Oracle Solaris Studio 12.2: Simple Performance Optimization Tool (SPOT) User's Guide

Part No: 821–2842
November 2010

110606@25097

Contents

Preface

The Simple Performance Optimization Tool (SPOT) can help you diagnose performance problems that can limit the speed of an application. Running your application with SPOT is complementary to running it under the Oracle Solaris Studio Performance Analyzer and looking at the resulting experiment.

Who Should Use This Book

This manual is intended for application developers with a working knowledge of Fortran, C, C++, or Java programming languages. Users of the performance tools need some understanding of the Solaris operating system, or the Linux operating system, and UNIX? operating system commands. Some knowledge of performance analysis is helpful but is not required to use the tools.

Accessing Oracle Solaris Studio Documentation

You can access the documentation at the following locations:

- The documentation is available from the documentation index page at `http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/index.html`.
- Online help for all components of the IDE is available through the Help menu, as well as through the F1 key, and through Help buttons on many windows and dialog boxes, in the IDE.
- Online help for the Performance Analyzer and the Thread Analyzer is available through the Help menu, as well as through the F1 key, and through Help buttons on many windows and dialog boxes, in the Performance Analyzer.
- Online help for DLight and dbxtool is available through the Help menu, as through the F1 Key, and through Help button on many windows and dialog boxes, in these tools.

Documentation in Accessible Formats

The documentation is provided in accessible formats that are readable by assistive technologies for users with disabilities. You can find accessible versions of documentation as described in the following table.

Type of Documentation	Format and Location of Accessible Version
Manuals and Tutorials	HTML from the Oracle Solaris Studio 12.2 collection on `http://docs.sun.com`
What's New in the Oracle Solaris Studio 12.2 Release (information that was included in the component Readmes in previous releases)	HTML from the Oracle Solaris Studio 12.2 collection on `http://docs.sun.com`
Man pages	In the installed product through the `man` command
Online help	HTML available through the Help menu Help buttons, and F1 key in the IDE, Performance Analyzer, DLight, and dbxtool.
Release notes	HTML from the Oracle Solaris Studio 12.2 collection on `http://docs.sun.com`

Related Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

Note – Oracle is not responsible for the availability of third-party web sites mentioned in this document. Oracle does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Oracle will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Documentation, Support, and Training

See the following web sites for additional resources:

- Documentation (`http://docs.sun.com`)
- Support (`http://www.oracle.com/us/support/systems/index.html`)
- Training (`http://education.oracle.com`) – Click the Sun link in the left navigation bar.

Oracle Welcomes Your Comments

Oracle welcomes your comments and suggestions on the quality and usefulness of its documentation. If you find any errors or have any other suggestions for improvement, go to `http://docs.sun.com` and click Feedback. Indicate the title and part number of the documentation along with the chapter, section, and page number, if available. Please let us know if you want a reply.

Oracle Technology Network (`http://www.oracle.com/technetwork/index.html`) offers a range of resources related to Oracle software:

- Discuss technical problems and solutions on the Discussion Forums (`http://forums.oracle.com`).
- Get hands-on step-by-step tutorials with Oracle By Example (`http://www.oracle.com/technology/obe/start/index.html`).
- Download Sample Code (`http://www.oracle.com/technology/sample_code/index.html`).

Typographic Conventions

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

TABLE P–1 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

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

TABLE P–2 Shell Prompts

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

CHAPTER 1

The Simple Performance Optimization Tool (SPOT)

The Simple Performance Optimization Tool (SPOT) can help you diagnose performance problems that can limit the speed of an application.

This chapter includes information about the following:

Introduction

The role of SPOT is complementary to running the application under the Oracle Solaris Studio Performance Analyzer, and looking at the resulting experiment. The profile generated by the Analyzer tells you where the time was spent in running your application. In certain situations, however, you may not be able to diagnose your application's problems just by examining its profile.

Some problems that cannot easily be solved by inspecting the application profile are:

- Is the time spent in the routine high because the routine itself is slow, or because the routine is called a large number of times?
- Is a line of code taking time because it misses cache or because it misses the translation lookaside buffer (TLB)?
- Are traps slowing down the application?
- Is the application reaching a memory bandwidth limit?

While you might be able to identify the causes of these issues by looking at the application's profile and running additional tools, you might not know what tools are available or which specific tool to use.

SPOT simplifies the process of performance analysis by running an application under a common set of tools and producing HTML reports of its findings, which provides the following benefits:

- By creating reports in HTML format, SPOT lets you place the reports on a server that can be accessed by an entire development team. For example, a SPOT report can be examined by remote colleagues, or referred to during a meeting. You could even email a URL of a particular line of source code, or disassembly, to a colleague for further review.
- The SPOT report archives the compiler build commands as well as the profile for the active parts of the application. By comparing the current application profile with an older profile, you can easily check for changed code or changed compiler build flags.
- SPOT can also profile the application according to the most frequently occurring hardware events, indicating which routines are encountering which problems.

Requirements for Using SPOT

Supported Platforms

SPOT runs on SPARC and x86 platforms. The specific details included in the SPOT reports are platform dependent. Some of the tools that SPOT uses are not available on x86 platforms, so instruction count data, bandwidth data, and trap data are not included when you run SPOT on these platforms.

Binaries Must Be Prepared Correctly

SPOT works on binaries compiled with the Sun Studio 12, Sun Studio 12 Update 1, or Oracle Solaris Studio 12.2 compilers, or the GCC for Sun Systems compilers starting with version 4.2.0, on a SPARC_based or x86–based system running the Solaris 10 5/08 operating system or a later Solaris 10 update.

