.stubs_1@0x10001fd4-0x100023d8: <0x10001fd4-0x100023d8>
.
.
.
/usr/people/n4733/demos/SpeedShop/generic/generic.c
main: <0x10002500-0x10002640>
    31 total cycles(0.00%) invoked 1 times, average 31 cycles/invoke
File '/usr/people/n4733/demos/SpeedShop/generic/generic.c':
Skipping source listing to line 91
92: void      sproctestgrandchild(void *);    /* sproc grandchild code */
93:
94: static    struct timeval  starttime;      /* starting time - first timestamp */
95: static    struct timeval  ttime;          /* last-recorded timestamp */
96: static    struct timeval  deltatime;
97:
98: int pagesize;
99:
100: main(unsigned argc, char **argv)
101: {

```

7: Analyzing Experiment Results

```

[101] 0x10002500      0x27bdffd0      addiu   sp,sp,-48      # 1
[101] 0x10002504      0xffbc0010      sd      gp,16(sp)      # 2
[101] 0x10002508      0xffbf0008      sd      ra,8(sp)       # 3
[101] 0x1000250c      0x3c010002      lui     at,0x2 # 3
[101] 0x10002510      0x2421eb28      addiu   at,at,-5336   # 4
[101] 0x10002514      0x0321e021      addu    gp,t9,at      # 5
[101] 0x10002518      0xaf40024      sw      a0,36(sp)     # 5
[101] 0x1000251c      0xaf5002c      sw      a1,44(sp)     # 6
102:   int     i;
103:
104:   /* initialize the timestamp */
105:   (void) gettimeofday(& starttime, NULL);
[105] 0x10002520      0x27848198      addiu   a0,gp,-32360  # 6
[105] 0x10002524      0x00002825      move    a1,zero       # 6
[105] 0x10002528      0x8f998064      lw      t9,-32664(gp) # 7
      <2 cycle stall for following instruction>
[105] 0x1000252c      0x0320f809      jalr    ra,t9 # 8
[105] 0x10002530      0000000000      nop     # 9
^---   11 total cycles(0.00%) executed   1 times, average 11 cycles.---^
106:
107:   /* set up to reap any children */
108:   (void) sigset(SIGCHLD, (SIG_PF)reapSig);
[108] 0x10002534      0x24040012      li      a0,18 # 1
[108] 0x10002538      0x8f858130      lw      a1,-32464(gp) # 2
[108] 0x1000253c      0x8f998068      lw      t9,-32664(gp) # 3
      <2 cycle stall for following instruction>
[108] 0x10002540      0x0320f809      jalr    ra,t9 # 4
[108] 0x10002544      0000000000      nop     # 5
^---   6 total cycles(0.00%) executed   1 times, average 6 cycles.---^
109:
110:   if(argc == 1) {
[110] 0x10002548      0x8fa20024      lw      v0,36(sp) # 1
[110] 0x1000254c      0x24030001      li      v1,1 # 2
      <2 cycle stall for following instruction>
[110] 0x10002550      0x1443000c      bne     v0,v1,0x10002584 # 3
[110] 0x10002554      0000000000      nop     # 4
      Preceding branch executed 1 times, taken 0 times.
^---   5 total cycles(0.00%) executed   1 times, average 5 cycles.---^
111:   Scriptstring(DEFAULT_SCRIPT);
[111] 0x10002558      0x8f848034      lw      a0,-32716(gp) # 1
      <1 cycle stall for following instruction>

```

```

[111] 0x1000255c    0x24848008    addiu   a0,a0,-32760    # 2
[111] 0x10002560    0x8f998138    lw      t9,-32456(gp)   # 2
[111] 0x10002564    0x0c000a14    jal     0x10002850     # 2
[111] 0x10002568    0000000000    nop     # 3
^---      4 total cycles(0.00%) executed    1 times, average  4 cycles.---^
112:      exit(0);
[112] 0x1000256c    0x00002025    move    a0,zero        # 1
[112] 0x10002570    0x8f99805c    lw      t9,-32676(gp)   # 2
      <2 cycle stall for following instruction>
[112] 0x10002574    0x0320f809    jalr    ra,t9          # 3
[112] 0x10002578    0000000000    nop     # 4
^---      5 total cycles(0.00%) executed    1 times, average  5 cycles.---^
[112] 0x1000257c    0x10000020    b       0x10002600     # 1
[112] 0x10002580    0000000000    nop     # 2
^---      0 total cycles(0.00%) executed    0 times, average  2 cycles.---^
113:    } else {
114:
115:      i = argc;
[115] 0x10002584    0x8fa60024    lw      a2,36(sp)      # 1
[115] 0x10002588    0xaf600000    sw      a2,0(sp)       # 2
.
.
.

```

Using the `-calipers` Option

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

```
% prof -calipers 0,1 executable_name.exp_type.id
```

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

```
% prof -calipers 1,2 executable_name.exp_type.id
```

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

```
% prof -calipers 1,2 executable_name.exp_type.id
```

Using the `-butterfly` Option

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

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

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

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

These exceptions are listed at the end of the report.

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

```
-----  
SpeedShop profile listing generated Thu Feb 12 13:52:09 1998  
  prof -butterfly generic.usertime.m10981  
    generic (n32): Target program  
      usertime: Experiment name  
      ut:cu: Marching orders  
R4400 / R4000: CPU / FPU  
      1: Number of CPUs  
      175: Clock frequency (MHz.)  
  
