Developer Magic™: Performance Analyzer User's Guide

Document Number 007–2581–004

New Features

Developer Magic™: Performance Analyzer User's Guide 007–2581–004

This revision of the *Developer Magic: Performance Analyzer User's Guide* supports the 2.7 release of the ProDev Workshop tools.

Record of Revision

Version	*Description*
Revision level	Month Year Original Printing.
2.7	June 1998 Revised to reflect changes for the ProDev WorkShop 2.7 release, including the ability to present SpeedShop data within WorkShop.

Contents

Page

Page

Page

Page

Page

Page

Figures

Page

Page

Page

Page

Tables

About This Guide

This publication documents the ProDev WorkShop Performance Analyzer for release 2.7, running on IRIX systems.

This release of the WorkShop toolkit requires the following software levels:

- IRIX 6.2 or higher
- MIPSpro 7.2.1
- SpeedShop 1.3.1

Related Publications

The following documents contain additional information that may be helpful:

- *SpeedShop User's Guide*
- *C Language Reference Manual*
- *C++ Language System Library*
- *C++ Language System Overview*
- *C++ Language System Product Reference Manual*
- *C++ Programmer's Guide*
- *Developer Magic: WorkShop Pro MPF User's Guide*
- *Developer Magic: Debugger User's Guide*
- *Developer Magic: Static Analyzer User's Guide*
- *Developer Magic: ProDev WorkShop Overview*
- *Fortran 77 Language Reference Manual*
- *MIPSPro 7 Fortran 90 Commands and Directives Reference Manual*

Obtaining Publications

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

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

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

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

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

Conventions

The following conventions are used throughout this document:

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

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. You can contact us in any of the following ways:

- Send us electronic mail at the following address:

  `techpubs@sgi.com`

- Contact your customer service representative and ask that a PV be filed.

- Call our Software Publications Group in Eagan, Minnesota, through the Customer Service Call Center, using either of the following numbers:

  1–800–950–2729 (toll free from the United States and Canada)

  +1–612–683–5600

- Send a facsimile of your comments to the attention of Software Publications Group in Eagan, Minnesota, at fax number +1–612–683–5599.

We value your comments and will respond to them promptly.

Introduction to the Performance Analyzer [1]

The Performance Analyzer helps you understand your program in terms of performance. If there are areas in which performance can be improved, it helps you find those areas and make the changes. This chapter provides a brief introduction to the Performance Analyzer tools and describes how to use them to solve performance problems. It includes the following sections:

- Performance Analyzer Overview, see Section 1.1, page 1.
- The Performance Analyzer Tools, see Section 1.2, page 1.
- Sources of Performance Problems, see Section 1.3, page 4.
- Interpreting Performance Analyzer Results, see Section 1.4, page 6.

1.1 Performance Analyzer Overview

To conduct performance analysis, you first run an experiment to collect performance data. Specify the objective of your experiment through a task menu or with the SpeedShop command `ssrun`(1). The Performance Analyzer reads the required data and provides charts, tables, and annotated code to help you analyze the results.

There are three general techniques for collecting performance data:

- Counting. This involves counting the exact number of times each function or basic block has been executed. This requires *instrumenting* the program; that is, inserting code into the executable to collect counts.
- Profiling. The program's program counter (PC), call stack, and/or resource consumption are periodically examined and recorded. For a list of resources, see Section 1.4.2, page 7.
- Tracing. Events that impact performance, such as reads and writes, system calls, floating-point exceptions, and memory allocations, reallocations, and frees, can be traced.

1.2 The Performance Analyzer Tools

This section describes the major windows in the Performance Analyzer toolset. The main window (see Figure 1, page 2) contains the following major areas:

- The function list area, which shows functions with their performance metrics.
- The system resource usage chart, which shows the mode of the program at any time.
- The time line, which shows when sample events occur in the experiment and controls the scope of analysis for the Performance Analyzer views.

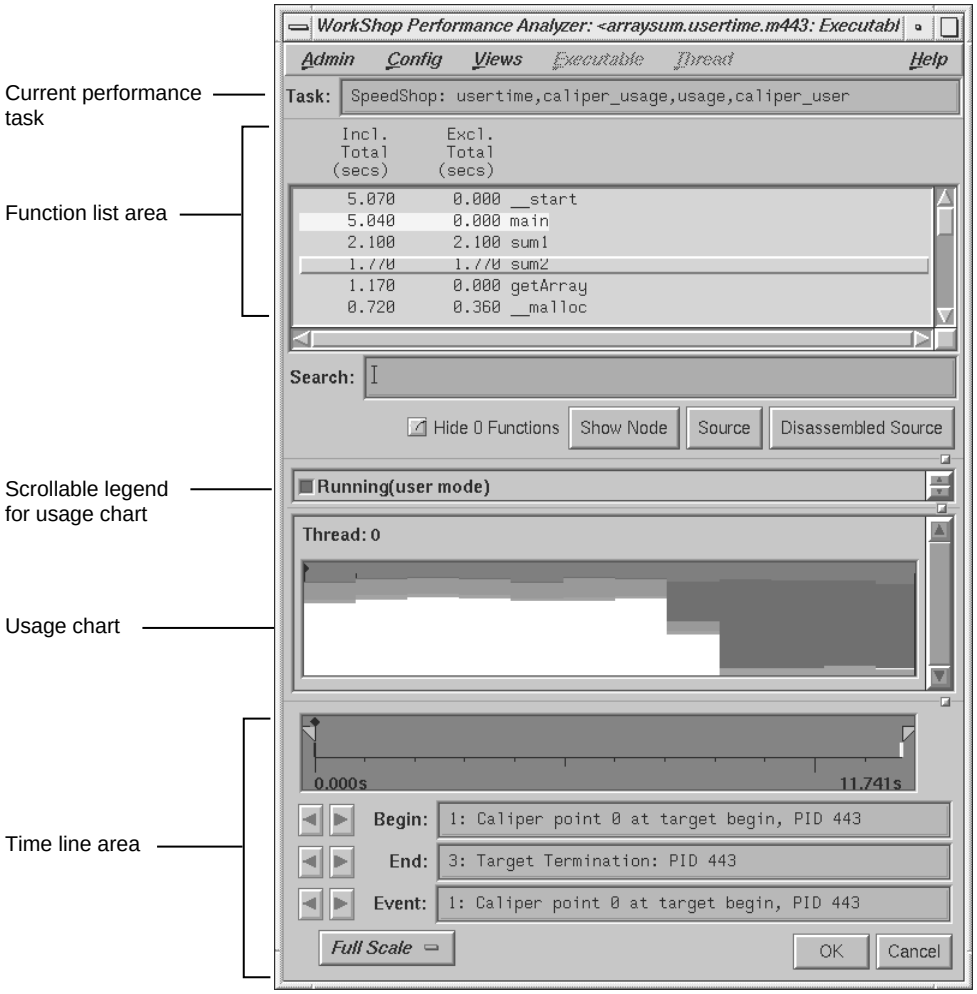

Figure 1. Performance Analyzer Main Window

The following supplemental views bring up their own windows:

- `Usage View (Graphs)`, containing charts that indicate resource usage and the occurrence of samples corresponding to time intervals set by the time line calipers (see Figure 3, page 9).
- `Usage View (Numerical)`, providing the actual resource usage values corresponding to time intervals set by the time line calipers (see Figure 4, page 10).
- `Call Graph View`, displaying the selected function, with its metrics, in a graphical format that shows the functions that called it (callers) and the functions it called (callees) (see Figure 6, page 12).
- `Butterfly View`, showing the selected function along with the callers and callees (see Figure 7, page 13).
- `Call Stack`, displaying the contents of the call stack at the selected event or I/O trace (see Figure 11, page 19).
- `Malloc Error View`, displaying each `malloc` error (bad memory frees) that occurred in the experiment, the number of times the `malloc` occurred (a count is kept of calls to `malloc` that occurred with identical call stacks), and the call stack corresponding to the selected `malloc` error.
- `Leak View`, displaying each memory leak that occurred in your experiment, its size, the number of times the leak occurred at that location during the experiment, and the call stack corresponding to the selected leak.
- `Malloc View`, displaying each `malloc` (whether or not it caused a problem) that occurred in your experiment, its size, the number of times the `malloc` occurred (a count is kept of `malloc` calls with identical call stacks), and the call stack corresponding to the selected `malloc`.
- `Heap View`, displaying a map of memory indicating how blocks of memory were used in the time interval set by the time line calipers (see Figure 10, page 18).
- `I/O View`, displaying a chart devoted to I/O system calls. It identifies up to 10 files involved in I/O (see Figure 12, page 20).
- `Working Set View`, measuring the coverage of your executable and the dynamic shared objects (DSOs) that make up your executable. It indicates instructions, functions, and pages that were not used when the experiment was run (see Figure 13, page 21).

- Cord analyzer, accessed from `sscord`(1), works in conjunction with the `Working Set View` to let you try out different working set configurations to improve performance (see Figure 14, page 22).
- `Source View` with performance annotations, displaying performance metrics adjacent to the corresponding line of source code (see Figure 8, page 15) .
- `Disassembled Source` view with performance annotations, displaying the performance metrics adjacent to the corresponding machine code. For the `Ideal Time/Pixie` experiment, the `Disassembled Source` view can show where and why the processor may have stalled during the execution of an instruction.

1.3 Sources of Performance Problems

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

- CPU speed and availability
- I/O processing
- Memory size and availability
- Bugs
- Instruction cache and data cache sizes
- Any of the above in different phases

The following sections describe these sources of performance problems in more detail.

1.3.1 CPU-Bound Processes

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.

1.3.2 I/O-Bound Processes

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.

1.3.3 Memory-Bound Processes

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. 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.

1.3.4 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`).

1.3.5 Performance Phases in Programs

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.

1.4 Interpreting Performance Analyzer Results

Before we discuss the mechanics of using the Performance Analyzer, let's look at these features that help you understand the behavior of your processes:

- The time line display shows the experiment as a set of events over time and provides calipers to allow the user to specify an interval of interest. See the following section, and for more information, see Section 4.4.4, page 56.
- `The Usage View (Graphs)` displays process resource usage data in the form of strip charts and event charts. See Section 1.4.2, page 7.
- `The Usage View (Numerical)` presents a textual display of the process and system-wide resource usage data. See Section 1.4.3, page 8.
- The Function List Area displays the program's functions with associated performance metrics. See Section 1.4.4, page 10.
- The `Call Graph View` presents the target program as nodes and arcs, along with associated metrics. See Section 1.4.5, page 11.
- The `Butterfly View` presents a selected function along with the functions that called it and the functions that it called. See Section 1.4.6, page 12.
- `Source View` with performance annotations, see Section 1.4.7, page 14.
- `Disassembled Source` with performance annotations, see Section 1.4.8, page 15.
- `Malloc Error View`, `Leak View`, `Malloc View`, and `Heap View`, see Section 1.4.9, page 16.
- The `Call Stack View` shows the path through functions that led to an event. See Section 1.4.10, page 18.
- `I/O View` displays a chart of the number of bytes for each I/O transfer. See Section 1.4.11, page 19.
- The `Working Set View` displays a list of the DSOs in the program, with information on the efficiency of use of the text (instruction) pages. See Section 1.4.12, page 20.
- The cord analyzer lets you explore the working set behavior of an executable or DOS. See Section 1.4.13, page 21.

The following sections describe these features in more detail.

1.4.1 The Time Line Display

Have you ever considered timing a program with a stopwatch? The Performance Analyzer time line serves the same function. The time line shows where each sample event in the experiment occurred. By setting sample traps at phase boundaries, you can analyze metrics on a phase-by-phase basis. The simplest metric, time, is easily recognized as the space between events. The triangular icons are calipers; they let you set the scope of analysis to the interval between the selected events.

Figure 2, page 7, shows the time line portion of the Performance Analyzer window with typical results. Event number 4 is selected; it is labeled according to the caliper number, `third`. You can see from the graph that the phase between the selected event and event number 5 is taking more of the program's time than any of the other phases.

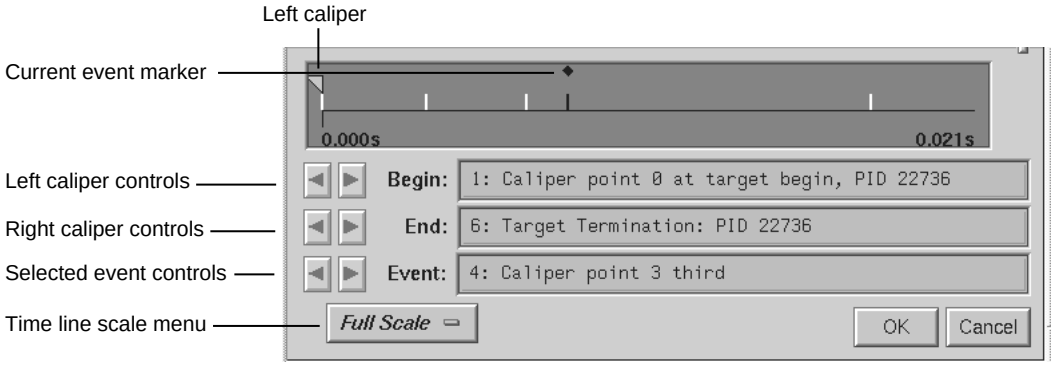

Figure 2. Typical Performance Analyzer Time Line

1.4.2 Resource Usage Graphs

The Performance Analyzer lets you look at how different resources are consumed over time. It produces a number of resource usage graphs that are tied to the time line (see Figure 3, page 9, which shows six of the graphs available). These resource usage graphs indicate trends and let you pinpoint problems within phases.

Resource usage data refers to items that consume system resources. They include

- User and system time

- Page faults
- Context switches
- The size of reads and writes
- Read and write counts
- Poll and I/O calls
- Total system calls
- Process signals received
- Process size

Resource usage data is not always recorded (written to file) at each sample point. If you discover inconsistent behavior within a phase, you can set new sample points and break the phase down into smaller phases.

You can analyze resource usage trends in the charts in `Usage View (Graphs)` and can view the numerical values in the `Usage View (Numerical)` window.

1.4.3 Usage View (Numerical)

The usage graphs show the patterns; the textual usage views let you view the aggregate values for the interval specified by the time line calipers. Figure 4, page 10, shows a typical `Usage View (Numerical)` window.

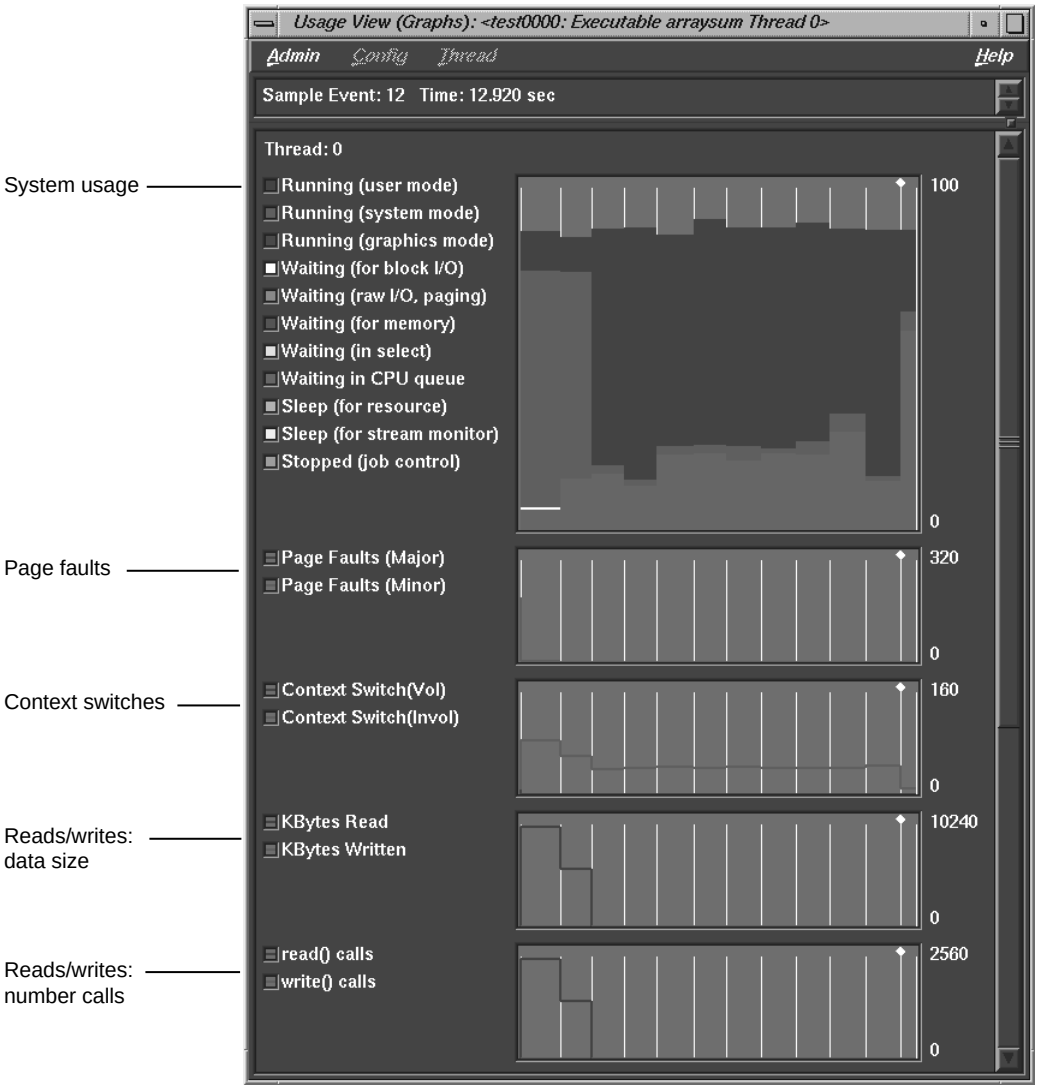

Figure 3. Typical Resource Usage Graphs

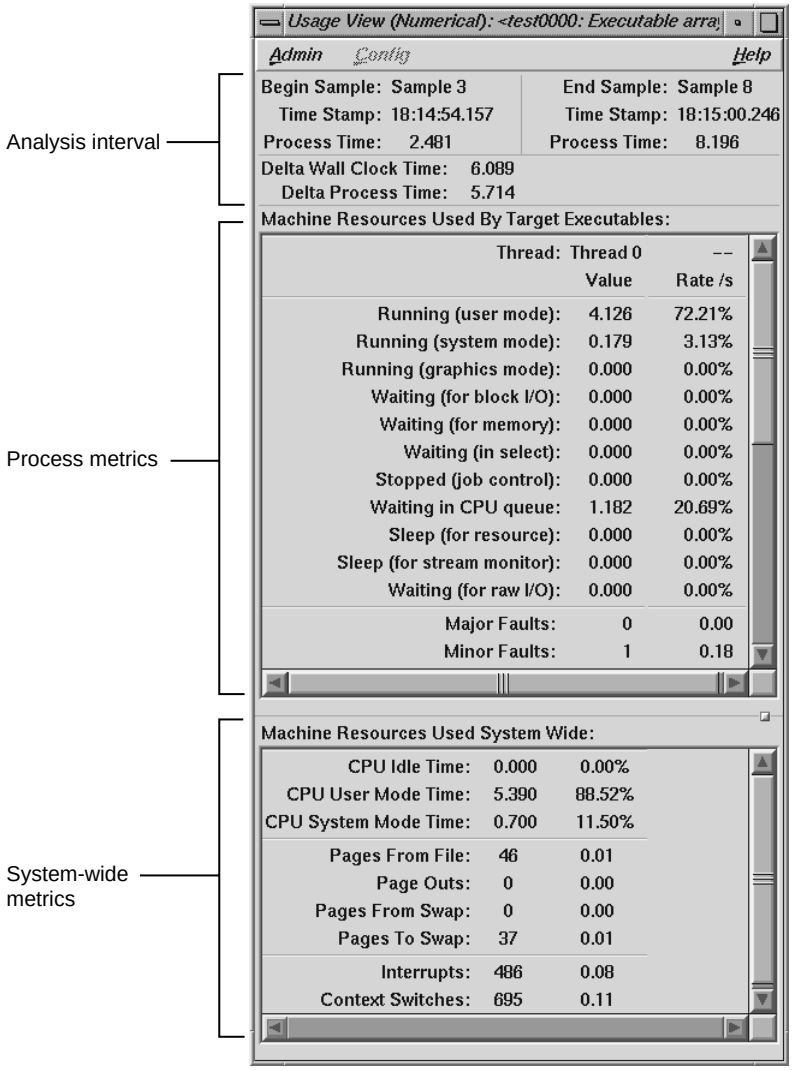

Figure 4. Typical Textual Usage View

1.4.4 The Function List Area

The function list displays all functions in the source code, annotated by performance metrics and ranked by the criterion of your choice, such as counts or one of the time metrics. Figure 5, page 11, shows an example of the function list, ranked by inclusive CPU time.

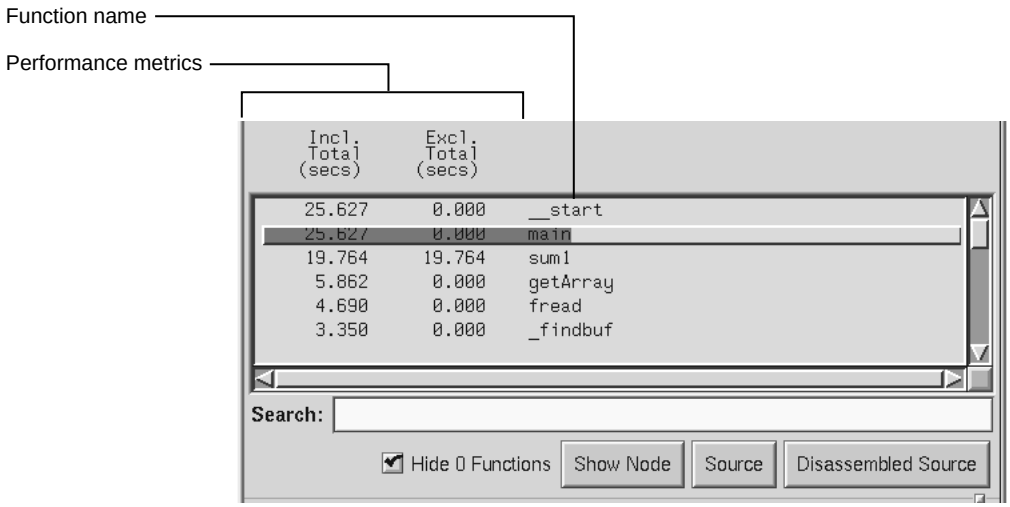

Figure 5. Typical Performance Analyzer Function List Area

You can configure how functions appear in the function list area by selecting `Preferences...` in the `Config` menu. It lets you select which performance metrics display, whether they display as percentages or absolute values, and the style of the function name. The `Sort...` selection in the `Config` menu lets you order the functions in the list by the selected metric. Both selections disable those metric selections that were not collected in the current experiment.

1.4.5 Call Graph View

In contrast to the function list, which provides the performance metrics for functions, the call graph puts this information into context by showing you the relationship between functions. The call graph displays functions as nodes and calls as arcs (displayed as lines between the nodes). The nodes are annotated with the performance metrics; the arcs come with counts by default and can include other metrics as well.

In Figure 6, page 12, for example, the inclusive time spent by the function `main` is 8.107 seconds. Its exclusive time was 0 seconds, meaning that the time was actually spent in called functions. The `main` function can potentially call three functions. The `Call Graph View` indicates that in the experiment, `main` called three functions: `getArray`, which consumed 1.972 seconds; `sum1`, which consumed 3.287 seconds; and `sum2`, which consumed 2.848 seconds.

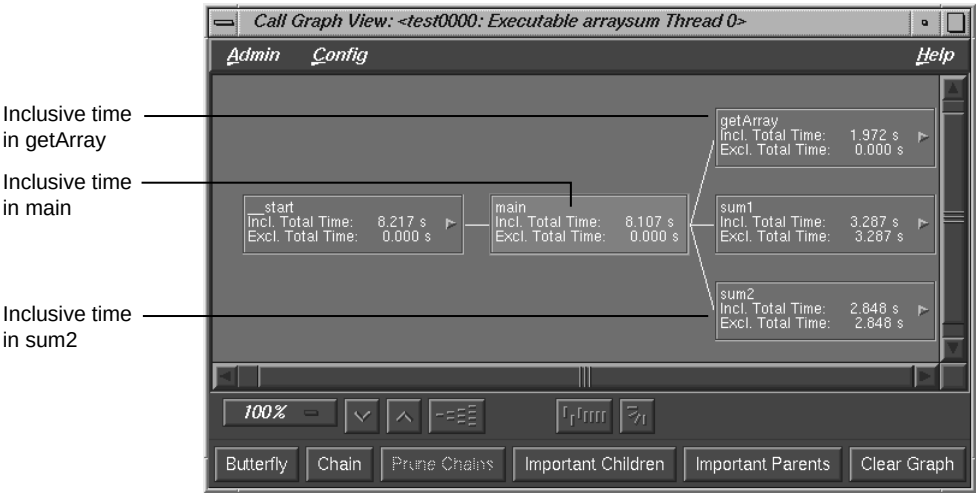

Figure 6. Typical Performance Analyzer Call Graph

1.4.6 Butterfly View

The `Butterfly View` shows a selected routine in the context of functions that called it and functions it called. For an illustration, see Figure 7, page 13.

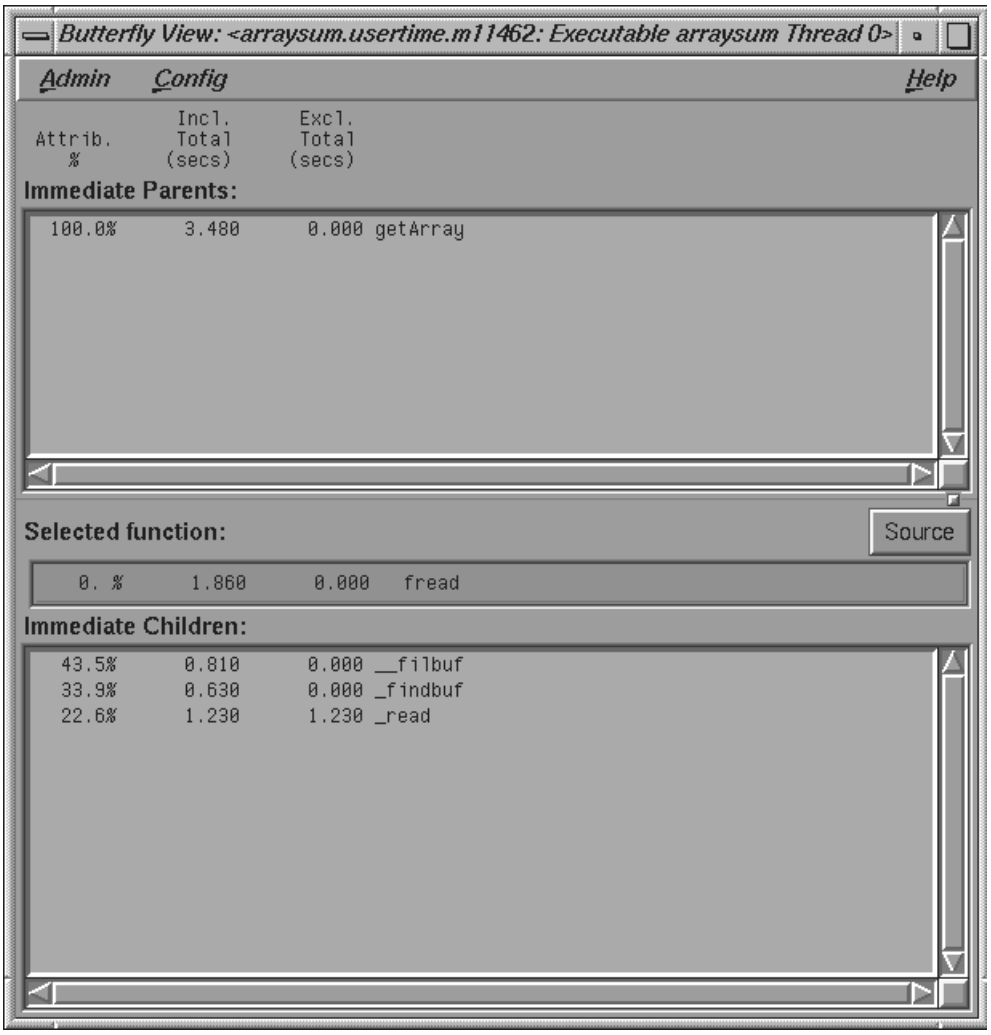

Figure 7. Butterfly View

Select a function to be analyzed by clicking on it in the function list area of the main Performance Analyzer window. The `Butterfly View` window displays the function you click on as the selected function.

The two main parts of the `Butterfly View` window identify the immediate parents and the immediate children of the selected function. In this case, the

term *immediate* means they either call the selected function directly or are called by it directly.

The columns of data in the illustration show:

- The percentage of the sort key (inclusive time, in the illustration) attributed to each caller or callee.
- The time the function and any functions it called required to execute.
- The time the function alone (excluding other functions it called) required to execute.

You can also display the address from which each function was called by selecting the `Show All Arcs Individually` from the `Config` menu.

1.4.7 Source View with Performance Annotations

The Performance Analyzer lets you view performance metrics by source line in the `Source View` (see Figure 8, page 15) or by machine instruction in the `Disassembled Source` view. Displaying performance metrics is set in the `Preferences` dialog box, accessed from the `Display` menu in the `Source View` and `Disassembled Source` view. The Performance Analyzer sets thresholds to flag lines that consume more than 90% of a total resource. These indicators appear in the metrics column and on the scroll bar.

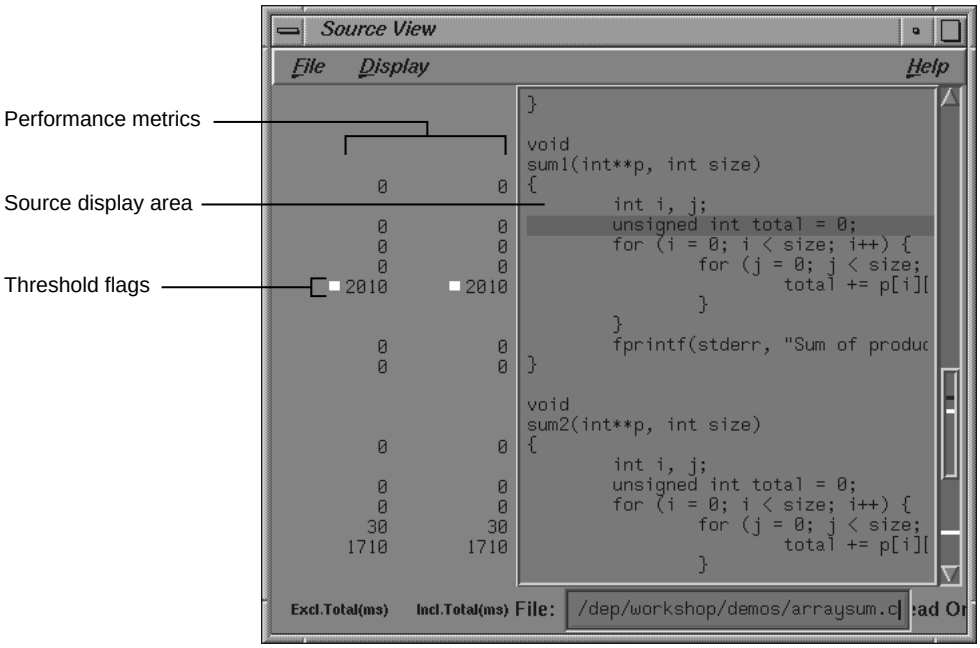

Figure 8. Detailed Performance Metrics by Source Line

1.4.8 Disassembled Code with Performance Annotations

The Performance Analyzer also lets you view performance metrics by machine instruction (see Figure 9, page 16). You can view any of the performance metrics that were measured in your experiment. If you ran an `Ideal Time/Pixie` experiment, you can get a special three-part annotation that provides information about stalled instructions.

The yellow bar spanning the top of three columns in this annotation indicates the first instruction in each basic block. The first column labeled `Clock` in the annotation displays the clock number in which the instruction issues relative to the start of a basic block. If you see clock numbers replaced by quotation marks (“), it means that multiple instructions were issued in the same cycle. The column labeled `Stall` shows how many clocks elapsed during the stall before the instruction was issued. The column labeled `Why` shows the reason for the stall. There are three possibilities:

- `B` - Branch delay
- `F` - Function unit delay

- `O` - Operand has not arrived yet

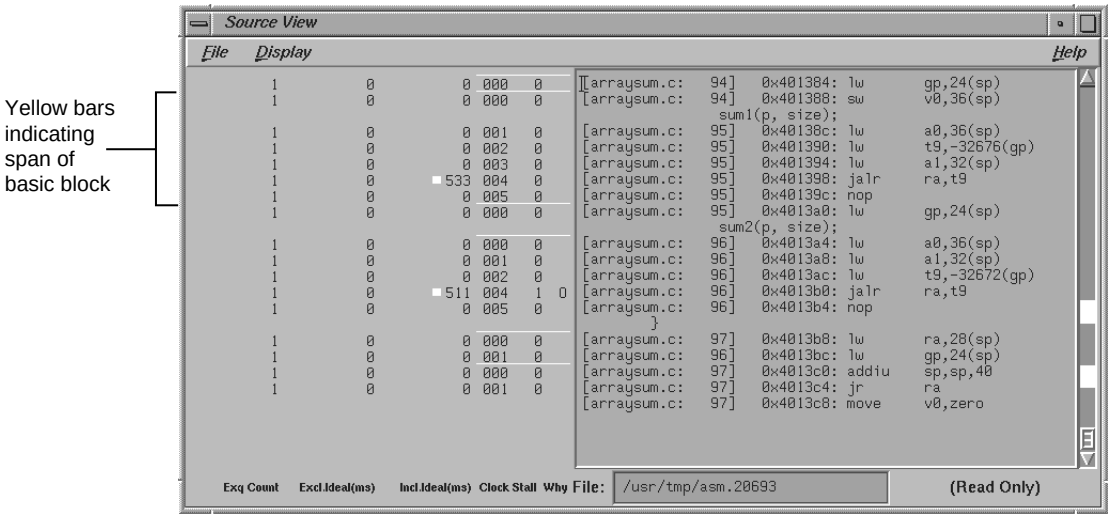

Figure 9. Disassembled Code with Stalled Clock Annotations

1.4.9 Malloc Error View, Leak View, Malloc View, and Heap View

The Performance Analyzer lets you look for memory problems. The `Malloc Error View`, `Leak View`, `Malloc View`, and `Heap View` windows address two common types of memory problems that can inhibit performance:

- Memory leakage, see Section 1.4.9.1, page 17
- Bad calls to `free`, see Section 1.4.9.2, page 17.

The difference between these windows lies in the set of data that they collect. `Malloc Error View` displays all `malloc` errors. When you run a memory leak experiment and problems are found, a dialog box displays suggesting you use `Malloc Error View` to see the problems. `Leak View` shows memory leak errors only. `Malloc View` shows each `malloc` operation whether faulty or not. `Heap View` displays a map of heap memory that indicates where both problems and normal memory allocations occur and can tie allocations to memory addresses. The first two views are better for focusing on problems; the latter two views show the big picture.

1.4.9.1 Memory Leakage

Memory leakage occurs when a program dynamically allocates memory and fails to deallocate that memory when it is through using the space. This causes the program size to increase continuously as the process runs. A simple indicator of this condition is the Total Size strip chart on the `Usage View (Graphs)` window. The strip chart only indicates the size; it does not show the reasons for an increase.

`Leak View` displays each memory leak in the executable, its size, the number of times the leak occurred at that location, and the corresponding call stack (when you select the leak), and is thus the most appropriate view for focusing on memory leaks.

A region allocated but not freed is not necessarily a leak. If the calipers are not set to cover the entire experiment, the allocated region may still be in use later in the experiment. In fact, even when the calipers cover the entire experiment, it is not necessarily wrong if the program does not explicitly free memory before exiting, since all memory is freed anyway on program termination.

The best way to look for leaks is to set sample points to bracket a specific operation that should have no effect on allocated memory. Then any area that is allocated but not freed is a leak.

1.4.9.2 Bad Frees

A bad `free` (also referred to as an anti-leak condition) occurs when a program frees some structure that it had already freed. In many such cases, a subsequent reference picks up a meaningless pointer, causing a segmentation violation. Bad calls to `free` are indicated in both `Malloc Error View` and in `Heap View`. `Heap View` identifies redundant calls to `free` in its memory map display. It helps you find the address of the freed structure, search for the `malloc` event that created it, and find the `free` event that released it. Hopefully, you can determine why it was prematurely freed or why a pointer to it was referenced after it had been freed.

`Heap View` also identifies unmatched calls to `free` in an information window. An unmatched `free` is a `free` that does not have a corresponding allocation in the same interval. As with leaks, the caliper settings may cause false indications. An unmatched `free` that occurs in any region not starting at the beginning of the experiment may not be an error. The region may have been allocated before the current interval and the unmatched `free` in the current interval may not be a problem after all. A segment identified as a bad `free` is definitely a problem; it has been freed more than once in the same interval.

A search facility is provided in `Heap View` that allows the user to find the allocation and deallocation events for all blocks containing a particular virtual address.

The `Heap View` window lets you analyze memory allocation and deallocation between selected sample events in your experiment. `Heap View` displays a memory map that indicates calls to `malloc` and `realloc`, bad deallocations, and valid deallocations during the selected period, as shown in Figure 10, page 18. Clicking an area in the memory map displays the address.

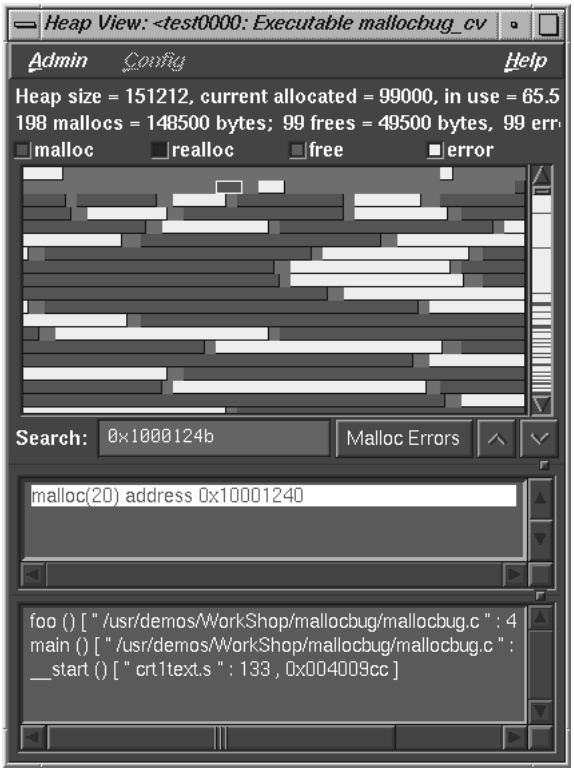

Figure 10. Typical Heap View Display Area

1.4.10 Call Stack View

The Performance Analyzer allows you to recall call stacks at sample events, which helps you reconstruct the calls leading up to an event so that you can

relate the event back to your code. Figure 11, page 19, shows a typical call stack. It corresponds to sample event #3 in an experiment.