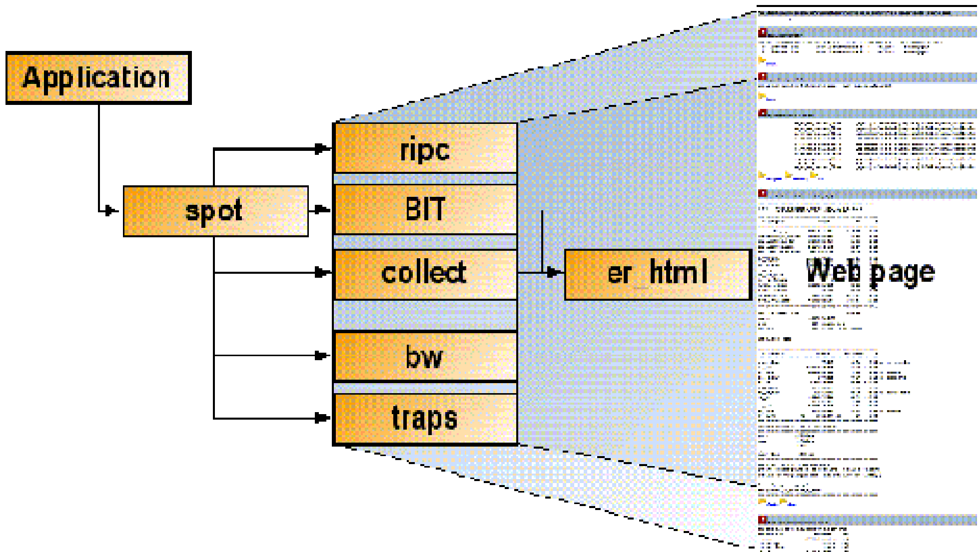
When using a Sun Studio or Oracle Solaris Studio compiler. you must compile with optimizations by using the `-O` option or the `-xO[`*n*`]` option. When using a GCC compiler, no particular optimization level is required.

Using the `-g` option to generate debug information when compiling the binary allows SPOT to attribute time and processor events back to the source code lines that caused them. For C++ programs, the `-g` option alone (with no optimization level specified) turns off inlining of functions, which can have a significant performance impact. So it is better to use the `-g0` (zero) option, which turns on debugging information and does not affect inlining of functions.

A binary compiled as described includes information called annotations that let the tools instrument the application to generate the counts of the number of calls to routines and the number of times each individual instruction is executed.

The Architecture of SPOT

SPOT uses several tools to collect data and generate its report.



- The `rpic` tool collects performance counter information over the run of a program and produces a text summary of the stall time that each processor event contributed to the run time of the program. For example, it reports the number of seconds spent waiting for data located in memory.
- The Binary Improvement Tool (`BIT`) instruments an application (provided it is compiled as described in "Binaries Must Be Prepared Correctly" on page 10) and generates information on the number of times each routine is called, the number of times each individual instruction is executed, and the instruction frequency for each assembly language instruction.
- The `collect` tool is used by SPOT to profile the application over time and, when you request extended information, profile where the processor events occur.
- The `bw` tool collects system-wide bandwidth utilization data (if possible for the target platform) when you request extended data and are running SPOT with root privileges.
- The `traps` tool is a wrapper for `trapstat`, which is included in the Oracle Solaris operating system. It collects trap data when you request extended data and are running SPOT with root privileges.

- The `er_html` tool is a wrapper for the `er_print` tool, which generates a set of hyperlinked HTML files from a Performance Analyzer experiment (the data collected by the `collect` tool and the `BIT` tool).
- The `spot_diff` tool produces a report that compares multiple SPOT reports.

You can run each of these tools as a stand-alone tool. Only `er_html` and `spot_diff` produce HTML output when run them outside of SPOT.

CHAPTER 2

Running SPOT on Your Application

Running SPOT

You can run SPOT on your application in two ways:

- By starting your application with the `spot` command:

  `spot` *application_command arguments*

  SPOT will run the application multiple times, collect data with each tool over the entire run of the application, and generate a report.
- By using the `spot` command to attach SPOT to an existing process that is executing the application:

  `spot -P` *process_id*

  SPOT will attach to the running process, use each tool for a short period of time to collect data, and generate a report on the process.

Running SPOT with the `-X` option produces the most information. However, when you use this option, SPOT takes longer to collect the data. When you run SPOT with root privileges and this option, SPOT collects bandwidth utilization and trap data. On some processors, SPOT also collects hardware counter profiles.

Command Line Options

You can use the following options with the `spot` command.

Data Collection Option

`-X`	Request extended statistics. If you run SPOT with root privileges, SPOT will collect system wide bandwidth consumption data and system wide trap statistics. Use a dedicated system when collecting this data. On some processors, SPOT will also collect hardware counter profiles of the application using the performance counters identified by `ripc` as large contributors to the overall stall time. To do, it will run the application up to four additional times.

Attach to Running Process Options

`-P` *process_id*	Attach SPOT to the running process *process_id*.
`-T` *seconds*	Set *seconds* as the duration of attachment of each of the five probes to the process. The default is 60 seconds per probe, for a total of 300 seconds.

Output Options

You can specify the following options to determine the location and content of the SPOT report. The `-d` and `-o` options work together to determine the location and the name of the subdirectory that contains the report.

`-d` *directory*	Write the SPOT report to a subdirectory of *directory*. By default, SPOT writes the report to a subdirectory of the current directory.
`-o` *subdirectory*	Write the SPOT report to *subdirectory*. By default, the subdirectory is named `spot_run`*n*, where *n* is a unique number.

`-D` *n*	Set the verbosity level of debug information to be reported. The value of n can be: 0: No debug output 1: Normal level of debug information (default) 2: Full debug information The debug information is written to the `debug.log` file in the SPOT report.
`-q`	Do not write SPOT output to `stdout` (same as `-D 0`).
`-v`	Write the current version and detailed debugging information to `stdout` (same as `-D 2`).

Other Options

`-c` *path*	Specify a path for the Oracle Solaris Studio components used by SPOT. This option is useful if you want to override the default compiler and use a compiler installed in a different location.
`-V`	Print SPOT version information and exit.
`-h`	Print help information.

CHAPTER 3

Understanding SPOT Reports

Reports Produced by SPOT

When you run SPOT, it produces the following directories and files, which it writes to the current directory unless you specified a different directory with the `-d` option:

`spot_run`*n*/	A subdirectory for each run that contains the SPOT report for the run. *n* is a unique number for each run. You can specify a different subdirectory name with the `-o` option.
`spot_summary.html`	A report that lists SPOT runs with the date and time of each run.
`spot_diff.html`	A report that compares SPOT data from different runs in a tabular format.
`spot_diff.source_list.html`	A report that lists the compiler used to compile the application, and the source files compiled.