Experiment notes--  
  From file generic.usertime.m10981:  
  Caliper point 0 at target begin, PID 10981  
    /usr/demos/SpeedShop/linpack.demos/c/generic  
  Caliper point 1 at exit(0)
```



```
-----
Summary of statistical callstack sampling data (usertime)--
```

```
      809: Total Samples
        0: Samples with incomplete traceback
    24.270: Accumulated Time (secs.)
     30.0: Sample interval (msecs.)
-----
```

```
Function list, in descending order by exclusive time
-----
```

[index]	excl.secs	excl.%	cum.%	incl.secs	incl.%	samples	function
(dso: file, line)							
[4]	22.770	93.8%	93.8%	22.770	93.8%	759	anneal
(generic: generic.c, 1573)							
[6]	1.020	4.2%	98.0%	1.020	4.2%	34	slaveusertime
(dlslave.so: dlslave.c, 22)							
[9]	0.210	0.9%	98.9%	0.210	0.9%	7	cvtttrap
(generic: generic.c, 317)							
[12]	0.120	0.5%	99.4%	0.120	0.5%	4	_pm_create_special
(libc.so.1: pm.c, 191)							
[14]	0.090	0.4%	99.8%	0.090	0.4%	3	_migr_policy_args_init
(libc.so.1: pm.c, 398)							
[10]	0.030	0.1%	99.9%	0.180	0.7%	6	iofile
(generic: generic.c, 464)							
[11]	0.030	0.1%	100.0%	0.150	0.6%	5	_doscan_f
(libc.so.1: inline_doscan.c, 615)							
[1]	0.000	0.0%	100.0%	24.270	100.0%	809	__start
(generic: crt1text.s, 101)							
[2]	0.000	0.0%	100.0%	24.270	100.0%	809	main
(generic: generic.c, 101)							
[3]	0.000	0.0%	100.0%	24.270	100.0%	809	Scriptstring
(generic: generic.c, 184)							
[5]	0.000	0.0%	100.0%	22.770	93.8%	759	usertime
(generic: generic.c, 1377)							
[15]	0.000	0.0%	100.0%	0.090	0.4%	3	dirstat
(generic: generic.c, 348)							
[16]	0.000	0.0%	100.0%	0.090	0.4%	3	_pread
(libc.so.1: preadSCI.c, 33)							
[13]	0.000	0.0%	100.0%	0.120	0.5%	4	_fullocale
(libc.so.1: _locale.c, 77)							
[7]	0.000	0.0%	100.0%	1.020	4.2%	34	libdso

7: Analyzing Experiment Results

```
(generic: generic.c, 619)
 [8]      0.000  0.0% 100.0%      1.020  4.2%          34  dlslave_routine
(dlslave.so: dlslave.c, 7)
```

 Butterfly function list, in descending order by inclusive time

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

		93.8%	22.770			22.770	usrtime
(@0x10005c30; generic: generic.c, 1393) [5]							
[4]	93.8%	22.770	93.8%	22.770	22.770	anneal [4]	

```

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

```

7: Analyzing Experiment Results

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

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 a `bbcounts` experiment on an R4000 processor with a clock frequency of 100 megahertz, but you want to generate a report for an R10000 processor with a clock frequency of 196 megahertz, the `prof` command would be the following:

```
% prof -r10000 -clock 196 generic.bbcounts.m4561
```

Generating Reports for Multiprocessed Executables

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

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

The experiment ID, *id*, consists of one or two letters (designating the process type) and the process ID number. See Table 1-1, page 9 for the letter codes and their meanings. This naming convention avoids the potential conflict of multiple threads attempting to write simultaneously to the same file.

Determining Program Overhead

You can determine the overhead of a parallel program by including the `-overhead` argument on the `prof` command line. The overhead information will be included at the end of the usual report.

You can get the overhead report for any experiment, but it may be the most useful for the following experiments:

- `pcsamp`
- `usertime`
- `bbcounts`
- The hardware counter experiments (`*_hwc` and `*_hwctime`)

Use the following steps to generate an overhead report on a system with multiple processors.

1. Run the `bbcounts` experiment on the executable:

```
% ssrun -bbcounts linpackd
```

The `ssrun` command generates the following files, each from a different processor, for an 8-processor program:

```
linpackd.bbcunts.m422744  
linpackd.bbcunts.p421778  
linpackd.bbcunts.p422191  
linpackd.bbcunts.p422252  
linpackd.bbcunts.p422313  
linpackd.bbcunts.p422620  
linpackd.bbcunts.p422704  
linpackd.bbcunts.p422731
```

2. Combine the experiment files into one experiment file using the `ssaggregate(1)` command.