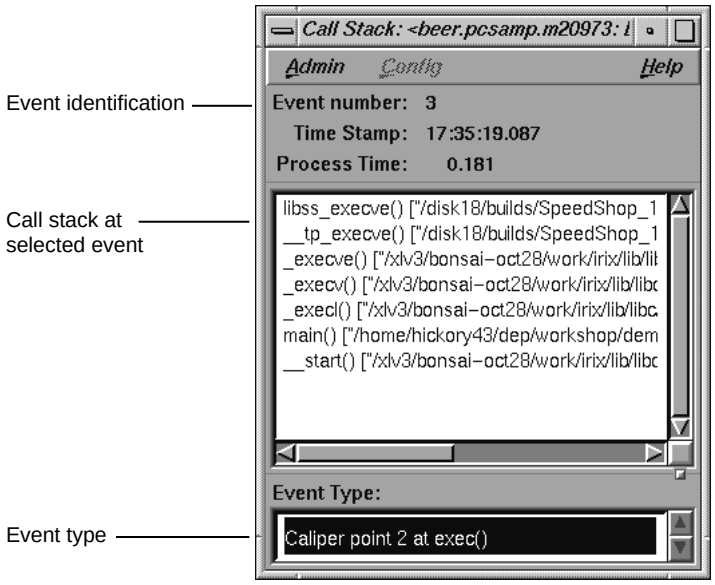

Figure 11. Typical Call Stack

1.4.11 I/O View

`I/O View` helps you determine the problems in an I/O-bound process. It produces a graph of all I/O system calls and identifies up to 10 files involved in I/O. By selecting an event with the left mouse button, you can display the call stack corresponding to the event in the `Call Stack View`. See Figure 12.

Figure 12. I/O View

1.4.12 Working Set View

`Working Set View` measures the coverage of the dynamic shared objects (DSOs) that make up your executable (see Figure 13). It indicates instructions, functions, and pages that were not used when the experiment was run. It shows the coverage results for each DSO in the DSO list area. Clicking a DSO in the list displays its pages with color coding to indicate the coverage of the page.

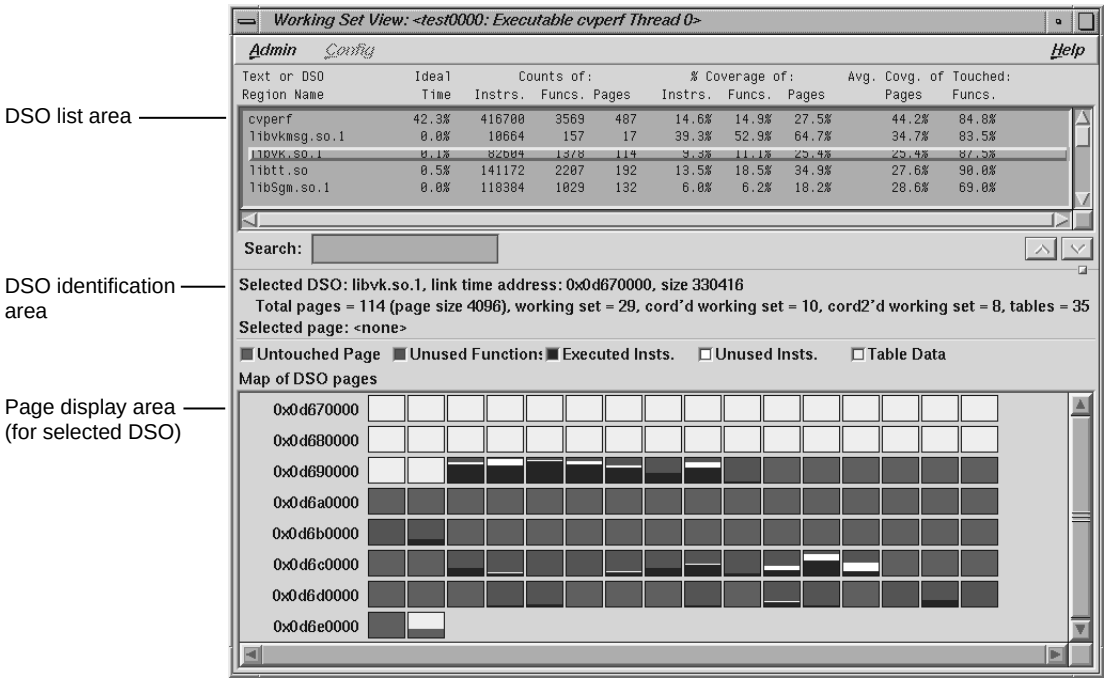

Figure 13. Working Set View

1.4.13 Cord Analyzer

The cord analyzer is not actually part of the Performance Analyzer and is invoked by typing `sscord` at the command line. The cord analyzer (see Figure 14, page 22) lets you explore the working set behavior of an executable or dynamic shared library (DSO). With it you can construct a feedback file for input to `cord` to generate an executable with improved working-set behavior.

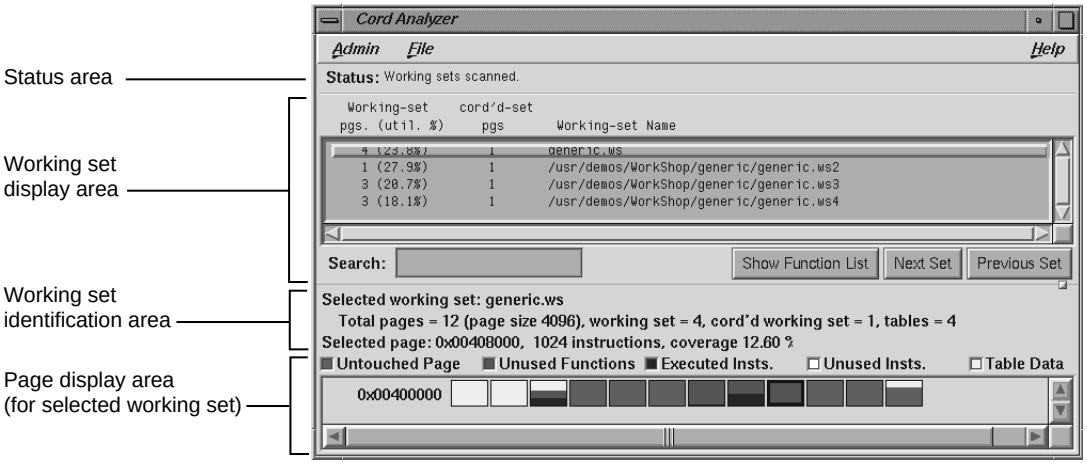

Figure 14. Cord Analyzer

Performance Analyzer Tutorial [2]

This chapter presents a tutorial for using the Performance Analyzer and covers these topics:

- Tutorial Overview, see Section 2.1, page 23
- Tutorial Setup, see Section 2.2, page 23
- Analyzing the Performance Data, see Section 2.3, page 25

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

2.1 Tutorial Overview

This tutorial is based on a sample program called `arraysum`. The `arraysum` program goes through the following steps:

1. Defines the size of an array (2,000 by 2,000).
2. Creates a 2,000-by-2,000 element array, gets the size of the array, and reads in the elements.
3. Calculates the array total by adding up elements in each column.
4. Recalculates the array total differently, by adding up elements in each row.

It is more efficient to add the elements in an array row-by-row, as in step 4, than column-by-column, as in step 3. Because the elements in an array are stored sequentially by rows, adding the elements by columns potentially causes page faults and cache misses. The tutorial shows you how you can detect symptoms of problems like this and then zero in on the problem. The source code is located in `/usr/demos/WorkShop/performance` if you want to examine it.

2.2 Tutorial Setup

You need to compile the program first so that you can use it in the tutorial.

1. Change to the `/usr/demos/WorkShop/performance` directory.

You can run the experiment in this directory or set up your own directory.

2. Compile the `arraysum.c` file by entering the following:

   `make arraysum`

   This will provide you with an executable for the experiment, if one does not already exist.

3. From the command line, enter the following:

   `cvd arraysum&`

   The `Debugger Main View` window is displayed. You need the Debugger to specify the data to be collected and run the experiment.

4. Choose `User Time/Callstack Sampling` from the `Select Task` submenu in the `Perf` menu.

   This is a performance task that will return the time your program is actually running and the time the operating system spends performing services such as I/O and executing system calls. It includes the time spent in each function.

5. If you want to watch the progress of the experiment, choose `Execution View` in the `Views` menu. Then click `Run` in the `Debugger Main View` window.

   This starts the experiment. When the status line indicates that the process has terminated, the experiment has completed. The main Performance Analyzer window is displayed automatically. The experiment may take 1 to 3 minutes, depending on your system. The output file will appear in a newly created directory, named `test0000`.

You can also generate an experiment using the `ssrun`(1) command with the `-workshop` option, naming the output file on the `cvperf`(1) command. In the following example, the output file from `ssrun` is `arraysum.usertime.m2344`.

```
% ssrun -workshop -usertime arraysum
% cvperf arraysum.usertime.m2344
```

If you are analyzing your experiment on the same machine you generate it, you do not need the `-workshop` option. If the `_SPEEDSHOP_OUTPUT_FILENAME` is set to a file name, such as `my_prog`, the experiment file from the example above would be `my_prog.m2345`. See the `ssrun`(1) or the *Speedshop User's Guide* for more SpeedShop environment variables.

2.3 Analyzing the Performance Data

Performance analysis experiments are set up and run in the Debugger window; the data is analyzed in the main Performance Analyzer window. The Performance Analyzer can display any data generated by the `ssrun`(1) command, by any of the Debugger window performance tasks (which use the `ssrun`(1) command), or by `pixie`(1).

Note: Again, the timings and displays shown in this tutorial could be quite different from those on your system. For example, setting caliper points in the time line may not give you the same results as those shown in the tutorial, because the program will probably run at a different speed on your system.

1. Examine the main Performance Analyzer window, which is invoked automatically if you created your experiment file from the `cvd` window.

   The Performance Analyzer window now displays the information from the new experiment (see Figure 15, page 26).

2. Look at the usage chart in the Performance Analyzer window.

   The first phase is I/O-intensive. The second phase, during which the calculations took place, shows high user time.

3. Select `Usage View (Graphs)` from the `Views` menu.

   The `Usage View (Graphs)` window displays as in Figure 16, page 27. It shows high read activity and high system calls in the first phase, confirming our hypothesis that it is I/O-intensive.

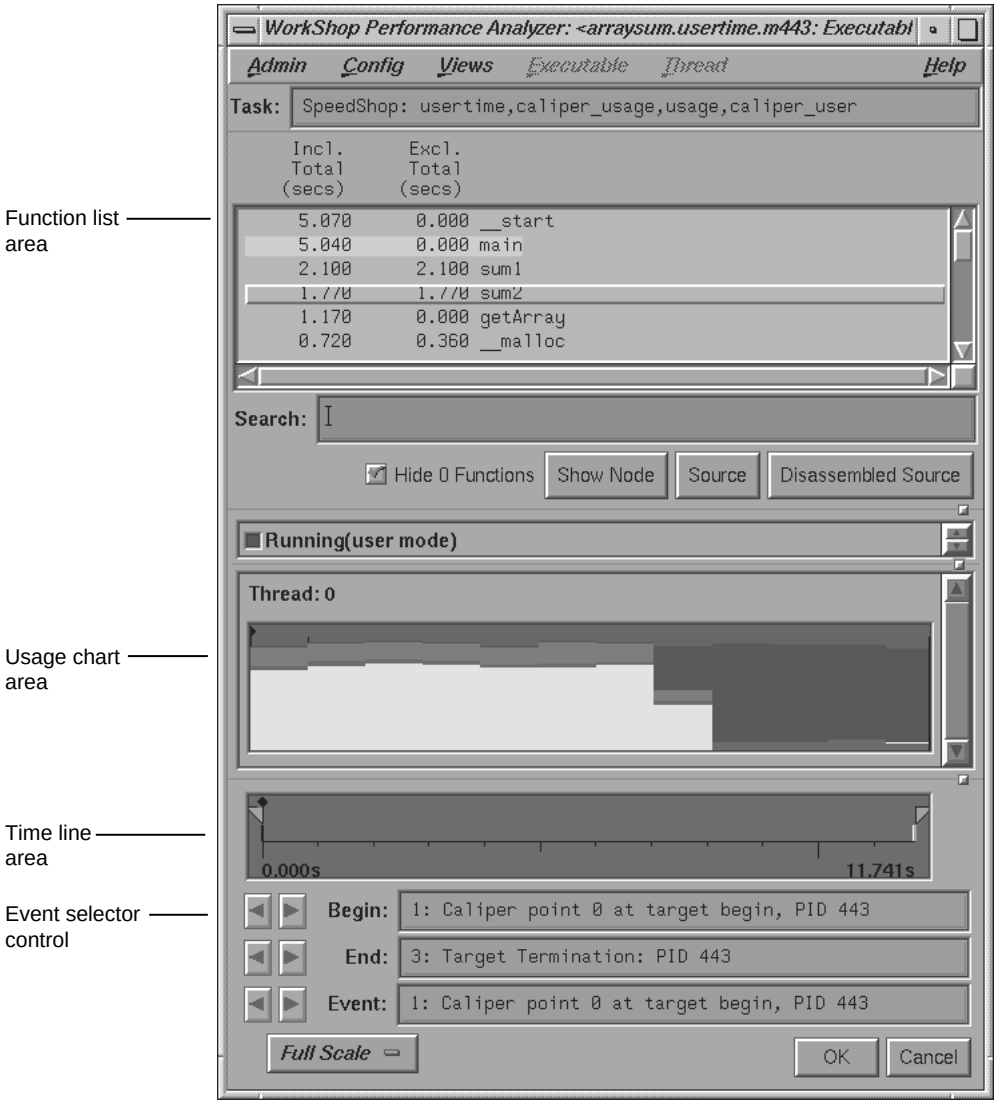

Figure 15. Performance Analyzer Main Window—`arraysum` Experiment

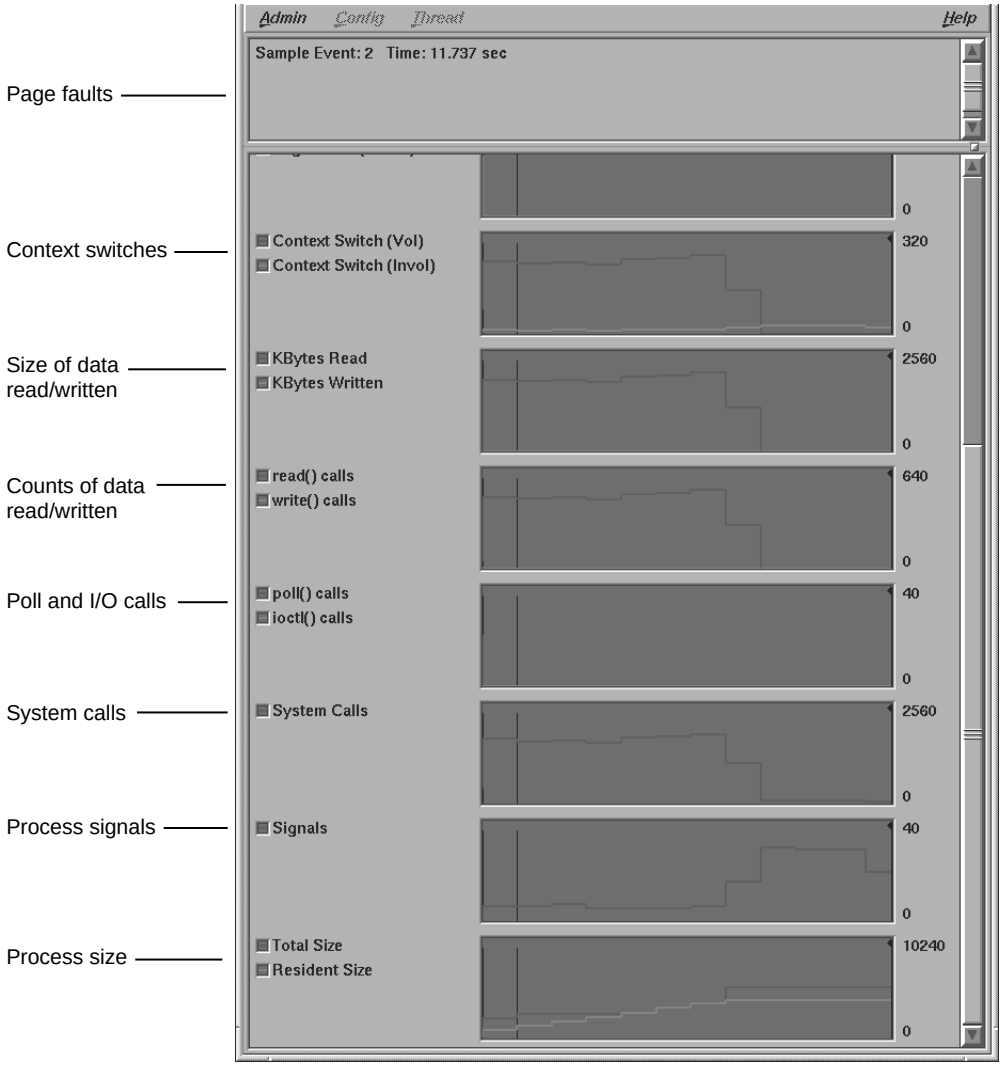

Figure 16. `Usage View (Graphs)`—`arraysum` Experiment

As a side note, scroll down to the last chart, which indicates that the maximum total size of the process is reached at the end of the first phase and does not grow thereafter.

4. Select `Call Stack View` from the `Views` menu.

The call stack displays for the selected event. An event refers to a sample point on the time line (or any usage chart).

At this point, no events have been selected so the call stack is empty. To define events, you can add calls to `ssrt_caliper_point` to record caliper points in the source file, set a sample trap from the WorkShop Debugger window, or set pollpoint calipers on the time line. (For more information on the `ssrt_caliber_point` function, see the `ssapi`(3) man page.) See Figure 17, page 28, for an illustration of how the `Call Stack View` responds when various caliper points are recorded.

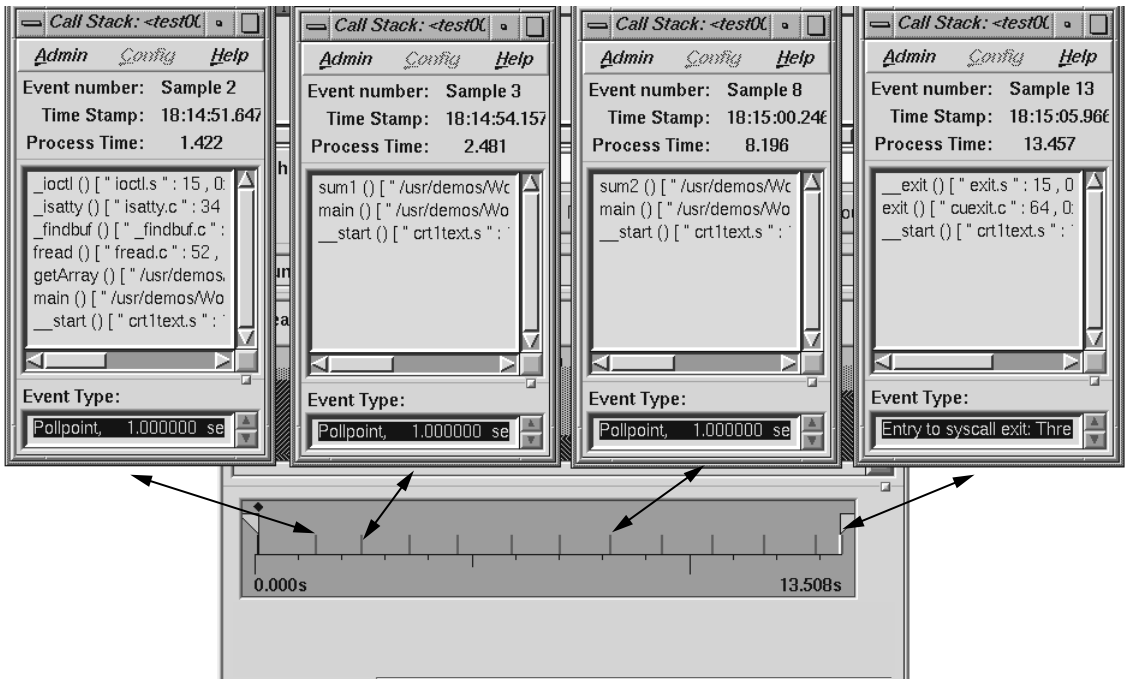

Figure 17. Significant Call Stacks in the `arraysum` Experiment

5. Return to the Performance Analyzer window and pull down the sash to expose the complete function list.

   This shows the inclusive time (that is, time spent in the function and its called functions) and exclusive time (time in the function itself only) for each function. As you can see, more time is spent in `sum1` than in `sum2`.

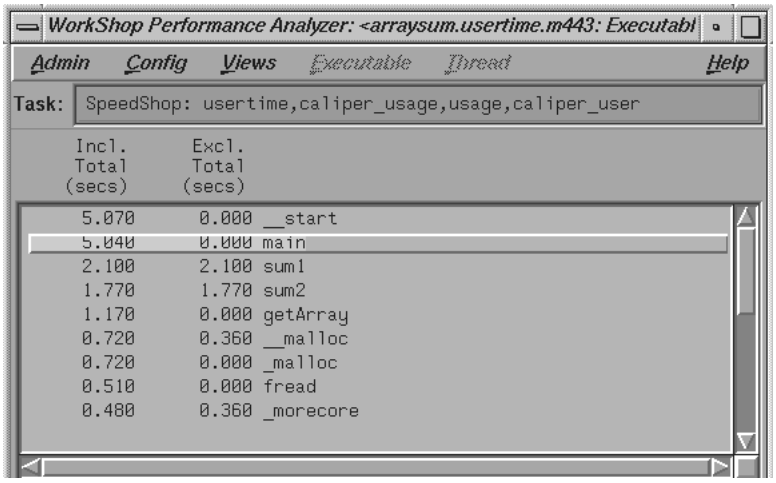

Figure 18. Function List Portion of Performance Analyzer Window

6. Select `Call Graph` from the `Views` menu and click on the `Butterfly` button.

   The call graph provides an alternate means of viewing function performance data. It also shows relationships, that is, which functions call which functions. After the `Butterfly` button is clicked, the `Call Graph View` window appears, as shown in Figure 19, page 30. The `Butterfly` button takes the selected function (or most active function if none is selected) and displays it with the functions that call it and those that it calls.

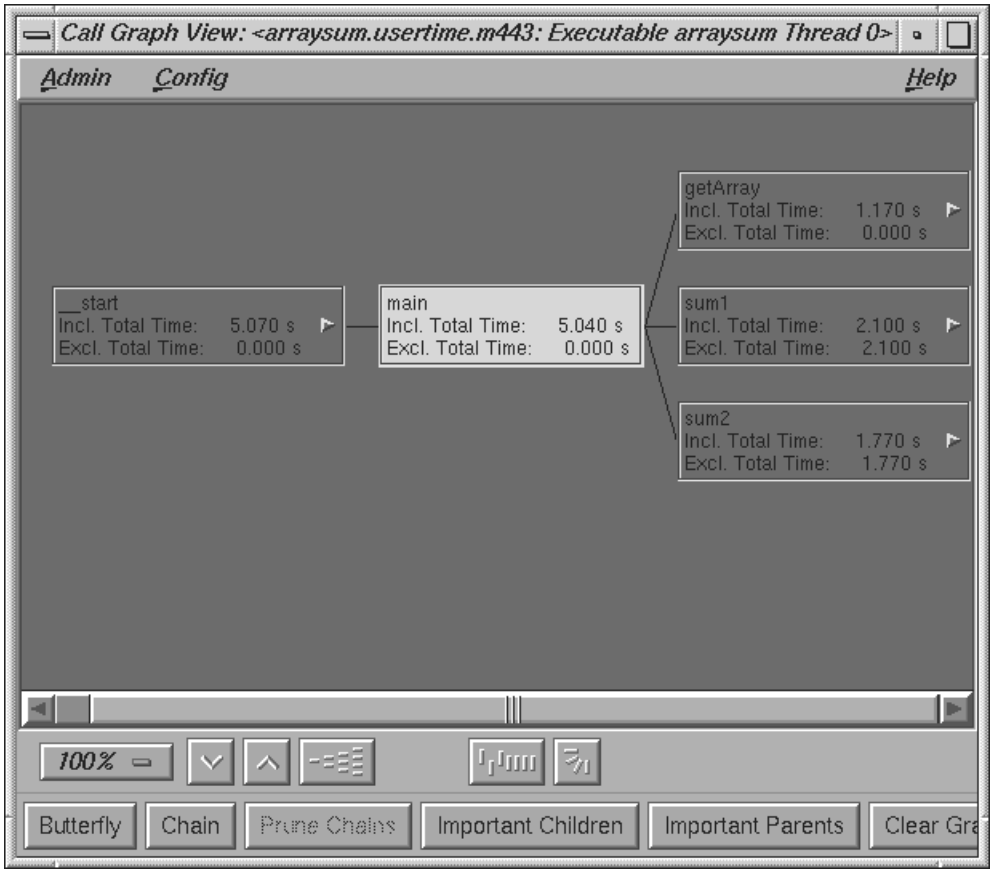

Figure 19. `Call Graph View`—`arraysum` Experiment

7. Select `Close` from the `Admin` menu in the `Call Graph View` window to close it. Return to the main Performance Analyzer window.
8. Select `Usage View (Numerical)` from the `Views` menu.

   The `Usage View (Numerical)` window appears as shown in Figure 20, page 31.

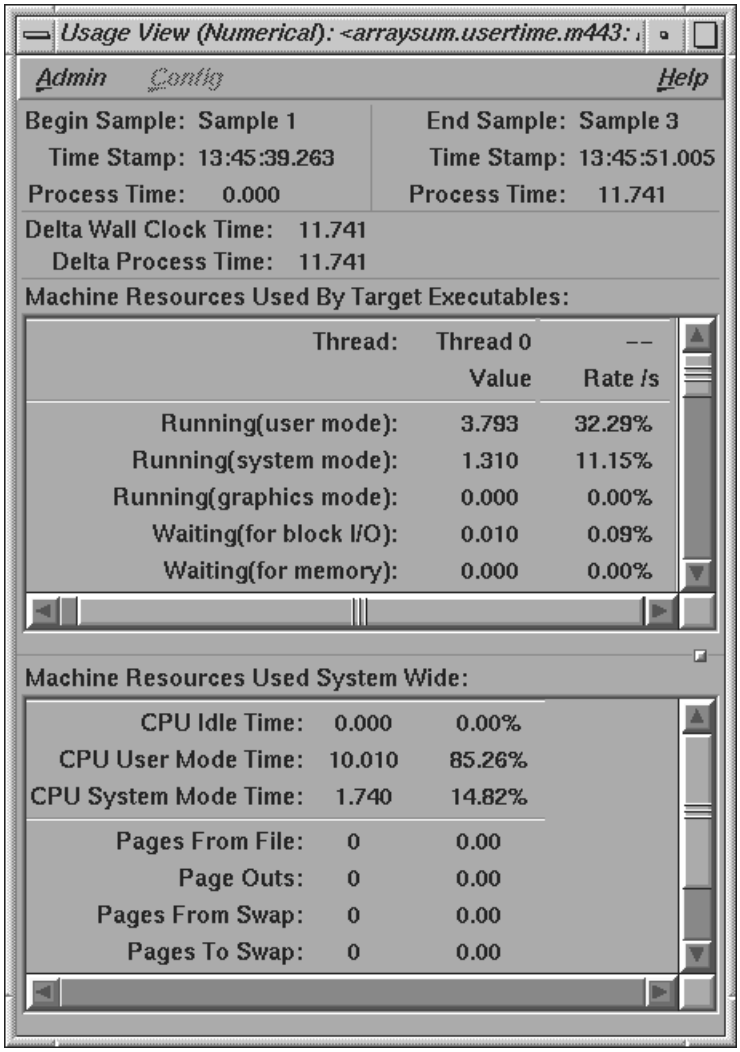

Figure 20. Viewing a program in the `Usage View (Numerical)` window

9. Return to the main Performance Analyzer window, select `sum1` from the function list, and click `Source`.

   The `Source View` window displays as shown in Figure 21, page 32, scrolled to `sum1`, the selected function. The annotation column to the left of

the display area shows the performance metrics by line. Lines consuming more than 90% of a particular resource appear with highlighted annotations.

Notice that the line where the total is computed in `sum1` is seen to be the culprit, consuming 2,100 milliseconds. As in the other WorkShop tools, you can make corrections in `Source View`, recompile, and try out your changes.

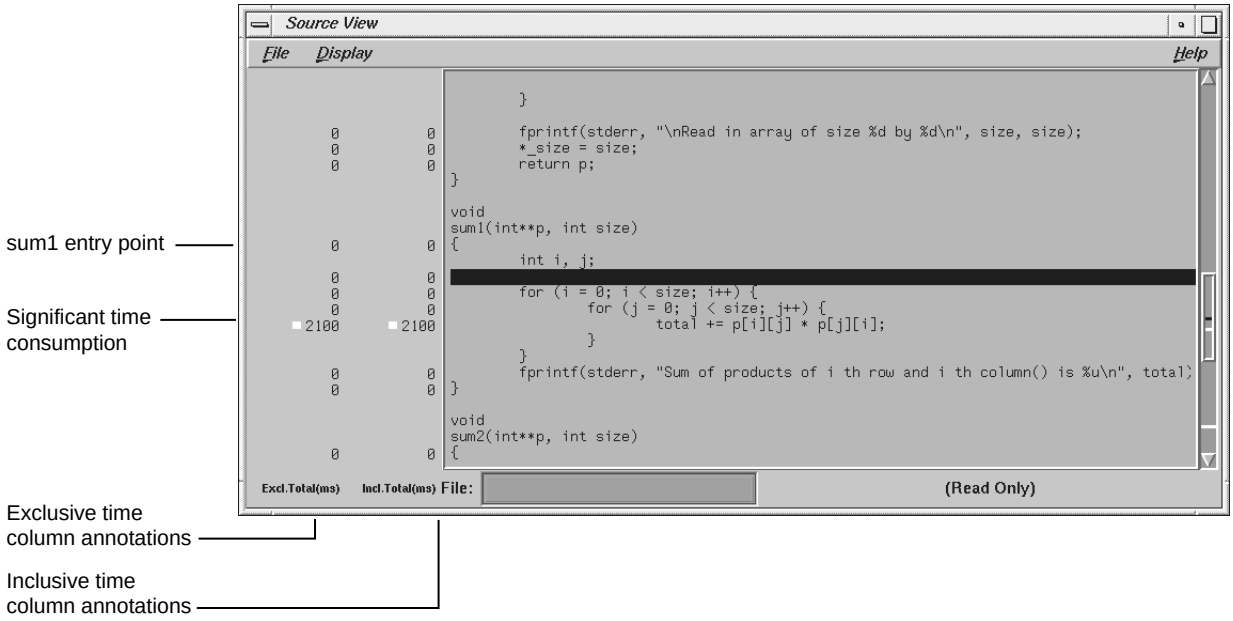

Figure 21. `Source View` with Performance Metrics—`arraysum` Experiment

At this point, we have uncovered one performance problem: the `sum1` algorithm is inefficient. As a side exercise, you may want to take a look at the performance metrics at the assembly level. To do this, return to the main Performance Analyzer window, select `sum1` from the function list, and click `Disassembled Source`. The disassembly view displays the assembly language version of the program with the performance metrics in the annotation column.

10. Close any windows that are still open.

This concludes the tutorial.

Setting Up Performance Analysis Experiments [3]

In performance analysis, you set up the experiment, run the executable, and analyze the results. To make setup easier, the Performance Analyzer provides predefined tasks that help you establish an objective and ensure that the appropriate performance data will be collected. This chapter tells you how to conduct performance tasks and what to look for.

It covers these topics:

3.1 Experiment Setup Overview

Performance tuning typically consists of examining machine resource usage, breaking down the process into phases, identifying the resource bottleneck within each phase, and correcting the cause. Generally, you run the first experiment to break your program down into phases and run subsequent experiments to examine each phase individually. After you have solved a problem in a phase, you should then reexamine machine resource usage to see if there is further opportunity for performance improvement.

Each experiment has these steps:

1. Specify the performance task.

   The Performance Analyzer provides *predefined tasks* for conducting experiments. When you select a task, the Performance Analyzer automatically enables the appropriate performance data items for collection.

   The predefined tasks ensure that only the appropriate data collection is enabled. Selecting too much data can bog down the experiment and skew the data for collection. If you need a mix of performance data not available in the predefined tasks, you can select `Custom` from the `Select Task` submenu. It lets you enable combinations of the data collection options.

2. Specify where to capture the data.

   If you want to gather information for the complete program, this step is not needed. If you want data at specific points in the process, you need to set sample traps. See Section 3.3, page 35, for a brief description of traps or Chapter 4, "Setting Traps," in the *Developer Magic: Debugger User's Guide* for an in-depth discussion.

   The Performance Analyzer records samples at the beginning and end of the process automatically. If you want to analyze data within phases, set sample traps at the beginning of each phase and at intermediate points.

3. Specify the experiment configuration parameters.

   This step is not necessary if you use the defaults; if you want to make configuration changes, select `Configs` from the `Perf` menu. The dialog box lets you specify a number of configuration options, many of which depend on the experiment you plan to run. The dialog box in Figure 23, page 51, shows the runtime configuration choices, and the options are described in Section 4.3, page 50.

4. Run the program to collect the data.

   You run the experiment from the WorkShop Debugger window. If you are running a small experiment to capture resource usage, you may be able to watch the experiment in real time in the Process Meter. SpeedShop stores the results in the designated experiment subdirectory.

5. Analyze the results.

   After the experiment completes, you can look at the results in the Performance Analyzer window and its associated views. Use the calipers to get information for phases separately.

3.2 Selecting a Performance Task

To set up a Performance Analyzer experiment, choose a task from the `Select Task` submenu in the `Perf` menu in the Debugger Main View (see Figure 22, page 35).

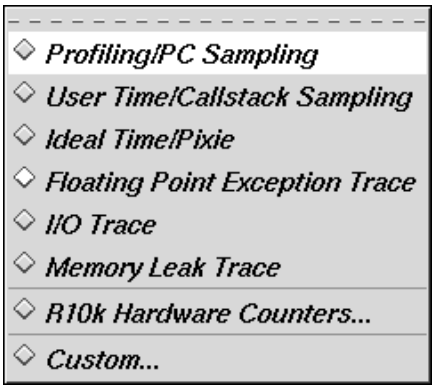

Figure 22. `Select Task` Submenu

Selecting a task enables data collection. The mode indicator in the upper right corner of the Main View changes from `Debug Only` to `Performance`.

3.3 Setting Sample Traps

Sample traps allow you to record data when a specified condition occurs. You set them from the debugger main view, Trap Manager, or Source View. For a thorough discussion of setting traps, see Chapter 4, “Setting Traps,” in the *Developer Magic: Debugger User’s Guide*.

> **Note:** In order for trap-based caliper points to work, you must activate the `Attach Debugger` toggle on the `Runtime` tab window. That window is available from the `Configs...` menu item on the `Perf` menu of the debugger window.

You can define sample traps:

- At function entry or exit points
- At source lines
- For events
- Conditionally
- Manually during an experiment

Sample traps at function entry and exit points are preferable to source line traps, because they are more likely to be preserved as your program evolves. This better enables you to save a set of traps in the Trap Manager in a file for subsequent reuse.

Manual sample traps are triggered when you click the `Sample` button in the Debugger Main View. They are particularly useful for applications with graphical user interfaces. If you have a suspect operation in an experiment, a good technique is to take a manual sample before and after you perform the operation. You can then examine the data for that operation.

3.4 Understanding Predefined Tasks

If you are unfamiliar with performance analysis, it is very easy to request more data collection than you actually need. Doing so can slow down the Performance Analyzer and skew results. To help you record data appropriate to your current objective, WorkShop provides predefined combinations of tasks, which are available in the `Select Task` submenu in the `Perf` menu (see Figure 22, page 35). These tasks are described in the following sections. When you select a task, the required data collection is automatically enabled.

3.4.1 `Profiling/PC Sampling`

Use the `Profiling/PC Sampling` task selection when you are identifying which parts of your program are using the most CPU time. PC profiling results in a statistical histogram of the program counter. The exclusive CPU time is presented as follows:

- By function in the function list
- By source line in `Source View`
- By instruction in `Disassembly View`
- Machine resource usage data at 1-second intervals and at sample points

3.4.2 `User Time/Callstack Sampling`

Use the `User Time/Callstack Sampling` task selection to tune a CPU-bound phase or program. It enables you to display the time spent in the CPU by function, source line, and instruction. This task records the following:

- The call stack every 3 milleseconds (ms)

- Machine resource usage data at 1-second intervals and at sample points

Data is measured by periodically sampling the call stack. The program's call stack data is used to do the following:

- 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).
- Attribute inclusive user time to all the functions above the one currently being executed.

The time spent in a procedure is determined by multiplying the number of times an instruction for that procedure appears in the stack by the average time interval between call stacks. Call stacks are gathered whether the program was running or blocked; hence, the time computed represents the total time, both within and outside the CPU. If the target process was blocked for a long time as a result of an instruction, that instruction will show up as having a high time.

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

3.4.3 Ideal Time/Pixie

Use the `Ideal Time/Pixie` task selection to tune a CPU-bound phase. This task provides exact counts with theoretical times. The analysis determines the cost on a per-basic block basis; it does not deal with data dependencies between basic blocks. It is very useful when used in conjunction with the `Profiling/PC Sampling` task. This approach lets you examine actual versus ideal time. The difference is the time spent as a result of the following:

- The `load` operations, which take a minimum of two cycles if the data is available in the cache and much longer if the data has to be accessed from the swap area, secondary cache, or main memory
- The `store` operations, which cause the CPU to stall if the write buffer in the CPU gets filled
- Time spent with the CPU stalled as a result of data dependencies between basis blocks

This task records the following:

- Basic block counts
- Counts of branches taken

- Machine resource usage data at 1-second intervals and at sample points
- Function pointer traces with counts

The following results can be displayed in the function list, the `Source View`, and the `Disassembly View`:

- The ideal time, which is the product of the number of the number of times each machine instruction executes, the cycle time of the machine, and the estimated number of cycles per execution.
- Execution counts.
- Resulting machine instructions.
- A count of resulting loads, stores, and floating-point instructions.
- An approximation of the time spent with the CPU stalling (caused by data and functional unit interlocks).

The task requires instrumentation of the target executable. This involves dividing the code into basic blocks, which are a set of instructions with a single entry point, a single exit point, and no branches within. Counter code is inserted at the beginning of each basic block.

After the instrumented executable runs, the Performance Analyzer multiplies the number of times a basic block was executed by the number of instructions in it. This yields the total number of instructions executed as a result of that basic block (and similarly for other specific kinds of instructions, like loads or stores).

3.4.4 `Floating Point Exception Trace`

Use the `Floating Point Exception Trace` task selection when you suspect that large, unaccounted for periods of time are being spent in floating-point exception handlers. The task records the call stack at each floating-point exception. The number of floating-point exceptions is presented as follows:

- By function in the function list
- By source line in the `Source View`
- By instruction in `Disassembly View`

To observe the pattern of floating-point exceptions over time, look in the floating-point exceptions event chart in the `Usage View (Graphical)` window.