Example Program

The following `test` example program was run with SPOT to generate the reports discussed in this chapter. The program has three routines, each of which targets a different kind of event:

- The `fp_routine` routine does floating point computation on three 80 MB arrays. The routine has floating point operations, and also (because of the size of the array) significant amounts of memory traffic, which appears as read and write memory bandwidth consumption.
- The `cache_miss` routine is a test of memory latency. Each pointer chase in the key loop brings in another cacheline, resulting in many cache misses, and also a significant amount of memory read bandwidth.
- The `tlb_miss` routine is identical to the `cache_miss` routine except for the way it is called. The reason for duplicating the code is to show clearly the location in the code where the events are happening. This routine brings in a new TLB page on every pointer chase in the key loops, so the routine encounters both cache and TLB misses.

```
#include <stdio.h>
#include <stdlib.h>

void fp_routine(double *out, double *in1, *double *in2, int n)
{
  for (int i=0; i<n; i++) (out[i]=in1[i]+in2[i];)
}

int** cache_miss(int **array, int size, int step)
{
  for (int i=0; i<size-step; i++){array[i]=(int*)&array[i+step];}
  for (int i=size-step; i<size; i++)
     {array[i]=(int*)&array[i-size+step];}

  int ** cp=(int**)array[0];
  for (int i=0; i<size*16; i++) {cp= (int**)*cp;}
}

int** tlb_miss(int **array, int size, int step)
{
  for (int i=0; i<size-step; i++){array[i]=(int*)&array[i+step];}
  for (int i=size-step’ i<size’ i++)
  {array[i]=(int*)&array[i-size+step];}

  int ** cp=(int**)array[0];
  for (int i=0; i<size*16; i++) {cp= (int**)*cp;}
  return cp;
}

void main()
{
  double * out, *in1, *in2;
  int **array;

  out=(double*) calloc(sizeof(double),10*1024*1024);
  in1=(double*) calloc(sizeof(double),10*1024*1024);
```

```
  in2=(double*) calloc(sizeof(double),10*1024*1024);
  for (int rpt=0; rpt <100; rpt++)
fp_routine(out,in1,in2,10*1024*1024);
  free(out);
  free(in1);
  free(in2);

  array=(int**)calloc(sizeof(int*),10*1024*1024);
  cache_miss(array,10*1024*1024,64/sizeof(int*));
  tlb_miss(array,10*1024*1024,8192/sizeof(int*));
  free (array);
}
```

The program was compiled with the Oracle Solaris Studio 12.2 c compiler:

```
cc -g -O -o test test.c
```

The `spot_summary` Report

The `spot_summary.html` report is updated each time you run SPOT in the current directory. The report list the run number, application, and date and time of each run, the date and time the file was last updated, and the version of SPOT that was used for the runs.

SPOT Summary

The SPOT Diff Report contains tables with the key metrics.
The tables are completed after the spot run finishes.

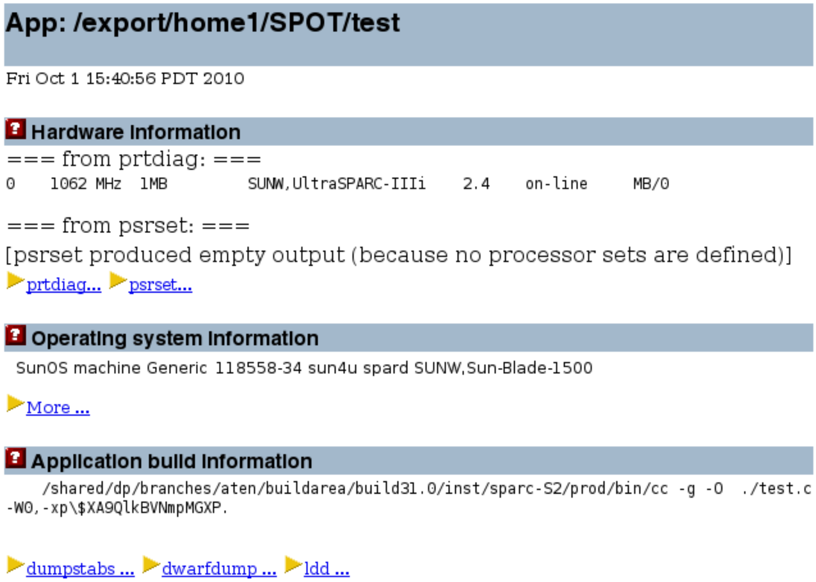
Run number	Application	Date and time collected
1	/export/home1/SPOT/test	Thu Sep 16 17:29:55 PDT 2010
2	/export/home1/SPOT/test	Thu Sep 16 18:41:01 PDT 2010
3	/export/home1/SPOT/test	Tue Sep 21 16:22:23 PDT 2010
4	/export/home1/SPOT/test	Tue Sep 21 16:59:48 PDT 2010
	Tue Sep 21 16:59:48 PDT 2010	

The SPOT Run Report

The SPOT report for each run of your application with SPOT includes a section for each of the files that SPOT writes to the subdirectory for that run. To display the report, point your web browser to the `index.html` file in the subdirectory.

Runtime System and Build Information

The Hardware information, Operating system information, and Application build information sections of the SPOT report list details on the system on which SPOT was run on the application and on how the application was compiled. This information can help you to reproduce the same results at another time.



Processor Events

The Application stall information section displays information collected by the `ripc` tool about what processor events were encountered during the run of the application. The processor has event counters that are incremented either each time an event occurs or each cycle during the duration of an event. Using these counters, SPOT can determine values for the cache miss rate, or the number of cycles lost due to cache misses. The information is displayed in several text subsections.

Note – You can run `ripc` as a stand-alone tool. Type `ripc -h` for a list of the command line options, and consult the `ripc`(1) man page for more information.