```
% ssaggregate -e linpackd.bbcunts* -o bbcunts.8ps
```

3. Execute the `prof` command with the `-overhead` option to create the data file.

```
% prof -overhead bbcunts.8ps >result
```

The result file contains the usual `bbcunts` experiment output with the overhead information at the end. Note that some lines are wrapped here to accommodate page width:

SpeedShop profile listing generated Fri Jun 25 09:21:27 1999

```
prof -overhead bbcunts.8ps  
    linpackd (n32): Target program  
          bbcunts: Experiment name  
          it:cu: Marching orders  
R10000 / R10010: CPU / FPU  
          16: Number of CPUs  
          195: Clock frequency (MHz.)
```

Experiment notes--

```
    From file bbcunts.8ps:  
    Caliper point 0 at target begin, PID 422744  
        linpackd  
    Caliper point 0 at target begin, PID 422620  
        linpackd  
    Caliper point 0 at target begin, PID 422731  
        linpackd  
    Caliper point 0 at target begin, PID 422704  
        linpackd  
    Caliper point 0 at target begin, PID 422252  
        linpackd
```

```

Caliper point 0 at target begin, PID 421778
    linpackd
Caliper point 0 at target begin, PID 422191
    linpackd
Caliper point 0 at target begin, PID 422313
    linpackd
Caliper point 1 at exit(0)

```

```

-----
Summary of bccounts time data (bccounts)--
    29877509668: Total number of instructions executed
    20592366537: Total computed cycles
    105.602: Total computed execution time (secs.)
    0.689: Average cycles / instruction

```

```

-----
Function list, in descending order by exclusive bccounts time

```

```

-----
[index]  excl.secs  excl.%   cum.%      cycles  instructions  calls
function (dso: file, line)
-----
    [1]      72.955   69.1%   69.1%   14226219437  24895879414   140
__mp_slave_wait_for_work (libmp.so: mp_parallel_do.s, 593)
    [2]      30.344   28.7%   97.8%    5917081268  4669997342   772633
daxpy (linpackd: linpack.f, 495)

```

```

.
.
.

```

```

-----
OpenMP Report

```

```

-----
Parallelization Overhead: 00.000%
Load Imbalance: 00.076%
Insufficient Parallelism: 69.085%
Barrier Loss: 00.002%
Synchronization Loss: 00.000%
Other Model-specific Overhead: 00.000%

```

The parallel model used in the program was OpenMP, although other parallel models (such as MPI and pthreads) are supported. The categories for which information is returned vary depending on the model. The OpenMP categories have the following meanings:

- **Parallelization Overhead:** the percentage of the program's time spent doing work necessary to a parallel program, such as distributing loop iterations and data among the processors. The percentage is negligible for this program.
- **Load Imbalance:** the percentage of a program's time lost because work was not spread evenly across the processors. This number would be 0 if each processor had exactly the same amount of work.
- **Insufficient Parallelism:** the percentage of a program's time in which the processors were not working in parallel. The number returned for this program tells us that about two-thirds of the program time was not parallelized.
- **Barrier Loss:** the percentage of the program's time used by the barrier process. This is not the time processors spent waiting at barriers.
- **Synchronization Loss:** the percentage of the program's time used by the other synchronization processes.
- **Other Model-specific Overhead:** the percentage of a program's time spent in other OpenMP (in this case) processes.

The same aggregated experiment file created above can be used by the `cvperf(1)` command to display overhead information in its own window. For an example, see the *ProDev WorkShop: Performance Analyzer User's Guide*.

Generating Compiler Feedback Files

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

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

For more information, see your compiler man page or the `cord(1)` man page.

Comparing Experiment Results

After running experiments, you can compare experiment results by using the `scompare` command. This command can be used to analyze performance data

generated by `ssrun` and produce a comparison report. This can be particularly useful when comparing the effects of different optimization techniques, for example, or when comparing different experiments for the same application.

The `sscompare` command has the following syntax:

```
sscompare [-by type] [-individual] [-left caliper] [-output format]
          [-path pathname] [-percentages] [-precision digits] [rough]
          [-right caliper] [statistics] [-ut_exclusive] [-ut_inclusive]
          [-width characters]
```

The comparison report contains a legend and a table of performance data. Each input file and the type of performance data it contains is listed in the legend with a numeric column key. The table contains multiple columns of data. There is one column for each experiment file (if the `individual` option is used) or one for each statistic (if the `statistics` option was used).

The following example demonstrates this command's use. In this example, a SpeedShop PC sampling experiment is run on the OpenMP implementation of the NAS Conjugate Gradient Parallel Benchmark using four threads. The resulting experiment files are then compared side-by-side using `sscompare`:

```
% setenv OMP_NUM_THREADS 4
% ssrun -fpcsampx cg.A

NAS Parallel Benchmarks 2.3 OpenMP C version - CG Benchmark
Size:      14000
Iterations: 15

    ...
    ...
    ...