3.4.5 I/O Trace

Use the `I/O Trace` task selection when your program is being slowed down by I/O calls, and you want to find the responsible code. This task records call stacks at every `read` and `write` system call, along with file descriptor information and the number of bytes read or written.

The number of bytes read and written is presented as follows:

- By function in the function list
- By source line in the `Source View`
- By instruction in the `Disassembly View`

3.4.6 Memory Leak Trace

Use the `Memory Leak Trace` task selection to determine where memory leaks and bad calls to `free` may occur in a process. The task records the call stacks, address, and number of bytes at every `malloc`, `realloc`, and `free` call. The bytes currently allocated by `malloc` (that might represent leaks) and the list of double calls to `free` are presented in `Malloc Error View` and the other memory analysis views. The number of bytes allocated by `malloc` is presented:

- By function in the function list
- By source line in the `Source View`
- By instruction in the `Disassembly View`

3.4.7 R10000 Hardware Counters

If you are running your application on a system using the R10000 series CPU, you can use the `R10k Hardware Counters` task selection once you have focused in on where your problem is coming from. This task gives low-level, detailed information about hardware events. It counts the following events:

- The cycles counter, which is incremented on each clock cycle.
- Issued instructions.

- Graduated instructions. The graduated instruction counter is incremented by the number of instructions that were graduated on the previous cycle.
- Issued loads.
- Graduated loads.
- Issued stores.
- Graduated stores.
- Issued store conditionals.
- Graduated store conditionals.
- Graduated floating-point instructions. This counter is incremented by the number of floating-point instructions that graduated on the previous cycle.
- Decoded branches.
- Primary cache quadword writeback.
- Secondary cache quadword writeback.
- TLB (task lookaside buffer) misses. This counter is incremented on the cycle after the TLB mishandler is invoked.
- Correctable secondary cache data array ECC errors.
- Mispredicted branches.
- Primary instruction cache misses. This counter is incremented one cycle after an instruction fetch request is entered into the miss handling table.
- Primary data cache misses. This counter is incremented on the cycle after a primary cache data refill is begun.
- Secondary instruction cache misses. This counter is incremented after the last 16-byte block of a 64-byte primary instruction cache line is written into the instruction cache.
- Secondary data cache misses. This counter is incremented on the cycle after the second 16-byte block of a primary data cache line is written into the data cache.
- Instruction misprediction.
- Data misprediction.

- External interventions.
- External interventions, secondary cache.
- External invalidations.
- External invalidation, secondary cache.
- Virtual coherency conditions.
- Store/prefetch exclusive to clean block in secondary cache.
- Store/prefetch exclusive to shared block in secondary cache.

You can also choose hardware counter profiling based on either PC sampling or call stack sampling.

3.4.8 `Custom`

Use the `Custom` task selection when you need to collect a combination of performance data that is not available through the predefined tasks. Selecting `Custom` brings up the same tab panel screen displayed by the `Configs...` selection (see Figure 23, page 51).

The `Custom` task lets you select and tune the following:

- Sampling data. This includes profiling intervals, counter size, and whether `rld`(1) will be involved in data collection.
- Tracing data. This includes `malloc` and `free` trace, I/O system call trace, and floating-point exception trace.
- Recording intervals. This includes the frequency of data recording for usage data or usage or call stack data at caliper points. You can also specify this with marching orders. (For more information on marching orders, see the `ssrun`(1) man page.)
- Call stack. This includes sampling intervals and the type of timing.
- Ideal experiments. This specifies whether or not the basic block count data is collected.
- Hardware counter specification. This specifies the hardware event you want to count, the counter overflow value, and the profiling style (PC or call stack). Hardware counter experiments are possible only on R10000 systems.

- Runtime. This specifies the same as those listed for the `Configs` menu selection. See Section 3.1, page 33.

Remember the basic warnings in this chapter about collecting data:

- Too much data can slow down the experiment.
- Call stack profiling is not compatible with count operations or PC profiling.
- If you combine count operations with PC profiling, the results will be skewed due to the amount of instrumented code that will be profiled.

Performance Analyzer Reference [4]

This chapter provides detailed descriptions of the Performance Analyzer toolset, including:

- Selecting Performance Tasks, see Section 4.1, page 43.
- Specifying a Custom Task, see Section 4.2, page 45.
- Specifying the Experiment Configuration, see Section 4.3, page 50.
- The Performance Analyzer Main Window, see Section 4.4, page 52.
- Usage View (Graphs), see Section 4.5, page 68.
- Process Meter, see Section 4.6, page 72.
- Usage View (Numerical), see Section 4.7, page 73.
- I/O View, see Section 4.8, page 75.
- Call Graph View, see Section 4.9, page 76.
- Butterfly View, see Section 4.10, page 85.
- Analyzing Memory Problems, see Section 4.11, page 86.
- Call Stack, see Section 4.12, page 97.
- Analyzing Working Sets, see Section 4.13, page 98.

4.1 Selecting Performance Tasks

You choose performance tasks from the `Select Task` submenu of the `Perf` menu in Debugger View. You should have an objective in mind before you start an experiment. The tasks ensure that only the appropriate data collection is enabled. Selecting too much data can slow down the experiment and skew the data for collection.

The tasks are summarized in Table 1, page 44. The Task column identifies the task as it appears in the `Select Task` menu of the WorkShop Debugger's `Perf` menu. The Clues column provides an indication of symptoms and situations appropriate for the task. The Data Collected column indicates performance data set by the task. Note that call stacks are collected

automatically at sample points, poll points, and process events. The Description column describes the technique used.

Table 1. Summary of Performance Analyzer Tasks

Task	Clues	Data Collected	Description
`Profiling/PC Sampling`	CPU-bound	• PC Profile Counts • Fine-Grained Usage (1 sec.) • Call stacks at sample points	Tracks CPU time spent in functions, source code lines, and instructions. Useful for CPU-bound conditions. CPU time metrics help you separate CPU-bound from non-CPU-bound instructions.
`User Time/Callstack Sampling`	Not CPU-bound	• Fine-Grained Usage (1 sec.) • Call Stack Profiling (30 ms) • Call stacks at sample points	Tracks the user time spent by function, source code line, and instruction.
`Ideal Time/Pixie`	CPU-bound	• Basic Block Counts • Fine-Grained Usage (1 sec.) • Call stacks at sample points	Calculates the ideal time, that is, the time spent in each basic block with the assumption of one instruction per machine cycle. Useful for CPU-bound conditions. Ideal time metrics also give counts, total machine instructions, and loads / stores / floating point instructions. It is useful to compare ideal time with the CPU time in an "Identify high CPU time functions" experiment.
`Floating Point Exception Trace`	High system time in usage charts; presence of floating point operations; NaNs	• FPE Exception Trace • Fine-Grained Usage (1 sec.) • Call stacks at sample points	Useful when you suspect that time is being wasted in floating-point exception handlers. Captures the call stack at each floating-point exception. Lists floating-point exceptions by function, source code line, and instruction.

Task	Clues	Data Collected	Description
`I/O trace`	Process blocking due to I/O	• I/O system call trace • Fine-Grained Usage (1 sec.) • Call stacks at sample points	Captures call stacks at every `read` and `write`. The file description and number of bytes are available in I/O View.
`Memory Leak Trace`	Swelling in process size	• `malloc`/`free` trace • Fine-Grained Usage (1 sec.) • Call stacks at sample points	Determines memory leaks by capturing the call stack, address, and size at all `malloc`, `realloc`, and `free` routines and displays them in a memory map. Also indicates double `free` routines.
`R10k Hardware Counters...`	Need more detailed information	• Wide range of hardware-level counts	On R10000 systems only, returns low-level information by counting hardware events in special registers. An overflow value is assigned to the relevant counter. The number of overflows is returned.
`Custom...`		• Call stacks at sample points • User's choice	Lets you select the performance data to be collected. Remember that too much data can skew results.

4.2 Specifying a Custom Task

When you choose `Custom...` from the `Select Task` submenu in the `Perf` menu in the Main View, a dialog box appears. This section provides an explanation of most of the windows involved in setting up a custom task.

The `Custom... Runtime` and `HWC Spec` (the hardware counters) windows are identical to the `Configs... Runtime` and `HWC Spec` windows. For an illustration of `Runtime`, see Figure 23, page 51. For information on `HWC Spec`, see Section 3.4.7, page 39.

4.2.1 Specifying Data to be Collected

Data is collected and recorded at every sample point. The following data collection methods are available:

- Call stack (the `CallStack` window). See the following section.
- Basic block counts (the `Ideal` window). See Section 4.2.1.2, page 46.

- PC profile counts (the `PC Sampling` window). See Section 4.2.1.3, page 47.

4.2.1.1 Call Stack Profiling

The Performance Analyzer performs call stack data collection automatically. There is no instrumentation involved. This corresponds to the SpeedShop `usertime` experiment.

The `CallStack` window lets you choose from real time, virtual time, and profiling time and specify the sampling interval.

Real time is also known as *wall-clock time* and *total time*. It is the total time a program takes to execute, including the time it takes waiting for a CPU.

Virtual time is also called *process virtual time*. It is the time spent when a program is actually running, as opposed to when it is swapped out and waiting for a CPU or when the operating system is in control, such as performing I/O for the program.

Profiling time is time the process has actually been running on the CPU, whether in user or system mode. This is the default for the `usertime` experiment. The is also called *CPU time* or *user time*.

For the sampling interval, you can select one of the following intervals:

- `Standard` (every 30 milleseconds)
- `Fast` (every 20 milliseconds)
- `Custom` (enter your own interval)

4.2.1.2 Basic Block Count Sampling

Basic block counts are translated to ideal CPU time (as shown in the SpeedShop `ideal` experiment) and are displayed at the function, source line, and machine line levels. The assumptions made in calculating ideal CPU time are as follows:

- Each instruction and system call takes exactly one cycle.
- Potential floating-point interlocks and memory latency time (cache misses and memory bus contention) are ignored.

The end result might be better described as *ideal user CPU time*.

The `Ideal` window lets you select the counter size, either 16 or 32 bits, and the option to use `rld`(1) profiling.

The data is gathered by first instrumenting the target executable. This involves dividing the executable into basic blocks consisting of sets of machine instructions that do not contain branches into or out of them. A few instructions are inserted for every basic block to increment a counter every time that basic block is executed. The basic block data is actually generated, and when the instrumented target executable is run, the data is written out to disk whenever a sample trap fires. Instrumenting an executable increases its size by a factor of three and greatly modifies its performance behavior.

Caution: Running the instrumented executable causes it to run more slowly. By instrumenting, you might be changing crucial resources; during analysis, the instrumented executable might appear to be CPU-bound, whereas the original executable is I/O-bound.

4.2.1.3 PC Profile Counts

Enabling PC profile counts causes the Program Counter (PC) of the target executable to be sampled every 10 milliseconds when it is in the CPU. PC profiling is a lightweight, high-speed operation done with kernel support. Every 10 milliseconds, the kernel stops the process if it is in the CPU, increments a counter for the current value of the PC, and resumes the process. It corresponds to the SpeedShop `pcsamp` experiment.

PC profile counts are translated to the actual CPU time displayed at the function, source line, and machine line levels. The actual CPU time is calculated by multiplying the PC hit count by 10 milliseconds.

A major discrepancy between actual CPU time and ideal CPU time indicates one or more of the following:

- Cache misses in a single process application.
- Secondary cache invalidations in a multiprocess application run on a multiprocessor.

Note: This comparison is inaccurate over a single run if you collect both basic block and PC profile counts simultaneously. In this situation, the ideal CPU time will factor out the interference caused by instrumenting; the actual CPU time will not.

A comparison between basic block counts and PC profile counts is shown in Table 2, page 48.

Table 2. Basic Block Counts and PC Profile Counts Compared

Basic Block Counts	PC Profile Counts
Used to compute ideal CPU time	Used to estimate actual CPU time
Data collection by instrumenting	Data collection done with the kernel
Slows program down	Has minimal impact on program speed
Generates an exact count	Approximates counts

4.2.2 Specifying Tracing Data

Tracing data records the time at which an event of the selected type occurred. There are five types of tracing data:

- `malloc` and `free` Heap Analysis, see Section 4.2.2.1, page 48.
- I/O (`read`, `write`) Operations, see Section 4.2.2.2, page 48.
- Floating-Point Exceptions, see Section 4.2.2.3, page 49.

Note: These features should be used with care; enabling tracing data adds substantial overhead to the target execution and consumes a great deal of disk space.

4.2.2.1 `malloc` and `free` Heap Analysis

Tracing `malloc` and `free` allows you to study your program's use of dynamic storage and to quickly detect memory leaks (`malloc` routines without corresponding `free` routines) and bad `free` routines (freeing a previously freed pointer). This data can be analyzed in the `Malloc Error View`, `Leak View`, `Malloc View`, and `Heap View` (see Section 4.11, page 86).

4.2.2.2 I/O (`read`, `write`) Operations

I/O tracing records every I/O-related system call that is made during the experiment. It traces `read` and `write` system calls with the call stack at the time, along with the number of bytes read or written. This is useful for I/O-bound processes.

4.2.2.3 Floating-Point Exceptions

Floating-point exception tracing records every instance of a floating-point exception. This includes problems like underflow and NaN (not a number) values. If your program has a substantial number of floating-point exceptions, you may be able to speed it up by correcting the algorithms.

The floating-point exceptions are as follows:

- Overflow
- Underflow
- Divide-by-zero
- Inexact result
- Invalid operand (for example, infinity)

4.2.3 Specifying Polling Data

The following categories of polling data are available:

- Pollpoint Sampling, see Section 4.2.3.1, page 49.
- Call Stack Profiling, see Section 4.2.3.2, page 49.

Entering a positive nonzero value in their fields turns them on and sets the time interval at which they will record.

4.2.3.1 Pollpoint Sampling

Setting pollpoint sampling on the `Runtime` tab window enables you to specify a regular time interval for capturing performance data, including resource usage and any enabled sampling or tracing functions. Since pollpoint sampling occurs frequently, it is best used with call stack data only rather than other profiling data. Its primary utility is to enable you to identify boundary points for phases. In subsequent runs, you can set sample points to collect the profiling data at the phase boundaries.

4.2.3.2 Call Stack Profiling

Enabling call stack profiling in the `CallStack` tab window causes the call stack of the target executable to be sampled at the specified time interval (minimum of 10 milliseconds) and saved. The call stack continues to be

sampled when the program is not running: that is, while it is internally or externally blocked. Call stack profiling is used in the `User Time/Callstack Sampling` task to calculate total times.

You can choose the type of time you want to eventually display: real time, virtual time, or profiling time. See the glossary for definitions.

By setting the sampling interval to a higher number, you can sample more often and receive better finer grained results.

Call stack profiling is accomplished by the Performance Analyzer views and not by the kernel. As a result, it is less accurate than PC profiling. Collecting call stack profiling data is far more intrusive than collecting PC profile data.

Caution: Collecting basic block data causes the text of the executable to be modified. Therefore, if call stack profiling data is collected along with basic block counts, the cumulative total time displayed in `Usage View (Graphs)` is potentially erroneous.

Table 3, page 50, compares call stack profiling and PC profiling.

Table 3. Call Stack Profiling and PC Profiling Compared

PC Profiling	Call Stack Profiling
Done by kernel	Done by Performance Analyzer process
Accurate, nonintrusive	Less accurate, more intrusive
Used to compute CPU time	Used to compute total time

4.3 Specifying the Experiment Configuration

To specify the experiment configuration, choose `Configs...` from the `Perf` menu. See Figure 23, page 51, for an illustration of the resulting window. While you can access other tabs, the only ones that are active are the `Runtime` and `General` tabs.

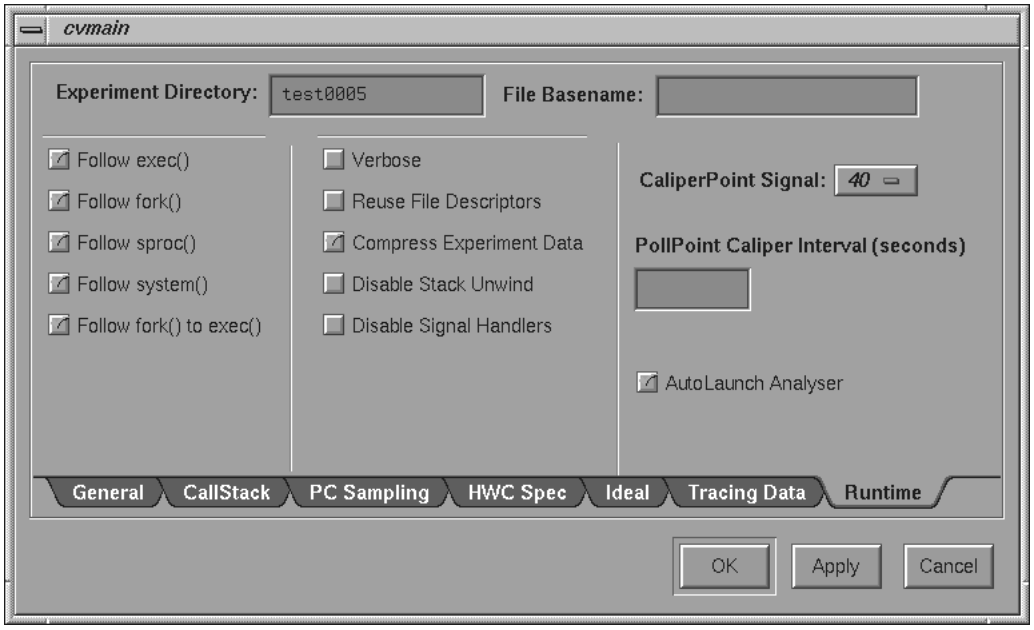

Figure 23. Runtime Configuration Dialog Box

4.3.1 Specifying the Experiment Directory

The `Experiment Directory` field lets you specify the directory where you want the data to be stored. The Performance Analyzer provides a default directory named `test0000` for your first experiment. If you use the default or any other name that ends in four digits, the four digits are used as a counter and will be incremented automatically for each subsequent session. Note that the Performance Analyzer does not remove (or overwrite) experiment directories. You need to remove directories yourself.

4.3.2 Other Options

The following configuration options are available on the `Runtime` display:

- Specifies the base name of the experiment file (if blank, it is the name of the executable).
- Lets you specify whether you want the Performance Analyzer to gather performance data for any processes launched by one or more of the following:

  - `exec()`
  - `fork()`
  - `sproc()`
  - `system()`
  - Tracks `fork()` to `exec()` processes.
- Other options:
  - `Verbose` output yields more explanatory information in the `Execution View`.
  - `Reuse File Descriptors` opens and closes the file descriptors for the output files every time performance data is to be written. If the target program is using `chdir()`, the `_SPEEDSHOP_REUSE_FILE_DESCRIPTORS` environment variable is set to the value selected by this configuration option.
  - `Compress Experiment Data` saves disk space.
  - `Disable Stack Unwind` suppresses the stack unwind as is done in the SpeedShop `usertime`, `totaltime`, and other call stack-based experiments.
  - `Disable Signal Handlers` disables the normal setting of signal handlers for all fatal and exit signals.
- `CaliperPoint Signal` sets the value of the signal sent by the sample button to cause the process to write out a caliper point. The default value is 40.
- `PollPoint Caliper Interval (seconds)` specifies the interval at which pollpoint caliper points are taken.
- `AutoLaunch Analyzer` launches the Performance Analyzer automatically when the experiment finishes.

4.4 The Performance Analyzer Main Window

The Performance Analyzer main window is used for analysis after the performance data has been captured. It contains a time line area indicating when events took place over the span of the experiment, a list of functions with

their performance data, and a resource usage chart. The following sections cover these topics:

- Task field, see the Section 4.4.1, page 54.
- Function list display and controls, see Section 4.4.2, page 54.
- Usage chart area, see Section 4.4.3, page 55.
- Time line area and controls, seeSection 4.4.4, page 56.
- Admin menu, see Section 4.4.5, page 57.
- Config menu, see Section 4.4.6, page 59.
- Views menu, see Section 4.4.7, page 66.
- Executable menu, see Section 4.4.8, page 67.
- Thread menu, see Section 4.4.9, page 68.

The Performance Analyzer main window can be invoked from the `Launch Tool` submenu in the Debugger `Admin` menu or from the command line, by typing one of the following:

`cvperf [-exp]` *directory*

`cvperf` *speedshop_exp_files*

`cvperf [-pixie]` *pixie.counts_files*

The arguments to these commands are as follows:

directory	A directory containing data from old WorkShop performance experiments.
speedshop_exp_files	One or more SpeedShop experiment files generated either by the `ssrun`(1) command or by using the `Select Task ...` submenu of the `Perf` menu on the WorkShop Debugger window.

pixie.counts_files	An output file from `pixie`(1) measuring code execution frequency. The `ideal` task generates a *pixie.counts_file*.

4.4.1 Task Field

The `Task` field identifies the task for the current experiment and is read-only. See Section 4.1, page 43, for a summary of the performance tasks. For an in-depth explanation of each task, refer to Section 3.4, page 36.

4.4.2 Function List Display and Controls

The function list area displays the program's functions with the associated performance metrics. It also provides buttons for displaying function performance data in other views. See Figure 24, page 54.

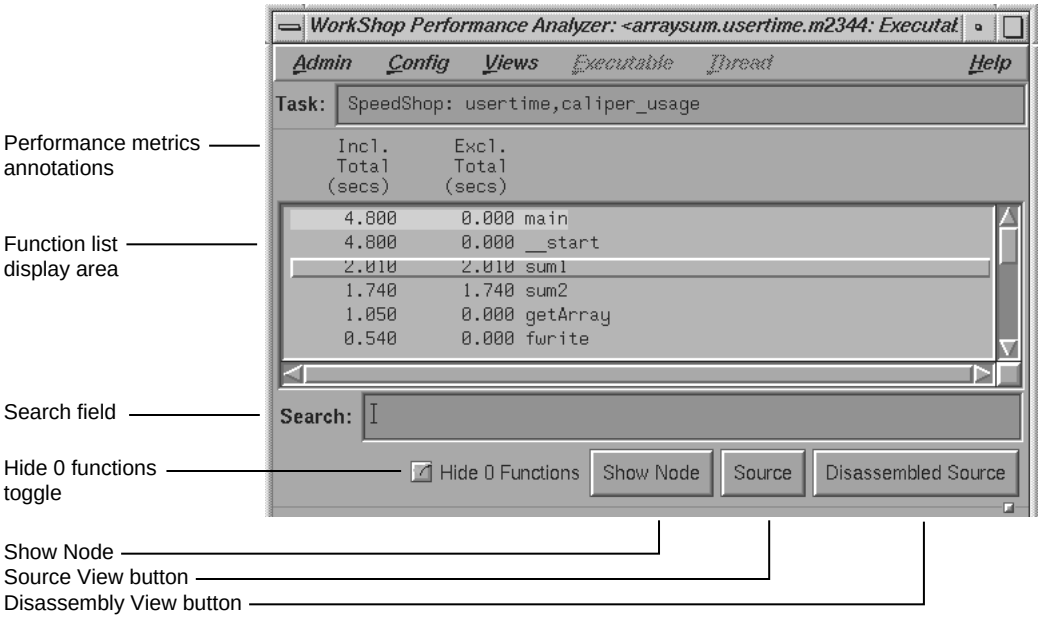

Figure 24. Typical Function List Area

The main features of the function list are:

Function list display area	Shows all functions in the program annotated with their associated performance data. The column headings identify the metrics. You select the performance data to display from the `Preferences...` selection in the `Config` menu. The order of ranking is set by the `Sort...` selection in the `Config` menu. The default order of sorting (depending on availability) is: 1. Inclusive time 2. Exclusive time 3. Counts
`Search` field	Lets you look for a function in the list and in any active views.
`Hide 0 Functions` toggle button	Lets you filter functions with 0 time from the list.
`Show Node` button	Displays the specified node in the `Call Graph View`.
`Source` button	Displays the `Source View` window corresponding to the selected function. The `Source View` window displays performance metrics in the annotation column. `Source View` can also be displayed by double-clicking a function in the function list or a node or arc in the call graph. This is discussed in the next section.
`Disassembled Source` button	Displays the `Disassembly View` window corresponding to the selected function. The `Disassembly View` is annotated with the performance metrics.

4.4.3 Usage Chart Area

The usage chart area in the Performance Analyzer main window displays the stripchart most relevant to the current task. The upper subwindow displays the legend for the stripchart, and the lower subwindow displays the stripchart itself. This gives you some useful information without having to open the `Usage View (Graphs)` window. Table 4, page 56, shows you the data displayed in the usage chart area for each task.

Table 4. Task Display in Usage Chart Area

Task	Data in Usage Chart Area
User Time/Callstack Sampling	User versus system time
Profiling/PC Sampling	User versus system time
Ideal Time/Pixie	User versus system time
Floating Point Exception Trace	Floating-point exception event chart
I/O Trace	`read()`, `write()` system calls
Memory Leak Trace	Process Size stripchart
R10000 Hardware Counters	Depends on experiment
Custom task	User versus system time, unless tracing data has been selected (see Trace tasks above)

4.4.4 Time Line Area and Controls

The time line shows when each sample event in the experiment occurred. Figure 2, page 7, shows the time line portion of the Performance Analyzer window with typical results.

4.4.4.1 The Time Line Calipers

The time line calipers let you define an interval for performance analysis. You can set the calipers in the time line to any two sample event points, using the caliper controls or by dragging them directly. The calipers appear solid for the current interval. If you drag them with the mouse (left or middle button), they appear dashed to give you visual feedback. When you stop dragging a caliper, it appears in outlined form denoting a tentative and as yet unconfirmed selection.

Specifying an interval is done as follows:

1. Set the left caliper to the sample event at the beginning of the interval.

   You can drag the left caliper with the left or middle mouse button or by using the left caliper control buttons in the control area. Note that calipers always snap to sample events. (It does not matter whether you start with the left or right caliper.)

2. Set the right caliper to the sample event at the end of the interval. This is similar to setting the left caliper.
3. Confirm the change by clicking the `OK` button in the control area.

   After you confirm the new position, the solid calipers move to the current position of the outlined calipers and change the data in all views to reflect the new interval.

   Clicking `Cancel` or clicking with the right mouse button before the change is confirmed restores the outlined calipers to the solid calipers.

4.4.4.2 Current Event Selection

If you want to get more information on an event in the time line or in the charts in the `Usage View (Graphs)`, you can click an event with the left button. The `Event` field displays the following:

- Event number
- Description of the trap that triggered the event

In addition, the `Call Stack View` window updates to the appropriate times, stack frames, and event type for the selected event. A black diamond-shaped icon appears in the time line and charts to indicate the selected event. You can also select an event using the event controls below the caliper controls; they work in similar fashion to the caliper controls.

4.4.4.3 Time Line Scale Menu

The time line scale menu lets you change the number of seconds of the experiment displayed in the time line area. The `Full Scale` selection displays the entire experiment on the time line. The other selections are time values; for example, if you select `1 min`, the length of the time line displayed will span 1 minute.

4.4.5 `Admin` Menu

The `Admin` menu and its options are shown in Figure 25, page 58. The `Admin` menu has selections common to the other WorkShop tools. The following selections are different in the Performance Analyzer:

`Experiment...`	Lets you change the experiment directory and displays a dialog box (see Figure 26, page 59).

`Save As Text...`	Records a text file with preference information selected in the view and displays a dialog box. You can use the default file name or replace it with another name in the `Selection` dialog box that displays. You can specify the number of lines to be saved. The data can be saved as a new file or appended to an existing one.

Figure 25. Performance Analyzer `Admin` Menu

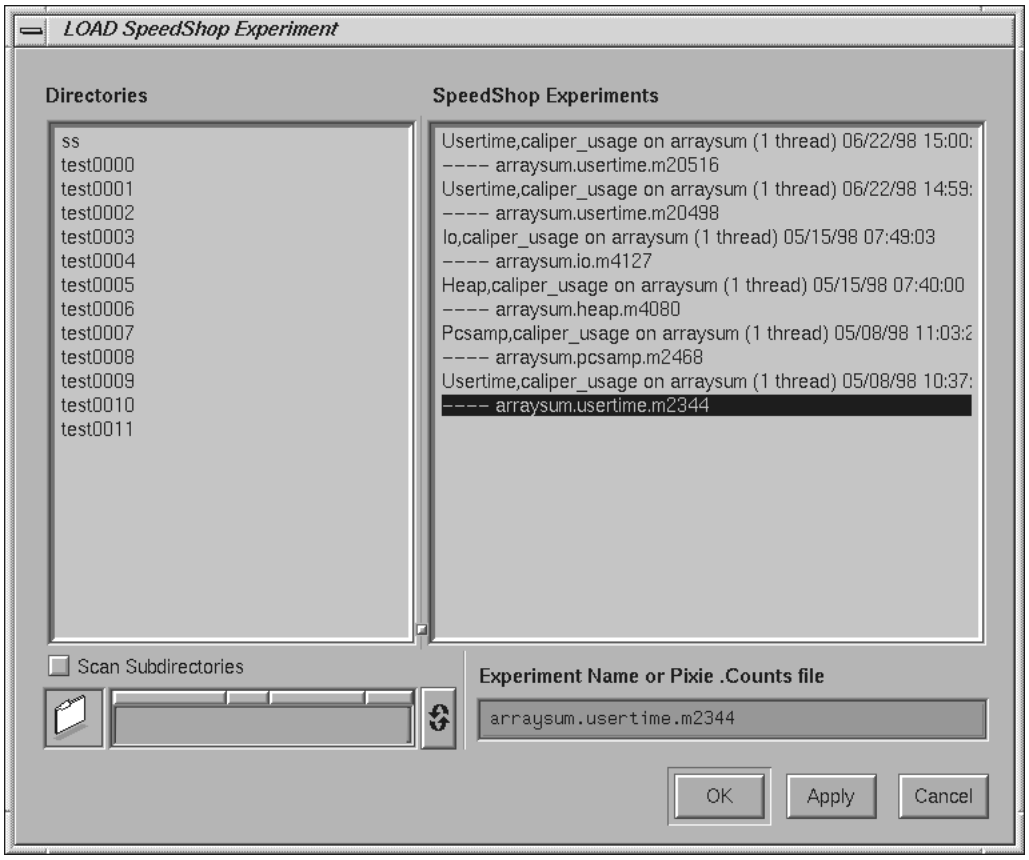

Figure 26. `Experiment` Window

4.4.6 Config Menu

The main purpose of the `Config` menu in the Performance Analyzer main window is to let you select the performance metrics for display and for ranking the functions in the function list. However, your selections also apply elsewhere, such as the `Call Graph View` window.

The selections in the `Config` menu are as follows:

`Preferences...` Lets you select which metrics display and whether they appear as absolute times and counts or percentages. Remember you can only select the types of metrics that were collected in the

experiment. You can also specify how C++ file names (if appropriate) are to display:

- `Demangled` shows the function and its argument types.
- `As Is` uses the translator-generated C-style name.
- `Function` shows the function name only.
- `Class::Function` shows the class and function.

For an illustration of the `Preferences...` window, see Figure 27, page 61.

`Sort...` Lets you establish the order in which the functions appear; this helps you find questionable functions. The default order of sorting (depending on availability) is:

1. Inclusive time or counts
2. Exclusive time or counts
3. Counts

For an illustration, see Figure 28, page 62.

The performance data selections for the `Preferences` and `Sort` dialog boxes are similar. The difference between the inclusive (`Incl.`) and exclusive (`Excl.`) metrics is that inclusive data includes a function's calls and exclusive data does not.

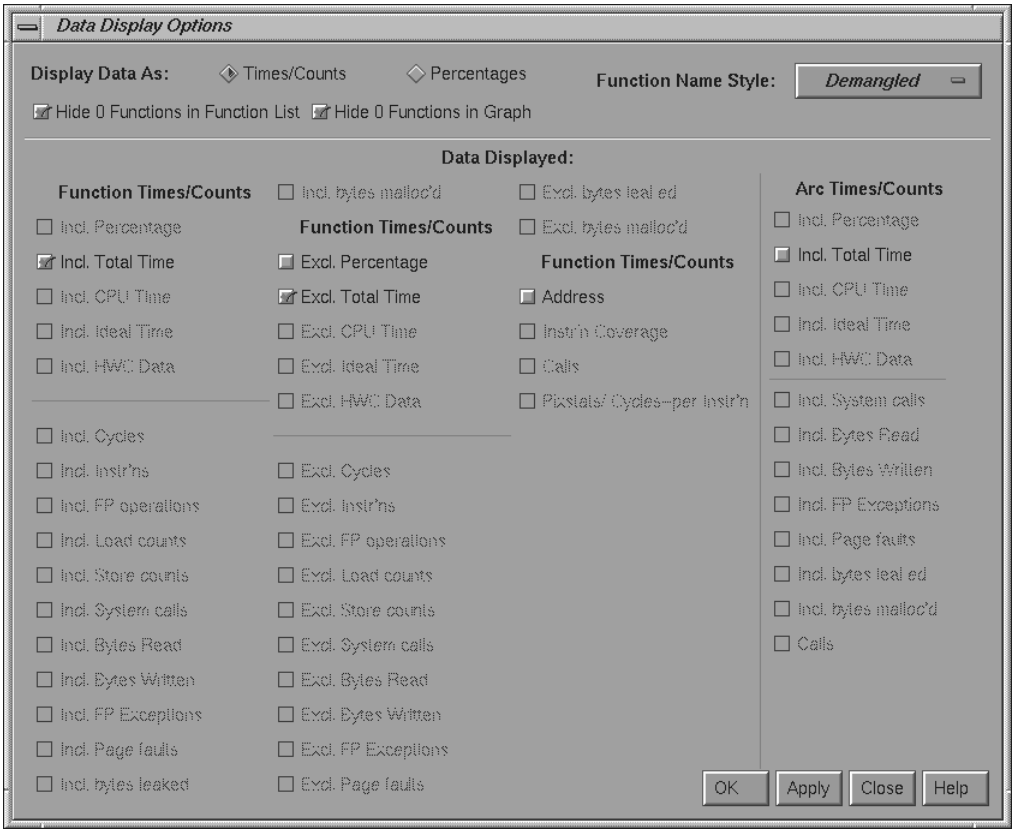

Figure 27. Performance Analyzer Data Display Options

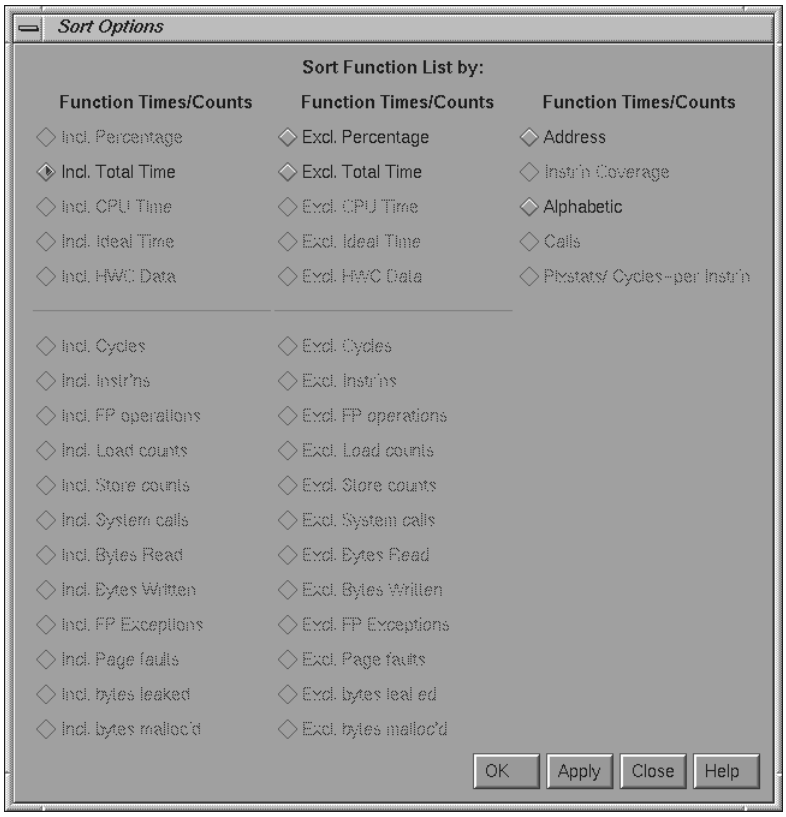

Figure 28. Performance Analyzer Sort Options

The toggle buttons in both the `Data Display Options` and `Sort Options` windows are as follows:

`Incl. Percentage`, `Excl. Percentage`

Percentage of the total time spent inside and outside of the CPU (by a function, source line, or instruction).

`Incl. Total Time`, `Excl. Total Time`

Time spent inside and outside of the CPU (by a function, source line, or instruction). It is calculated by multiplying the number of times the PC appears in any call stack by the average time interval between call stacks.

`Incl. CPU Time, Excl. CPU Time`

Time spent inside the CPU (by a function, source line, or instruction). It is calculated by multiplying the number of times a PC value appears in the profile by 10 ms.

`Incl. Ideal Time, Excl. Ideal Time`

Theoretical time spent by a function, source line, or instruction under the assumption of one machine cycle per instruction. It is useful to compare ideal time with actual.

`Incl. HWC Data, Excl. HWC Data`

Number of events measured.

`Incl. Cycles, Excl. Cycles`

Number of machine cycles.

`Incl. Instr'ns, Excl. Instr'ns`

Number of instructions.

`Incl. FP operations, Excl. FP operations`

Number of floating-point operations.

`Incl. Load counts, Excl. Load counts`

Number of `load` operations.

`Incl. Store counts, Excl. Store counts`

Number of `store` operations.

`Incl. System calls, Excl. System calls`

Number of system calls.

`Incl. Bytes Read, Excl. Bytes Read`

Number of bytes in a `read` operation.

`Incl. Bytes Written, Excl. Bytes Written`

Number of bytes in a `write` operation.

`Incl. FP Exceptions, Excl. FP Exceptions`

Number of floating-point exceptions.

`Incl. Page faults, Excl. Page faults`

Number of page faults.

`Incl. bytes leaked, Excl. bytes leaked`

Number of bytes leaked as a result of calls to `malloc` that were not followed by calls to `free`.

`Incl. bytes malloc'd, Excl. bytes malloc'd`

Number of bytes allocated in `malloc` operations.

`Address`

Address of the function.

`Instr'n Coverage`

A percentage of instructions (in the line or function) that were executed at least once.

`Calls`

Number of times a function is called.

`Pixstats/Cycles-per instr'n`

Shows how efficient the code is written to avoid stalls or to take advantage of super scalar operation. A cycles per-instruction count of 1.0 means that an instruction is executed every cycle. A count greater than 1.0 means some instructions took more than one cycle. A count less that 1.0 means that sometimes more than one instruction was executed at a given cycle. The R10000 can potentially execute up to 4 instructions on every cycle.

In the disassembly view, this metric turns into `pixstats`, which displays basic block boundaries and the cycle counts distribution for each instruction in the basic block.