Application stall information (using ripc)

```
=========================================================
Analysis Of Application Stall Behavior
=========================================================
Stall                           Ticks          Sec      %
=========================================================
ITLB-miss                    1,730,770      0.002    0.0%
DTLB-miss               14,717,602,440     13.497   15.9%
Instr. Issue             2,198,130,170      2.016    2.4%
D-Cache                 18,887,724,964     17.321   20.4%
E-Cache                 53,143,404,898     48.735   57.3%
RAW-miss                     9,113,005      0.008    0.0%
StoreQ                   2,598,679,076      2.383    2.8%
FPU Use                         63,144      0.000    0.0%
IU Use                     562,820,314      0.516    0.6%
---------------------------------------------------------
Total Stalltime         92,119,268,781     84.477   99.4%
---------------------------------------------------------
Total CPU Time                         85 Sec
Total Elapsed Time                    101 Sec
Total Cycle Count          92,689,463,161
Total Instr. Count         11,117,934,446
FP Instructions             1,048,826,376   9.4% of Total Instr.
IPC                                 0.120 (instr/time)
---------------------------------------------------------
Unfinished FPops                        0

--------------------------------------------------------
Cache Statistics
========================================================
  Name                       Events       Event/Inst.    %
========================================================
ITLB_miss                    1,730,770      0.000    0.0% of Instructions
IC_miss                    113,010,738      0.010    1.2% of IC Ref
EC_ic_miss                   3,111,406      0.000    2.8% of IC misses
DTLB_miss               14,717,602,440      1.324  132.4% of Instructions
DC_rd                    2,645,896,804      0.238   23.8%
DC_rd_miss                 699,324,418      0.063   26.4% of DC_rd
EC_rd_miss                 754,149,849      0.068  107.8% of DC_rd_miss
DC_wr                    1,608,168,681      0.145   14.5%
DC_wr_miss               1,704,250,368      0.153  106.0% of DC_wr
EC_wr_miss               1,704,249,061      0.153  100.0% of DC_wr_miss
EC_miss                    757,477,626      0.068
FP Inst.           A= 1,048,826,348   M=       82    9.4% of Total Instr.

=========================================
Maximum Resources Used By The Process
=========================================
Heap                      245768 KB
RSS                       246864 KB
Size                      247432 KB
System Time                    0 Sec
User   Time                   85 Sec
Est. Read  Bandwidth      543.915 MB/Sec
Est. Write Bandwidth      96.958 MB/Sec
=========================================================
Pairs Of Top Four Stall Counters:
[These counter pairs can be used with -h flag of collect
command to study application stall behavior more closely.]
=========================================================
#Rstall_storeQ,Re_DC_miss
#Cycle_cnt,Re_EC_miss
#Cycle_cnt,DTLB_miss


=============================================================
```

Analysis of Application Stall Behavior Section

The Analysis of Application Stall Behavior section shows the percentage of the total number of cycles lost to each type of processor event. The events are different on different processors. For example, an UltraSPARC IV+ has a third level of cache that is not present on previous generations of processors.

In this report for a run of the example code, the time is lost due to Data Cache misses, External Cache misses, and Data TLB misses. Together these three types of events account for more than 93% of the execution count of the benchmark:

- The Data Cache miss time represents time spent by load instructions that found their data in the External Cache.
- The External Cache miss time is accumulated by load instructions where the data was not resident in either the Data Cache or the External Cache, and had to be fetched from memory.
- The Data TLB miss time is caused by memory accesses where the TLB mapping is not resident in the on-chip TLB, and has to be fetched using a trap to the operating system.

The section also shows data that summarizes the efficiency of the entire run. The IPC is the number of instructions executed per cycle. The Grouping IPC is an estimate of what the IPC would be if the processor did not encounter any stall events.

A single line at the bottom of the section reports the number of unfinished floating point traps. These traps can occur in some exceptional circumstances on most UltraSPARC processors. They can take a significant time to complete, and are also hard to observe in the profiles. Most of the time this count should be zero, but if there are a large number of such events, it is definitely worth investigating what is causing them.

Cache Statistics Section

The Cache Statistics section reports the number of events that occurred as a proportion of the total number of opportunities for the events to occur, in other words, how much useful work the processor is achieving each cycle. An example is the number of cache misses as a proportion of cache references.

Maximum Resources Used By The Process Section

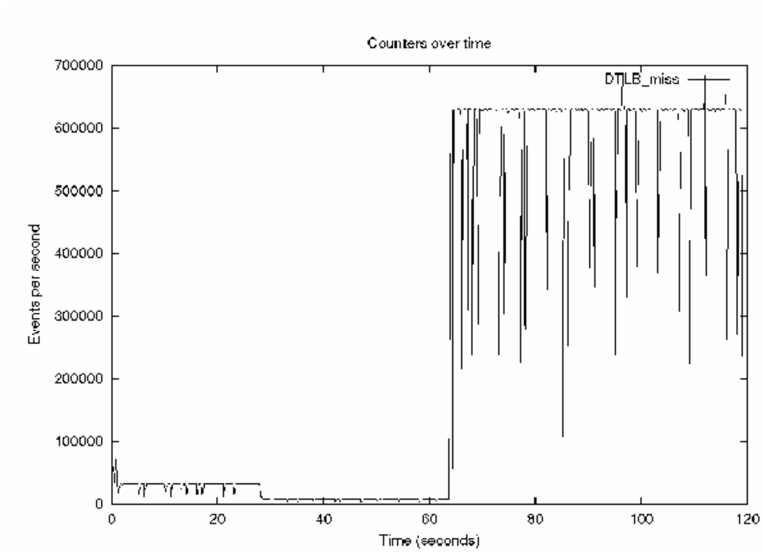
The Maximum Resources Used By The Process section shows data on the memory utilization for the application, and the user and system time.

Pairs of Top Four Stall Counters Section

The Pairs of Top Four Stall Counters section lists the performance counters that should be profiled if more detail is required.

Event Graph

If `rpic` locates the `gnuplot` software in the system's path, it also generates a graph of how the events occurred over the entire run time. For example, the following graph clearly shows three phases of the `test` application. The first two phases contain only a few TLB misses, but the graph shows large numbers of misses during the execution of the final `tlb_misses` routine.



Instruction Frequency Data

The Instruction frequency statistics from BIT section of the report shows information generated by the `BIT` tool on the frequency with which different assembly language instructions are used during the run of the application, providing a more detailed kind of instruction count.

`BIT` does not generate information about the performance of the application, but it does provide information about what the application is doing.

The section lists the number of instructions executed, and for these instructions, how many were located in the delay slot and how many were annulled (not executed).

The Annulled and In Delay Slot columns require some explanation. Every branch instruction has a delay slot, which is the instruction immediately following the branch. This instruction is executed together with the branch. The branch can annul the instruction in the delay slot so that the instruction is performed only if the branch is taken.