% sscompare -by function -individual -percentages cg.A.fpcsampx.*

1: Exclusive 'PC sampling' time for cg.A.fpcsampx.T0.m2675229
2: Exclusive 'PC sampling' time for cg.A.fpcsampx.T1.p2676575
3: Exclusive 'PC sampling' time for cg.A.fpcsampx.T2.p2678981
4: Exclusive 'PC sampling' time for cg.A.fpcsampx.T3.p2681324

    1          2          3          4
```

7: Analyzing Experiment Results

```
63.973% 74.819% 67.227% 63.820% conj_grad (cg.A: cg.c, 356)
 0.000% 22.881% 23.754% 24.255% __mp_slave_wait_for_work (libmp.so: mp_parallel_do.s, 593)
10.758%  1.608%  7.913% 10.943% __mp_barrier_nthreads (libmp.so: mp_barrier.c, 271)
16.161%  0.000%  0.000%  0.000% sparse (cg.A: cg.c, 709)
```

```
...
...
...
...
```

sscompare supports the following SpeedShop experiment types:

- usertime
- totaltime
- pcsamp
- bbcunts

Miscellaneous Commands

This chapter describes additional SpeedShop commands that are useful in analyzing application performance. It contains the following sections:

- "Using the `thrash` Command", page 135
- "Using the `squeeze` Command", page 136
- "Calculating the Working Set of a Program", page 137
- "Combining Multiple Experiment Files into One", page 139

Using the `thrash` Command

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

`thrash` Syntax

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

```
thrash args [-n count] [-s] [-w time]
```

- *args* can be one of the following options:
 - `-k n`: the amount of memory to access in kilobytes, where *n* is the number of kilobytes. The minimum value for *n* is the size of one page, or the value will be changed appropriately.
 - `-m n`: the amount of memory to access in megabytes, where *n* is the number of megabytes.
 - `-p n`: the amount of memory to access in pages, where *n* is the number of pages.
- `-n [count]`: the number of references to make before exiting. The default is 10,000.
- `-s`: sequential thrashing. The default is random.

- `-w time`: an integer amount of time, in seconds, `thrash` should sleep after thrashing but before exiting. The default is 0 seconds.

Effects of thrash

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

```
% thrash -m 4
```

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

```
10000 iterations
```

You can use `thrash` in conjunction with `ssusage(1)` and `squeeze(1)` to determine the approximate available working memory on a system, as described in "Calculating the Working Set of a Program", page 137.

Using the squeeze Command

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

squeeze Syntax

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

```
squeeze [unit] amount
```

The following arguments are used:

- `unit`: can be one of the following options indicating the unit of measure. If no option is specified, the default is megabytes.
 - `-k`: kilobytes
 - `-m`: megabytes

- -p: pages
- -%: a percentage of the installed memory
- *amount*: the amount of memory to be locked.

Effects of squeeze

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

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

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

Here is an example that locks down 4 megabytes of memory:

```
% 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. For all practical purposes, the working set of your program is the size of memory allocated. The following procedure assumes that you are running on a system that is either stand-alone or where the environment will not change while you are running these tests.

Procedure 8-1 Calculating the Working Set

1. Determine the working set of the kernel and other applications:

- Choose a machine that has a large amount of physical memory (enough to allow your target application to run without any paging other than at startup).
- Make sure that the machine is running a minimal number of applications that will remain fairly consistent for the duration of these steps.
- Run `thrash` with `ssusage` to determine the working set of the kernel and any other applications you have running.

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

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

When the `thrash` command completes, `ssusage` prints the resource usage of `thrash`. The value labeled `majf` gives the number of major page faults (that is, the number of faults that required a physical I/O). When you run on a machine with a large amount of physical memory, this value is the number of faults needed to start the program, which is the minimum number for any run. For more information on `ssusage`, see Chapter 5, "Collecting Data on Machine Resource Usage", page 69.

- As super user in a separate window, run the `squeeze` command to lock down an amount of memory.
- Rerun `thrash` with `ssusage`, as shown here:

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

- Repeat the previous two steps, increasing the amount of memory for `squeeze`, until the `majf` number begins to rise.

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

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

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

2. Determine the working set of the kernel and other applications:

- The applications that the machine is running should remain consistent with the machine in the first step.
- 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.