The following options are on the `Data Display Options` window only:

Display Data As: Times/Counts Percentages	Lets you choose whether you want to display your performance metrics as times and counts (for instance, the time a function required to execute) or as percentages (the percentage of the program's time a function used). The default is Times/Counts.
Hide 0 Functions in Function List and Hide 0 Functions in Graph	Lets you filter functions with 0 counts from the list or graph.
Incl. Percentage	Show inclusive percentages on the Call Graph View window.
Incl. Total Time	Show inclusive total time on the Call Graph View window.
Incl. CPU Time	Show inclusive CPU time on the Call Graph View window.
Incl. Ideal Time	Show inclusive ideal time on the Call Graph View window.
Incl. HWC Data	Show inclusive hardware counter data on the Call Graph View window.
Incl. System calls	Show inclusive system calls on the Call Graph View window.
Incl. Bytes Read	Show inclusive bytes read on the Call Graph View window.
Incl. Bytes Written	Show inclusive bytes written on the Call Graph View window.
Incl. FP Exceptions	Show inclusive floating–point exceptions on the Call Graph View window.
Incl. Page faults	Show inclusive page faults on the Call Graph View window.

`Incl. bytes leaked`	Show inclusive bytes leaked as a result of `malloc` operations not followed by matching `free` operations on the `Call Graph View` window.
`Incl. bytes malloc'd`	Show inclusive bytes allocated with a `malloc` operation on the `Call Graph View` window.
`Calls`	Show the number of calls to that function on the `Call Graph View` window.

The following option is on the `Sort Options` window only:

`Alphabetic`	Sort alphabetically by function name.

4.4.7 Views Menu

The `Views` menu in the Performance Analyzer (see Figure 29, page 67) provides the following selections for viewing the performance data from an experiment. Each view displays the data for the time interval bracketed by the calipers in the time line.

`Usage View (Graphs)`	Displays resource usage charts and event charts. See Section 4.5, page 68.
`Usage View (Numerical)`	Displays the aggregate values of resources used. See Section 4.7, page 73.
`I/O View`	Displays I/O events. See Section 4.8, page 75.
`Call Graph View`	Displays a call graph that shows functions and calls and their associated performance metrics. See Section 4.9, page 76.
`Butterfly View`	Displays the callers and callees of the function. See Section 4.10, page 85.
`Leak View`	Displays individual leaks and their associated call stacks. See Section 4.11.2, page 88.
`Malloc View`	Displays individual `malloc` routines and their associated call stacks. See Section 4.11.2, page 88.
`Malloc Error View`	Displays errors involving memory leaks and bad calls to `free`, indicating error locations and the total number of errors. See Section 4.11.2, page 88.
`Heap View`	Displays a map of heap memory showing `malloc`, `realloc`, `free`, and bad `free` operations. See Section 4.11.3, page 91.

`Call Stack`	Displays the call stack for the selected event and the corresponding event type. See Section 4.12, page 97.
`Working Set View`	Measures the coverage of the DSOs that make up the executable, noting which were not used. See Section 1.4.12, page 20.

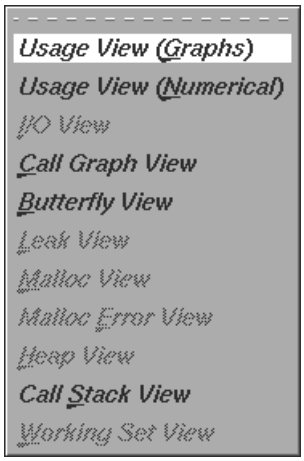

Figure 29. Performance Analyzer `Views` Menu

4.4.8 `Executable` Menu

If you enabled `Track Exec'd Processes` (in the Performance Panel) for the current experiment, the `Executable` menu will be enabled and will contain selections for any `exec`ed processes. These selections let you see the performance results for the other executables.

Note: The `Executable` menu is not enabled by an experiment generated by the `Select Task` submenu in the `Perf` menu of the WorkShop Debugger window, the `ssrun`(1) command, or any other method using SpeedShop functionality. It can only be enabled by experiments generated in older versions of WorkShop.

4.4.9 Thread Menu

If your process forked any processes, the `Thread` menu is activated and contains selections corresponding to the different threads. Selecting a thread displays its performance results.

Note: The `Thread` menu is not enabled by an experiment generated by the `Select Task` submenu in the `Perf` menu of the WorkShop Debugger window, the `ssrun`(1) command, or any other method using SpeedShop functionality. It can only be enabled by experiments generated in older versions of WorkShop.

4.5 `Usage View (Graphs)`

The `Usage View (Graphs)` window displays resource usage and event charts containing the performance data from the experiment. These charts show resource usage over time and indicate where sample events took place. Sample events are shown as vertical lines. Figure 30, page 68, shows the user versus system time and page faults graphs; Figure 31, page 69, shows the other graphs.

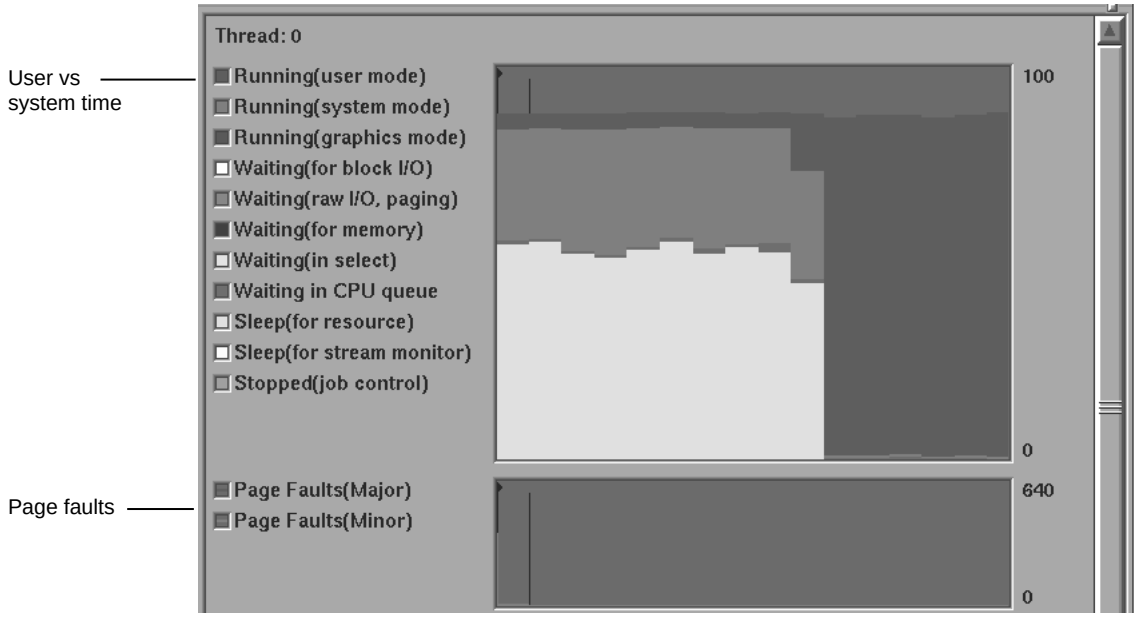

Figure 30. `Usage View (Graphs)` Window: Top Graphs

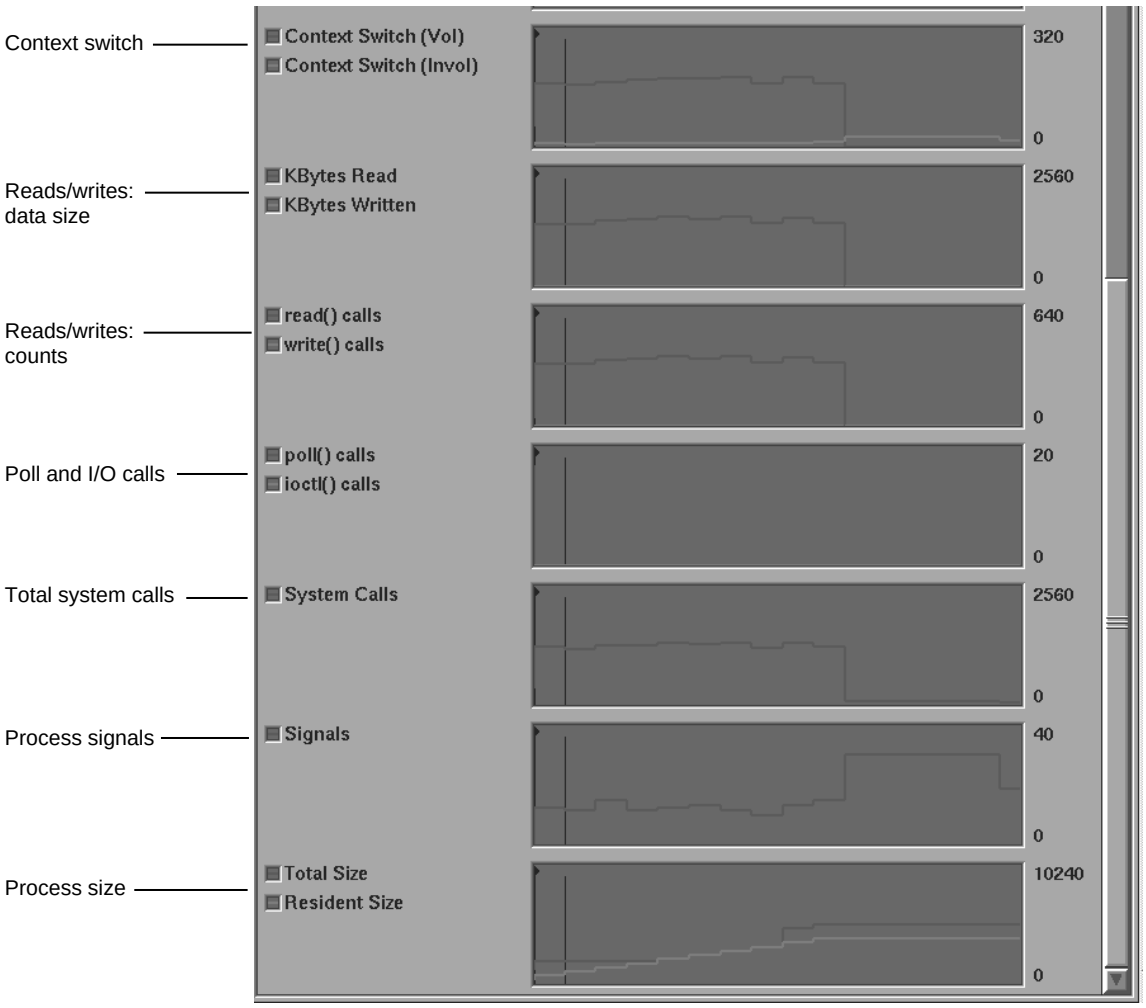

Figure 31. `Usage View (Graphs)` Window: Lower Graphs

4.5.1 Charts in the `Usage View (Graphs)` Window

The available charts in the `Usage View (Graphs)` Window are as follows:

`User vs system time` — Shows CPU usage. Whenever the system clock ticks, the process occupying the CPU is charged for the entire ten millisecond interval. The time is charged either as user or system time, depending

	on whether the process is executing in user mode or system mode. The graph provides these annotations to show how time is spent during an experiment's process: `Running (user mode)`, `Running (system mode)`, `Running (graphics mode)`, `Waiting (for block I/O)`, `Waiting (raw I/O, paging)`, `Waiting (for memory)`, `Waiting (in select)`, `Waiting in CPU queue`, `Sleep (for resource)`, `Sleep (for stream monitor)`, and `Stopped (job control)`.
`Page faults`	Shows the number of page faults that occur within a process. *Major faults* are those that require a physical read operation to satisfy; *minor faults* are those where the necessary page is already in memory but not mapped into the process address space. Each major fault in a process takes approximately 10 to 50 milliseconds. A high page-fault rate is an indication of a memory-bound situation.
`Context switch`	Shows the number of voluntary and involuntary context switches in the life of the process. *Voluntary context switches* are attributable to an operation caused by the process itself, such as a disk access or waiting for user input. These occur when the process can no longer use the CPU. A high number of voluntary context switches indicates that the process is spending a lot of time waiting for a resource other than the CPU. *Involuntary context switches* happen when the system scheduler gives the CPU to another process, even if the target process is able to use it. A high number of involuntary context switches indicates a CPU contention problem.
`KBytes Read` and `KBytes Written`	Shows the number of bytes transferred between the process and the operating system buffers, network connections, or physical devices. `KBytes Read` are transferred into the process address space; `KBytes Written` are transferred

	out of the process address space. A high byte-transfer rate indicates an I/O-bound process.
`read() calls and write() calls`	Shows the number of `read` and `write` system calls made by the process.
`poll() calls and ioctl() calls`	Shows the combined number of poll or select system calls (used in I/O multiplexing) and the number of I/O control system calls made by the process.
`System Calls`	Shows the total number of system calls made by the process. This includes the counts for the calls shown on the other charts.
`Signals`	Shows the total number of signals received by the process.
`Total Size and Resident Size`	Shows the total size of the process in pages and the number of pages resident in memory at the end of the time interval when the data is read. It is different from the other charts in that it shows the absolute size measured at the end of the interval and not an incremental count for that interval. If you see the process total size increasing over time when your program should be in a steady state, the process most likely has leaks and you should analyze it with `Leak View` and `Malloc View`.

4.5.2 Getting Event Information from the `Usage View (Graphs)` Window

The charts indicate trends; to get detailed data, click the relevant area on the chart, and the data displays at the top of the window. The left mouse button displays event data; the right mouse button displays interval data.

When you click the left mouse button on a sample event in a chart, the following actions take place:

- The point becomes selected, as indicated by the diamond marker above it. The marker appears in the time line, resource usage chart, and `Usage View (Graphs)` charts if the window is open.

- The current event line at the top of the window identifies the event and displays its time.
- The call stack corresponding to this sample point gets displayed in the `Call Stack` window (see Section 4.12, page 97).

Clicking a graph with the right mouse button displays the values for the interval if a collection is specified. If a collection is not specified, clicking a graph with the right mouse button displays the interval bracketed by the nearest sample events.

4.6 The Process Meter Window

The Process Meter lets you observe resource usage for a running process without conducting an experiment. To call the Process Meter, select `Process Meter` from the `Views` menu in the Debugger Main View.

A `Process Meter` window with data and its menus displayed appears in Figure 32, page 73. The `Process Meter` window uses the same `Admin` menu as the WorkShop Debugger tools.

The `Charts` menu options display the selected stripcharts in the `Process Meter` window.

The `Scale` menu adjusts the time scale in the stripchart display area such that the time selected becomes the end value.

You can select which usage charts and event charts display. You can also display sample point information in the `Status` field by clicking within the charts.

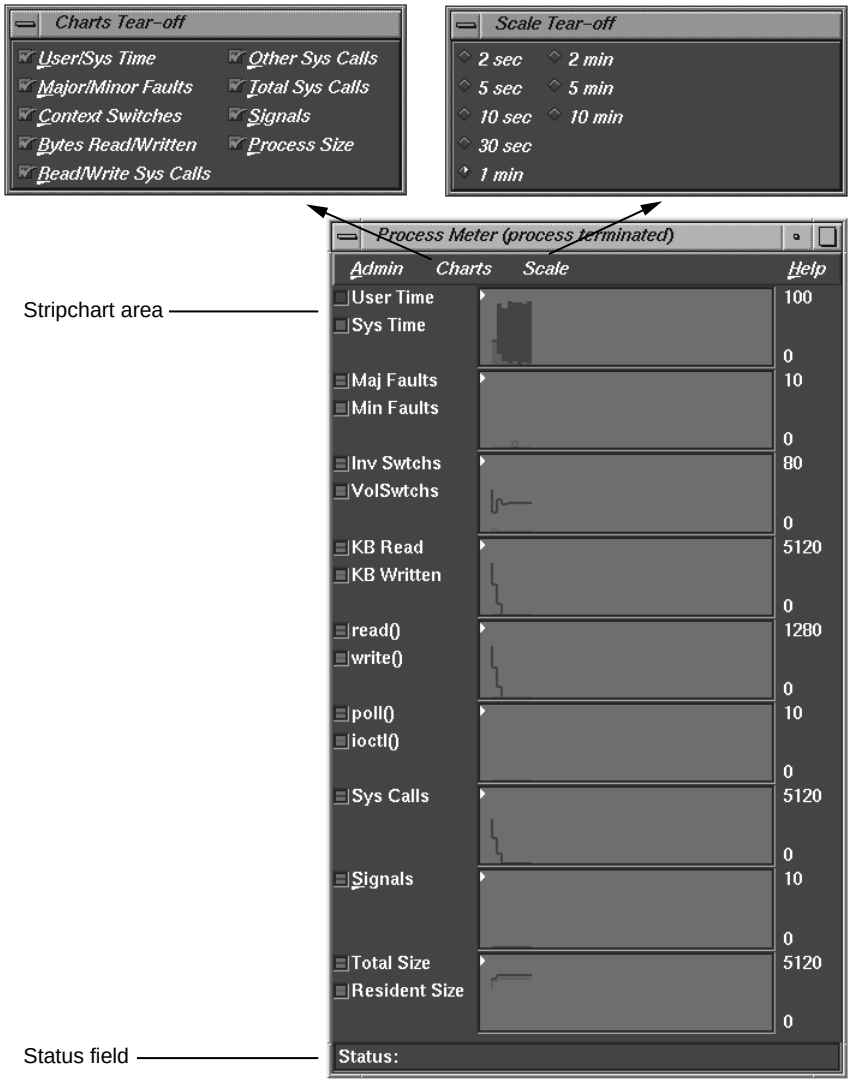

Figure 32. The `Process Meter` Window with Major Menus Displayed

4.7 `Usage View (Numerical)` Window

The `Usage View (Numerical)` window (see Figure 33, page 75) shows detailed, process-specific resource usage information in a textual format for the interval defined by the calipers in the time line area of the Performance

Analyzer main window. To display the `Usage View (Numerical)` window, select `Usage View (Numerical)` from the `Views` menu.

The top of the window identifies the beginning and ending events for the interval. The middle portion of the window shows resource usage for the target executable. The bottom panel shows resource usage on a system-wide basis. Data is shown both as total values and as per-second rates.

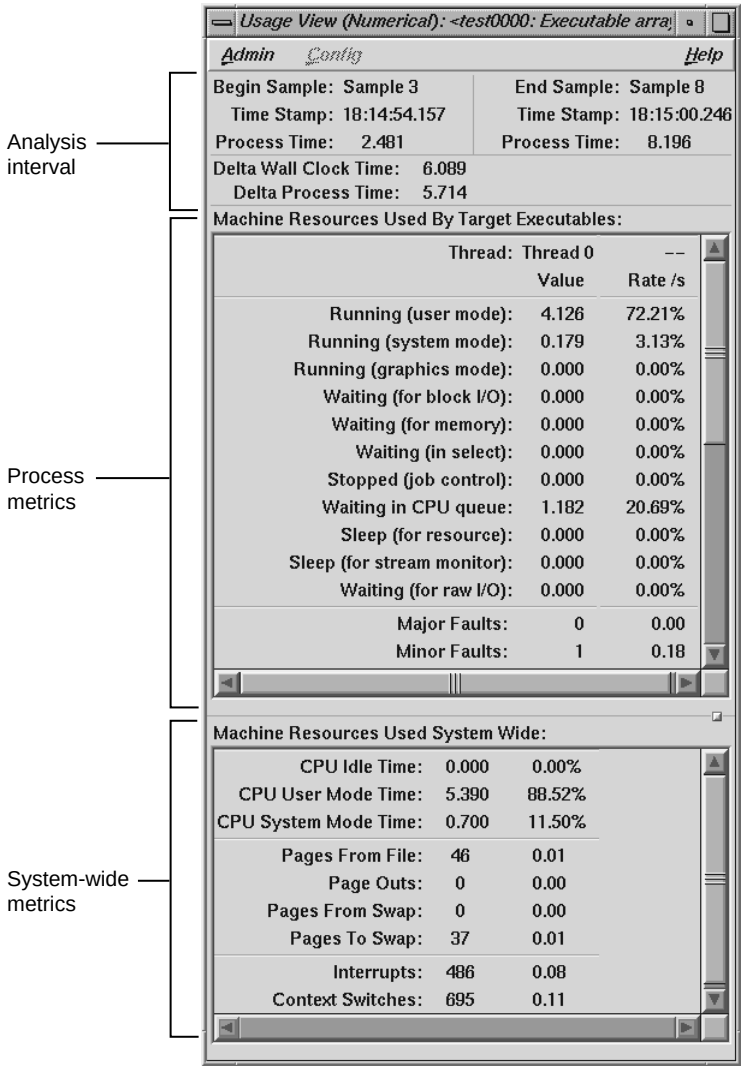

Figure 33. The `Usage View (Numerical)` Window

4.8 The `I/O View` Window

The `I/O View` window helps you determine the problems in an I/O-bound process. It produces graphs of all I/O system calls for up to 10 files involved in I/O. Clicking an I/O event with the left mouse button displays information

about it in the event identification field at the top of the `I/O View` window. See Figure 34, page 76.

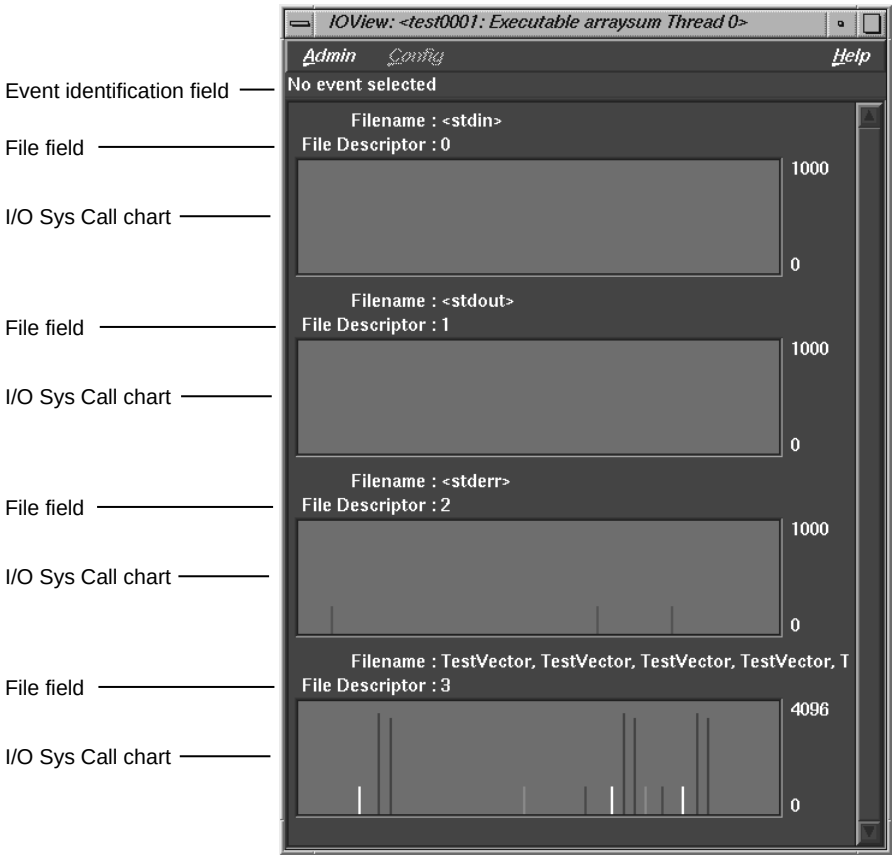

Figure 34. The `I/O View` Window

4.9 The `Call Graph View` Window

The `Call Graph View` window displays the functions as nodes, annotated with performance metrics, and their calls as connecting arcs (see Figure 35, page 77). Bring up the `Call Graph View` window by selecting `Call Graph View` from the `Views` menu.

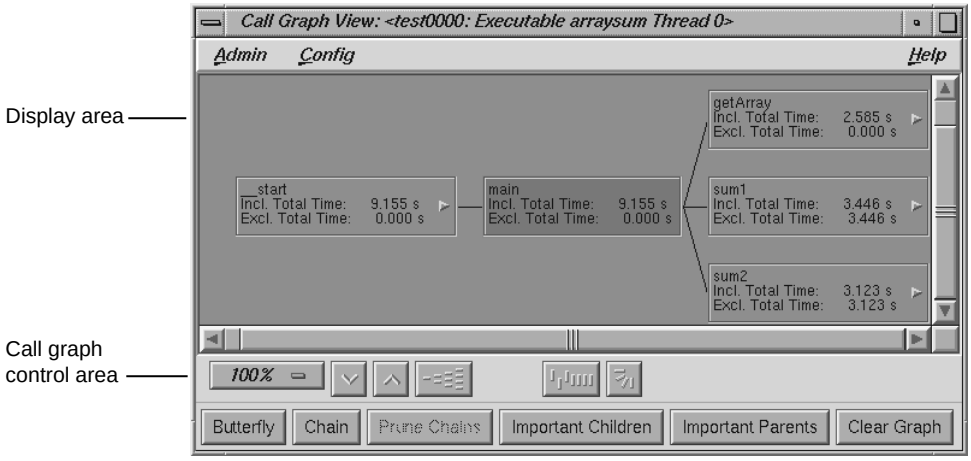

Figure 35. `Call Graph View` with Display Controls

Since a call graph can get quite complicated, the Performance Analyzer provides various controls for changing the graph display. The `Preferences` selection in the `Config` menu lets you specify which performance metrics display and also lets you filter out unused functions and arcs. There are two node menus in the display area; these let you filter nodes individually or as a selected group. The top row of display controls is common to all ProDev WorkShop graph displays. It lets you change scale, alignment, and orientation. See an overview in the *Developer Magic: ProDev WorkShop Overview*. The bottom row of controls lets you define the form of the graph. You can view the call graph as a butterfly graph, showing the functions that call and are called by a single function, or as a chain graph between two functions.

4.9.1 Special Node Icons

Although rare, nodes can be annotated with two types of graphic symbols:

- A right-pointing arrow in a node indicates an indirect call site. It represents a call through a function pointer. In such a case, the called function cannot be determined by the current methods.
- A circle in a node indicates a call to a shared library with a data-space jump table. The node name is the name of the routine called, but the actual target in the shared library cannot be identified. The table might be switched at run time, directing calls to different routines.

4.9.2 Annotating Nodes and Arcs

You can specify which performance metrics appear in the call graph, as described in the following sections.

4.9.2.1 Node Annotations

To specify the performance metrics that display inside a node, use the `Preferences` dialog box in the `Config` menu from the Performance Analyzer main view. (For an illustration of the `Preferences...` window, see Figure 27, page 61.)

4.9.2.2 Arc Annotations

Arc annotations are specified by selecting `Preferences...` from the `Config` menu in the `Call Graph View` window. (For an illustration of the `Preferences...` window, see Figure 27, page 61.) You can display the counts on the arcs (the lines between the functions). You can also display the percentage of calls to a function broken down by incoming arc. For an explanation of the performance metric items, see Section 4.4.6, page 59.

4.9.3 Filtering Nodes and Arcs

You can specify which nodes and arcs appear in the call graph as described in the following sections.

4.9.3.1 Call Graph Preferences Filtering Options

The `Preferences` selection in the `Call Graph View Config` menu also lets you hide functions and arcs that have 0 calls. See Figure 27, page 61.

4.9.3.2 Node Menu

There are two node menus for filtering nodes in the graph: the `Node` menu and the `Selected Nodes` menu. Both menus are shown in Figure 36, page 79.

The `Node` menu lets you filter a single node. It is displayed by holding down the right mouse button while the cursor is over the node. The name of the selected node appears at the top of the menu.

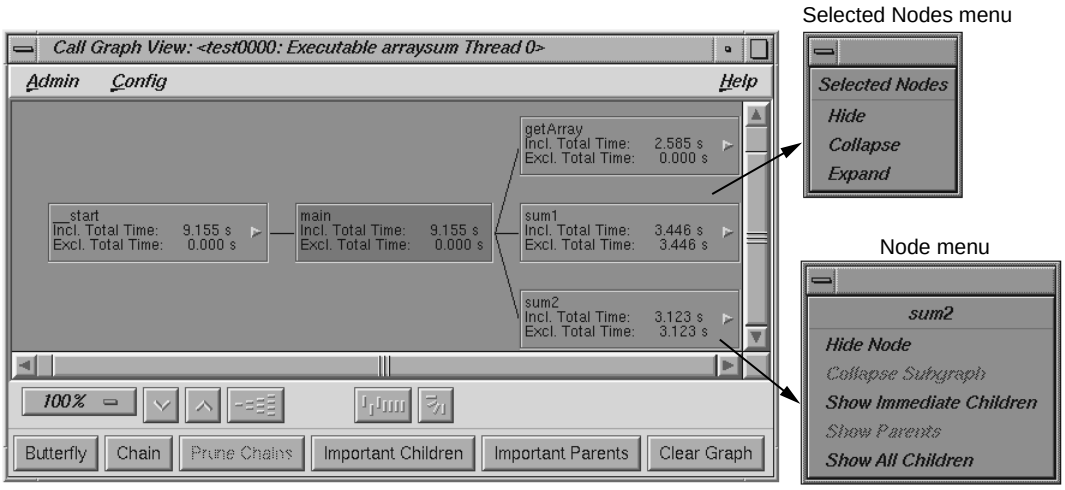

Figure 36. Node Menus

The `Node` menu selections are:

`Hide Node`	Removes the selected node from the call graph display.
`Collapse Subgraph`	Removes the nodes called by the selected node (and subsequently called nodes) from the call graph display.
`Show Immediate Children`	Displays the functions called by the selected node.
`Show Parents`	Displays all the functions that call the selected node.
`Show All Children`	Displays all the functions and the descendants called by the selected node.

4.9.3.3 `Selected Nodes` Menu

The `Selected Nodes` menu lets you filter multiple nodes. You can select multiple nodes by dragging a selection rectangle around them. You can also Shift-click a node, and it will be selected along with all the nodes that it calls. Holding down the right mouse button anywhere in the graph, except over a node, displays the `Selected Nodes` menu. The `Selected Nodes` menu selections are as follows:

`Hide`	Removes the selected nodes from the call graph display.
`Collapse`	Removes the nodes called by the selected nodes (and descendant nodes) from the call graph display.
`Expand`	Displays all the functions (descendants) called by the selected nodes.

4.9.3.4 Filtering Nodes through the Display Controls

The lower row of controls in the `Call Graph View` panel helps you reduce the complexity of a busy call graph.

You can perform these display operations:

`Butterfly`	Presents the call graph from the perspective of a single node (the target node), showing only those nodes that call it or are called by it. Functions that call it are displayed to the left and functions it calls are on the right. Selecting any node and clicking `Butterfly` redraws the display with the selected node in the center. The selected node is displayed and highlighted in the function list.
`Chain`	Lets you display all paths between a given source node and target node. The `Chain` dialog box is shown in Figure 37, page 81. You designate the source function by selecting it or entering it in the `Source Node` field and clicking the `Make Source` button. Similarly, the target function is selected or entered and then established by clicking the `Make Target` button. If you want to filter out paths that go through nodes and arcs with zero counts, click the toggle. After these selections are made, click `OK`.

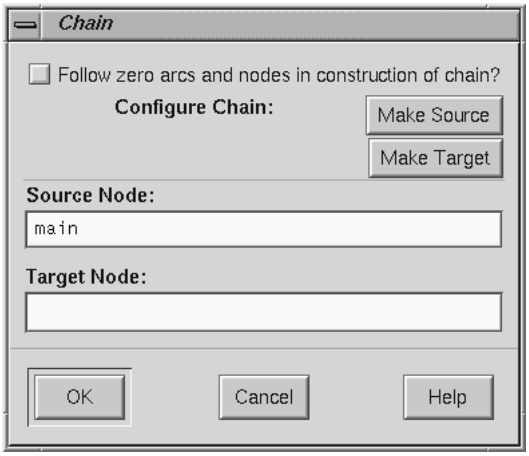

Figure 37. `Chain` Dialog Box

`Prune Chains` Displays a dialog box that provides two selections for filtering paths from the call graph (see Figure 38, page 81).

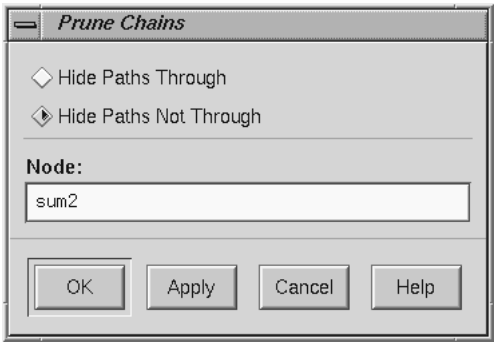

Figure 38. `Prune Chains` Dialog Box

The `Prune Chains` button is only activated when a chain mode operation has been performed. The dialog box selections are:

- The `Hide Paths Through` toggle removes from view all paths that go through the specified node. You must have a current node specified. Note that this operation is irreversible; you will not be able to redisplay the hidden paths unless you perform the `Chain` operation again.
- The `Hide Paths Not Through` toggle removes from view all paths except the ones that go through the specified node. This operation is irreversible.

`Important Children` Lets you focus on a function and its descendants and set thresholds to filter the descendants. You can filter the descendants either by percentage of the caller's time or by percentage of the total time. The `Threshold key` field identifies the type of performance time data used as the threshold. See Figure 39, page 82.

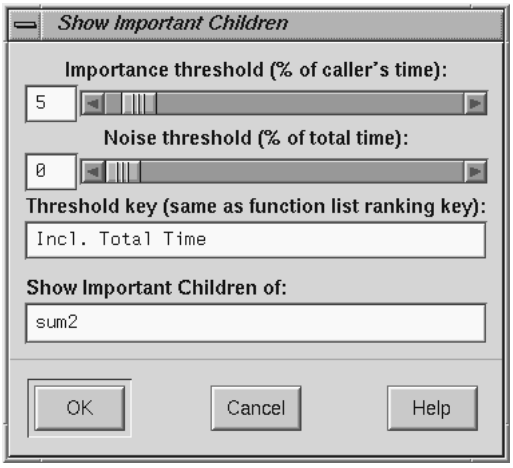

Figure 39. `Show Important Children` Dialog Box

`Important Parents` Lets you focus on the parents of a function, that is, the functions that call it. You can set thresholds to filter only those parents making a

significant number of calls, by percentage of the caller's time, or by percentage of the total time. The `Threshold key` field identifies the type of performance time data used as the threshold. See Figure 40, page 83.

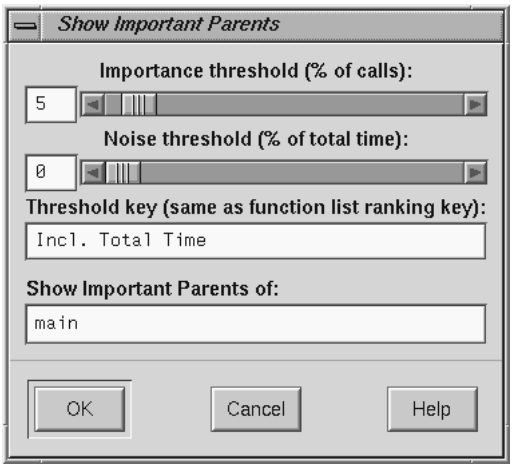

Figure 40. `Show Important Parents` Dialog Box

`Clear Graph` Removes all nodes and arcs from the call graph.

4.9.4 Other Manipulation of the Call Graph

The `Call Graph View` window provides facilities for changing the display of the call graph without changing the data content.

4.9.4.1 Geometric Manipulation through the Control Panel

The controls for changing the display of the call graph are in the upper row of the control panel (see Figure 41, page 84).

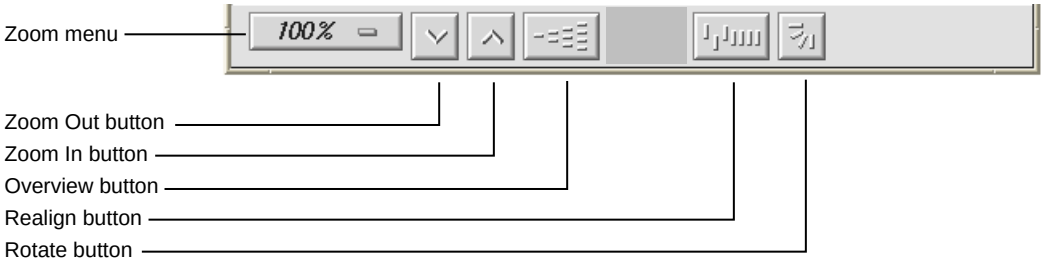

Figure 41. `Call Graph View` Controls for Geometric Manipulation

These facilities are:

`Zoom menu` button	Shows the current scale of the graph. If you click this button, a pop-up menu appears displaying other available scales. The scaling range is between 15% and 200% of the normal (100%) size.
`Zoom Out` button	Resets the scale of the graph to the next (available) smaller size in the range.
`Zoom In` button	Resets the scale of the graph to the next (available) larger size in the range.
`Overview` button	Invokes an overview pop-up display that shows a scaled down representation of the graph. The nodes appear in the analogous places on the overview pop-up, and a white outline can be used to position the main graph relative to the pop-up. Alternatively, the main graph may be repositioned by using its scroll bars.
`Realign` button	Redraws the graph, restoring the positions of any nodes that were repositioned.
`Rotate` button	Flips the orientation of the graph between horizontal (calling nodes at the left) and vertical (calling nodes at the top).

For more information on the graphical controls, see the *Developer Magic: ProDev WorkShop Overview* manual.

4.9.4.2 Using the Mouse in the `Call Graph View`

You can move an individual node by dragging it using the middle mouse button. This helps reveal obscured arc annotations.

You can select multiple nodes by dragging a selection rectangle around them. Shift-clicking a node selects the node along with all the nodes that it calls.

4.9.4.3 Selecting Nodes from the Function List

You can select functions from the function list of the Performance Analyzer window to be highlighted in the call graph. Select a node from the list and then click the `Show Node` button in the `Function List` window. The node will be highlighted in the graph.

4.10 Butterfly View

The `Butterfly View` shows a selected function, the functions that called it (the `Immediate Parents`), and the functions it calls (the `Immediate Children`). For an illustration, see Figure 7, page 13.

You can change the selected function by clicking on a new one in the function list area of the main Performance Analyzer window.

The `Attrib.%` column shows the percentage of the sort key (inclusive time, in the illustration) attributed to each caller or callee. The sort key varies according to the view; on an `I/O View`, for instance, it is by default inclusive bytes read. You can change the criteria for what is displayed in the columns and how the list is ordered by using the `Preferences...` and `Sort...` windows, both of which are accessed through the `Config` menu on the main Performance Analyzer menu.

You can display the addresses from which the caller functions call the selected function, and from which the selected function calls the callee functions, by `Show All Arcs Individually` on the `Config` menu. You can view the source code for the selected function (and the rest of the program) by clicking on the `Source` button.

If you want to save the data as text, select `Save As Text...` from the `Admin` menu.

4.11 Analyzing Memory Problems

The Performance Analyzer provides four tools for analyzing memory problems: `Malloc Error View`, `Leak View`, `Malloc View`, and `Heap View`. Setting up and running a memory analysis experiment is the same for all four tools. After you have conducted the experiment, you can apply any of these tools.

4.11.1 Conducting Memory Leak Experiments

To look for memory leaks or bad `free` routines, or to perform other analysis of memory allocation, run a Performance Analyzer experiment with `Memory Leak Trace` specified as the experiment task. You run a memory corruption experiment like any performance analysis experiment, by clicking `Run` in the Debugger Main View. The Performance Analyzer keeps track of each `malloc` (memory allocation), `realloc` (reallocation of memory), and `free`. The general steps in running a memory experiment are as follows:

1. Display the WorkShop Debugger, including the executable file (`generic`, in this case, from the `/usr/demos/SpeedShop` directory) as an argument.

   ```
   cvd generic &
   ```

2. Specify `Memory Leak Trace` as the experiment task.

   `Memory Leak Trace` is a selection on the `Perf` menu.