Instruction frequency statistics from BIT

```
Instruction frequencies of /export/home1/SPOT/test
Instruction              Executed     (%)
 TOTAL                 7963939877 (100.0)
 float ops             4194304000 ( 52.7)
 float ld st           3145728000 ( 39.5)
 load store            3502243842 ( 44.0)
 load                  2432696322 ( 30.5)
 store                 1069547520 ( 13.4)
-------------------------------------------
Instruction              Executed     (%)       Annulled   In Delay Slot
 TOTAL                 7963939877 (100.0)
 lddf                  2097152000 ( 26.3)           100               0
 add                   1415578442 ( 17.8)             0         5242878
 stdf                  1048576000 ( 13.2)             0       262143900
 faddd                 1048576000 ( 13.2)             0               0
 prefetch               791674576 (  9.9)             0               0
 br                     602931822 (  7.6)             0               0
 subcc                  602931622 (  7.6)             0               0
 lduw                   335544322 (  4.2)             0       335544320
 stw                     20971520 (  0.3)             4               8
```

Instruction counts for the entire application... Broken up by functions...

Note – BIT works by running a modified version of the application, *application_name*.instr, which contains instrumentation code that collects count data over the course of the run. For this instrumentation to work, you must have compiled the application with an optimization level of -xO1 or higher.

Note – You can run BIT as a stand-alone tool. Type bit -hfor a list of command line options, and consult the bit(1) man page for more information.

Bandwidth Data

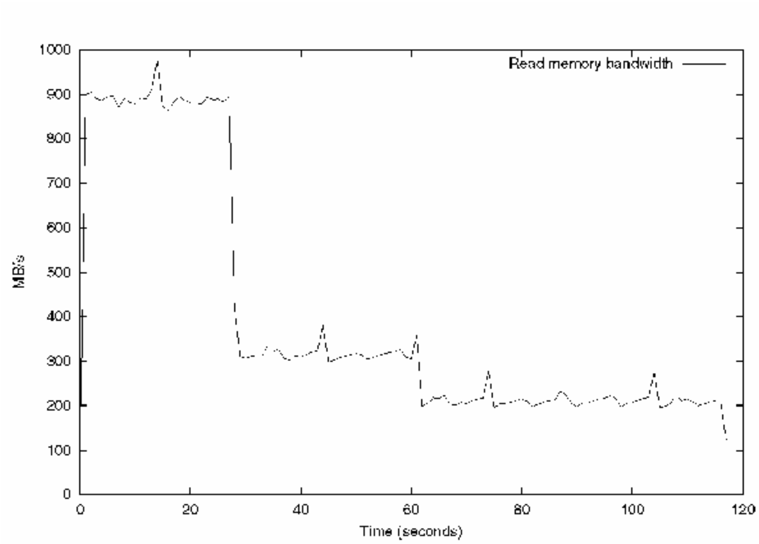
It is not possible to measure the bandwidth consumption of a single process, since one process can read memory that is attached to processors running other processes. Hence the bandwidth reported here is system-wide. A consequence is that it is not possible to attribute the memory activity to a single process if there are multiple processes running on the system.

SPOT collects bandwidth data if you run it with the -X option and root privileges. The average bandwidth consumption over the entire run of the test program is reported.

Bandwidth data

```
Bandwidth Report for krolik
-----------------------------------------------------------
 Read  memory bandwidth: 467.4 MB/sec (total bytes = 53425521536)
 Write memory bandwidth:  86.5 MB/sec (total bytes = 9889076224)
 Total memory bandwidth: 554.0 MB/sec (total bytes = 63314597760)
 Elapsed time   : 109.000 secs
-----------------------------------------------------------
```

If bw locates the gnuplot software in the system's path, it generates a graph of the bandwidth data. For example, the following graph shows the read memory bandwidth consumed over the entire run of the application. The fp_routine routine consumes the most bandwidth because it is three streams of data being used by the processor. The other two routines use less bandwidth because they are pointer chasing, and therefore testing memory latency.



Note – You can run bw as a stand-alone tool, outside of SPOT. Type bw -hfor a list of the command line options, and consult the bw(1) man page for more information.

Traps Data

The traps data section displays data that SPOT collects by running the trapstat software for the duration of the run of the application. SPOT collects this data when you run it with the -X option and with root privileges.

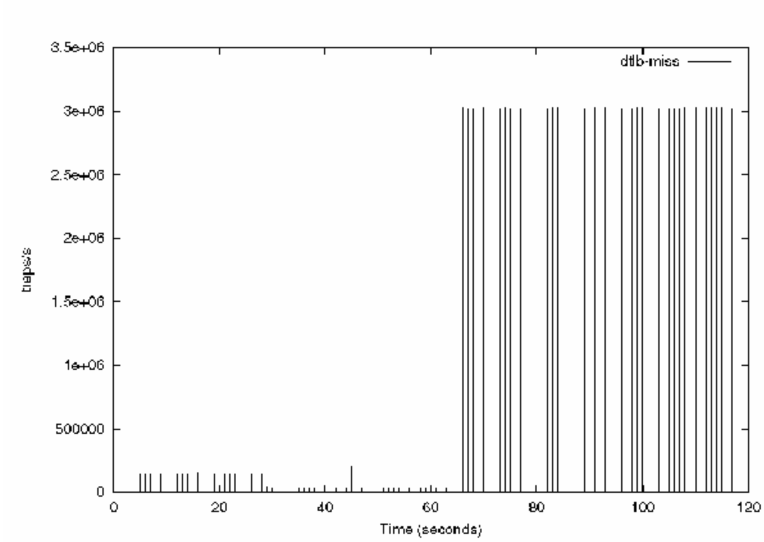
`trapstat` counts system-wide traps, not just the traps that are due to this process, so it is not possible to distinguish between traps generated by the application and those generated by other processes running on the system.

The table reports the average number of traps encountered per second.

traps data

```
-----------------------------------------------------------
             cleanwin      7038.4 (traps/sec)
            dtlb-miss   1493641.3 (traps/sec)
            dtlb-prot       239.9 (traps/sec)
         fill-kern-64     10082.8 (traps/sec)
         fill-user-32     82633.0 (traps/sec)
     fill-user-32-cln     12753.0 (traps/sec)
            fix-align         0.1 (traps/sec)
           flush-wins         8.8 (traps/sec)
          fp-disabled         0.1 (traps/sec)
          fp-xcp-other        0.1 (traps/sec)
              get-psr         0.0 (traps/sec)
            gethrtime      8815.1 (traps/sec)
                getts      1109.6 (traps/sec)
              int-vec       108.6 (traps/sec)
            itlb-miss     98420.5 (traps/sec)
              level-1        97.7 (traps/sec)
             level-10       100.0 (traps/sec)
             level-14       101.0 (traps/sec)
              level-4        22.5 (traps/sec)
              level-6         7.1 (traps/sec)
              level-9        82.9 (traps/sec)
           self-xcall         2.0 (traps/sec)
      spill-asuser-32      5692.7 (traps/sec)
  spill-asuser-32-cln      6555.0 (traps/sec)
        spill-kern-64     10163.9 (traps/sec)
        spill-user-32     83832.9 (traps/sec)
    spill-user-32-cln      1511.7 (traps/sec)
           syscall-32      3529.6 (traps/sec)
-----------------------------------------------------------
```

If `trapstat` locates the `gnuplot` software in the system's path, it also generates a graph of traps over time. The following graph shows number of TLB traps reported over the entire run of the `test` application. As expected, the traps reported by `trapstat` correspond to the traps reported by the performance counter on the processor.