- In another window, become super user.
- In this new window, run `squeeze` to lock down an amount of memory. The following example locks down 15 megabytes of memory:

```
% squeeze 15
```

- In the first window, rerun your program with `ssusage`.
- In the second window running `squeeze`, enter `ctrl-c` to cause `squeeze` to exit.
- Repeat these steps, using `squeeze` to lock down increasing amounts of memory until the `majf` number begins to rise.
- Deduct the amount squeezed at the point at which the application begins to page fault from the total amount of physical memory in the system. This computation determines the combined working set of your program, the kernel, and any other applications you have running.

3. Calculate the working set size of your program.

Deduct the amount of working memory calculated in step 1g from the combined working set size calculated in step 2h. This computation determines the approximate working set of your program.

Combining Multiple Experiment Files into One

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

8: Miscellaneous Commands

The `ssaggregate` command takes the following form:

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

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

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

The output from `prof` is as follows:

```
-----
SpeedShop profile listing generated Tue Nov 24 11:30:03 1998
  prof combo
    generic (n32): Target program
      pcsamp: Experiment name
pc,2,10000,0:cu: Marching orders
  R5000 / R5000: CPU / FPU
    1: Number of CPUs
    180: Clock frequency (MHz.)

Experiment notes--
  From file combo:
  Caliper point 0 at target begin, PID 14635
    /home/saffron02/speedshop/c/generic ll.u.cvt.d.i.f.dso ll.u.cvt.d.i.f.dso ll.u.cvt.d.i.f.dso
  Caliper point 0 at target begin, PID 14636
    /home/saffron02/speedshop/c/generic ll.u.cvt.d.i.f.dso ll.u.cvt.d.i.f.dso ll.u.cvt.d.i.f.dso
  Caliper point 1 at exit(0)

-----
Summary of statistical PC sampling data (pcsamp)--
    4012: Total samples
    40.120: Accumulated time (secs.)
    10.0: Time per sample (msecs.)
    2: Sample bin width (bytes)

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

  [1]      37.480  93.4%  93.4%       3748  anneal (generic: generic.c, 1573)
  [2]       1.450   3.6%  97.0%        145  slaveusertime (dlslave.so: dlslave.c, 22)
  [3]       0.490   1.2%  98.3%         49  _read (libc.so.1: read.s, 15)
  [4]       0.330   0.8%  99.1%         33  _xstat (libc.so.1: xstat.s, 12)
```



```
[5] 0.300 0.7% 99.8% 30 cvttrap (generic: generic.c, 317)
[6] 0.030 0.1% 99.9% 3  _write (libc.so.1: write.s, 15)
[7] 0.010 0.0% 99.9% 1  fread (libc.so.1: fread.c, 27)
[8] 0.010 0.0% 100.0% 1  _syscall (libc.so.1: syscall.s, 15)
    0.020 0.0% 100.0% 2  **OTHER** (includes excluded DSOs, rld, etc.)

40.120 100.0% 100.0% 4012 TOTAL
```

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

Glossary

basic block

A set of instructions with a single entry point, a single exit point, and no branches into or out of the set.

bead

A record in an experiment.

caliper points

A caliper point is a point at which you wish to mark your program so that later you may display performance taken between the marks (caliper points) you have set. A caliper point may be set at a particular location in the source, after a particular time interval, or when a particular signal is received by your program. An implicit caliper point is always present at the start of execution of the process. A final caliper point is set when the process calls `_exit`. Caliper points are numbered so you can select them with displaying performance data.

call stack

A software stack of functions and routines that represent the state of the program at any time. The functions and routines are listed in the reverse order, from top to bottom, in which they were called. If function *a* is immediately below function *b* in the stack, then *a* was called by *b*. The function at the bottom of the stack is the one currently executing.

context switch

The act of saving the state of one process and replacing it with that of another when both processes time-share a single processor.

counts

The number of times an event takes place during data gathering. For example, a count may be kept of the number of times a function executes.

CPU time

Process virtual time plus time spent when the system is running on behalf of the process, performing such tasks as executing a system call. This is the time returned in `pcsamp` and `usertime` experiments. It can be specified in an experiment by using the `ut,30000,2` marching orders.

dynamic shared object (DSO)

An object file that is similar in structure to an executable program, but it has no main program.

exclusive time

The execution time of a given function but not of any functions called by that function. See inclusive time.

graduated instruction

As a performance enhancement, when an R10000 system comes to a point in the execution of a program at which either of two paths might be taken, it begins to execute both paths until it knows for sure which path is correct. Graduated instructions are those on the path it will eventually follow. Issued instructions are those on the path it does not follow.

inclusive time

The execution time both of a given function and of any functions called by that function. See exclusive time.

overflow interval

As used by the hardware counter experiments, it is the number at which a hardware counter exceeds a preset value. See the `speedshop` man page, `dsc_hwc` experiment.