3. Run the experiment.

   You run experiments by clicking the `Run` button.

4. The `Performance Analyzer` window is displayed automatically with the experiment information.

   The `Performance Analyzer` window displays results appropriate to the task selected. Figure 42, page 87, shows the `Performance Analyzer` window after a memory experiment.

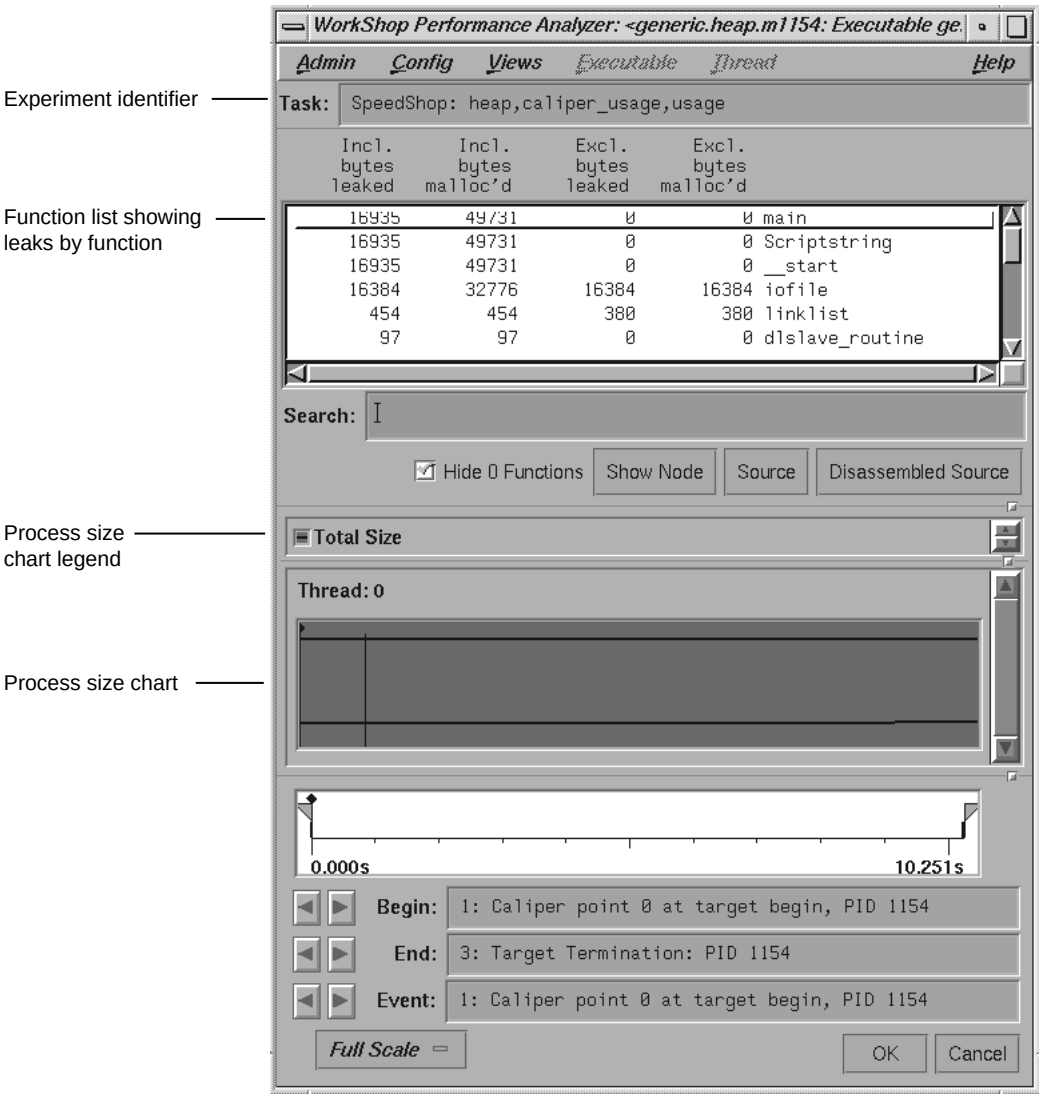

Figure 42. `Performance Analyzer` Window Displaying Results of a Memory Experiment

The function list displays inclusive and exclusive bytes leaked and allocated with `malloc` per function. Clicking `Source` brings up the `Source View`, which displays the function's source code annotated with bytes leaked and allocated by `malloc`. You can set other annotations in `Source View` and

the function list by choosing `Preferences...` from the `Config` menu in the `Performance Analyzer` window and selecting the desired items.

5. Analyze the results of the experiment in `Leak View` when doing leak detection and `Malloc Error View` when performing broader memory allocation analysis. To see all memory operations, whether problems or not, use `Malloc View`. To view memory problems within the memory map, use `Heap View`. See the following section for more information.

4.11.2 Using `Malloc Error View`, `Leak View`, and `Malloc View`

After you have run a memory experiment using the Performance Analyzer, you can analyze the results using `Malloc Error View` (see Figure 43, page 89), `Leak View` (see Figure 44, page 90), or `Malloc View` (see Figure 45, page 90). `Malloc View` is the most general, showing all memory allocation operations. `Malloc Error View` shows only those memory operations that caused problems, identifying the cause of the problem and how many times it occurred. `Leak View` displays each memory leak that occurs in your executable, its size, the number of times the leak occurred at that location during the experiment, and the corresponding call stack (when you select the leak).

Each of these views has three major areas:

- Identification area—This indicates which operation has been selected from the list. `Malloc View` identifies `malloc` routines, indicating the number of `malloc` locations and the size of all `malloc` operations in bytes. `Malloc Error View` identifies leaks and bad `free` routines, indicating the number of error locations and how many errors occurred in total. `Leak View` identifies leaks, indicating the number of leak locations and the total number of bytes leaked.
- List area—This is a list of the appropriate types of memory operations according to the type of view. Clicking an item in the list identifies it at the top of the window and displays its call stack at the bottom of the list. The list displays in order of size.
- Call stack area— This displays the contents of the call stack when the selected memory operation occurred. Figure 46, page 91, shows a typical `Source View` window with leak annotations. (You can change the annotations by using the `Preferences...` selection in the `Performance Analyzer Config` menu). Colored boxes draw attention to high counts.

Note: As an alternative to viewing leaks in `Leak View`, you can save one or more memory operations as a text file. Choose `Save As Text...` from the `Admin` menu, select one or more entries, and view them separately in a text file along with their call stacks. Multiple items are selected by clicking the first and then either dragging the cursor over the others or shift-clicking the last in the group to be selected.

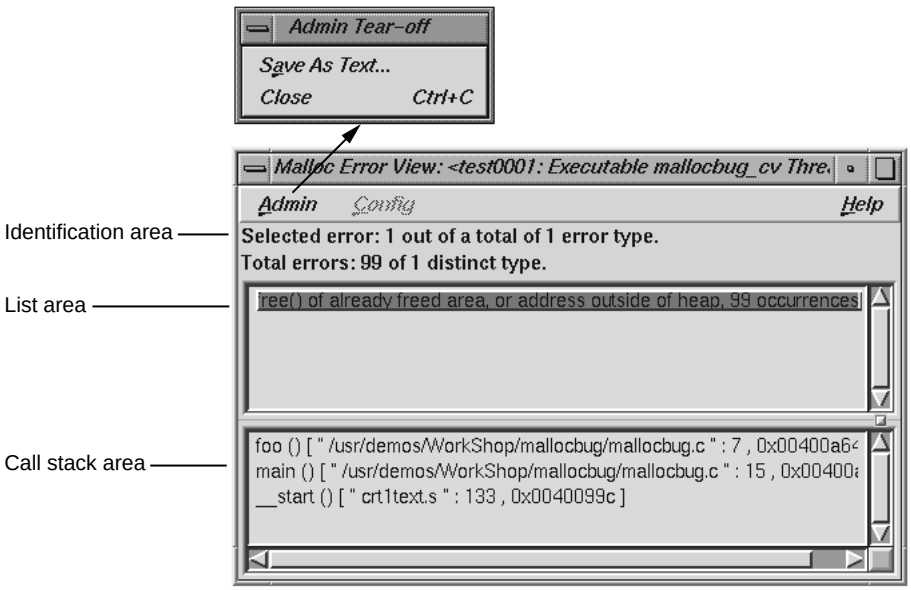

Figure 43. `Malloc Error View` Window with an `Admin` Menu

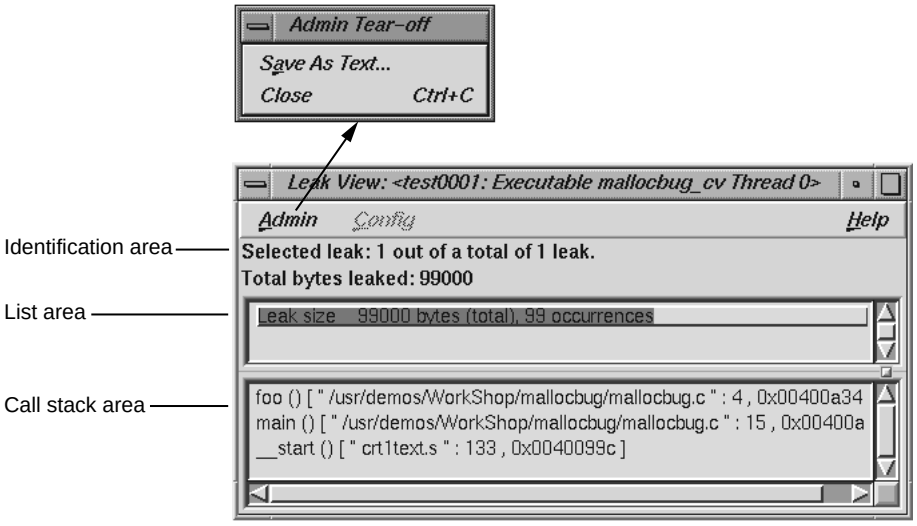

Figure 44. `Leak View` Window with an `Admin` Menu

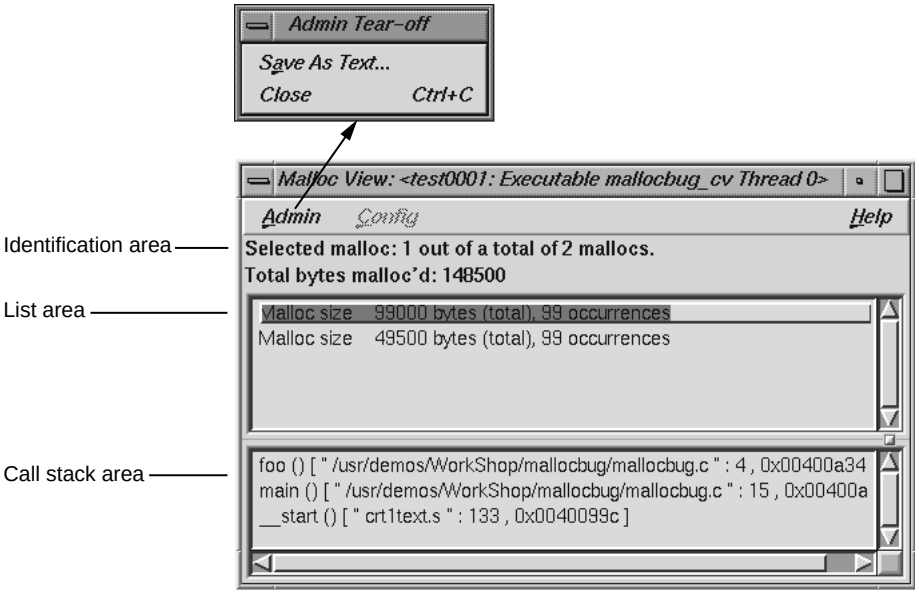

Figure 45. Malloc View Window with Admin Menu

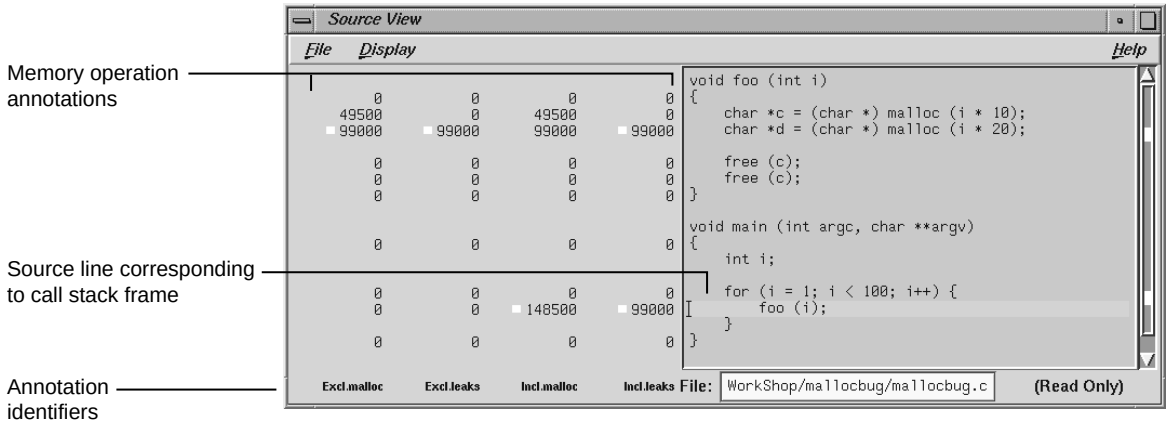

Figure 46. `Source View` Window with Memory Analysis Annotations

4.11.3 Analyzing the Memory Map with `Heap View`

`Heap View` lets you analyze data from experiments based on the `Memory Leak Trace` task. The `Heap View` window provides a memory map that shows memory problems occurring in the time interval defined by the calipers in the `Performance Analyzer` window. The map indicates the following memory block conditions:

- `malloc`—reserved memory space
- `realloc`—reallocated space
- `free`—open space
- error—bad `free` space
- unused space

In addition to the `Heap View` memory map, you can analyze memory leak data using these other tools:

- If you select a memory problem in the map and bring up the `Call Stack` window, it will show you where the selected problem took place and the state of the call stack at that time.
- The `Source View` window shows exclusive and inclusive `malloc` routines and leaks and the number of bytes used by source line.

4.11.3.1 `Heap View` Window

A typical `Heap View` window with its parts labeled appears in Figure 47, page 92.

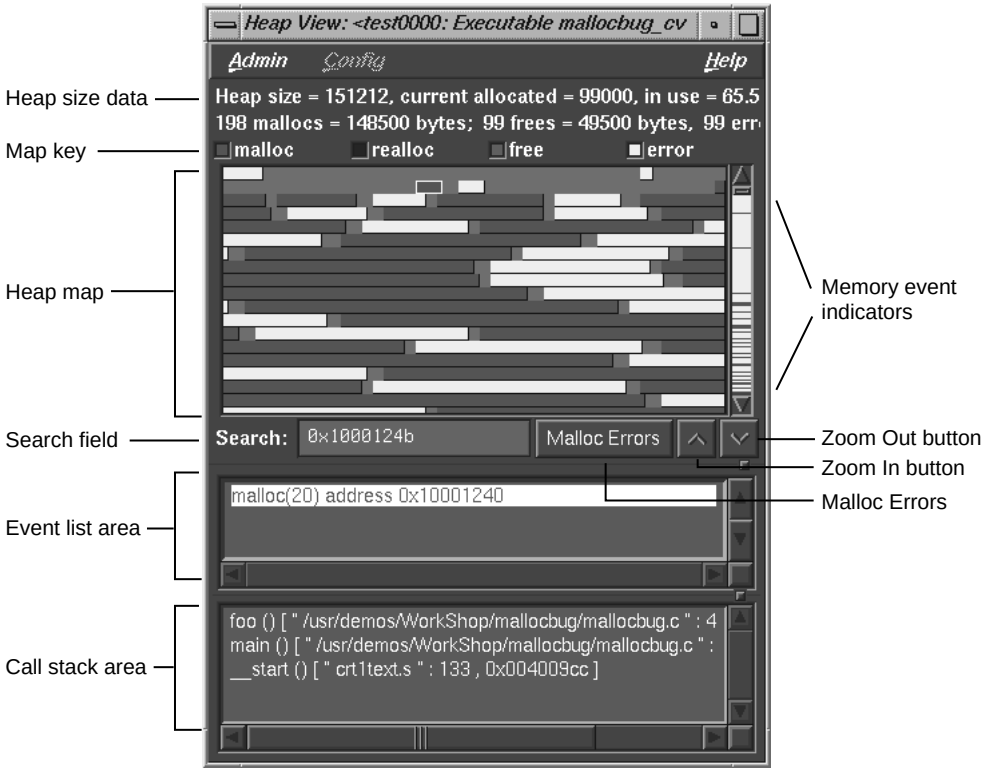

Figure 47. `Heap View` Window

The major features of a `Heap View` window are as follows:

Map key	Appears at the top of the heap map area to identify blocks by color. The actual colors depend on your color scheme.
Heap map	Shows heap memory as a continuous, wrapping horizontal rectangle. The memory addresses begin at the upper left corner and progress from left to right, row by row. The rectangle is broken

	up into color-coded segments according to memory use status. Clicking a highlighted area in the heap map identifies the type of problem, the memory address where it occurred, its size in the event list area, and the associated call stack in the call stack display area. Note in Figure 47, page 92, that there are only a few problems in the memory at the lower addresses and many more at the higher addresses.
Memory event indicators	The events appear color-coded in the scroll bar. Clicking an indicator with the middle button scrolls the display to the selected problem.
`Search` field	Provides two functions: • If you enter a memory address in the field, the corresponding position will be highlighted in the heap map. If there was a problem at that location, it will be identified in the event list area. If there is no problem, the event list area displays the address at the beginning of the memory block and its size. • If you hold down the left mouse button and position the cursor in the heap map, the corresponding address will display in the `Search` field.
Event list area	Displays the events occurring in the selected block. If only one event was received at the given address, its address is shown by default. If more than one event is shown, double-clicking an event will display its corresponding call stack.
Call stack area	Displays the call stack corresponding to the event highlighted in the event list area.
`Malloc Errors` button	Causes `malloc` errors and their addresses to display in the event list area. You can then enter the address of the `malloc` error in the `Search` field and press the `Enter` key to see the error's `malloc` information and its associated call stack.

`Zoom in` button	An upward-pointing arrow, it redisplays the heap area at twice the current size of the display. If you reach the limit, an error message displays.
`Zoom out` button	A downward-pointing arrow, it redisplays the heap area at half the current size (to a limit of one pixel per byte). If you reach the limit, an error message displays.

4.11.3.2 `Source View` `malloc` Annotations

Like `Malloc View`, if you double-click a line in the call stack area of the `Heap View` window, the `Source View` window displays the portion of code containing the corresponding line. The line is highlighted and indicated by a caret (^), with the number of bytes used by `malloc` in the annotation column. See Figure 46, page 91.

4.11.3.3 Saving `Heap View` Data as Text

Selecting `Save As Text...` from the `Admin` menu in `Heap View` lets you save the heap information or the event list in a text file. When you first select `Save As Text...`, a dialog box displays asking you to specify heap information or the event list. After you make your selection, the `Save Text` dialog box displays (see Figure 48, page 95). This lets you select the file name in which to save the `Heap View` data. The default file name is *experiment-filename*`.out`. When you click `OK`, the data for the current caliper setting and the list of unmatched `free` routines, if any, are appended to the specified file.

Note: The `Save As Text...` selection in the `File` menu for the `Source View` from `Heap View` saves the current file. No file name default is provided, and the file that you name will be overwritten.

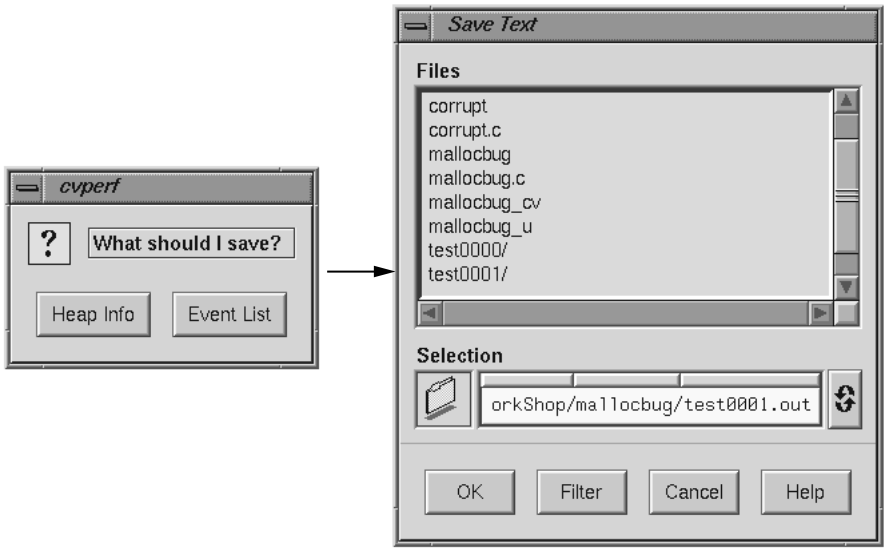

Figure 48. `Heap View Save Text` Dialog Boxes

4.11.4 Memory Experiment Tutorial

In this tutorial, you will run an experiment to analyze memory usage. The following short program generates memory problems that demonstrate how you can use the Performance Analyzer to detect memory problems.

1. Go to the `/usr/demos/WorkShop/mallocbug` directory. The executable `mallocbug` was compiled as follows:

   ```
   cc -g -o mallocbug mallocbug.c  -lc
   ```

2. Invoke the Debugger by typing:

   `cvd mallocbug`

3. Bring up a list of the performance tasks by selecting `Select Task` from the `Perf` menu.

4. Select `Memory Leak Trace` from the menu and click `Run` to begin the experiment. The program runs quickly and terminates.

5. The `Performance Analyzer` window appears automatically. A dialog box indicating `malloc` errors displays also.

6. Select `Malloc View` from the Performance Analyzer `Views` menu.

   The `Malloc View` window displays, indicating two `malloc` locations.

7. Select `Malloc Error View` from the Performance Analyzer `Views` menu.

   The `Malloc Error View` window displays, showing one problem, a bad `free`, and its associated call stack. This problem occurred 99 times

8. Select `Leak View` from the Performance Analyzer `Views` menu.

   The `Leak View` window displays, showing one leak and its associated call stack. This leak occurred 99 times for a total of 99,000 leaked bytes.

9. Double-click the function `foo` in the call stack area.

   The `Source View` window displays, showing the function's code, annotated by the exclusive and inclusive leaks and the exclusive and inclusive calls to `malloc`.

10. Select `Heap View` from the Performance Analyzer `Views` menu.

    The `Heap View` window displays the heap size and other information at the top. The heap map area of the window shows the heap map as a continuous, wrapping horizontal rectangle. The rectangle is broken up into color-coded segments, according to memory use status. The color key at the top of the heap map area identifies memory usage as `malloc`, `realloc`, `free`, or an error, or bad `free`. Notice also that color-coded indicators showing `malloc`, `realloc`, and bad `free` routines are displayed in the scroll bar trough. At the bottom of the heap map area are: the `Search` field, for identifying or finding memory locations; the `Malloc Errors` button, for finding memory problems; a `Zoom In` button (upward pointing arrow) and a `Zoom Out` button (downward pointing arrow).

    The event list area and the call stack area are at the bottom of the window. Clicking any event in the heap map area displays the appropriate information in these fields.

11. Click on any memory block in the heap map.

    The beginning memory address appears in the`Search` field. The event information displays in the event list area. The call stack information for the last event appears in the call stack area.

12. Select other memory blocks to try out this feature.

As you select other blocks, the data at the bottom of the `Heap View` window changes.

13. Double-click on a frame in the call stack area.

    A `Source View` window comes up with the corresponding source code displayed.

14. Close the `Source View` window.

15. Click the `Malloc Errors` button.

    The data in the `Heap View` information window changes to display memory problems. Note that a `free` may be unmatched within the analysis interval, yet it may have a corresponding `free` outside of the interval.

16. Click `Close` to leave the `Heap View` window.

17. Select `Exit` from the `Admin` menu in any open window to end the experiment.

4.12 The `Call Stack` Window

The `Call Stack` window, which is accessed from the Performance Analyzer `Views` menu, lets you get call stack information for a sample event selected from one of the Performance Analyzer views. See Figure 49, page 98.

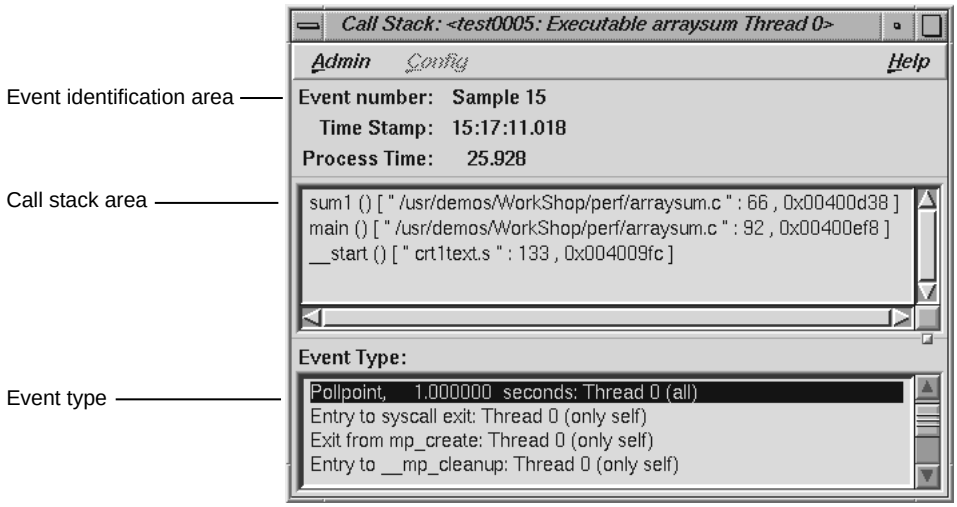

Figure 49. Performance Analyzer `Call Stack` Window

There are three main areas in the `Call Stack` window:

Event identification area

Displays the number of the event, its time stamp, and the time within the experiment. If you have a multiprocessor experiment, the thread will be indicated here.

Call stack area

Displays the contents of the call stack when the sample event took place.

Event type area

Highlights the type of event and shows the thread in which it was defined. It indicates, in parentheses, whether the sample was taken in all threads or the indicated thread only.

4.13 Analyzing Working Sets

If you suspect a problem with frequent page faults or instruction cache misses, conduct a working set analysis to determine if rearranging the order of your functions will improve performance. The term *working set* refers to those

executable pages, functions, and instructions that are actually brought into memory during a phase or operation of the executable. If more pages are required than can fit in memory at the same time, *page thrashing* (that is, swapping in and out of pages) may result, slowing down your program. Strategic selection of which pages functions appear on can dramatically improve performance in such cases. You do this by creating a file containing a list of functions, their sizes, and addresses called a *cord mapping file*. The functions should be ordered so as to optimize page swapping efficiency. This file is then fed into the `cord` utility, which rearranges the functions according to the order suggested in the `cord` mapping file. See the man page for `cord`(1).

Working set analysis is appropriate for:

- Programs that runs for a long time.
- Programs whose operation comes in distinct phases.
- Distributed shared objects (DSOs) that are shared among several programs.

4.13.1 Working Set Analysis Overview

WorkShop provides two tools to help you conduct working set analysis:

- `Working Set View` is part of the Performance Analyzer. It displays the working set of pages for each DSO that you select and indicates the degree to which the pages are used.
- The cord analyzer, `sscord`(1), is separate from the Performance Analyzer and is invoked by typing **`sscord`** at the command line. It displays a list of the working sets that make up a cord mapping file, shows their utilization efficiency, and, most importantly, computes an optimized ordering to reduce working sets.

Figure 50, page 100, presents an overview of the process of conducting working set analysis.

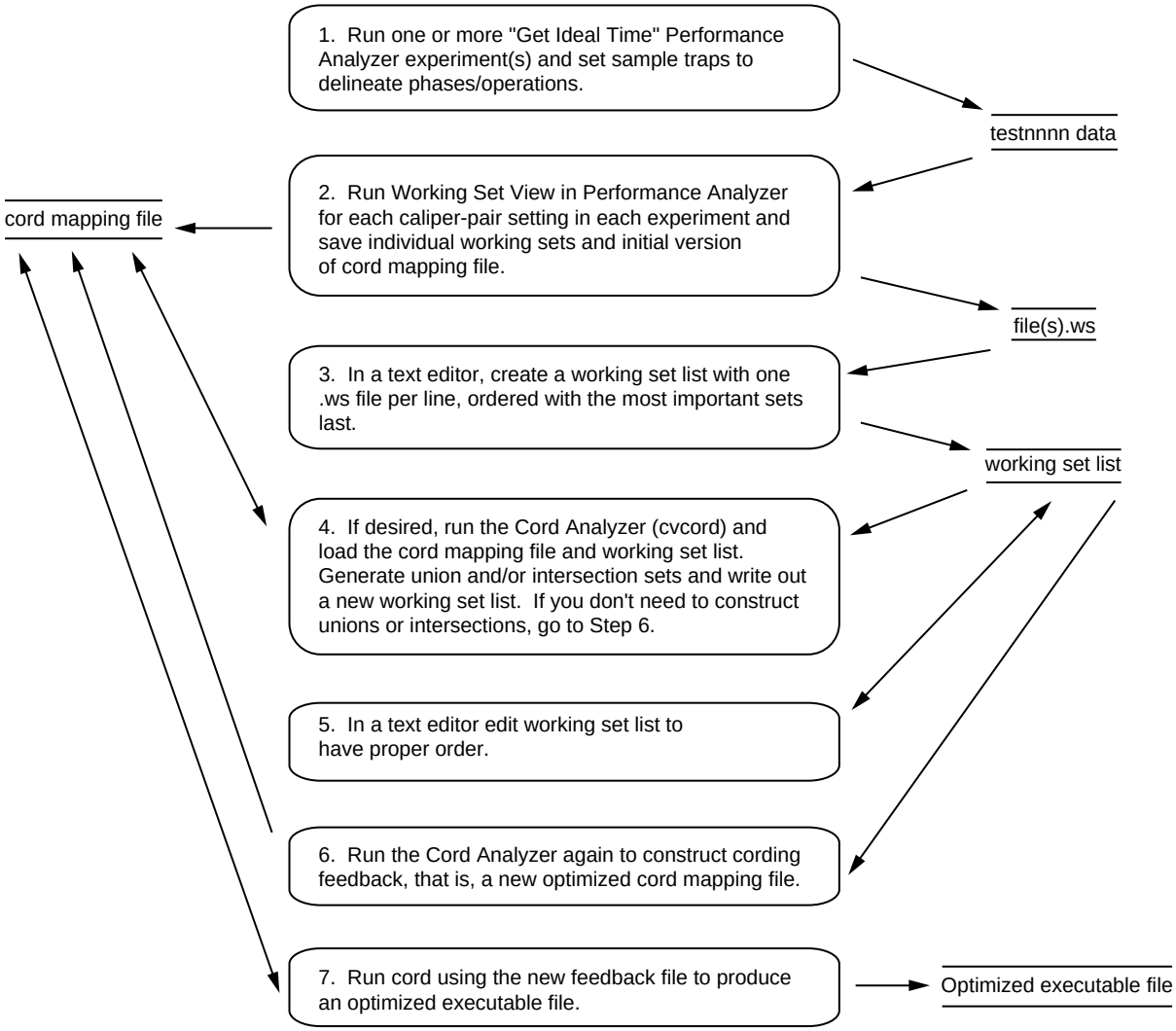

Figure 50. Working Set Analysis Process

First, conduct one or more Performance Analyzer experiments using the `Ideal Time/Pixie` task. Set sample traps at the beginning and end of each operation or phase that represents a distinct task. You can run additional experiments on the same executable to collect data for other situations in which it can be used.

After you have collected the data for the experiments, run the Performance Analyzer and select `Working Set View`. Save the working set for each phase or operation that you want to improve. Do this by setting the calipers to bracket each phase and select `Save Working Set` from the `Admin` menu.

Select `Save Cord Map File` to save the cord mapping file (for all runs and caliper settings). This need only be done once.

The next step is to create the *working set list file*, which contains all of the working sets you want to analyze using the cord analyzer. Create the working set list file in a text editor, specifying one line for each working set and in reverse order of priority, that is, the most important comes last.

The working set list and the cord mapping file serve as input to the cord analyzer. The working set list provides the cord analyzer with working sets to be improved. The cord mapping file provides a list of all the functions in the executable. The cord analyzer displays the list of working sets and their utilization efficiency. It lets you do the following:

- Construct gray-code cording feedback (the preferred method).
- Examine the page layout and the efficiency of each working set with respect to the original ordering of the executable.
- Construct union and intersection sets as desired.
- View the efficiency of a different ordering.
- Construct a new cord mapping file as input to the `cord` utility.

If you have a new order that you would like to try out, edit your working set list file in the desired order, submit it to the cord analyzer, and save a new cord mapping file for input to `cord`.

4.13.2 Working Set View

The `Working Set View` measures the coverage of the dynamic shared objects (DSOs) that make up your executable (see Figure 51, page 102). It indicates instructions, functions, and pages that were not used when the experiment was run. It shows the coverage results for each DSO in the DSO list area. Clicking a DSO in the list displays its pages with color-coding to indicate the coverage of the page.

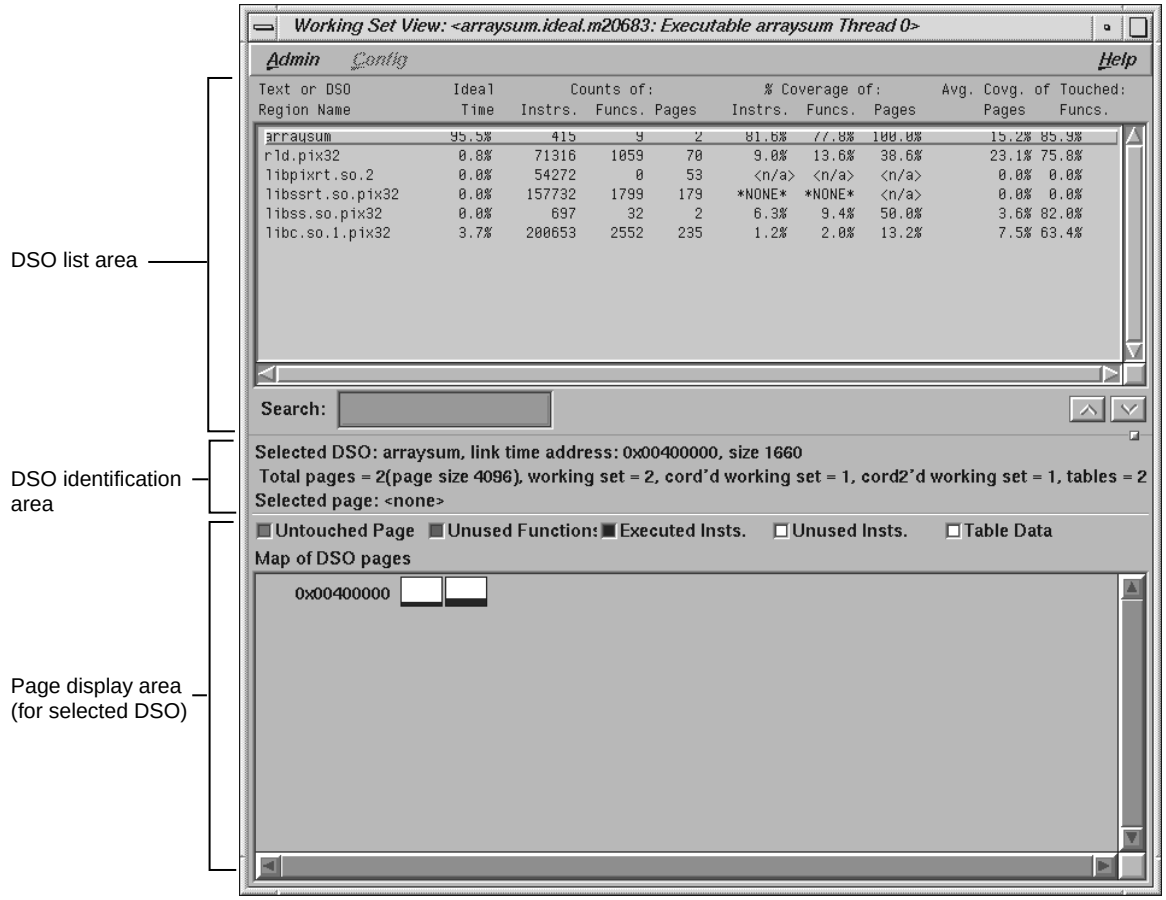

Figure 51. Working Set View

4.13.2.1 DSO List Area

The DSO list area displays coverage information for each DSO used by the executable. It has the following columns:

`Text or DSO Region Name`

Identifies the DSO.

`Ideal Time`

Lists the percentage of ideal time for the caliper setting attributed to the DSO.

`Counts of: Instrs.`

Lists the number of instructions contained in the DSO.

`Counts of: Funcs.`

Lists the number of functions contained in the DSO.

`Counts of: Pages`

Lists the number of pages occupied by the DSO.

`% Coverage of: Instrs.`

Lists the percentage obtained by dividing the number of instructions used by the total number of instructions in the DSO.

`% Coverage of: Funcs.`

Lists the percentage obtained by dividing the number of functions used by the total number of functions in the DSO.

`% Coverage of: Pages`

Lists the coverage obtained by dividing the number of pages touched by the total pages in the DSO.

`Avg. Covg. of Touched: Pages`

Lists the coverage obtained by dividing the number of instructions executed by the total number of instructions on those pages touched by the DSO.

`Avg. Covg. of Touched: Funcs`

Lists the average percentage use of instructions within used functions.

The `Search` field lets you perform incremental searches to find DSOs in the DSO list. (An incremental search goes to the immediately matching target as you enter each character.)

4.13.2.2 DSO Identification Area

The DSO identification area shows the address, size, and page information for the selected DSO. It also displays the address, number of instructions, and coverage for the page selected in the page display area.

4.13.2.3 Page Display Area

The page display area at the bottom of the `Working Set View` window shows all the pages in the DSO and indicates untouched pages, unused functions, executed instructions, unused instructions, and table data (related to `rld`(1)). It also includes a color legend at the top to indicate how pages are used.

Clicking a page displays its address, number of instructions, and coverage data in the identification area. Clicking a function in the function list of the main Performance Analyzer window highlights (using a solid rectangle) the page on which the function begins. Clicking the left mouse button on a page indicates the first function on the page by highlighting it in the function list area of the Performance Analyzer window. Similarly, clicking the middle button on a page highlights the function at the middle of the page, and clicking the right button highlights the button at the end of the page. For all three button clicks, the page containing the beginning of the function becomes highlighted. Note that left clicks typically highlight the page before the one clicked, since the function containing the first instruction usually starts on the previous page.

4.13.2.4 `Admin` Menu

The `Admin` menu of the `Working Set View` window provides the following menu selections:

`Save Working Set`

Saves the working set for the selected DSO. You can incorporate this file into a working set list file to be used as input to the Cord Analyzer.

`Save Cord Map File`

Saves all of the functions in the DSOs in a cord mapping file for input to the Cord Analyzer. This file corresponds to the feedback file discussed in the reference page for `cord`.

`Save Summary Data as Text`

Saves a text file containing the coverage statistics in the DSO list area.