Application HW Counter Profile Output

If you request extended information by running SPOT with the `-X` option, then SPOT profiles the application using the performance counters that contribute the most stall time to the run of the application. It generates several profiles of the application that indicate exactly where in the code the events are occurring.

For this run of the `test` application, it is apparent that the External Cache (EC) misses are mainly attributable to the `cache_miss` and `tlb_miss` routines.

Application HW counter profile output

```
Functions sorted by metric: Exclusive Rstall_storeQ Events

Excl.          Excl.         Excl.     Excl. Bit   Excl.      Name
Rstall_storeQ  Re_DC_miss    Bit Func  Inst Exec   Bit Inst
Events sec.    Events sec.   Count                 Annul
3.617          63.791        103       7963939877  104        <Total>
2.958          21.904        100       6553604200  100        fp_routine
0.572           0.             0                0    0        memset
0.044          20.339          1        705167420    2        tlb_miss
0.043          21.548          1        705167420    2        cache_miss
0.              0.             0                0    0        _start
0.              0.             1              837    0        main
```

More ...

```
Functions sorted by metric: Exclusive Cycle_cnt Events

Excl.        Excl.         Excl.     Excl. Bit   Excl.      Name
Cycle_cnt    Re_EC_miss    Bit Func  Inst Exec   Bit Inst
Events sec.  Events sec.   Count                 Annul
75.663       49.285        103       7963939877  104        <Total>
29.263        9.058        100       6553604200  100        fp_routine
23.665       19.778          1        705167420    2        tlb_miss
22.002       20.449          1        705167420    2        cache_miss
 0.733        0.             0                0    0        memset
 0.           0.             0                0    0        _start
 0.           0.             1              837    0        main
```

More ...

```
Functions sorted by metric: Exclusive Cycle_cnt Events

Excl.        Excl.      Excl.     Excl. Bit   Excl.      Name
Cycle_cnt    DTLB_miss  Bit Func  Inst Exec   Bit Inst
Events sec.  Events     Count                 Annul
75.700       172000516  103       7963939877  104        <Total>
29.335         4000012  100       6553604200  100        fp_routine
23.650       167000501    1        705167420    2        tlb_miss
21.979         1000003    1        705167420    2        cache_miss
 0.735               0    0                0    0        memset
 0.                  0    0                0    0        _start
 0.                  0    1              837    0        main
```

More ...

Clicking the More hyperlinks opens more detailed displays of source code (if you compiled the application with the `-g` option and the source code is accessible) and the disassembly code.

Application Profile Output

The Application Profile Output section shows a summary of which routines consumed the most run time.

Application profile output

```
Functions sorted by metric: Exclusive User CPU Time

Excl.     Incl.     Excl.     Excl.     Excl.     Excl. Bit  Excl.       Name
User CPU  User CPU  Sys. CPU  Wall      Bit Func  Inst Exec  Bit Inst
  sec.      sec.      sec.       sec.   Count                Annul
87.071    87.071    0.861     109.076   103       7963939877 104         <Total>
34.154    34.154    0.030      41.919     1        705167420    2        tlb_miss
29.581    29.581    0.030      37.656   100       6553604200 100         fp_routine
22.466    22.466    0.040      27.739     1        705167420    2        cache_miss
 0.871     0.871    0.761       1.761     0                0    0        memset
 0.       87.071    0.          0.        0                0    0        _start
 0.       87.071    0.          0.        1              837    0        main
```

More ...

Clicking the More hyperlink displays a page that allows exploration of the application in more depth.

```
HTML data from experiment(s):
 ./spot_run1/test.1.er ./spot_run1/bit.1.er

Functions sorted by metric: Exclusive User CPU Time

Excl.     Incl.     Excl.     Excl.     Excl.     Excl. Bit  Excl.       Name
User CPU  User CPU  Sys. CPU  Wall      Bit Func  Inst Exec  Bit Inst
  sec.      sec.      sec.       sec.   Count                Annul
87.071    87.071    0.861     109.076   103       7963939877 104         <Total>
34.154    34.154    0.030      41.919     1        705167420    2        [trimmed] tlb_miss src Caller-callee
29.581    29.581    0.030      37.656   100       6553604200 100         [trimmed] fp_routine src Caller-callee
22.466    22.466    0.040      27.739     1        705167420    2        [trimmed] cache_miss src Caller-callee
 0.871     0.871    0.761       1.761     0                0    0        [trimmed] memset  Caller-callee
 0.       87.071    0.          0.        0                0    0        _start
 0.       87.071    0.          0.        1              837    0        [trimmed] main src Caller-callee
```

The hyperlink at the top of each column lets you make that column the sort key for the data on the page.

The columns list the following data:

- The Excl User CPU column displays the amount of time spent in the source code corresponding to the routine shown on the right.
- The Incl User CPU column displays the amount of time spent in a given routine, plus the routines that routine calls, which is apparent when looking at the row for the `main` routine. No exclusive time attributed to that routine, but it has 120 seconds of inclusive time, which is all due to the routines that the `main` routine calls.
- The Excl Sys CPU column displays the system time attributed to the various routines.
- The Excl Wall column displays the number of seconds spent in a given routine. For a single threaded application, this time is the sum of the user time, system time, and various other wait and sleep times. For a multithreaded application, it is the time spent by the master thread, which in many cases might not be actively doing work.
- The Excl Bit Func column reports the number of times that each function is called. This count does not extend to library functions, so the routine `_memset`, which is in a library, is attributed with a count of zero even through it is called multiple times.