PC

Program counter. A register that contains the address of the instruction that is currently executing.

process virtual time

Time spent when a program is actually running. This does not include either 1) the time spent when the program is swapped out and waiting for a CPU or 2) the time

when the operating system is in control, such as executing a system call for the program. The marching orders `ut , 30000 , 1` return process virtual time.

rld

The runtime linker. This is invoked when a dynamic executable is run. It maps in shared objects used by the executable, resolves relocations as `ld` does at static link time, and allocates common, if required.

statistical data

Sampling. The results from this method of data gathering vary from run to run.

system time

The time the operating system spends performing services for a program, such as executing system calls and I/O.

TLB

Translation lookaside buffer. This is hardware used by the CPU to quickly translate a virtual address (such as the name of a variable) to a physical memory address.

TDT model

Target Description Table model. A CPU model used to calculate ideal time.

user time

The same as CPU time.

wall-clock time

Total time a program takes to execute, including the time it takes waiting for a CPU. This is real time, not computer time. The marching orders `ut , 30000 , 0` return wall-clock time.

Index

A

API
 setting calipers, 11

B

basic block counting
 inclusive, 63
 overview bbcounts experiment
 overview, 5
bbcounts CPU time, 63
bbcounts experiment
 effects, 96
 tutorial experiments
 bbcounts basic block counting, 25, 44
block counting, 62
bugs , 2
butterfly option, 124

C

cache thrashing, 2
calipers
 automatic, 93
 pollpoint
 time oriented, 93
 sample traps
 using the debugger, 96
 sample traps calipers, 93
 setting calipers, 92
 time-oriented, 94
calipers option, 123
calipers option to prof, 11
-calipers , 11

commands in SpeedShop, 3
compiler feedback files, 132
compiler optimization restrictions, 72
cord, 132
CPU time calculations, 63
CPU-bound processes, 1
cy_hwc experiment, 55
cy_hwctime experiment, 58

D

data display anomalies, 72
dc_hwc experiment , 56
dc_hwctime experiment, 58
debugger
 setting calipers, 11, 93, 96
 using ssrun, 85
demo program SpeedShop
 C and C++, 13
dsc_hwc experiment, 56
dsc_hwctime experiment, 58
DSOs shared libraries, 6

E

environment variables, 74
 _RLD_LIST, 96
 _SPEEDSHOP_CALIPER_POINT_SIG, 74, 93, 95
 _SPEEDSHOP_DEBUG_NO_SIG_TRAPS , 78
 _SPEEDSHOP_DEBUG_NO_STACK_UNWIND, 78
 _SPEEDSHOP_EXPERIMENT_TYPE, 77, 96
 _SPEEDSHOP_FILE_BUFFER_LENGTH, 78
 _SPEEDSHOP_HWC_COUNTER_NUMBER
 , 57, 74

- `_SPEEDSHOP_HWC_COUNTER_OVERFLOW`, 57, 75
 - `_SPEEDSHOP_HWC_COUNTER_PROF_NUMBER` num , 74
 - `_SPEEDSHOP_INIT_DEFERRED_SIG`, 77
 - `_SPEEDSHOP_INSTR_ARGS`, 75
 - `_SPEEDSHOP_MARCHING_ORDERS`, 77, 96
 - `_SPEEDSHOP_OUTPUT_DIRECTORY`, 75
 - `_SPEEDSHOP_OUTPUT_FILENAME`, 75
 - `_SPEEDSHOP_OUTPUT_NOCOMPRESS`, 75
 - `_SPEEDSHOP_POLLPOINT_CALIPER_POINT`, 75, 94
 - `_SPEEDSHOP_REUSE_FILE_DESCRIPTOR`s, 75
 - `_SPEEDSHOP_RLD` full_path_name, 75
 - `_SPEEDSHOP_SAMPLING_MODE`, 77
 - `_SPEEDSHOP_SBRK_BUFFER_ADDR`, 75
 - `_SPEEDSHOP_SBRK_BUFFER_LENGTH`, 75, 78
 - `_SPEEDSHOP_SHUTDOWN_SIG`, 77
 - `_SPEEDSHOP_SILENT`, 76
 - `_SPEEDSHOP_TRACE_EXEC` [True | False], 76
 - `_SPEEDSHOP_TRACE_FORK`, 76
 - `_SPEEDSHOP_TRACE_FORK_TO_EXEC`, 76
 - `_SPEEDSHOP_TRACE_MPI_RANKS`, 77
 - `_SPEEDSHOP_TRACE_SPROC`, 76
 - `_SPEEDSHOP_TRACE_SYSTEM`, 77
 - `_SPEEDSHOP_VERBOSE`, 76
 - `_SSMALLOC_NO_BUFFERING`, 76
 - examples
 - c tutorial, 13
 - fortran tutorial, 33
 - exec system call, 6
 - executable requirements
 - calipers, 72
 - executables
 - calculating a working set, 137
 - execution times, 63
 - experiment data
 - controlling output file, 73
 - file name examples, 73
 - experiment data files
 - combining, 139
 - performance data, 9
 - experiment types, 51
 - experiments
 - cy_hwc, 55
 - dc_hwc, 56
 - dsc_hwc, 56
 - fpe, trace floating-point exceptions, 53
 - gfp_hwc, 56
 - hardware counter, 54, 108
 - heap trace, 53
 - I/O trace, 65
 - isc_hwc, 56
 - pcsamp, 67
 - pcsamp and bbcounts, 64
 - prof_hwc, 57
 - selecting, 51
 - ssrun setup, 71
 - tlb_hwc, 56
 - totaltime, 4
 - types of, 51
 - usertime, 68
- F**
- file preparation, 62
 - floating-point exception trace
 - experiment description, 53
 - overview, 5
 - fork processes, 6
 - Fortran
 - files for tutorial, 33
 - limitations, multiprocessor executables, 72
 - Fortran tutorial, 33
 - fpcsampx, 67
 - fpe trace experiment , 53
 - tutorial experiments
 - fpe trace floating-point exceptions, 29
 - fsc_hwc experiment, 56
 - fsc_hwctime experiment, 58