`Save Page Data as Text`

Saves a text file containing the coverage statistics for each page in the DSO.

`Save All Data as Text`

Saves a text file containing the coverage statistics in the DSO list area and for each page in the selected DSO.

`Close`

Closes the `Working Set View` window.

4.13.3 Cord Analyzer

The cord analyzer is not actually part of the Performance Analyzer; it is discussed in this part of the manual because it works in conjunction with the `Working Set View`. The cord analyzer lets you explore the working set behavior of an executable or shared library (DSO). With it you can construct a feedback file for input to the `cord`(1) utility to generate an executable with improved working set behavior. Invoke the cord analyzer at the command line using the following syntax:

`sscord -fb` *fb_file* `-wsl` *ws_list_file* `-ws` *ws_file* `-v|-V` *executable*

The `sscord` command accepts the following arguments:

`-fb` *fb_file*	Specifies a single text file to use as a feedback file for the executable. It should have been generated either from a Performance Analyzer experiment on the executable or DSO, or from the cord analyzer. If no `-fb` argument is given, the feedback file name will be generated as *executable*`.fb`.
`-wsl` *ws_list_file*	Specifies a single text file name as input; the working set list consists of the working set files whose names appear in the input file. Each file name should be on a single line.
`-ws` *ws_file*	Specifies a single working set file name.
`-v|-V`	Verbose output. If specified, mismatches between working sets and the executable or DSO are noted.
executable	Specifies a single executable file name as input.

The `Cord Analyzer` window is shown in Figure 52, page 107, with its major areas and menus labeled.

4.13.3.1 Working Set Display Area

The working set display area of the `Cord Analyzer` window shows all of the working sets included in the working set list file. It has the following columns:

`Working-set pgs. (util. %)`

Lists the number of pages in the working set and the percentage of page space that is utilized.

`cord'd set pgs`

Specifies the minimum number of pages for this set, that is, the number of pages the working set would occupy if the program or DSO were reordered optimally for that specific working set.

`Working-set Name`

Identifies the path for the working set.

Note that when the function list is displayed, double-clicking a function displays a plus sign (+) in the working set display area to the left of any working sets that contain the function.

4.13.3.2 Working Set Identification Area

The working set identification area shows the name of the selected working set. It also shows the number of pages in the working set list, in the selected working set, and in the `cord`ed working set, and the number of pages used as tables. It also provides the address for the selected page, its size, and its coverage as a percentage.

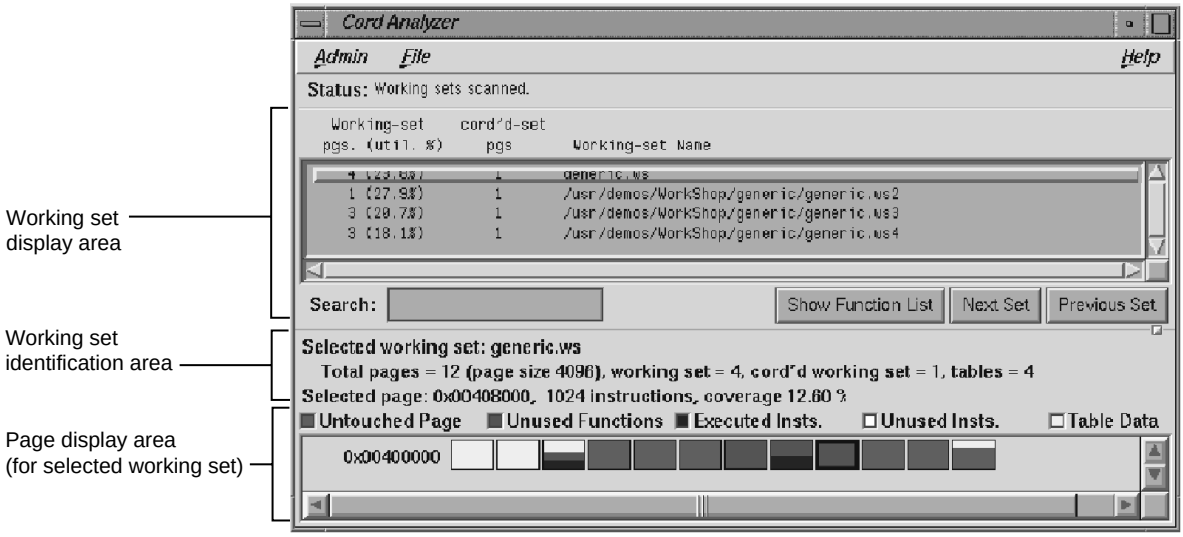

Figure 52. The `Cord Analyzer` Window

4.13.3.3 Page Display Area

The page display area at the bottom of the window shows the starting address for the DSO and its pages, and their use in terms of untouched pages, unused functions, executed instructions, unused instructions, and table data (related to `rld`). It includes a color legend at the top to indicate how pages are used.

4.13.3.4 Function List

The `Function List` window displays all the functions in the selected working set. It contains the following columns:

`Use`	Count of the working sets containing the function.
`Address`	Starting address for the function.
`Insts.`	Number of instructions in the function.
`Function (File)`	Name of the function and the file in which it occurs.

When the `Function List` window is displayed, clicking a working set in the working set display area displays a plus sign (+) in the function list to the left of any functions that the working set contains. Similarly, double-clicking a

function displays a plus sign in the working set display area to the left of any working sets that contain the function.

The `Search` field lets you do incremental searches for a function in the `Function List`.

4.13.3.5 `Admin` Menu

The `Admin` menu contains the standard `Admin` menu commands in WorkShop views (see Appendix A, in the *Developer Magic: Debugger User's Guide*). It has the following command specific to the cord analyzer:

`Save Working Set List`

Saves a new working set list with whatever changes you made to it in the session.

4.13.3.6 `File` Menu

The `File` menu contains the following selections:

`Delete All Working Sets`

Removes all the working sets from the working set list. It does not delete any files.

`Delete Selected Working Set`

Removes the selected working set from the working set list.

`Add Working Set`

Includes a new working set in the working set list.

`Add Working Set List from File`

Adds the working sets from the specified list to the current working set file.

`Construct Gray-code Cording Feedback`

Generates an ordering to minimize the working sets for the highest priority set first. It then orders it to minimize the transitions between the first set and the second, then compacting the second, and minimizing the transitions between it and the third, and so on. Gray code is believed to be superior

to weighted ordering, but you might want to experiment with them both.

`Construct Weighted Cording Feedback`

Finds as many distinct affinity sets as it can and orders them to minimize the working sets for their operations in a weighted priority order.

`Construct Union of Selected Sets`

Displays a new working set built as a union of working sets. This is the same as an `OR` of the working sets.

`Construct Intersection of Selected Sets`

Displays a new working set built from the intersection of the specified working sets. This is the same as an `AND` of the working sets.

`Read Feedback File`

Loads a new cord mapping file into the Cord Analyzer.

Using Tester [5]

This chapter describes the Tester usage model. It shows the general approach of applying Tester for coverage analysis. It contains these sections:

5.1 Tester Overview

WorkShop Tester is a UNIX-based software quality assurance toolset for dynamic test coverage over any set of tests. The term *covered* means the test has executed a particular unit of source code. In this product, units are functions, individual source lines, arcs, blocks, or branches. If the unit is a branch, covered means it has been executed under both true and false conditions. This product is intended for software and test engineers and their managers involved in the development, test, and maintenance of long-lived software projects.

WorkShop Tester provides these general benefits:

- Provides visualization of coverage data, which yields immediate insight into quality issues at both engineering and management levels
- Provides useful measures of test coverage over a set of tests/experiments
- Lets you view the coverage results of a dynamically shared object (DSO) by executables that use it
- Provides comparison of coverage over different program versions
- Provides tracing capabilities for arguments and function arcs that go beyond traditional test coverage tools
- Supports programs written in C, C++, and Fortran
- Is integrated into the CASEVision family of products
- Allows users to build and maintain higher quality software products

There are two versions of Tester:

- `cvcov` is the command line version of the test coverage program.
- `cvxcov` is the GUI version of the test coverage program.

Most of the functionality is available from either program, although the graphical representations of the data are available only from `cvxcov`, the GUI tool.

5.1.1 Test Coverage Data

Tester provides the following basic coverage:

- Basic block—how many times was this basic block executed?
- Function—how many times was this function executed?
- Branch—did this condition take on both TRUE and FALSE values?

You can also request the following coverage information:

- Arc—was function `F` called by function `A` and function `B`? Which arcs for function `F` were **not** taken?
- Source line coverage—how many times has this source line been executed and what percentage of source lines is covered?
- Argument—what were the maximum and minimum values for argument `X` in function `F` over all tests?
- When the target program `execs`, `forks`, or `sprocs` another program, only the main target is tested, unless you specify which executables are to be tested, the parent and/or child programs.

  Note: When you compile with the `-g` flag, you may create assembly blocks and branches that can never be executed, thus preventing "full" coverage from being achieved. These are usually negligible. However, if you compile with the`01` flag (the default), you can increase the number of executable blocks and branches.

5.1.2 Types of Experiments

You can conduct Tester coverage experiments for:

- Separate tests
- A set of tests operating on the same executable
- A list of executables related by `fork`, `exec`, or `sproc` commands

- A test group of executables sharing a common dynamically shared object (DSO)

5.1.3 Experiment Results

Tester presents the experiment results in these reports:

- Summary of test coverage, including user parameterized dynamic coverage metric
- List of functions, which can be sorted by count, file, or function name and filtered by percentage of block, branch, or function covered
- Comparison of test coverage between different versions of the same program
- Source or assembly code listing annotated with coverage data
- Breakdown of coverage according to contribution by tests within a test set or test group

The graphical user interface lets you view test results in different contexts to make them more meaningful. It provides:

- Annotated function call graph highlighting coverage by counts and percentage (ASCII function call graph supported as well)
- Annotated Source View showing coverage at the source language level
- Annotated Disassembly View showing coverage at the assembly language level
- Bar chart summary showing coverage by functions, lines, blocks, branches, and arcs

5.1.4 Multiple Tests

Tester supports multiple tests. You can:

- Define and run a test set to cover the same program.
- Define and run a test group to cover programs sharing a common DSO. This approach is useful if you want to test different client programs that bind with the same libraries.
- Automate test execution via command line interface as well as GUI mode.

5.1.5 Test Components

Each test is a named object containing the following:

- Instrumentation file—This describes the data to be collected.
- Executable—This is the program being instrumented for coverage analysis.
- Executable list—If the program you are testing can `fork`, `exec`, or `sproc` other executables and you want these other executables included in the test, then you can specify a list of executables for this purpose.
- Command—This defines the program and command line arguments.
- Instrumentation directory—The instrumentation directory contains directories representing different versions of the instrumented program and related data. Instrumentation directories are named `ver##<n>` where `n` is the version number. Several tests can share the same instrumentation directory. This is true for tests with the same instrumentation file and program version. The instrumentation directory contains the following files, which are automatically generated:

  ```
  <program|DSO>.Arg     optional arg trace file
  <program|DSO>.Binmap  basic block & branches bitmap file
  <program|DSO>.Graph   arc data
  <program|DSO>.Log     instrumentation log file (cvinstr)
  <program|DSO>.Map     function map file
  <program|DSO>_Instr   instrumented executable
  ```

  As part of instrumentation, you can filter the functions to be included or excluded in your test, through the directives INCLUDE, EXCLUDE, and CONSTRAIN.

- Experiment results—Test run coverage results are deposited in a results directory. Results directories are named `exp##<n>` where `n` corresponds to the instrumentation directory used in the experiment. There is one results directory for each version of the program in the instrumentation directory for this test. Note that results are not deposited in the instrumentation directory because the instrumentation directory may be shared by other tests. The results directory is different when you run the test with or without the `-keep` option.

  When you run your test without the `-keep` option the results directory contains the following files:

`COV_DESC`	Description file of experiment.

COUNTS_<exe>	Counts file for each executable; <exe> is an executable file name.
USER_SELECTIONS	Instrumentation criteria.

When you run your test with the -keep option the results directory contains the following files:

COV_DESC	Description file of experiment.
COUNTS_ <exe>	Counts file for each executable; <exe> is an executable file name.
USER_SELECTIONS	Instrumentation criteria.
ARGTRACE_<n>	Argument trace database; <n> is a unique number for each process.
COUNTS_<n>	Basic block and branch counts database.
DESC	Experiment description file.
FPTRACE_<n>	Function pointer tracing database.
LOG	Experiment log file (cvmon).
TRAP	N/A.
USAGE_<n>	N/A.

There are also soft links of the instrumentation data files in the results directory to the instrumentation directory described above.

5.2 Usage Model

This section is divided into three parts:

- Section 5.2.1, page 115, shows the general steps in conducting a test.
- Section 5.3, page 122, discusses using scripts to automate your testing.
- Section 5.3.1, page 124, describes strategies using multiple tests.

5.2.1 Single Test Analysis Process

In performing coverage analysis for a single test, you typically go through the following steps:

1. Plan your test.

Test tools are only as good as the quality and completeness of the tests themselves.

2. Create (or reuse) an instrumentation file.

   The instrumentation file defines the coverage data you wish to collect in this test. You can define:

   - COUNTS—three types of count items perform tracking. `bbcounts` tracks execution of basic blocks. `fpcounts` counts calls to functions through function pointers. `branchcounts` tracks branches at the assembly language level.
   - INCLUDE/EXCLUDE—lets you define a subset of functions to be covered. INCLUDE adds the named functions to the current set of functions. EXCLUDE removes the named functions from the set of functions. Simple pattern matching is supported for pathnames and function names. The basic component for inclusion/exclusion is of the form:

     `<shared library | program name>:<functionlist>`

     INCLUDE, EXCLUDE, and CONSTRAIN (see below) play a major role in working with DSOs. Tester instruments all DSOs in an executable whether you are testing them or not, so it is necessary to restrict your coverage accordingly. By default, the directory `/usr/tmp/cvinstrlib/CacheExclude` is used as the excluded DSOs cache and `/usr/tmp/cvinstrlib/CacheInclude` as the included DSOs cache. If you wish to override these defaults, set the CVINSTRLIB environment variable to the desired cache directory.
   - CONSTRAIN—equivalent to EXCLUDE *, INCLUDE < **subset**>. Thus, the only functions in the test will be those named in the CONSTRAIN subset. You can constrain the set of functions in the program to either a list of functions or a file containing the functions to be constrained. The function list file format is:

     ```
     function_1
     function_2
     function_3
     ...
     ```

     You can use the `-file` option to include an ASCII file containing all the functions as follows:

     `CONSTRAIN -file` *filename*

- TRACE—lets you monitor argument values in the functions over all experiments. The only restriction is that the arguments must be of the following basic types: `int`, `char`, `long`, `float`, `double`, or `pointer` (treated as a 4-byte `unsigned int`). MAX monitors the maximum value of an argument. MIN monitors the minimum value of an argument. BOUNDS monitors both the minimum and maximum values. RETURN monitors the function return values.

The default instrumentation file `/usr/WorkShop/usr/lib/WorkShop/Tester/default_instr_file` contains:

```
COUNTS -bbcounts -fpcounts -branchcounts
EXCLUDE libc.so.1:*
EXCLUDE libC.so:*
EXCLUDE libInventor.so:*
EXCLUDE libMrm.so.1:*
EXCLUDE libUil.so.1:*
EXCLUDE libX11.so.1:*
EXCLUDE libXaw.so:*
EXCLUDE libXawI18n.so:*
EXCLUDE libXext.so:*
EXCLUDE libXi.so:*
EXCLUDE libXm.so.1:*
EXCLUDE libXmu.so:*
EXCLUDE libXt.so:*
EXCLUDE libcrypt.so:*
EXCLUDE libcurses.so:*
EXCLUDE libdl.so:*
EXCLUDE libfm.so:*
EXCLUDE libgen.so:*
EXCLUDE libgl.so:*
EXCLUDE libil.so:*
EXCLUDE libks.so:*
EXCLUDE libmf.so:*
EXCLUDE libmls.so:*
EXCLUDE libmutex.so:*
EXCLUDE libnsl.so:*
EXCLUDE librpcsvc.so:*
EXCLUDE libsocket.so:*
EXCLUDE libtbs.so:*
EXCLUDE libtermcap.so:*
EXCLUDE libtermlib.so:*
```

```
EXCLUDE libtt.so:*
EXCLUDE libview.so:*
EXCLUDE libw.so:*
EXCLUDE nis.so:*
EXCLUDE resolv.so:*
EXCLUDE straddr.so:*
EXCLUDE tcpip.so:*
```

The excluded items are all dynamically shared objects that might interfere with the testing of your main program.

Note: If you do not use the `default_instr_file` file, functions in shared libraries will be included by default, unless your instrumentation file excludes them.

The minimum instrumentation file contains the line:

```
COUNTS -bbcounts
```

You create an instrumentation file using your preferred text editor. Comments are allowed only at the beginning of a new line and are designated by the "#" character. Lines can be continued using a back slash (\) for lists separated with commas. White space is ignored. Keywords are case insensitive. Options and user-supplied names are case sensitive. All lines are additive to the overall experiment description.

Here is a typical instrument file:

```
COUNTS -bbcounts -fpcounts -branchcounts
# defines the counting options, in this case,<
# basic blocks, function pointers, and branches.
CONSTRAIN program:abc, xdr*, functionF, \
classX::methodY, *::methodM, functionG
# constrains the set of functions in the
# ''program'' to the list of user specified functions
TRACE BOUNDS functionF(argA)
# traces the upper and lower values of argA
TRACE MAX classX::methodY(argZ)
# traces the maximum value of argZ
EXCLUDE libc.so.1:*
...
```

Note: Instrumentation can increase the size of a program two to five times. Using DSO caching and sharing can alleviate this problem.

3. Apply the instrument file to the target executable(s).

This is the instrumentation process. You can specify a single executable or more than one if you are creating other processes through `fork`, `exec`, or `sproc`.

The command line interface command is `runinstr`. The graphical user interface equivalent is the `Run Instrumentation` selection in the `Test` menu.

The effect of performing a run instrument operation is shown in Figure 53, page 119. An instrumentation directory is created (`.../ver##<n>`). It contains the instrumented executable and other files used in instrumentation.

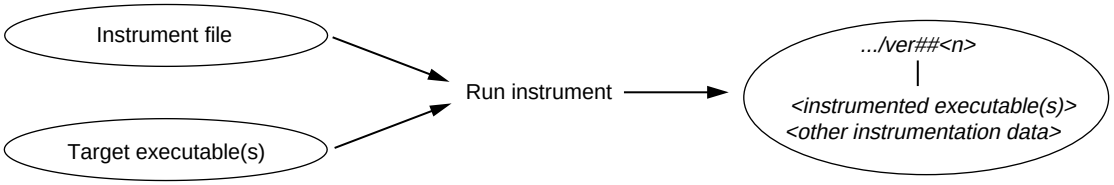

Figure 53. Instrumentation Process

4. Create the test directory.

   This part of the process creates a test data directory (`test0000`) containing a test description file named `TDF`. See Figure 54, page 119.

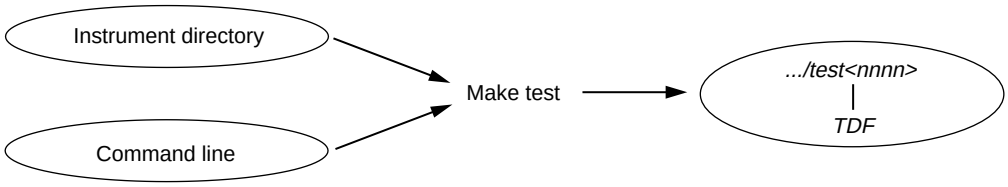

Figure 54. Make Test Process

Tester names the test directory `test0000` by default and increments it automatically for subsequent make test operations. You can supply your own name for the test directory if you prefer.

The TDF file contains information necessary for running the test. A typical `TDF` file contains the test name, type, command-line arguments, instrument

directory, description, and list of executables. In addition, for a test set or test group, the TDF file contains a list of subtests.

Note that the Instrument Directory can be either the instrumentation directory itself (such as ver##0) or a directory containing one or more instrumentation subdirectories.

The command line interface command is `mktest`. The graphical user interface equivalent is the `Make Test` selection in the `Test` menu.

5. Run the instrumented version of the executable to collect the coverage data.

   This creates a subdirectory (`exp##0`) under the test directory in which results from the current experiment will be placed. See Figure 55, page 120. The commands to run a test use the most recent instrumentation directory version unless you specify a different directory.

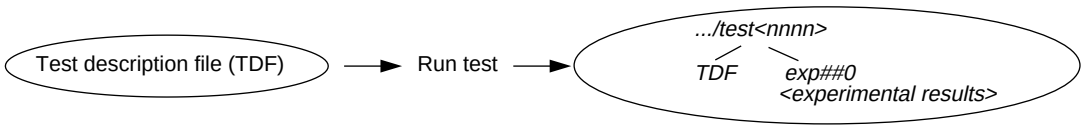

Figure 55. Run Test Process

The command-line interface command is `runtest`. The graphical user interface equivalent is the `Run Test` selection in the `Test` menu.

6. Analyze the results.

   Tester provides a variety of column-based presentations for analyzing the results. The data can be sorted by a number of criteria. In addition, the graphical user interface can display a call graph indicating coverage by function and call.

   The Tester interface provides many kinds of queries for performing analysis on a single test. Table 5, page 121, shows query commands for a single test that are available either from the command line or the graphical user interface Queries menu.

Table 5. Common Queries for a Single Test

Command Line	Graphical User Interface	Description
lsarc	`List Arcs`	Shows the function arc coverage. An *arc* is a call from one function to another.
`lsblock`	`List Blocks`	Shows basic block count information.
`lsbranch`	`List Branches`	Shows the count information for assembly language branches.
lsfun	`List Functions`	Shows coverage by function.
`lssum`	`List Summary`	Provides a summary of overall coverage.
`lstrace`	`List Argument Traces`	Shows the results of argument tracing, including argument, type, and range.
lsline	`List Line Coverage`	Shows coverage for native source lines.
`cattest`	`Describe Test`	Describes the test details.
`diff`	`Compare Test`	Shows the difference in coverage between programs.
`lsinstr`	`List Instrumentation`	Show instrumentation details for a test.

Other queries are accessed differently from either interface.

- `lscall`—Shows a function graph indicating caller and callee functions and their counts. From the graphical user interface, function graphs are accessed from a `Call Tree View` (`Views` menu selection).
- `lssource`—Displays the source or assembly code annotated with the execution count by line. From the graphical user interface, you access source or assembly code from a `Source View` (using the `Source` button) or a `Disassembly View` (using the `Disassembly` button), respectively.

The queries available in the graphical user interface are shown in Figure 56, page 122.

Figure 56. The `Queries` Menu from the Main Tester Window

5.3 Automated Testing

Tester is best suited to automated testing of command-line programs, where the test behavior can be completely specified at the invocation. Command-line programs let you incorporate contextual information, such as environment variables and current working directory.

Automated testing of server processes in a client-server application proceeds basically the same as single-program cases except that startup time introduces a new factor. Tester can substantially increase the startup time of your target process so that the instrumented target process will run somewhat slower than the standard, uninstrumented one. Tests which start a server, wait a while for it to be ready, and then start the client will have to wait considerably longer. The additional time depends on the size and complexity of the server process itself and on how much and what kind of data you have asked Tester to collect. You will have to experiment to see how long to wait.

Automated testing of interactive or nondeterministic tests is somewhat harder. These tests are not completely determined by their command line; they can produce different results (and display different coverage) from the same command line, depending upon other factors, such as user input or the timing of events. For tests such as these, Tester provides a `-sum` argument to the runtest command. Normally each test run is treated as an independent event, but when you use `runtest -sum`, the coverage from each run is added to the coverage from previous runs of the same test case. Other details of the coverage measurement process are identical to the first case.

In each case, you first need to instrument your target program, then run the test, sum the test results if desired, and finally analyze the results. There are two general approaches to applying `cvcov` in automated testing

- If you have not yet created any test scripts or have a small number of tests, you should create a script that makes each test individually and then runs the complete test set. See Example 1, which shows a script that automates a test program called `target` with different arguments:

Example 1: Making Tests and Running Them

```
# instrument program
cvcov runinstr -instr_file instrfile mypath/target
# test machinery
# make all tests
cvcov mktest -cmd ‘‘target A B C’’ -testname test0001
cvcov mktest -cmd ‘‘target D E F’’ -testname test0002
...
# define testset to include all tests
cvcov lstest > mytest_list
cvcov mktset -list mytest_list -testname mytestset
# run all tests in testset and sum up results
cvcov runtest mytestset
```

- If you have existing test scripts of substantial size or an automated test machinery setup, then you may find it straightforward to embed Tester by replacing each test line with a script containing two Tester command lines for making and running the test and then accumulating the results in a testset, such as in Example 2. Of course, you can also rewrite the whole test machinery as described in Example 1, page 123.

Example 2: Applying a Make-and-Run Script

```
# instrument program
cvcov runinstr -instr_file instrfile mypath/target
# test machinery
# make and run all tests
make_and_run ‘‘target A B C’’
make_and_run ‘‘target D E F’’
...
# make testset
cvcov lstest > mytestlist
cvcov mktset -list mytestlist -testname mytestset
# accumulate results
```

```
cvcov runtest mytestset
```

where the `make_and_run` script is:

```
#!/bin/sh
testname=‘cvcov mktest -instr_dir /usr/tmp -cmd ‘‘$*’’’
testname=‘expr ‘‘$testname’’ : ‘‘.*Made test directory: ‘.*’’’’
cvcov runtest $testname
```

Note that both examples use simple testset structures—these could have been nested hierarchically if desired.

After running your test machinery, you can use `cvcov` or `cvxcov` to analyze your results. Make sure that your test machinery does not remove the products of the test run (even if the test succeeds), or it may destroy the test coverage data.

5.3.1 Additional Coverage Testing

After you have created and run your first test, you typically need additional testing. Here are some scenarios.

- You can define a test set so that you can vary your coverage using the same instrumentation. You can analyze the new tests singly or you can combine them in a set and look at the cumulative results. If the tests are based on the same executable, they can share the same instrumentation file. You can also have a test set with tests based on different executables but they should have the same instrumentation file.
- You can change the instrumentation criteria to gather different counts, examine a different set of functions, or perform argument tracing differently.
- You can create a script to run tests in batch mode (command line interface only).
- You can run different programs that use a common dynamically shared object (DSO) and accumulate test coverage for a test group containing the DSO.
- You can run the same tests using the same instrumentation criteria for two versions of the same program and compare the coverage differences.
- You can run a test multiple times and sum the result over the runs. This is typically used for GUI-based applications.

As you conduct more tests, you will be creating more directories. A typical coverage testing hierarchy is shown in Figure 57, page 125.

There are two different instrumentation directories, `ver##0` and `ver##1`. The test directory `test0000` contains results for a single experiment that uses the instrumentation from `ver##0`. (Note that the number in the name of the experiment results directory corresponds to the number of the instrumentation directory.) Test directory `test0001` has results for two experiments corresponding to both instrumentation directories, `ver##0` and `ver##1`.

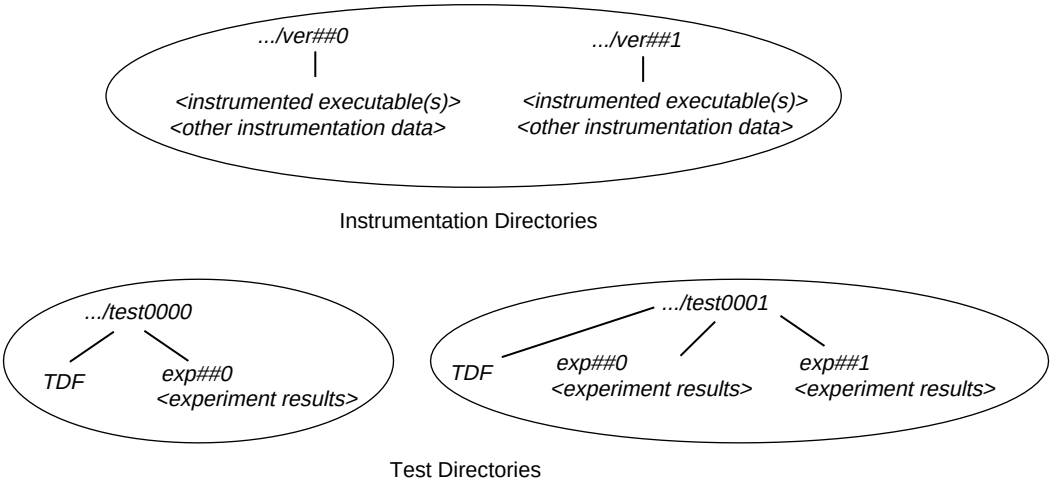

Figure 57. Typical Coverage Testing Hierarchy

Tester Command Line Interface Tutorial [6]

The tutorials in this chapter are based on simple programs written in C. To run them, you need the C compiler. The chapter is broken down into these sections:

- Section 6.1, page 127, shows you how to run the script that creates the files needed for the tutorials.
- Section 6.2, page 128, takes you through the steps of performing coverage analysis for a single test.
- Section 6.3, page 132, discusses creating additional tests to achieve full coverage.
- Section 6.4, page 139, explains how to fine-tune a test set to eliminate redundant tests.
- Section 6.5, page 142, explains how you would use a test group to analyze the coverage of a dynamically shared object (DSO) in different executables sharing the DSO.

Note that if you are going to run these tutorials, you must run them in order; each tutorial builds on the results of previous tutorials.

If you would rather have the test data built automatically, run the following script:

```
/usr/demos/WorkShop/Tester/setup_Tester_demo
```

If at any time a command syntax is not clear, enter the following:

`cvcov help <` *commandname* `>`

6.1 Setting Up the Tutorials

1. Enter the following commands to set up the tutorials:

   ```
   % cp -r /usr/demos/WorkShop/Tester /usr/tmp/tutorial
   % cd /usr/tmp/tutorial
   % echo ABCDEFGHIJKLMNOPQRSTUVWXYZ > alphabet
   % make -f Makefile.tutorial copyn
   ```

   This moves some scripts and source files used in the tutorial to `/usr/tmp/tutorial`, creates a test file named `alphabet`, and makes a

simple program, `copyn`, which copies *n* bytes from a source file to a target file.

2. To see how the program works, try a simple test by typing:

```
% copyn alphabet targetfile 10
% cat targetfile
ABCDEFGHIJ
```

You should see the first 10 bytes of `alphabet` copied to `targetfile`.

6.2 Tutorial #1 - Analyzing a Single Test

Tutorial #1 discusses the following topics:

- Instrumenting an executable
- Making a test
- Running a test
- Analyzing test coverage data

6.2.1 Instrumenting an Executable

This is the first step in providing test coverage. The user defines the instrumentation criteria in an instrumentation file.

1. Enter the following to see the instrumentation directives in the file `tut_instr_file` used in the tutorials:

```
% cat tut_instr_file
COUNTS -bbcounts -fpcounts -branchcounts
CONSTRAIN main, copy_file
TRACE BOUNDS copy_file(size)
```

We will be getting all counting information (blocks, functions, branches, and arcs) for the two functions specified in the CONSTRAIN directive, `main` and `copy_file`. We will also be tracing the *size* argument for the `copy_file` function.

2. Enter the following command to instrument `copyn`:

```
% cvcov runinstr -instr_file tut_instr_file copyn
cvcov: Instrument "copyn" of version "0" succeeded.
```

Directory ver##0 has been created by default. This contains the instrumented executable, copyn_Instr, and other instrumentation data.

6.2.2 Making a Test

A *test* defines the program and arguments to be run, instrument directory, executables, and descriptive information about the test.

1. Enter the following to make a test:

   ```
   % cvcov mktest -cmd "copyn alphabet targetfile 20"
   ```

   You will see the following message:

   ```
   cvcov: Made test directory:
   "/usr/var/tmp/tutorial/test0000"
   ```

   Directory test0000 has been created by default. It contains a single file, TDF, the test description file.

   Note: The directory /usr/var/tmp is linked to /usr/tmp.

2. Enter the following to get a textual listing of the test:

   ```
   % cvcov cattest test0000
   Test Info              Settings
   ------------------------------------------------------
   Test                   /usr/var/tmp/tutorial/test0000
   Type                   single
   Description
   Command Line           copyn alphabet targetfile 20
   Number of Exes         1
   Exe List               copyn
   Instrument Directory   /usr/var/tmp/tutorial
   Experiment List
   ```

6.2.3 Running a Test

To run a test, we use technology from the WorkShop Performance Analyzer. The instrumented process is set to run, and a monitor process (cvmon) captures test coverage data by interacting with the WorkShop process control server (cvpcs).

1. Enter the following command:

   ```
   % cvcov runtest test0000
   ```

2. You will see the following message:

   ```
   cvcov: Running test "/usr/var/tmp/tutorial/test0000" ...
   ```

Now the directory `test0000` contains the directory `exp##0`, which contains the results of the first test experiment.

6.2.4 Analyzing Test Coverage Data

You can analyze test coverage data many ways. In this tutorial, we will illustrate a simple top-down approach. We will start at the top to get a summary of overall coverage, proceed to the function level, and go finally to the actual source lines.

1. Enter the following to get the summary:

   ```
   % cvcov lssum test0000
   ```

   You will see the display shown in Example 3.

Example 3: lssum Example

```
% cvcov lssum test0000

Coverages                    Covered     Total       % Coverage    Weight
-------------------------------------------------------------------------
Function                     2           2           100.00%       0.400
Source Line                  17          35          48.57%        0.200
Branch                       0           10          0.00%         0.200
Arc                          8           18          44.44%        0.200
Block                        19          42          45.24%        0.000
Weighted Sum                                         58.60%        1.000
```

Notice that although both functions have been covered, we have incomplete coverage for source lines, branches, arcs, and blocks.

Note: Items are highlighted on your screen to emphasize null coverage. As a convention in this manual, we are showing highlighting or user input in boldface.

2. Enter the following to look at the line count information for the `main` function:

   ```
   % cvcov lssource main test0000
   ```

   This produces a source listing annotated with counts, shown in Example 4.

Example 4: lssource Example

```
% cvcov lssource main test0000
Counts  Source
------------------------------------------------------------------------
          #include <stdio.h>
          #include <sys/types.h>
          #include <sys/stat.h>
          #include <fcntl.h>
          #define OPEN_ERR         1
          #define NOT_ENOUGH_BYTES 2
          #define SIZE_0           3

          int copy_file();
          main (int argc, char *argv[])
1       {
              int bytes, status;
1           if( argc < 4){
0               printf("copyn: Insufficient arguments.\n");
0               printf("Usage: copyn f1 f2 bytes\n");
0               exit(1);
              }
1           if( argc > 4 ) {
0               printf("Error: Too many arguments\n");
0               printf("Usage: copyn f1 f2 bytes\n");
0               exit(1);
              }
1           bytes = atoi(argv[3]);
1           if(( status = copy_file(argv[1], argv[2], bytes)) >0){
0              switch ( status) {
                   case SIZE_0:
0                      printf("Nothing to copy\n");
0                      break;
                   case NOT_ENOUGH_BYTES:
0                      printf("Not enough bytes\n");
0                      break;
                   case OPEN_ERR:
0                      printf("File open error\n");
0                      break;
                 }
0              exit(1);
              }
1       }
```

```
   int copy_file( source, destn, size)
          char *source, *destn;
          int size;
1       {
             char *buf;
             int fd1, fd2;
             struct stat fstat;
1          if( (fd1 = open( source, O_RDONLY)) <= 0){
0             return OPEN_ERR;
             }
1          stat( source, &fstat);
1          if( size <= 0){
0             return SIZE_0;
             }
1          if( fstat.st_size < size){
0             return NOT_ENOUGH_BYTES;
             }
1          if( (fd2 = creat( destn, 00777)) <= 0){
0             return OPEN_ERR;
             }
1          buf = (char *)malloc(size);

1         read( fd1, buf, size);
1         write( fd2, buf, size);
1         return 0;
0       }
```

Notice that the 0-counted lines appear in a highlight color. In this example, the lines with 0 counts occur where there is an error condition. This is our first good look at branch and block coverage at the source line level. The branch and block coverage in the summary are at the assembly language level.

6.3 Tutorial #2 - Analyzing a Test Set

In the second tutorial, we are going to create additional tests with the objective of achieving 100% overall coverage. From examining the source code in Example 4, page 131, it seems that the 0-count lines in `main` and `copy_file` are due to error-checking code that is not tested by `test0000`.

Note: This tutorial needs `test0000`, which was created in the previous tutorial.

The script `tut_make_testset` is supplied to demonstrate how to set up this test set.

1. Enter **`sh -x tut_make_testset`** to run the script.

   Example 5, page 133, shows the first portion of the script (as it runs), in which the individual tests are created. The `tut_make_testset` script uses `mktest` to create eight additional tests. The tests `test0001` and `test0002` pass too few and too many arguments, respectively. `test0003` attempts to copy from a nonexistent file named `no_file`. `test0004` attempts to pass 0 bytes, which is illegal. `test0005` attempts to copy 20 bytes from a file called `not_enough`, which contains only one byte. In `test0006`, we attempt to write to a directory without proper permission. `test0007` tries to copy too many bytes. In `test0008`, we attempt to copy from a file without read permission.

Example 5: `tut_make_testset` Script: Making Individual Tests

```
% sh -x tut_make_testset
+ cvcov mktest -cmd copyn alphabet target -des not enough arguments
cvcov: Made test directory: "/usr/var/tmp/tutorial/test0001"

+ cvcov mktest -cmd copyn alphabet target 20 extra_arg \
-des too many arguments
cvcov: Made test directory: "/usr/var/tmp/tutorial/test0002"

+ cvcov mktest -cmd copyn no_file target 20 -des cannot access file
cvcov: Made test directory: "/usr/var/tmp/tutorial/test0003"

+ cvcov mktest -cmd copyn alphabet target 0 -des pass bad size arg
cvcov: Made test directory: "/usr/var/tmp/tutorial/test0004"

+ echo a

+ cvcov mktest -cmd copyn not_enough target 20 -des not enough data \
(less bytes than requested) in original file
cvcov: Made test directory: "/usr/var/tmp/tutorial/test0005"

+ cvcov mktest -cmd copyn alphabet /usr/bin/target 20 \
-des cannot create target executable due to permission problems
cvcov: Made test directory: "/usr/var/tmp/tutorial/test0006"

+ ls -ld /usr/bin
drwxr-xr-x    3 root     sys         3584 May 12 18:25 /usr/bin
```

```
+ cvcov mktest -cmd copyn alphabet targetfile 200
-des size arg too big
cvcov: Made test directory: "/usr/var/tmp/tutorial/test0007"

+ cvcov mktest -cmd copyn /usr/adm/sulog targetfile 20 \
-des no read permission on source file
cvcov: Made test directory: "/usr/var/tmp/tutorial/test0008"
```

After the individual tests are created, the script uses `mktset` to make a new test set and `addtest` to include the new tests in the set. Example 6, page 134, shows the portion of the script in which the test set is created and the individual tests are added to the test set.