- The Excl Bit Instr Exec column counts the dynamic number of instructions executed during the run of the application for each routine.
- The Excl Bit Instr Annul column shows a count of the instructions that were annulled (not executed) during the run.

On the right side of the page, the Name column contains links to the routines:

- The trimmed link goes to a trimmed-down version of the disassembly of the routine. The trimming is done to remove parts of the code that have no time or events attributed to them.
- The routine name link goes to the complete disassembly for the routine. This file can be very large since often many routines share the same source file. So the trimmed link is frequently the more appropriate one to use.
- The src link goes to the source code for the routine. This link is available only if the program was compiled with the `-g` or `-g0` option.
- The Caller-callee link goes to the caller-callee page, which indicates which routines call which other routines, and how the time is attributed between them.

Source Code Page

The source code report for a routine shoes how time is attributed at the source code level. For example, the source code report for the `tlb_misses` routine, the line starting with ## and highlighted has a high count for user time and dynamic instruction count for one of the processor events. The source code report also includes compiler commentary about the two loops in the code that are shown.

```
                                                     20. int** tlb_miss(int **array, int size, int step)
   0.070      1        34072089    1                 21. {

                                                    Source loop below has tag L5
   0.         0               0    0                 22.   for (int i=0; i<size-step; i++){array[i]=(int*)&array[i+step];}

                                                    Source loop below has tag L6
   0.         0            4117    0                 23.   for (int i=size-step; i<size; i++)
   0.         0            2566    1                 24.     {array[i]=(int*)&array[i-size+step];}
                                                     25.
   0.         0               3    0                 26.   int ** cp=(int**)array[0];

                                                    Source loop below has tag L7
## 34.084     0       671088643    0                 27.   for (int i=0; i<size*16; i++) {cp= (int**)*cp;}
                                                     28.   return cp;
   0.         0               2    0                 29. }
                                                     30.
```

Disassembly Page

The disassembly page holds more specific information. A hot line of disassembly is highlighted. The execution counts for the individual assembly language instructions are shown, so you can see that the loop is entered once and iterated nearly 170 million times. The hyperlinks let you rapidly navigate to either line of source code that generated the disassembly instruction of the target of a branch instruction.

```
                                                   Source loop below has tag L4
                                                    16.   for (int i=0; i<size*16; i++) {cp= (int**)*cp;}
    0.        0              1    0                      [16]    11288:  cmp         %o1, 0
    0.        0              1    0                      [16]    1128c:  ble,pn      %icc,0x112a8
    0.        0              1    0                      [16]    11290:  clr         %o2
    0.        0              1    0                      [15]    11294:  add         %o1, -1, %o4
## 22.216     0      167772160    0                      [16]    11298:  inc         %o2
    0.        0      167772160    0                      [16]    1129c:  cmp         %o2, %o4
    0.        0      167772160    0                      [16]    112a0:  ble,pt      %icc,0x11298
    0.190     0      167772160    0                      [16]    112a4:  ld          [%o3], %o3
                                                    17.   return cp;
                                                    18. }
    0.        0              1    0                      [18]    112a8:  retl
    0.        0              1    0                      [18]    112ac:  mov         %o3, %o0
                                                    19.
                                                    20. int** tlb_miss(int **array, int size, int step)
                                                    21. {
```

Caller-Callee Page

The caller-callee page shows information for the functions that call the routine (callers) and the functions that the routing calls (callees).

```
Function Name: _start
Attr.     Attr.     Attr. Bit   Attr.      Name
User CPU  Bit Func  Inst Exec   Bit Inst
  sec.    Count                 Annul
87.071      0               0     0       <Total>
 0.         0               0     0      *_start
87.071      0               0     0       main

Function Name: cache_miss
Attr.     Attr.     Attr. Bit   Attr.      Name
User CPU  Bit Func  Inst Exec   Bit Inst
  sec.    Count                 Annul
22.466      0               0     0       main
 0.         1       705167420     2       <Total>
22.466      1       705167420     2      *cache_miss

Function Name: fp_routine
Attr.     Attr.     Attr. Bit   Attr.      Name
User CPU  Bit Func  Inst Exec   Bit Inst
  sec.    Count                 Annul
29.581      0               0     0       main
 0.       100      6553604200   100       <Total>
29.581    100      6553604200   100      *fp_routine

Function Name: main
Attr.     Attr.     Attr. Bit   Attr.      Name
User CPU  Bit Func  Inst Exec   Bit Inst
  sec.    Count                 Annul
87.071      0               0     0       _start
 0.         1             837     0       <Total>
 0.         1             837     0      *main
34.154      0               0     0       tlb_miss
29.581      0               0     0       fp_routine
22.466      0               0     0       cache_miss
 0.871      0               0     0       memset
```

The caller-callee information is complex to read. In each section, the routine of focus is indicated by an asterisk.

For example, in the section for the main routine, that routine has an asterisk to the left of its name. The _start routine and the <Total> routine (a synthetic metric representing the run time of the entire application) are listed above the main routine. This information indicates that the main routine is called by the _start routine. The four routines listed after the main routine are routines that are called by the main routine.

The first column lists the attributed user CPU time. About 88 seconds are attributed to the _start routine. These seconds are the time that _start spends calling the main routine. The attributed time for the main routine is 0, indicating that no time is spent in that routine. The attributed time for the four routines called by main adds up to the 88 seconds.

The section of the page for the fp_routine routine shows that almost 30 seconds are spent by the main routine calling the fp_routine routine. However, in this case, all of that time is spent directly in the fp_routine routine.

Note – The profile data is collected with the collect tool, so it is stored as a Performance Analyzer experiment and you can also examine it with the Performance Analyzer or the er_print tool. For more information, see the analyzer(1) and er_print(2) man pages.

You can also convert experiment data collected by the collect tool to HTML format by using the er_html tool as a stand-alone tool. Type er_html -hfor a list of the command line options, and consult the er_html(1) man page for more information.

The spot_diff Report

SPOT automatically runs the spot_diff script after each data collection run. This tool compares the new run with the preceding ones. The output from the spot_diff script is the spot_diff.html file, which contains several tables that compare SPOT experiment data.. Large differences are highlighted to alert you to possible performance problems.