G

gfp_hwc experiment, 56
gfp_hwctime experiment, 58
gi_hwc experiment, 55
gi_hwctime experiment, 58

H

hardware counter experiment reports, 108
hardware counter experiments
 cy_hwc, 55
 cy_hwctime, 58
 dc_hwc, 56
 dc_hwctime, 58
 dsc_hwc, 56
 dsc_hwctime, 58
 fsc_hwctime, 58
 gfp_hwc, 56
 gfp_hwctime, 58
 gi_hwc, 55
 gi_hwctime, 58
 _hwc, 55
 _hwctime, 57
 ic_hwc, 56
 ic_hwctime, 58
 introduction, 54
 isc_hwc, 56
 isc_hwctime, 58
 prof_hwc, 57
 prof_hwctime, 59
 tlb_hwc, 56
 tlb_hwctime, 58
 tutorial experiments, 23, 41
hardware counter numbers, 59
hardware counter tools, 54
heap trace, 53
_hwc hardware counter experiments, 55
_hwctime hardware counter experiments, 57

I

I/O trace experiment, 65
I/O-bound processes, 1
ic_hwc experiment, 56
ic_hwctime experiment, 58
inclusive basic block counting, 63
introduction to performance analysis, 1
isc_hwc experiment, 56
isc_hwctime experiment, 58

L

libfpe_ss.so
 overview, 5
libmalloc.so
 overview, 6
libpixrt.so
 overview, 6
libraries
 libss.so, 96
 libssrt.so, 96
 linking in SpeedShop, 94
 overview, 5
libss.so, 5
libssrt.so
 overview, 5

M

machine resource usage, 69
marching orders, 78
 experiment specifier, 79
memory
 locking, 136
memory-bound processes, 2
MP Fortran limitations, 72
MPI
 with ssrun, 86

MPI message-passing paradigms, 6
multiprocessor executables, 6
 profiling, 129

O

OpenMP
 and `ssrun`, 92
 OpenMP support, 7

P

pc sampling
 pcsamp experiment
 overview, 4
pcsamp experiment, 39
 example, 83
 PC sampling program, 67
 tutorial experiments
 PC sampling tutorial, 20
perfex, 54
performance analysis, 1
 introduction, 1
 phases, 7
performance problems
 bugs, 2
 cache thrashing, 2
 CPU-bound processes, 1
 I/O-bound processes, 1
 memory-bound processes, 2
 program phases, 2
prof
 -butterfly example profiling
 inclusive basic block counts, 112
 options, 100
 output, 105
 overview, 3
 -S example, 120
 syntax, 100
 using with `ssrun`, 99

prof command
 butterfly option, 124
 calipers option, 123
prof_hwc experiment, 57
prof_hwctime experiment, 59
profiling
 bbcounts experiment experiments
 bbcounts bbcounts experiment reports, 110
 -clock option, 101
 -dis option, 102
 -dis option prof
 -dis example, 114
 -dsolist option, 102
 -exclude option, 102
 -feedback option, 102
 fpe trace experiment experiments
 fpe fpe trace experiment reports, 113
 -fpe_type option, 102
 hardware counter experiments, 108
 -heavy option, 102
 -lines option, 103
 machine scheduler option reports
 for different machine models, 128
 -only option, 103
 pcsamp experiment experiments
 pcsamp pcsamp experiment reports, 107
 processor scheduler option option, 104
 -quit option, 103
 -S option, 104, 120
 usertime experiment experiments
 usertime usertime experiment reports, 106
program overhead, 129
program phases, 2
pthreads, 6
 and `ssrun`, 91