Example 6: `tut_make_testset` Script: Making and Adding to the Test Set

```
+ cvcov mktset -des full coverage testset -testname tut_testset
cvcov: Made test directory: "/usr/var/tmp/tutorial/tut_testset"

+ cvcov addtest test0000 tut_testset
cvcov: Added "/usr/var/tmp/tutorial/test0000" to "tut_testset"

+ cvcov addtest test0001 tut_testset
cvcov: Added "/usr/var/tmp/tutorial/test0001" to "tut_testset"

+ cvcov addtest test0002 tut_testset
cvcov: Added "/usr/var/tmp/tutorial/test0002" to "tut_testset"

+ cvcov addtest test0003 tut_testset
cvcov: Added "/usr/var/tmp/tutorial/test0003" to "tut_testset"

+ cvcov addtest test0004 tut_testset
cvcov: Added "/usr/var/tmp/tutorial/test0004" to "tut_testset"

+ cvcov addtest test0005 tut_testset
cvcov: Added "/usr/var/tmp/tutorial/test0005" to "tut_testset"

+ cvcov addtest test0006 tut_testset
cvcov: Added "/usr/var/tmp/tutorial/test0006" to "tut_testset"

+ cvcov addtest test0007 tut_testset
cvcov: Added "/usr/var/tmp/tutorial/test0007" to "tut_testset"

+ cvcov addtest test0008 tut_testset
```

```
cvcov: Added "/usr/var/tmp/tutorial/test0008" to "tut_testset"
```

2. Enter **`cvcov cattest tut_testset`** to check that the new test set was created correctly.

   This is shown in Example 7, page 135. The index numbers in brackets in the subtest list are used to identify the individual tests as part of a test set. This index is used to list the contribution of each test.

Example 7: Contents of the New Test Set

```
% cvcov cattest tut_testset
Test Info               Settings
-------------------------------------------------------
Test                    /usr/var/tmp/tutorial/tut_testset
Type                    set
Description             full coverage testset
Number of Exes          1
Exe List                copyn
Number of Subtests      9
Subtest List
                        [0] /usr/var/tmp/tutorial/test0000
                        [1] /usr/var/tmp/tutorial/test0001
                        [2] /usr/var/tmp/tutorial/test0002
                        [3] /usr/var/tmp/tutorial/test0003
                        [4] /usr/var/tmp/tutorial/test0004
                        [5] /usr/var/tmp/tutorial/test0005
                        [6] /usr/var/tmp/tutorial/test0006
                        [7] /usr/var/tmp/tutorial/test0007
                        [8] /usr/var/tmp/tutorial/test0008
Experiment List
```

3. Enter the following to run the tests in the test set:

   ```
   % cvcov runtest tut_testset
   ```

   By applying the `runtest` command to the test set, we can run all the tests together. See Example 8, page 136. Note that when you run a test set, only tests without results are run; tests that already have results will not be run again. In this case, `test0000` has already been run. If you need to rerun a test, you can do so using the `-force` flag.

Example 8: Running the New Test Set

```
% cvcov runtest tut_testset
cvcov: Running test "/usr/var/tmp/tutorial/test0000" ...
cvcov: Running test "/usr/var/tmp/tutorial/test0001" ...
copyn: Insufficient arguments.
Usage: copyn f1 f2 bytes
cvcov: Running test "/usr/var/tmp/tutorial/test0002" ...
Error: Too many arguments
Usage: copyn f1 f2 bytes
cvcov: Running test "/usr/var/tmp/tutorial/test0003" ...
File open error
cvcov: Running test "/usr/var/tmp/tutorial/test0004" ...
Nothing to copy
cvcov: Running test "/usr/var/tmp/tutorial/test0005" ...
Not enough bytes
cvcov: Running test "/usr/var/tmp/tutorial/test0006" ...
File open error
cvcov: Running test "/usr/var/tmp/tutorial/test0007" ...
Not enough bytes
cvcov: Running test "/usr/var/tmp/tutorial/test0008" ...
File open error
```

4. Enter **cvcov lssum tut_testset** to list the summary for the test set.

   Example 9, page 136, shows the results of the tests in the new test set with `lssum`.

Example 9: Examining the Results of the New Test Set

```
% cvcov lssum tut_testset
Coverages                   Covered     Total       % Coverage    Weight
-------------------------------------------------------------------------
Function                    2           2           100.00%       0.400
Source Line                 35          35          100.00%       0.200
Branch                      9           10          90.00%        0.200
Arc                         18          18          100.00%       0.200
Block                       39          42          92.86%        0.000
Weighted Sum                                        98.00%        1.000
```

5. Enter **cvcov lssource main tut_testset** to see the coverage for the individual source lines as shown in Example 10, page 137.

Example 10: Source with Counts

```
% cvcov lssource main tut_testset
Counts  Source
------------------------------------------------------------------------
          #include <stdio.h>
          #include <sys/types.h>
          #include <sys/stat.h>
          #include <fcntl.h>
          #define OPEN_ERR         1
          #define NOT_ENOUGH_BYTES 2
          #define SIZE_0           3

          int copy_file();

          main (int argc, char *argv[])
9       {
              int bytes, status;

9           if( argc < 4){
1               printf("copyn: Insufficient arguments.\n");
1               printf("Usage: copyn f1 f2 bytes\n");
1               exit(1);
              }
8           if( argc > 4 ) {
1               printf("Error: Too many arguments\n");
1               printf("Usage: copyn f1 f2 bytes\n");
1               exit(1);
              }
7           bytes = atoi(argv[3]);
7           if(( status = copy_file(argv[1], argv[2], bytes)) >0){
6              switch ( status) {
                     case SIZE_0:
1                      printf("Nothing to copy\n");
1                      break;
                     case NOT_ENOUGH_BYTES:
2                      printf("Not enough bytes\n");
2                      break;
                     case OPEN_ERR:
3                      printf("File open error\n");
3                      break;
                  }
6              exit(1);
```

```
        }
1       }

          int copy_file( source, destn, size)
          char *source, *destn;
          int size;
7       {
              char *buf;
              int fd1, fd2;
              struct stat fstat;
7           if( (fd1 = open( source, O_RDONLY)) <= 0){
2               return OPEN_ERR;
              }
5           stat( source, &fstat);
5           if( size <= 0){
1               return SIZE_0;
              }
4           if( fstat.st_size < size){
2               return NOT_ENOUGH_BYTES;
              }
2           if( (fd2 = creat( destn, 00777)) <= 0){
1               return OPEN_ERR;
              }
1           buf = (char *)malloc(size);

1          read( fd1, buf, size);
1          write( fd2, buf, size);
1          return 0;
0       }
```

As you look at the source code, notice that all lines are covered.

6. Enter **`cvcov lssource -asm main tut_testset`** to see the coverage for the individual assembly lines.

   When we list the assembly code using `lssource -asm`, we find that not all blocks and branches are covered at the assembly level. This is due to compilation with the `-g` flag, which adds debugging code that can never be executed.

   Enter **`cvcov lsline tut_testset`** to see the coverage at the source line level. Notice that 100% of the lines have been covered.

6.4 Tutorial #3 - Optimizing a Test Set

Tester lets you look at the individual test coverages in a test set. When you put together a set of tests, you may want to improve the efficiency of your coverage by eliminating redundant tests. The `lsfun`, `lsblock`, and `lsarc` commands all have the `-contrib` option, which displays coverage result contributions by individual tests. We will now look at the contributions by tests for the test set we just ran, `tut_testset`.

Note: This tutorial needs `tut_testset` and all its subtests; these were created in the previous tutorial.

1. Enter **`cvcov lsfun -contrib -pretty tut_testset`** to see the function coverage test contribution.

   Example 11, page 139, shows how the test set covers functions. Note that the subtests are identified by index numbers; use `cattest` if you need to map these results back to the test directories.

Example 11: Test Contributions by Function

```
% cvcov lsfun -contrib -pretty tut_testset
Functions     Files         Counts
---------------------------------------
main          copyn.c       9
copy_file     copyn.c       7

Functions    Files       [0]    [1]    [2]    [3]    [4]    [5]
-----------------------------------------------------------------
main         copyn.c     1      1      1      1      1      1
copy_file    copyn.c     1      0      0      1      1      1

Functions    Files       [6]    [7]    [8]
-------------------------------------------
main         copyn.c     1      1      1
copy_file    copyn.c     1      1      1
```

At the function level, each test covers both functions except for Tests [1] and [2]. The information here is not sufficient to tell us if we have optimized the test set. To do this, we must look at contributions at the arc and block levels. Tester shows arc and block coverage information by test when you apply the `-contrib` flag to `lsarc` and `lsblock`, respectively.

2. Enter the following to see the arc coverage test contribution.

   ```
   % cvcov lsarc -contrib -pretty tut_testset
   ```

Example 12, page 140, shows the individual test contributions. Notice that Tests [5] and [7] have identical coverage to each other; so do Tests [3] and [8].

We can get additional information by looking at block coverage, confirming our hypothesis about redundant tests.

Example 12: Arc Coverage Test Contribution Portion of Report

Callers	Callees	Line	Files	[0]	[1]	[2]	[3]	[4]	[5]
main	copy_file	27	copyn.c	1	0	0	1	1	1
main	printf	17	copyn.c	0	1	0	0	0	0
main	printf	18	copyn.c	0	1	0	0	0	0
main	exit	19	copyn.c	0	1	0	0	0	0
main	printf	22	copyn.c	0	0	1	0	0	0
main	printf	23	copyn.c	0	0	1	0	0	0
main	exit	24	copyn.c	0	0	1	0	0	0
main	atoi	26	copyn.c	1	0	0	1	1	1
main	printf	30	copyn.c	0	0	0	0	1	0
main	printf	33	copyn.c	0	0	0	0	0	1
main	printf	36	copyn.c	0	0	0	1	0	0
main	exit	39	copyn.c	0	0	0	1	1	1
copy_file	_open	50	copyn.c	1	0	0	1	1	1
copy_file	_stat	53	copyn.c	1	0	0	0	1	1
copy_file	_creat	60	copyn.c	1	0	0	0	0	0
copy_file	_malloc	63	copyn.c	1	0	0	0	0	0
copy_file	_read	65	copyn.c	1	0	0	0	0	0
copy_file	_write	66	copyn.c	1	0	0	0	0	0

Callers	Callees	Line	Files	[6]	[7]	[8]
main	copy_file	27	copyn.c	1	1	1
main	printf	17	copyn.c	0	0	0
main	printf	18	copyn.c	0	0	0
main	exit	19	copyn.c	0	0	0
main	printf	22	copyn.c	0	0	0
main	printf	23	copyn.c	0	0	0
main	exit	24	copyn.c	0	0	0
main	atoi	26	copyn.c	1	1	1
main	printf	30	copyn.c	0	0	0
main	printf	33	copyn.c	0	1	0
main	printf	36	copyn.c	1	0	1
main	exit	39	copyn.c	1	1	1

```
copy_file   _open       50          copyn.c     1      1      1
copy_file   _stat       53          copyn.c     1      1      0
```

3. Enter the following to see the test contribution to block coverage:

   ```
   % cvcov lsblock -contrib -pretty tut_testset
   ```

   If you examine the results, you will see that Tests [5] and [7] and Tests [3] and [8] are identical.

   Now we can try to tune the test set. If we can remove tests with redundant coverage and still achieve the equivalent overall coverage, then we have tuned our test set successfully. Since the arcs and blocks covered by Test [7] are also covered by Test [5], we can remove either one of them without affecting the overall coverage. The same analysis holds true for Tests [3] and [8].

4. Delete `test0007` and `test0008` as shown in Example 13, page 141. Then rerun the test set and look at its summary.

   Note that the coverage is retabulated without actually rerunning the tests. The test summary shows that overall coverage is unchanged, thus confirming our hypothesis.

Example 13: Test Set Summary after Removing Tests [8] and [7]

```
% cvcov deltest test0008 tut_testset
cvcov: Deleted "/usr/var/tmp/tutorial/test0008" from "tut_testset"

% cvcov deltest test0007 tut_testset
cvcov: Deleted "/usr/var/tmp/tutorial/test0007" from "tut_testset"

% cvcov runtest tut_testset
cvcov: Running test "/usr/var/tmp/tutorial/test0000" ...
cvcov: Running test "/usr/var/tmp/tutorial/test0001" ...
cvcov: Running test "/usr/var/tmp/tutorial/test0002" ...
cvcov: Running test "/usr/var/tmp/tutorial/test0003" ...
cvcov: Running test "/usr/var/tmp/tutorial/test0004" ...
cvcov: Running test "/usr/var/tmp/tutorial/test0005" ...
cvcov: Running test "/usr/var/tmp/tutorial/test0006" ...

% cvcov lssum tut_testset
Coverages                   Covered     Total       % Coverage    Weight
-------------------------------------------------------------------------
Function                    2           2           100.00%       0.400
```

```
Source Line              35          35          100.00%       0.200
Branch                   9           10          90.00%        0.200
Arc                      18          18          100.00%       0.200
Block                    39          42          92.86%        0.000
Weighted Sum                                     98.00%        1.000
```

6.5 Tutorial #4 - Analyzing a Test Group

Test groups are used when you are conducting tests on executables that use a common dynamically shared object (DSO). The results will be limited to whatever constraints you set on the DSO and thus will not include branches, arcs, and other code that lie outside the executables.

> **Note:** This tutorial may be run independently of the previous tutorials. However, it does use `copyn`. If you have run the other tutorials previously, the instrumentation directory `ver##1` will be created for the new executable; otherwise, `ver##0` is created when `copyn` is compiled.

In this tutorial, we will test coverage for a DSO called `libc.so.1`, which is shared by `copyn`, the executable from the previous tutorials, and a simple application called `printtest`. The script `tut_make_testgroup` is provided to run this tutorial.

1. Run the script by typing **`tut_make_testgroup`**

   The `tut_make_testgroup` script creates the test group and its subtests. Example 14, page 143, shows the results of running the initial preparation part of the script using `sh -x`.

   First, the script makes the two applications, `printtest` and `copyn`. The next step is to instrument the programs. The script stores the instrumentation data for `printtest` in a subdirectory called `print_instr_dir` and the `copyn` data in `copyn_instr_dir`.

   The script then makes test directories for the applications and names them `print_test0000` and `copyn_test0000`, respectively. It makes a test group called `tut_testgroup` and adds both tests to it.

   The `mktgroup` command is the only one that we have not used previously in the tutorials. `mktgroup` creates the test group. As a final part of the preparation, the script performs a `cattest` command to show the contents of the test group.

Example 14: Setting up a Test Group

```
% sh -x tut_make_testgroup
+ make -f Makefile.tutorial all
           /usr/bin/cc -g -o printtest printtest.c  -lc

+ cvcov runinstr -instr_dir print_instr_dir -instr_file tut_group_instr_file printtest
cvcov: Instrument "printtest" of version "0" succeeded.

+ cvcov runinstr -instr_dir copyn_instr_dir -instr_file tut_group_instr_file copyn
cvcov: Instrument "copyn" of version "0" succeeded.

+ cvcov mktest -cmd printtest 10 2 3 -instr_dir print_instr_dir -testname print_test0000
cvcov: Made test directory: "/usr/var/tmp/tutorial4/print_test0000"

+ cvcov mktest -cmd copyn tut4_instr_file targetfile -instr_dir copyn_instr_dir -testname
 copyn_test0000
cvcov: Made test directory: "/usr/var/tmp/tutorial4/copyn_test0000"

+ cvcov mktgroup -des Group sharing libc.so.1 -testname tut_testgroup libc.so.1
cvcov: Made test directory: "/usr/var/tmp/tutorial4/tut_testgroup"

+ cvcov addtest print_test0000 tut_testgroup
cvcov: Added "/usr/var/tmp/tutorial4/print_test0000" to "tut_testgroup"

+ cvcov addtest copyn_test0000 testgroup
cvcov: Added "/usr/var/tmp/tutorial4/copyn_test0000" to "tut_testgroup"

+ cvcov cattest tut_testgroup
Test Info              Settings
---------------------------------------------------------------
Test                   /usr/var/tmp/tutorial4/tut_testgroup
Type                   group
Description            Group sharing libc.so.1
Number of Objects      1
Object List            libc.so.1
Number of Subtests     2
Subtest List
                         [0] /usr/var/tmp/tutorial4/print_test0000
                         [1] /usr/var/tmp/tutorial4/copyn_test0000
Experiment List
```

Finally, the script runs the test group and performs the queries shown in Example 15, page 144.

Example 15: Examining Test Group Results

```
+ cvcov runtest tut_testgroup
cvcov: Running test "/usr/var/tmp/tutorial4/print_test0000" ...
2
3
10
cvcov: Running test "/usr/var/tmp/tutorial4/copyn_test0000" ...
copyn: Insufficient arguments.
Usage: copyn f1 f2 bytes
+ cvcov lssum tut_testgroup
Coverages                   Covered     Total       % Coverage    Weight
-------------------------------------------------------------------------
Function                    33          1777        1.86%         0.400
Source Line                 438         25525       1.72          0.200
Branch                      27          10017       0.27%         0.200
Arc                         31          6470        0.48%         0.200
Block                       363         27379       1.33%         0.200
Weighted Sum                                        1.24%         1.000
+ cvcov lsfun -pretty -contrib -pat printf tut_testgroup
Functions    Files       Counts
-----------------------------------
printf       doprnt.c    5

Functions    Files       [0]    [1]
---------------------------------------
printf       doprnt.c    3      2
+ cvcov lsfun -pretty -contrib -pat sscanf tut_testgroup
Functions    Files       Counts
-----------------------------------
sscanf       scanf.c     3

Functions    Files       [0]    [1]
---------------------------------------
sscanf       scanf.c     3      0
```

You can use any of the query commands to look at test group results that we used in other tutorials. This tutorial is for illustrative purposes only. Notice that the overall coverage of the C library is poor and that the summary is too general. It is useful, however, to look at individual functions to see how they were covered between the two executables. Performing a list function for

`printf` indicates that it was adequately convered, three times by `printtest` (Test [0]) and twice by `copyn` (Test [1]). On the other hand, checking `sscanf` coverage shows that it was covered three times by Test [0] but not all by Test [1].

Tester Command Line Reference [7]

This chapter describes the cvcov commands. It contains two parts:

- Section 7.1, page 147, describes the command arguments that are common to more than one command
- Section 7.2, page 149, describes the specifications with descriptions for each command

A complete description of the cvcov commands, including individual arguments, is available in the man pages by typing:

```
man cvcov
```

7.1 Common cvcov Options

This section contains descriptions of some cvcov flags and variables that are common to more than one command.

[-ver]

Displays the version of cvcov. Note that there are no other arguments permitted; you enter: cvcov -ver

[-v *versionnumber]*

Allows you to specify a version of the instrumentation or experiment directory other than the most recent, which is the default.

[-contrib]

Shows the list of tests that contributed to coverage for the particular query.

[-exe *exe_name*]

Lets you specify an executable for coverage testing. This is used when there are multiple executables involved, as in testing processes created by the fork, exec, or sproc command.

`[-instr_dir` *instr_dir*`]`

Allows you to specify an instrumentation directory other than the current working directory, which is the default.

`[-instr_file` *instr_file*`]`

Specifies the instrumentation file, which is an ASCII description of the instrumentation criteria you have selected.

`[-list` *list_file*`]`

Specifies a file containing a list of test names to be made part of a test set or group. If no `-list` option is specified, an empty test set will be created.

`[-r]`

(Recursion) Lets you specify tests in a hierarchy of subdirectories.

`[-arg]`

Displays functions with their arguments.

`[-pretty]`

Displays output aligned in columns. Without `-pretty`, the output is in columns but more condensed.

`[-sort]`

Sorts the output by the specified criteria, as follows:

`function`	Alphabetically by function
`diff`	By differences in the counting information for coverage type
`caller`	Alphabetically by calling function
`callee`	Alphabetically by called function
`count`	By counts for current coverage type
`file`	Alphabetically by file name
`type`	Alphabetically by argument type

`[-functions]`

Displays list of constrained functions.

`[-pat` *func_pattern*]

Lets you enter a pattern instead of a complete function name. The pattern can be of the form `func_name`, `dso_:func_name`, or `'dso:*'`.

`experiment | test_name`

Lets you specify either the experiment subdirectory or the test directory. The test directory is typically of the form `test<nnnn>`, where `<nnnn>` is a number in a sequence counting from 0000. You can specify your own name. The test directory contains all information about a test including the experiment directory. The experiment directory is typically of the form `exp##<n>`, where `<n>` is a sequential number, counting from 0.

7.2 cvcov Command Syntax and Description

This section contains the syntax and description for all `cvcov` commands in the command line interface. If you need information on command arguments that are not described in this section, please refer back to Section 7.1, page 147.

The most general command is the `help` command, as follows:

`cvcov help` *command_name*

The `help` command prints help on the specified command. If the optional command name is not specified, it prints help for all the commands.

The rest of the commands are divided up into these categories:

- General test commands
  - `cvcov cattest`
  - `cvcov lsinstr`
  - `cvcov lstest`
  - `cvcov mktest`
  - `cvcov rmtest`

  - `cvcov runinstr`
  - `cvcov runtest`
- Coverage analysis commands
  - `cvcov lssum`
  - `cvcov lsfun`
  - `cvcov lsblock`
  - `cvcov lsbranch`
  - `cvcov lsarc`
  - `cvcov lscall`
  - `cvcov lsline`
  - `cvcov lssource`
  - `cvcov lstrace`
  - `cvcov diff`
- Test set commands
  - `cvcov mktset`
  - `cvcov addtest`
  - `cvcov deltest`
  - `cvcov optimize`
- Test group command
  - `cvcov mktgroup`

7.2.1 General Test Commands

The following commands support the creation, inspection, modification, and deletion of tests:

`cvcov cattest [-r]` *test_name*

Describes the test details for a test, test set, or test group. Example 16, shows the ASCII display for a single test.

Example 16: `cattest` Example

```
% cvcov cattest test0000
Test Info               Settings
-------------------------------------------------------------
Test                    /disk2/tutorial/tutorial/test0000
Type                    single
Description
Command Line            copyn alphabet targetfile 20
Number of Exes          1
Exe List                copyn
Instrument Directory    /disk2/tutorial/tutorial/
Experiment List
                          exp##0
                          exp##1
```

Example 17 shows the ASCII report for a test set without recursion.

Example 17: `cattest` Example without `-r`

```
% cvcov cattest tut_testset
Test Info               Settings
-------------------------------------------------------------
Test                   /disk2/tutorial/tutorial/tut_testset
Type                   set
Description            full coverage testset
Number of Exes         1
Exe List               copyn
Number of Subtests     9
Subtest List
                       [0] /disk2/tutorial/tutorial/test0000
                       [1] /disk2/tutorial/tutorial/test0001
                       [2] /disk2/tutorial/tutorial/test0002
                       [3] /disk2/tutorial/tutorial/test0003
                       [4] /disk2/tutorial/tutorial/test0004
                       [5] /disk2/tutorial/tutorial/test0005
                       [6] /disk2/tutorial/tutorial/test0006
                       [7] /disk2/tutorial/tutorial/test0007
                       [8] /disk2/tutorial/tutorial/test0008
Experiment List
                       exp##0
```

Example 18, shows the ASCII report for a nested test set.

Example 18: `cattest` Example with `-r`

```
% cvcov cattest -r tut_testset

Test Info               Settings
-------------------------------------------------------
Test                    /disk2/tutorial/tutorial/tut_testset
Type                    set
Description             full coverage testset
Number of Exes          1
Exe List                copyn
Number of Subtests      9
Subtest List
                        /disk2/tutorial/tutorial/test0000
                        /disk2/tutorial/tutorial/test0001
                        /disk2/tutorial/tutorial/test0002
                        /disk2/tutorial/tutorial/test0003
                        /disk2/tutorial/tutorial/test0004
                        /disk2/tutorial/tutorial/test0005
                        /disk2/tutorial/tutorial/test0006
                        /disk2/tutorial/tutorial/test0007
                        /disk2/tutorial/tutorial/test0008
Experiment List
                        exp##0
```

`cvcov lsinstr [-exe]` *exe_name* `[-functions] [-v` *versionnumber*`]` *test_name*

Displays the instrumentation information for a particular test. *exe_name* is the executable targeted for query. The main program is the default if no executable is specified. The `-functions` parameter shows the functions that are included in the coverage experiment. The *versionnumber* parameter allows you to specify the version of the program that was instrumented. You can specify the test directory using the *test_name* parameter. See Example 19.

Example 19: `lsinstr` Example

```
% cvcov lsinstr test0000
Instrumentation         Info
-------------------------------------------------------
Executable              copyn
Version                 0
```

```
Instrument Directory    /x/tmp/carol/
Instrument File         tut_instr_file
Criteria                RBPA
Instrumented Objects    copyn_Instr(2.57X)
                          libc.so.1_RBP_Instr(1.07X)
Argument Tracing        copy_file(size[bounds]) [copyn.c]
```

`cvcov lstest [-r] [` *test_name...*`]`

Lists the test directories in the current working directory. Note that the *test_name* parameter will accept regular expressions for `lstest`.

`cvcov mktest -cmd` *cmd_line* `[-des` *description*`] [-instr_dir` *directoryname*`] [-testname` *test*`] [`*exe1 exe2* `...]`

Creates a test directory. You specify the program and command line options for the program to be tested. This includes any redirection for `stdin`, `stderr`, or `stdout` as run from the Bourne shell. The `-cmd` qualifier is mandatory, even if it only includes the program name. If no executables are specified, only the main program is tested. Example 20, shows an example of `mktest`, followed by `cattest` to display the contents of the Test Description File (TDF).

Example 20: Test Description File Examples

```
% cvcov mktest -cmd "copyn tut_instr_file targetfile"
cvcov: Made test directory: /d/Tester/tutorial/test0002

% cvcov cattest test0002
Test Info               Settings
-------------------------------------------------------
Test                    /d/Tester/tutorial/test0002
Type                    single
Description
Command Line            copyn tut_instr_file targetfile
Number of Exes          1
Exe List                copyn
Instrument Directory    /d/Tester/tutorial
Experiment List
```

`cvcov rmtest [-r]` *test_name* `...`

Removes tests and test sets. Note that the *test_name* parameter will accept regular expressions for `rmtest`. It is recommended

to separate the test set directory from its test subdirectories and the instrument directory. In this way, `rmtest` will not remove instrumentation data or subtests if you choose to remove the test set only.

`cvcov runinstr [-instr_dir` *instr_dir*`][-instr_file` *instr_file*`] [-v` *versionnumber*`]` *executable*

Adds code to the target executable to enable you to capture coverage data, according to the criteria you specify. The instrument file is an ASCII description of the instrumentation criteria for the experiment.You can also specify the version of the executable and instrument directory.

You can capture basic block counts, function pointer counts, and branch counts (at the assembly language level). You can use INCLUDE, EXCLUDE, or CONSTRAIN to modify the set of functions covered. CONSTRAIN lets you define a set of functions for the test.

`cvcov runtest [ -bitcount ][ -compress ][-force] [-keep] [-sum] [-v` *versionnumber*`] [-noarc] [-rmsub]` *test_name*

Runs a test or a set of tests. The `-bitcount` flag compresses count data file to be 1-bit-per-count. This option can decrease the database size up to 32 times, although branch count information will be lost. The `-compress` flag compresses the experiment database using the standard utility `compress`. The `-force` flag forces the test to be run again even if an experiment is present. It uses WorkShop performance tool technology to set up the instrumented process, run the process, and monitor the run, collecting counting information upon exit. The `-keep` flag retains all performance data collected in the experiment. By default, the performance data is not retained, because it is not required by the coverage tool. The `-sum` flag accumulates (sum over) the coverage data into the existing experiment results. This allows users to run and rerun the same test and accumulate the results in one place.

The `-noarc` flag prevents arc information from being saved in the test database. With the `-noarc` flag, all arc-related queries will not work (for example, `lsarc` and `lscall`). The `-rmsub` flag removes results for individual subtests for a test set or test

group. There will be no data to query if you are querying a subtest. `-noarc` and `-rmsub` save disk space.

7.2.2 Coverage Analysis Commands

Once the data has been collected from the test experiments, the user can analyze the data. There are special commands for the various types of coverage available.

`cvcov lssum [-exe` *exe_name*`] [-weight` *func_factor* `:`*line_factor* `:` *branch_factor* `:` *arc_factor* `:` *block_factor*`]` *experiment* `|` *test_name*

Shows the overall coverage based on the user-defined weighted average over function, line, block, branch, and arc coverage. Example 21, shows a typical `lssum` report.

Example 21: `lssum` Example

```
% cvcov lssum test0000
Coverages        Covered     Total       % Coverage    Weight
-------------------------------------------------------------
Function         2           2           100.00%       0.400
Source Line      17          35          48.57%        0.200
Branch           0            10           0.00%        0.200
Arc              8           18          44.44%        0.200
Block            19          42          45.24%        0.000
Weighted Sum                             58.60%        1.000
```

`cvcov lsfun [-arg] [-bf` *filter_type block_filter_value*`] [-blocks] [-branches] [-contrib] [-exe` *exe_name*`] [-ff` *filter_type func_filter_value*`] [-pat` *func_pattern*`] [-pretty] [-rf` *filter_type branch_filter_value*`] [-sort` *count* `|` *file* `|` *function*`]` *experiment* `|`*test_name*

Lists coverage information for the specified functions in the program that was tested. Several sorting, matching, and filtering techniques are available. For example, you can show the list of functions that have 0 counts (were not covered) in alphabetical order. You can display arguments with the `-arg` flag. Example 22, shows a typical `lsfun` ASCII report.

Example 22: `lsfun` Example

```
% cvcov lsfun -pretty -sort function test0000
Functions    Files       Counts
```

```
-----------------------------------
copy_file    copyn.c     1
main         copyn.c     1
```

Note: C++ inline functions are not counted as functions.

`cvcov lsblock [-addr] [-arg] [-contrib] [-exe` *exe_name*`] [-pat` *func_pattern*`] [-pretty] [-sort` *count*`|` *file* `|` *function*`]` *experiment*`|` *test_name*

Displays a list of blocks for one or more functions and the count information associated with each block. Blocks are identified by the line numbers in which they occur. If there are multiple blocks in a line, blocks subsequent to the first are shown in order with an index number in parentheses. Be careful before listing all blocks in the program, since this can produce a lot of data. The `-addr` flag show blocks with the PC range instead of the source line number range. Example 23 shows a typical `lsblock` ASCII report.

Example 23: `lsblock` Example%

```
cvcov lsblock -pat main -pretty test0000
Blocks      Functions    Files       Counts
-----------------------------------------------
13~16       main         copyn.c     1
17~17       main         copyn.c     0
18~18       main         copyn.c     0
19~19       main         copyn.c     0
21~21       main         copyn.c     1
22~22       main         copyn.c     0
23~23       main         copyn.c     0
24~24       main         copyn.c     0
26~26       main         copyn.c     1
26~27       main         copyn.c     1
27~27       main         copyn.c     1
28~28       main         copyn.c     0
28~28(2)    main         copyn.c     0
28~28(3)    main         copyn.c     0
28~28(4)    main         copyn.c     0
30~30       main         copyn.c     0
31~31       main         copyn.c     0
33~33       main         copyn.c     0
34~34       main         copyn.c     0
```

```
36~36       main         copyn.c     0
37~37       main         copyn.c     0
39~39       main         copyn.c     0
41~41       main         copyn.c     0
43~43       main         copyn.c     1
43~43(2)    main         copyn.c     0
43~43(3)    main         copyn.c     1
```

cvcov lsbranch [-addr] [-arg] [-exe*exe_name*] [-pat *func_pattern*] [-pretty][-sort *function* | *file*] *experiment* | *test_name*

Lists coverage information for branches in the program, including the line number at which the branch occurs. Branch coverage counts assembly language branch instructions that are both taken and not taken. The -addr flag show blocks with the PC range instead of the source line number range.

Example 24, shows a typical branch coverage ASCII report. Note that branches with incomplete or null coverage are highlighted (**boldfaced**).

Example 24: lsbranch Example

```
% cvcov lsbranch -pretty -sort function test0000
Line        Functions    Files       Taken       Not Taken
-------------------------------------------------------------
50          copy_file    copyn.c     1           0
54          copy_file    copyn.c     1           0
57          copy_file    copyn.c     1           0
60          copy_file    copyn.c     1           0
16          main         copyn.c     1           0
21          main         copyn.c     1           0
27          main         copyn.c     1           0
28          main         copyn.c     0           0
28(2)       main         copyn.c     0           0
28(3)       main         copyn.c     0           0
```

cvcov lsarc [-arg] [-callee *callee_pattern*] [-caller *caller_pattern*] [-contrib][-exe *exe_name*] [-pretty][-sort *caller* | *callee*| *count* | *file*] *experiment* | *test_name*

Shows *arc coverage*, that is, the number of arcs taken out of the total possible arcs. An arc is a function caller-callee pair. Both *callee_pattern* and *caller_pattern* can be specified in the same way

as *func_pattern* (used with the `-pat` option) as shown under Section 7.1, page 147.

Example 25, shows a typical `lsarc` ASCII report.

Example 25: `lsarc` Example

```
% cvcov lsarc -callee printf -pretty test0001
Callers    Callees     Line       Files       Counts
-----------------------------------------------------
main        printf      17          copyn.c     1
main        printf      18          copyn.c     1
main        printf      22          copyn.c     0
main        printf      23          copyn.c     0
main        printf      30          copyn.c     0
main        printf      33          copyn.c     0
main        printf      36          copyn.c     0
```

`cvcov lscall [-arg] [-exe` *exe_name*`][-node` *func_name*`] [-pretty] [-r]` *experiment* `|`*test_name*

Lists the call graph for the executable with counts for each function. The contribution to this coverage by each test is shown in a separate column. Example 26, shows a typical `lscall` ASCII report. `N/A` means the node is excluded.

Example 26: `lscall` Example

```
% cvcov lscall -pretty test0000
Graph            Counts
--------------------------------
main             1
     copy_file     1
        _open      N/A
        _stat...   N/A
        _creat     N/A
        _malloc... N/A
        _read      N/A
        _write     N/A
     printf...     N/A
     exit...       N/A
     atoi          N/A
```

A function that has more than one parent and has children is called a *subnode.* Using `-r` will display the subnodes. Subnodes are given their own starting

point in the textual call graph. They are identified by a trailing ellipsis (...). For example, see `printf`, `exit`, and `malloc` in Example 26.

`cvcov lsline [-arg] [-exe` *exe_name*`] [-pat` *func_pattern*`][-pretty] [-sort` *function* | *file*`]` *experiment* | *test_name*

Lists the coverage for native source lines. Use `-arg` to show arguments for functions. If no executable is specified, the main program is the default. Use `-pretty` to provide column-aligned output. See Example 27.

Example 27: `lsline` Example

```
% cvcov lsline -pretty -pat main test0000
Functions   Files      Covered   Total     % Coverage
-----------------------------------------------------
main        copyn.c    6         20        30.00%
```

`cvcov lssource [-asm] [-exe` *exe_name*`]` *function experiment test_name*

Displays the source annotated with line counts. The `-asm` switch displays the assembly level source code annotated with line counts. Lines with 0 counts are highlighted to show the absence of coverage. This is useful for mapping to the source level blocks and branches that were not covered. Lines in functions that were not included in the test appear without count annotations.

Example 28, shows a segment of a typical `lssource` ASCII report.

Note: `lssource` requires the code to be compiled with the `-g` option.

Example 28: `lssource` Example

```
% cvcov lssource main test0000
Counts  Source
-------------------------------------------------------------
          #include <stdio.h>
          #include <sys/stat.h>
          #include <sys/types.h>
          #include <fcntl.h>

          #define OPEN_ERR         1
          #define NOT_ENOUGH_BYTES 2
```

```
          #define SIZE_0          3

          int copy_file();

          main (int argc, char *argv[])
1       {
             int bytes, status;

1          if( argc < 4){
0             printf(''copyn: Insufficient arguments.\n'');
0             printf(''Usage: copyn f1 f2 bytes\n'');
0             exit(1);
           }
1          if( argc > 4 ) {
0             printf(''Error: Too many arguments\n'');
0             printf(''Usage: copyn f1 f2 bytes\n'');
0             exit(1);
           }
1          bytes = atoi(argv[3]);
```

`cvcov lstrace [-exe` *exe_name*`] [-pat` *func_pattern*`] [-pretty][-sort` *function* | *type*`]`*experiment* | *test_name*

Shows the argument tracing information. Example 29, shows a typical `lstrace` ASCII report. The Range column shows upper and lower bounds. A dash (`-`) means that side of the bound has not been counted.

Example 29: `lstrace` Example

```
% cvcov lstrace -pretty testtrace
Arguments       Type    Range
--------------------------------
main(argc)      int     -, 4
copy_file(size) int     1, 20
```

Note: `lstrace` requires the code to be compiled with the `-g` option.

`cvcov diff [-arg] [-exe` *exe_name*`] [-functions] [-pretty][-sort diff | function]` *experiment1 experiment2*

Shows the difference in coverage for different versions of the same program. Example 30, shows an example of the `diff` command applied to two tests (although you should make sure that the comparison is relevant). Example 31, page 161, shows `diff` applied to different instrumentations of the same test.

Example 30: `diff` between Two Tests

```
% cvcov diff test0000/exp##0 test0001/exp##0

Experiment 1:    test0000/exp##0
Experiment 2:    test0001/exp##0

Coverages            Exp 1       Exp 2        Differences
-----------------------------------------------------------
Function Coverage    2(100.00%)  1(50.00%)    1(50.00%)
Source Line Coverage 17(48.57%)  5(14.29%)    12(34.29%)
Branch Coverage      0(0.00%)    0(0.00%)     0(0.00%)
Arc Coverage         8(44.44%)   3(16.67%)    5(27.78%)
Block Coverage       19(45.24%)  4(9.52%)     15(35.71%)
```

Example 31: `diff` between Different Instrumentations of the Same Test

```
% cvcov diff test0000/exp##0 test0000/exp##1
Experiment 1:    test0000/exp##0
Experiment 2:    test0000/exp##1

Coverages            Exp 1        Exp 2       Differences
-----------------------------------------------------------
Function Coverage    2(100.00%)    2(100.00%)    0(0.00%)
Source Line Coverage 17(48.57%)    17(47.22%)    0(1.35%)
Branch Coverage      0(0.00%)      0(0.00%)      0(0.00%)
Arc Coverage         8(44.44%)     8(44.44%)     0(0.00%)
Block Coverage       19(45.24%)    19(44.19%)    0(-1.05%)
```

7.2.3 Test Set Commands

A test set is a named collection of tests and other test sets. Test sets can be hierarchical. For example, `compiler_language_suite` might include `C++_suite`, `C_suite`, and `Fortran_suite`, where `Fortran_suite` is a test

set with subdirectories. The following commands support creation, inspection, modification, and deletion of test sets. Both `addtest` and `deltest` are also used with test groups, described in the next section.

`cvcov mktset [-des` *description*`] [-list` *list_file*`][-testname` *test*`]`

Makes a test set. If no test name is specified, the command assigns one automatically.

`cvcov addtest` *test_name test_set_name* | *test_group*

Adds a test or test set to a test set or test group.

`cvcov deltest` *test_name test_set_name* | *test_group*

Removes a test or test set from a test set or test group.

Note: Do not use UNIX commands `mv` and `cp` to rename or copy test sets because they are constructed with absolute files paths.