You can call the spot_diff script from the command line for situations where you need greater control over particular experiments. For example, to generate a spot_diff report called exp_1-exp_2-diff.html comparing experiment_1 and experiment_2, you would type:

```
spot_diff -e experiment_1 -e experiment_2 -o exp1-exp2-diff
```

For more information, see the spot_diff(1) man page.

The spot_diff report examined in this section was automatically generated by SPOT after running SPOT twice on the test example application. For the first run, test was compiled with the -xO2 option. For the second run, the application was compiled with the -fast option. The output from the first run was recorded in the test_xO2_1 directory. The output from the second run was recorded in the test_fast_1 directory.

Summary of Key Experiment Metrics Table

The Summary of Key Metrics section of the report compares several top-level metrics for the two experiments. You can see that compiling with higher optimization causes both the runtime and the number of executed instructions to decrease. It is also apparent that the total number of bytes read and written to the bus is similar, but because the second experiment ran faster, its bus bandwidth is correspondingly higher.

Summary of Key Experiment Metrics

Click on row heading to sort

Metric	test_O2_1	test_fast_1
Elapsed Time (s)	183	2
User Time (s)	149	1
System Time (s)	0	0
Instr Count (Mln)	15,141	587
IPC	0.09	0.25
BW (read, MB/s)	351.7	410.2
BW (write, MB/s)	56.9	267.5
Bus reads (MB)	58,375	820
Bus writes (MB)	9,442	535
RSS (MB)	246864	246864
Machine	krolik	krolik

Summary of Top Stalls Tables

The top causes for stalls are displayed in two tables, one by percent execution time and the other in absolute seconds. Depending on your preference or the application you are observing, one of the tables might be more useful than the other in identifying a performance problem.

Summary of Top Stalls (listed as percentage)

Click on row heading to sort

Metric	test_O2_1	test_fast_1
D-Cache	11.3%	6.0%
DTLB-miss	9.1%	0.8%
E-Cache	61.4%	23.2%
FP	6.9%	1.8%
FPU_Use	1.2%	0.1%
IU_Use	0.4%	1.2%
Instr_Issue	1.5%	7.3%
StoreQ	0.6%	36.3%
Total Stalltime	85.6%	75.1%

Summary of Top Stalls (listed in seconds)

Click on row heading to sort

Metric	test_O2_1	test_fast_1
D-Cache	16.0	0.0
DTLB-miss	13.0	0.0
E-Cache	91.0	0.0
FPU_Use	1.0	0.0
IU_Use	0.0	0.0
Instr_Issue	2.0	0.0
StoreQ	0.0	0.0
Total Stalltime	127.0	0.0

For the example used here, it might be more useful to look at the top stalls listed in seconds because the two runs are doing the same work. The table shows that the optimizations enabled by the `-fast` option significantly reduce the stalls. By clicking the column head hyperlinks in the table to go to the SPOT experiment profiles for the two runs, you can learn that:

- Prefetch instructions are responsible for reducing the cache stalls.
- Better code scheduling eliminated back-to-back floating point operations, which reduced the Floating Point Use stalls.

Bit Instruction Counts Table

In both cases, the binary was compiled with an optimization level higher than `-xO1`, so SPOT was able to collected instruction count data. This data is displayed in the Bit Instruction Counts Report.

Bit Instruction Counts Report

Units are million instructions
Only opcodes and opcode groups > 0.05*TOTAL are printed.

Opcode	test_O2_1	test_fast_1
TOTAL	11,953	107
add	4,592	31
br	1,405	3
faddd	1,048	10
float_ld_st	3,145	31
float_ops	4,194	41
lddf	2,097	20
load	2,432	20
load_store	3,502	52
stdf	1,048	10
store	1,069	31
subcc	1,405	3

The difference in instruction counts between the two runs is primarily due to unrolling (and to a lesser extent, inlining) done when compiling with the `-fast` option, which greatly reduces the number of branches and loop-related calculations.

Only instructions that show both high variance between experiments and a high total count are displayed in this table.

You can see more detailed bit data by clicking the column head hyperlinks and looking at the instruction frequency statistics in the two experiment profiles.

Flags Table

In the Flags report, you can see that the only difference between the two experiments is the optimization option.

Flags Report

test_O2_1	test_fast_1
/shared/dp/branches/aten/buildarea/build31.0/inst/sparc-S2/prod/bin/cc -g **-x02** Source files compiled with this compiler	/shared/dp/branches/aten/buildarea/build31.0/inst/sparc-S2/prod/bin/cc -g **-fast** Source files compiled with this compiler

Traps Table

While the total number of Data TLB traps in the two experiments is roughly the same, the trap rate, as shown in the Traps report, is higher in fast experiment because that binary runs in less time. All other trap rates, which you can see in the hyperlinked experiment reports, are too low to report.

Traps Report

Trap	test_O2_1	test_fast_1
dtlb-miss	931,725	95,516
fill-kern-64	7,014	106,558
spill-kern-64	7,081	106,642

Time Spent in Top Functions Tables

The time spent in top functions is displayed in two tables, one in percentage of time and one in seconds of execution time. In both tables, it is apparent that the `cache_miss()`, `fp_routine()`, and `tlb_miss()` functions are inlined when the application is compiled with the `-fast` option, but not when it is compiled with the `-x02` option.

Percent Time Spent in Top Functions

Function	test_O2_1	test_fast_1
cache_miss	14.6%	-
fp_routine	62.9%	-
tlb_miss	22.0%	-

Seconds Spent in Top Functions

Function	test_O2_1	test_fast_1
Total User Time	152.88	1.83
cache_miss	22.38	-
fp_routine	96.11	-
tlb_miss	33.58	-

Notes on the SPOT Reports

Here are some final things to be aware of when using SPOT reports:

- All of the data and commands used to generate a report are recorded in the same directory as the report.
- Each SPOT run directory contains a Performance Analyzer experiment (`test.1.er`) that is used to generate the run profile. You can load this experiment into the Performance Analyzer or the `er_print` tool if further investigation of the profile is necessary.
- The SPOT run directory contains log files for the various stages of the run. These log files report any error conditions that were encountered. The `debug.log` file contains a transcript of all of the commands used to generate the run report.
- The SPOT run report contains information that might be considered confidential, so take care in handling the report. Examples of information that the report might contain are:
  - The `spot` command that ran the binary.
  - The location of the binary and where it was run.
  - The location of the compiler used to build the binary.
  - The source code files that contain routines where significant time was spent.

Index

S

T