R

rearranging procedures, 132
reordering code regions, 62

_RLD_LIST variable , 96
 run-time environment variables, 74

S

setup ssrun, 71
 signals
 setting calipers, 11, 95
 SpeedShop
 overview, 3
 speedshop api , 6
 SpeedShop demo
 Fortran, 34
 SpeedShop hardware counter experiments
 introduction, 54
 SpeedShop libraries, 96
 libss.so libraries, 5
 linking libss.so, 94
 _SPEEDSHOP_CALIPER_POINT_SIG variable
 , 74, 93, 95
 _SPEEDSHOP_DEBUG_NO_SIG_TRAPS variable
 , 78
 _SPEEDSHOP_DEBUG_NO_STACK_UNWIND
 variable , 78
 _SPEEDSHOP_EXPERIMENT_TYPE variable
 , 77, 96
 _SPEEDSHOP_FILE_BUFFER_LENGTH variab, 78
 _SPEEDSHOP_HWC_COUNTER_NUMBER, 57
 _SPEEDSHOP_HWC_COUNTER_NUMBER
 variable , 74
 _SPEEDSHOP_HWC_COUNTER_OVERFLOW, 57
 _SPEEDSHOP_HWC_COUNTER_OVERFLOW
 variable , 75
 _SPEEDSHOP_HWC_COUNTER_PROF_NUMBER
 num variable, 74
 _SPEEDSHOP_INIT_DEFERRED_SIGNAL
 variable, 77
 _SPEEDSHOP_INSTR_ARGS variable, 75
 _SPEEDSHOP_MARCHING_ORDERS variable
 , 77, 96
 _SPEEDSHOP_OUTPUT_DIRECTORY variable , 75

_SPEEDSHOP_OUTPUT_FILENAME variable , 75
 _SPEEDSHOP_OUTPUT_NOCOMPRESS
 variable , 75
 _SPEEDSHOP_POLLPOINT_CALIPER_POINT
 environment variable, 93, 94
 _SPEEDSHOP_POLLPOINT_CALIPER_POINT
 variable
 , 75
 _SPEEDSHOP_REUSE_FILE_DESCRIPTOR
 variable , 75
 _SPEEDSHOP_RLD variable, 75
 _SPEEDSHOP_SAMPLING_MODE variable, 77
 _SPEEDSHOP_SBRK_BUFFER_ADDR variable, 75
 _SPEEDSHOP_SBRK_BUFFER_LENGTH
 variable, 75, 78
 _SPEEDSHOP_SHUTDOWN_SIG variable, 77
 _SPEEDSHOP_SILENT variable, 76
 _SPEEDSHOP_TARGET_FILE variable, 96
 _SPEEDSHOP_TRACE_EXEC variable, 76
 _SPEEDSHOP_TRACE_FORK variable, 76
 _SPEEDSHOP_TRACE_FORK_TO_EXEC
 variable, 76
 _SPEEDSHOP_TRACE_MPI_RANKS variable, 77
 _SPEEDSHOP_TRACE_SPROC variable, 76
 _SPEEDSHOP_TRACE_SYSTEM variable, 77
 _SPEEDSHOP_VERBOSE variable
 , 76
 sproc system call, 6
 squeeze
 calculating a working set, 137
 locking memory , 136
 overview, 3
 scompare, 10
 overview, 3
 _SSMALLOC_NO_BUFFERING variable, 76
 ssrt_caliper_point, 6, 72
 ssrt_caliper_point calipers, 93, 94
 ssrun
 and OpenMP, 92
 effects, 96
 flags, 82

- MPI programs , 86
- overview, 3
- overview ssrun
 - steps prof, 8
- pthread programs, 91
- syntax, 82
- using a debugger, 85
- v option example, 85
- ssrun command
 - examples, 83
 - syntax, 81
- ssrun setup, 71
- ssusage
 - calculating a working set, 137
 - machine resource usage, 69
 - overview, 3
- statistical call stack profiling
 - overview usertime experiment
 - overview, 4
- statistical hardware counter sampling
 - overview hardware counter experiments
 - overview hwc experiments, 4
- stripped executables programs, 72
- system call, 6

T

- thrash

- calculating a working set, 137
 - overview, 4
- thrash paging behavior, 135
- tlb_hwc experiment, 56
- tlb_hwctime experiment, 58
- totaltime experiment, 4
 - totaltime call stack profiling, 68
- Tutorial
 - c, 13
- tutorial experiments
 - PC sampling, 39

U

- user environment variables, 76
- usertime experiment
 - restrictions, 72
- tutorial experiments
 - call stack profiling, 16
 - usertime call stack profiling, 36
- usertime call stack profiling, 68

W

- working set , 137