`cvcov optimize [ -blocks ][ -branches ][ -cbb` *filter_type bb_filter_value* `][ -cbr` *filter_type br_filter_value*`] [ -exe` *exe_name*`] [ -pat` *func_pattern*`] [ -pretty ][ -stat ]`*experiment...*|*test_name* `...`

Selects the minimum set of tests that give the same coverage or meet the given coverage criteria as the given set. The `-blocks` flag shows block coverage for all the selected tests. The `-branches` flag shows branch coverage for all the selected tests. The `-cbb` *filter_type bb_filter_value* gives the basic block coverage criteria for test selection. The rules are the same as the flag `-bf` of the `lsfun` command. The `-cbr` *filter_type br_filter_value* gives the branch coverage criteria for test selection. The rules are the same as the flag `-rf` of `lsfun` command. The `-exe` *exe_name* option lets you specify which executable is targeted for test optimization. If no executable is specified, the main program is the default. The `-pat` *pattern* option lets you specify DSO patterns for calculation of coverage on test selection. The `-pretty` flag aligns column output. The `-stat` flag prints out block and branch coverage for all the selected tests. Without this option, cumulative coverages for block and branch are given. The *experiment ...*|*test_name ...* option lets you specify names of experiments or tests to be optimized. Example 32, demonstrates how test sets are

optimized. In this case, optimizing is applied to all tests matching the expression `test00*`.

Example 32: Optimizing Test Sets

```
% cvcov optimize -pretty -blocks -branches test00*

Test                Block Coverage    Branch Coverage
-----------------------------------------------------
test0000            41.54%            0.00%
test0001            7.69%             10.00%
test0002            7.69%             10.00%
test0003            9.23%             20.00%
test0004            9.23%             20.00%
test0005            6.15%             20.00%
test0006            1.54%             10.00%
Total Coverage      83.08%            90.00%
```

7.2.4 Test Group Commands

A *test group* is a collection of programs to be tested that have a common dynamically shared object (DSO). The coverage testing is limited to activity with the DSO so that the arcs and branches that terminate outside of the DSO will not be included. See descriptions of `addtest` and `deltest` in the previous section as well as the following command.

`cvcov mktgroup [-des` *description*`] [-list` *list_file*`] [-testname` *test*`]` *target1 target2* `...`

This command creates a test group that can contain other tests or test groups. The targets are either the target libraries or DSOs.

Note: Do not use UNIX commands `mv` and `cp` to rename or copy test groups because they are constructed with absolute files paths.

Tester Graphical User Interface Tutorial [8]

This chapter provides a tutorial for the Tester graphical user interface. It covers these topics:

8.1 Setting Up the Tutorial

If you have already set up a tutorial directory for the command line interface tutorial, you can continue to use it. If you remove the subdirectories, your directory names will match exactly; if you leave the subdirectories in, you can add new ones as part of this tutorial.

If you would like the test data built automatically, run the following script:

```
/usr/demos/WorkShop/Tester/setup_Tester_demo
```

To set up a tutorial directory from scratch, do the following; otherwise you can skip the rest of this section.

1. Enter the following:

   ```
   % cp -r /usr/demos/WorkShop/Tester /usr/tmp/tutorial
   % cd /usr/tmp/tutorial
   % echo ABCDEFGHIJKLMNOPQRSTUVWXYZ > alphabet
   % make -f Makefile.tutorial copyn
   ```

   This moves some scripts and source files used in the tutorial to `/usr/tmp/tutorial`, creates a test file named `alphabet`, and makes a simple program, `copyn`, which copies *n* bytes from a source file to a target file.

2. To see how the program works, try a simple test by typing the following at the command line:

   ```
   % copyn alphabet targetfile 10
   % cat targetfile
   ```

```
ABCDEFGHIJ
```

You should see the first 10 bytes of `alphabet` copied to `targetfile`.

8.2 Tutorial #1 — Analyzing a Single Test

Tutorial #1 discusses the following topics:

- Invoking the graphical user interface.
- Instrumenting an executable.
- Making a test.
- Running a test.
- Analyzing the results of a coverage tes.

These topics are all covered in the following section.

8.2.1 Invoking the Graphical User Interface

You typically call up the graphical user interface from the directory that will contain your test subdirectories. This section tells you how to invoke the Tester graphical user interface and describes the main window.

1. Enter **cvxcov** from the tutorial directory to bring up the Tester main window.

   Figure 58, page 168, shows the main Tester window with all its menus displayed.

   Note: You can also access Tester from the `Admin` menu in other WorkShop tools.

2. Observe the features of the Tester window.

   The `Test Name` field is used to display the current test. You can switch to different tests through this field.

   Test results display in the coverage display area. You display the results by choosing an item from the `Queries` menu. You also can select the format of the data from the `Views` menu.

   The `Source` button lets you bring up the standard `Source View` window with Tester annotations. `Source View` shows the counts for each line

included in the test and highlights lines with 0 counts. Lines from excluded functions display but without count annotations.

The `Disassembly` button brings up the `Disassembly View` window for assembly language source. It operates in a similar fashion to the `Source` button.

The `Contribution` button displays a separate window with the contributions to the coverage made by each test in a test set or test group.

A sort button lets you sort the test results by such criteria as function, count, file, type, difference, caller, or callee. The criteria available (shown by the name of the button) depend on the current query.

The status area displays status messages regarding the test.

The area below the status area will display special query-specific fields when you make queries.

You can launch other WorkShop applications from the `Launch Tool` submenu of the `Admin` menu. The applications include the Build Analyzer, Debugger, Parallel Analyzer, Performance Analyzer, and Static Analyzer.

You will find an icon version of the `Execution View` labeled `cvxcovExec`. It is a shell window for viewing test results as they would appear on the command line.

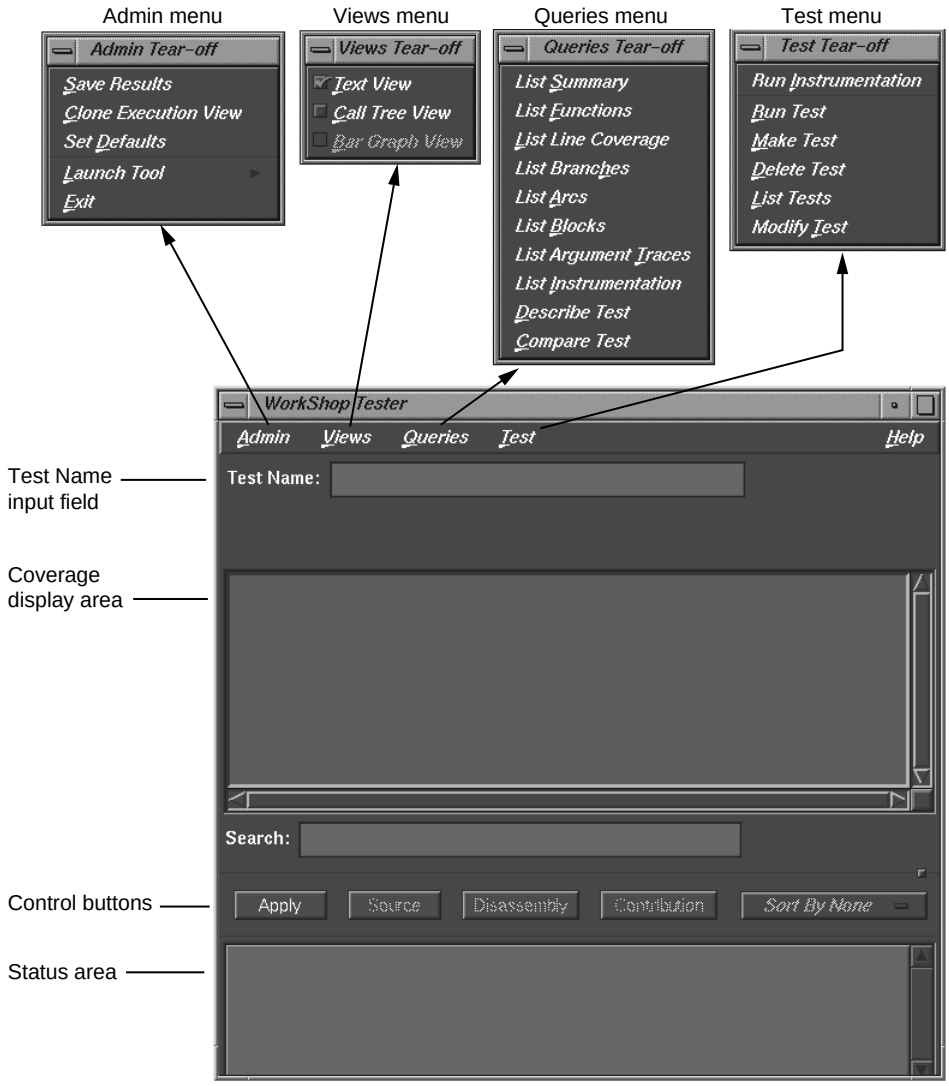

Figure 58. Main Tester Window

Instrumenting an Executable

The first step in providing test coverage is to define the instrumentation criteria in an instrumentation file.

3. On the command line or from `Execution View`, enter the following to see the instrumentation directives in the file `tut_instr_file` used in the tutorials:

```
% cat tut_instr_file
COUNTS -bbcounts -fpcounts -branchcounts
CONSTRAIN main, copy_file
TRACE BOUNDS copy_file(size)
```

   We will be getting all counting information (blocks, functions, source lines, branches, and arcs) for the two functions specified in the CONSTRAIN directive, `main` and `copy_file`. We will also be tracing the *size* argument for the `copy_file` function.

4. Select `Run Instrumentation` from the `Test` menu.

   This process inserts code into the target executable that enables coverage data to be captured. The dialog box shown in Figure 59, page 169, displays when `Run Instrumentation` is selected from the `Test` menu.

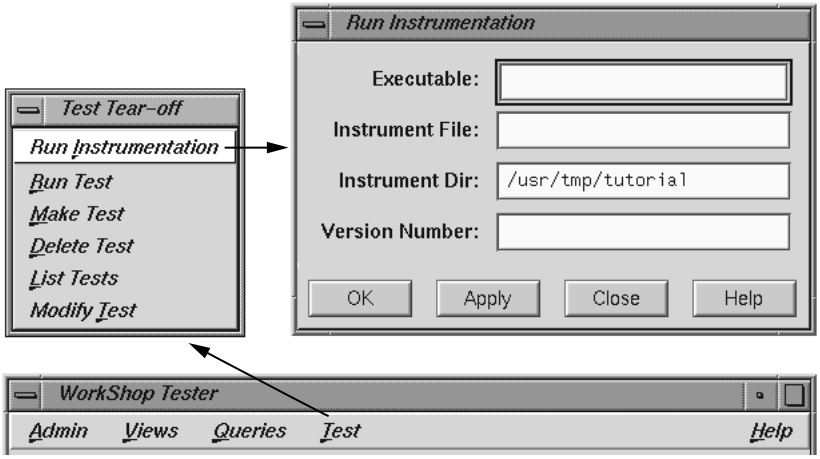

Figure 59. Running Instrumentation

5. Enter **`copyn`** in the `Executable` field.

   The `Executable` field is required, as indicated by the red highlight. You enter the executable in this field.

6. Enter **`tut_instr_file`** in the `Instrument File` field.

The Instrument File field lets you specify an instrumentation file containing the criteria for instrumenting the executable. In this tutorial, we use the file tut_instr_file, which was described earlier.

7. Leave the Instrument Dir and Version Number fields as is.

   The Instrument Dir field indicates the directory in which the instrumented programs are stored. A versioned directory is created (the default is ver##*n*, where *n* is 0 the first time and is incremented automatically if you subsequently change the instrumentation). The version number *n* helps you identify the instrumentation version you use in an experiment. The experiment results directory will have a matching version number. The instrument directory is the current working directory; it can be set from the Admin menu.

8. Click OK.

   This executes the instrumentation process. If there are no problems, the dialog box closes and the message Instrumentation succeeded displays in the status area with the version number created.

Making a Test

A *test* defines the program and arguments to be run, the instrumentation criteria, and descriptive information about the test.

9. Select Make Test from the Test menu.

   This creates a test directory. Figure 60, page 171, shows the Make Test window.

   You specify the name of the test directory in the Test Name field, in this case test0000. The field displays a default directory test<*nnnn*>, where *nnnn* is 0000 the first time and incremented for subsequent tests. You can edit this field if necessary.

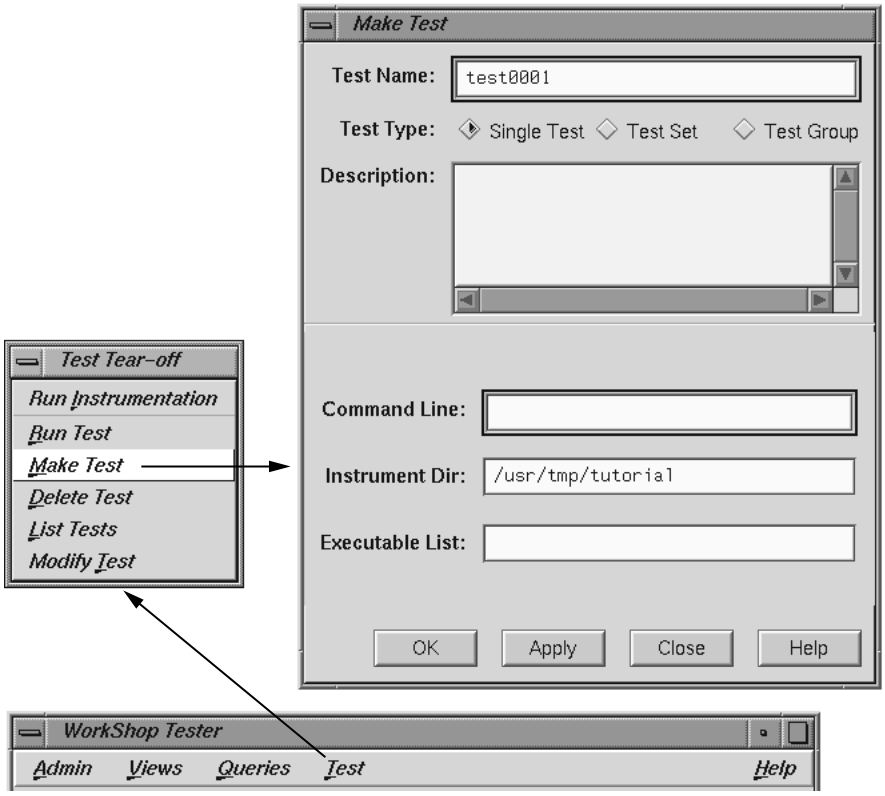

Figure 60. Selecting `Make Test`

10. Enter a description of the test in the `Description` field.

    This is optional, but can help you differentiate between tests you have created.

11. Enter the executable to be tested with its arguments in the `Command Line` field, in this example:

    `copyn alphabet targetfile 20`

    This field is mandatory, as indicated by its highlighting.

12. Leave the remaining fields as is.

Tester supplies a default instrumentation directory in the `Instrument Dir` field. The `Executable List` field lets you specify multiple executables when your main program `fork`s, `exec`s, or `sproc`s other processes.

13. Click `OK` to perform the make test operation with your selections.

    The results of the make test operation display in the status area of the main Tester window.

    Running a Test

    To run a test, we use technology from the WorkShop Performance Analyzer. The instrumented process is set to run, and a monitor process (`cvmon`) captures test coverage data by interacting with the WorkShop process control server (`cvpcs`).

14. Select `Run Test` from the `Test` menu.

    The dialog box shown in Figure 61, page 173, displays. You enter the test directory in the `Test Name` field. You can also specify a version of the executable in the `Version Number` field if you do not want to use the latest, which is the default. The `Force Run` toggle forces the test to be run again even if a test result already exists. The `Keep Performance Data` toggle retains all the performance data collected in the experiment. The `Accumulate Results` toggle sums over the coverage data into the existing experiment results. Both `No Arc Data` and `Remove Subtest Expt` toggles retain less data in the experiments and are designed to save disk space.

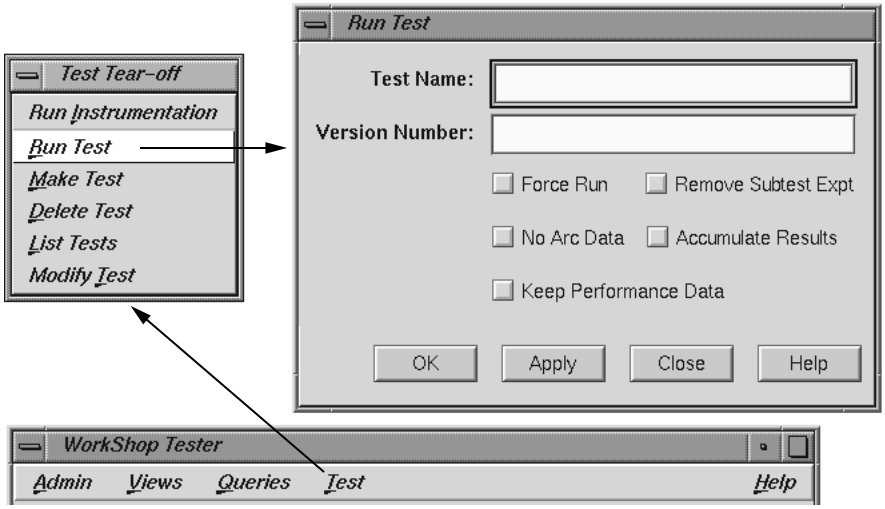

Figure 61. `Run Test` Dialog Box

15. Enter **`test0000`** in the `Test Name` field.

16. Click `OK` to run the test with your selections.

    When the test completes, a status message showing completion displays and you will have data to be analyzed. You can observe the test as it runs in `Execution View`.

    Analyzing the Results of a Coverage Test

    You can analyze test coverage data in many ways. In this tutorial, we will illustrate a simple top-down approach. We will start at the top to get a summary of overall coverage, proceed to the function level, and finally go to the actual source lines.

    Having collected all the coverage data, now you can analyze it. You do this through the `Queries` menu in the main Tester window.

17. Enter **`test0000`** in the `Test Name` field in the main window and select `List Summary` from the `Queries` menu.

    This loads the test and changes the main window display as shown in Figure 62, page 174. The query type (in this case, `List Summary`) is indicated above the display area. Column headings identify the data, which displays in columns in the coverage display area. The status area is

shortened. The query-specific fields (in this case, coverage weighting factors) that appear below the control buttons and status area are different for each query type. You can change the numbers and click `Apply` to weight the factors differently. The `Executable List` button brings up the `Target List` dialog box. It displays a list of executables used in the experiment and lets you select different executables for analysis. You can select other experiments from the experiment menu (`Expt`).

`List Summary` shows the coverage data (number of coverage hits, total possible hits, percentage, and weighting factor) for functions, source lines, branches, arcs, and blocks. The last coverage item is the weighted average, obtained by multiplying individual coverage averages by the weighting factors and summing the products.

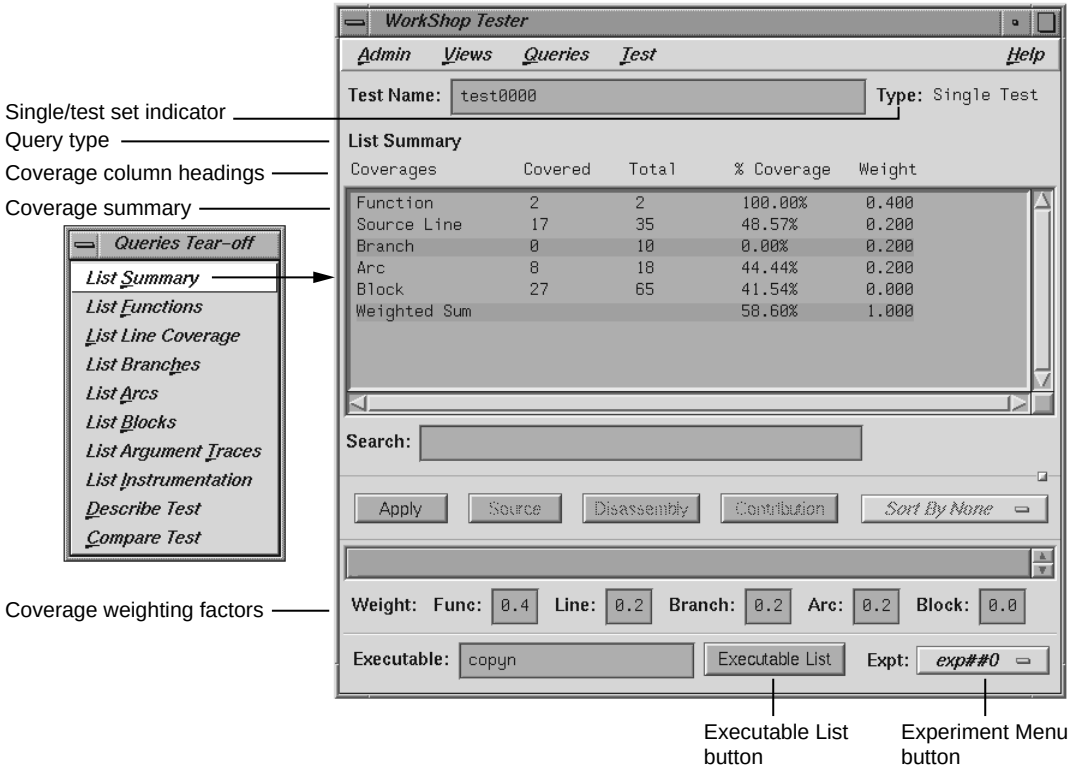

Figure 62. `List Summary Query` Window

18. Select `List Functions` from the `Queries` menu.

    This query lists the coverage data for functions specified for inclusion in this test. The default version is shown in Figure 63, with the available options.

Figure 63. `List Functions` Query with Options

If there are functions with 0 counts, they will be highlighted. The default column headings are `Functions`, `Files`, and `Counts`.

19. Click the `Blocks` and `Branches` toggles.

    The `Blocks` and `Branches` toggle buttons let you display these items in the function list. Figure 64, page 176, shows the display area with `Blocks` and `Branches` enabled.

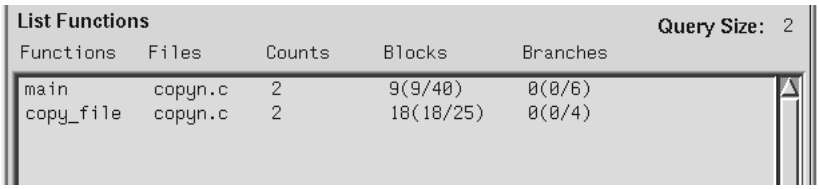

Figure 64. `List Functions` Display Area with `Blocks` and `Branches`

The `Blocks` column shows three values. The number of blocks executed within the function is shown first. The number of blocks covered out of the total possible for that function is shown inside the parentheses. If you divide these numbers, you will arrive at the percentage of coverage.

Similarly, the `Branches` column shows the number of branches covered, followed by the number covered out of the total possible branches. The term *covered* means that the branch has been executed under both true and false conditions.

20. Select the function `main` in the display area and click `Source`.

    The `Source View` window displays with count annotations as shown in Figure 65, page 177. Lines with 0 counts are highlighted in the display area and in the vertical scroll bar area. Lines in excluded functions display with no count annotations.

21. Click the `Disassembly` button in the main window.

    The `Disassembly View` window displays with count annotations as shown in Figure 66, page 177. Lines with 0 counts are highlighted in the display area and in the vertical scroll bar area.

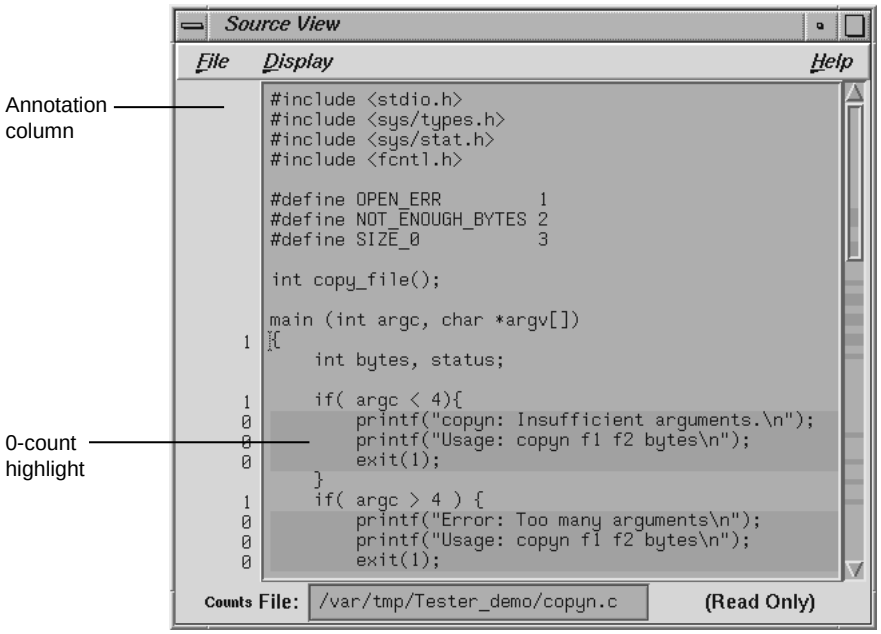

Figure 65. `Source View` with Count Annotations

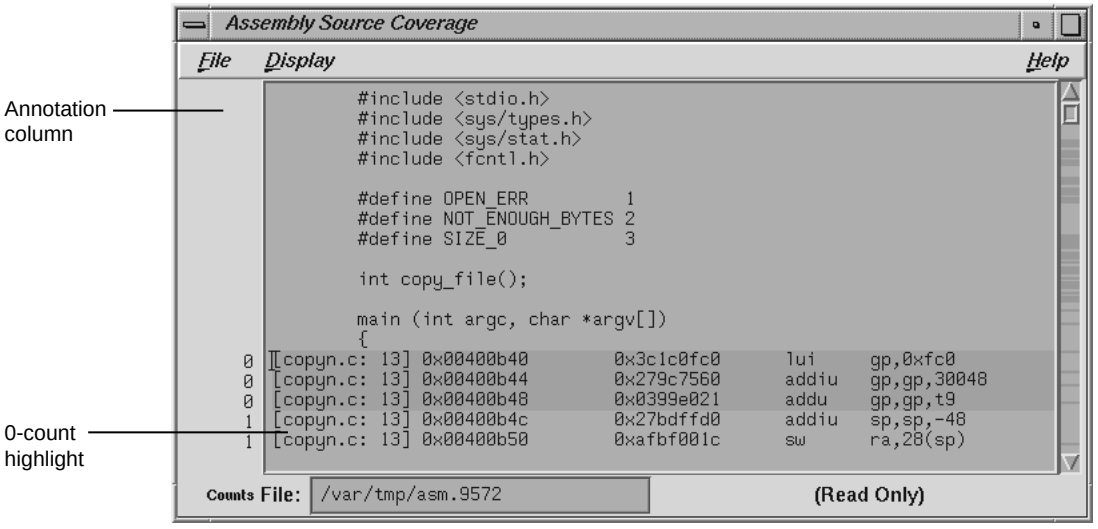

Figure 66. `Disassembly View` with Count Annotations

8.3 Tutorial #2 — Analyzing a Test Set

In the second tutorial, we are going to create additional tests with the objective of achieving 100% overall coverage. From examining the source code, it seems that the 0-count lines in `main` and `copy_file` are due to error-checking code that is not tested by `test0000`.

Note: This tutorial needs `test0000`, which was created in the previous tutorial.

1. Select `Make Test` from the `Test` menu.

   This displays the `Make Test` dialog box. It is easy to enter a series of tests. Using the `Apply` button in the dialog box instead of the `OK` button completes the task without closing the dialog box. The `Test Name` field supplies an incremented default test name after each test is created.

   We are going to create a test set named `tut_testset` and add to it 8 tests in addition to `test0000` from the previous tutorial. The tests `test0001` and `test0002` pass too few and too many arguments, respectively. `test0003` attempts to copy from a file named `no_file` that does not exist. `test0004` attempts to pass 0 bytes, which is illegal. `test0005` attempts to copy 20 bytes from a file called `not_enough`, which contains only one byte. In `test0006`, we attempt to write to a directory without proper permission. `test0007` tries to pass too many bytes. In `test0008`, we attempt to copy from a file without read permission.

   The following steps show the command line target and arguments and description for the tests in the tutorial. The descriptions are helpful but optional. Figure 67, page 179, shows the features of the dialog box you will need for creating these tests.

2. Enter **`copyn alphabet target`** in the `Command Line` field, **`not enough arguments`** in the `Description` field, and click `Apply` (or simply press the `Return` key) to make `test0001`.

3. Enter **`copyn alphabet target 20 extra_arg`** in the `Command Line` field, **`too many arguments`** in the `Description` field, and click `Apply` to make `test0002`.

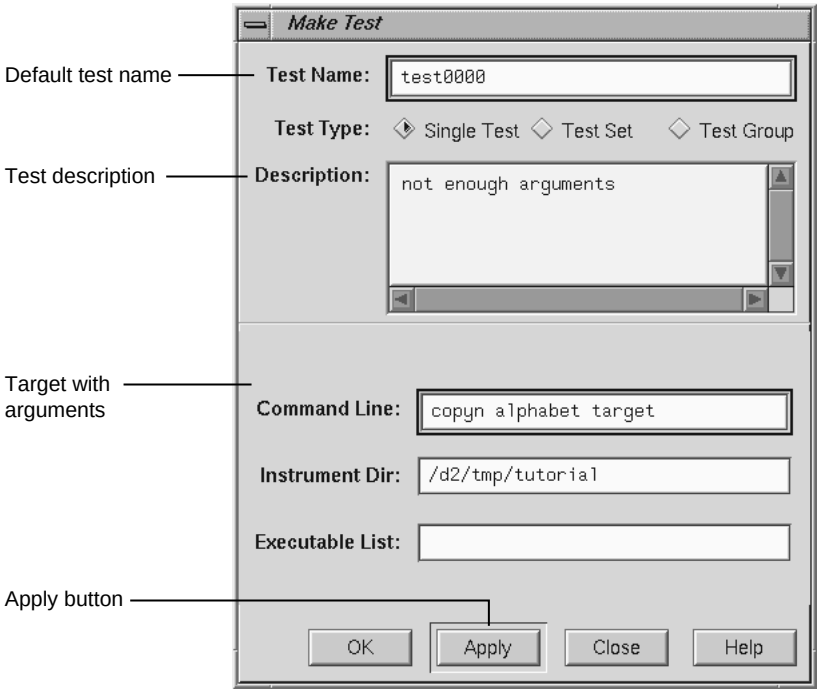

Figure 67. `Make Test` Dialog Box with Features Used in Tutorial

4. Enter **`copyn no_file target 20`** in the `Command Line` field, **`cannot access file`** in the `Description` field, and click `Apply` to make `test0003`.
5. Enter **`copyn alphabet target 0`** in the `Command Line` field, **`pass bad size arg`** in the `Description` field, and click `Apply` to make `test0004`.
6. Enter **`copyn not_enough target 20`** in the `Command Line` field, **`not enough data`** in the `Description` field, and click `Apply` to make `test0005`.
7. Enter **`copyn alphabet /usr/bin/target 20`** in the `Command Line` field, **`cannot create target executable due to permission problems`** in the `Description` field, and click `Apply` to make `test0006`.

8. Enter **`copyn alphabet targetfile 200`** in the `Command Line` field, **`size arg too big`** in the `Description` field, and click `Apply` to make `test0007`.

9. Enter **`copyn /usr/etc/snmpd.auth targetfile 20`** in the `Command Line` field, **`no read permission on source file`** in the `Description` field, and click `Apply` to make `test0008`.

   We now need to create the test set that will contain these tests.

10. Click the `Test Set` toggle in the `Test Type` field.

    This changes the dialog box as shown in Figure 68, page 180.

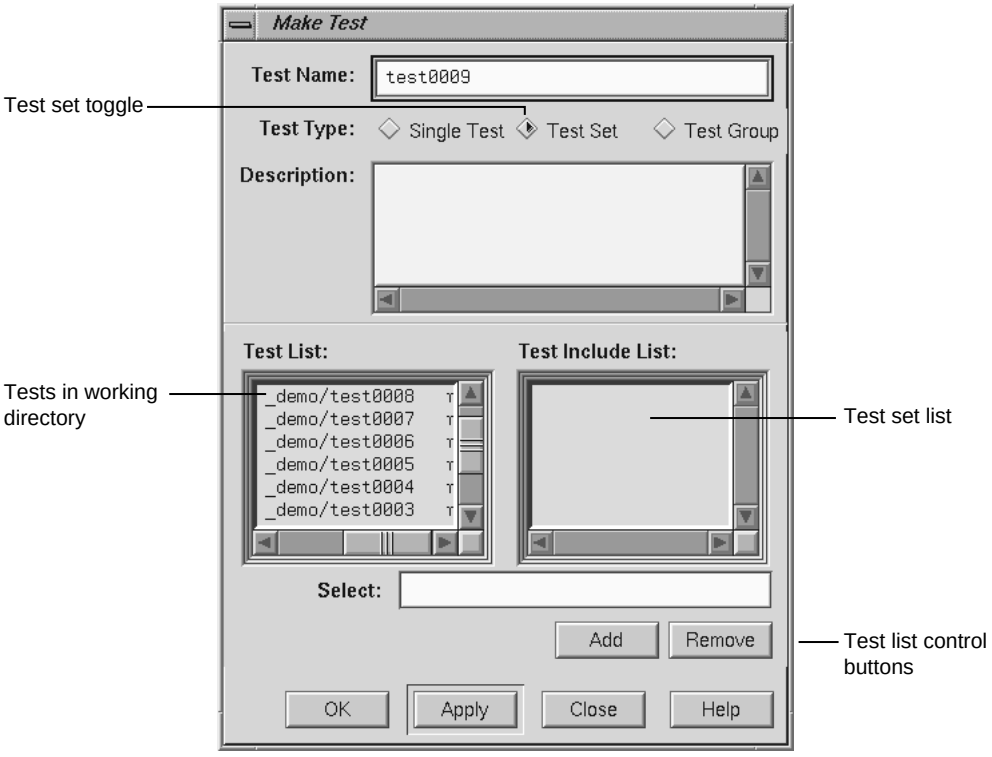

Figure 68. `Make Test` Dialog Box for Test Set Type

11. Change the default in the `Test Name` field to **`tut_testset`**.

This is the name of the new test set. Now we have to add the tests to the test set.

12. Select the first test in the `Test List` field and click `Add`.

    This displays the selected test in the `Test Include List` field, indicating that it will be part of the test set after you click `OK` (or `Apply` and `Close`).

13. Repeat the process of selecting a test and clicking `Add` for each test in the `Test List` field. When all tests have been added to the test set, click `OK`.

    This saves the test set as specified and closes the `Make Test` dialog box.

14. Enter **`tut_testset`** in the `Test Name` field and select `Describe Test` from the `Queries` menu.

    This displays the test set information in the display area of the main window.

15. Select `Run Test` from the `Test` menu, enter **`tut_testset`** in the `Test Name` field in the `Run Test` dialog box.

    This runs all the tests in the test set.

16. Enter **`tut_testset`** in the `Test Name` field in the main Tester window and select `List Summary` from the `Queries` menu.

    This displays a summary of the results for the entire test set.

17. Select `List Functions` from the `Queries` menu.

    This step serves two purposes. It enables the `Source` button so that we can look at counts by source line. It displays the list of functions included in the test, from which we can select functions to analyze.

18. Click the `main` function, which is displayed in the function list, and click the `Source` button.

    This displays the source code, with the counts for each line shown in the annotations column. Note that the counts are higher now and full coverage has been achieved at the source level (although not necessarily at the assembly level).

8.4 Tutorial #3 — Exploring the Graphical User Interface

The rest of this chapter shows you how to use the graphical user interface (GUI) to analyze test data. The GUI has all the functionality of the command

line interface and in addition shows the function calls, blocks, branches, and arcs graphically.

For a discussion of applying Tester to test set optimization, refer to Section 6.4, page 139. To learn more about test groups, see Section 6.5, page 142. Although these are written for the command line interface, you can use the graphical interface to follow both tutorials.

1. Enter **`test0000`** in the `Test Name` field of the main window and press the `Return` key.

   Since `test0000` has incomplete coverage, it is more useful for illustrating how uncovered items appear.

2. Select `List Functions` from the `Queries` menu.

   The list of functions displays in the text view format.

3. Select `Call Tree View` from the `Views` menu.

   The Tester main window changes to call graph format. Figure 69, page 183, shows a typical call graph. Initially, the call graph displays the `main` function and its immediate callees.

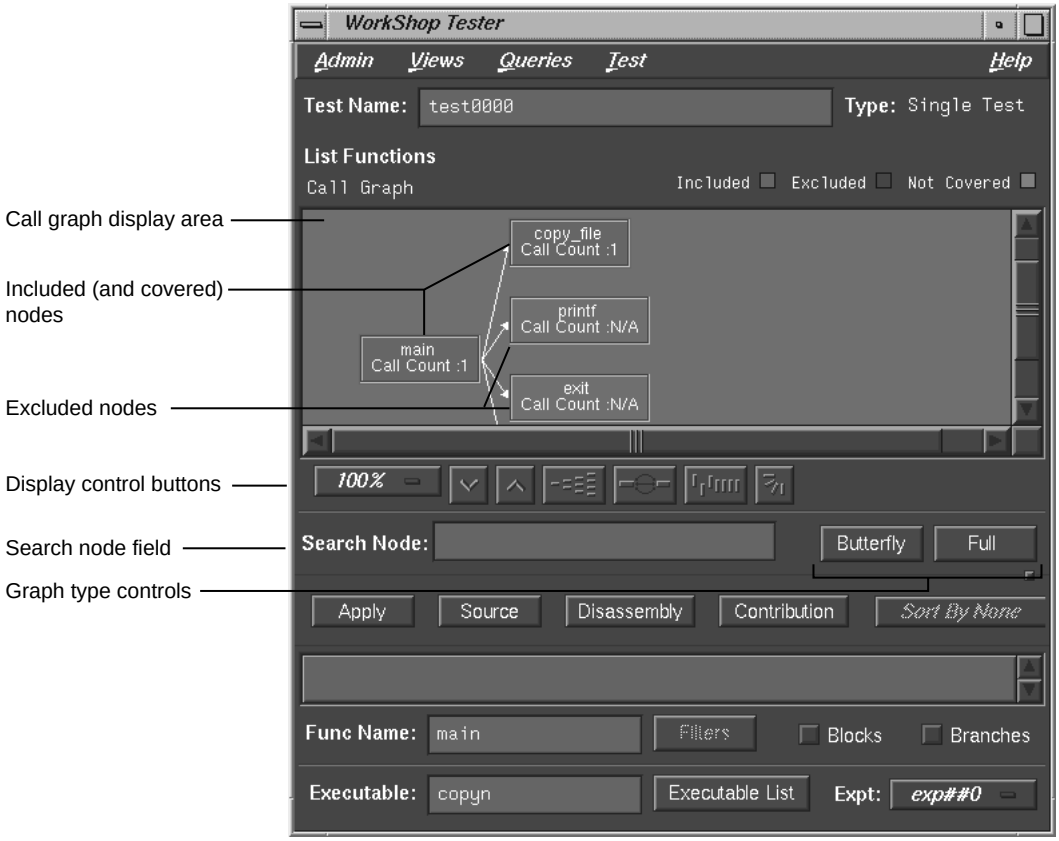

Figure 69. Call Graph for `List Functions` Query

The call graph displays functions as nodes and calls as connecting arrows. The nodes are annotated by call count information. Functions with 0 counts are highlighted. Excluded functions when visible appear in the background color.

The controls for changing the display of the call graph are just below the display area (see Figure 70, page 184).

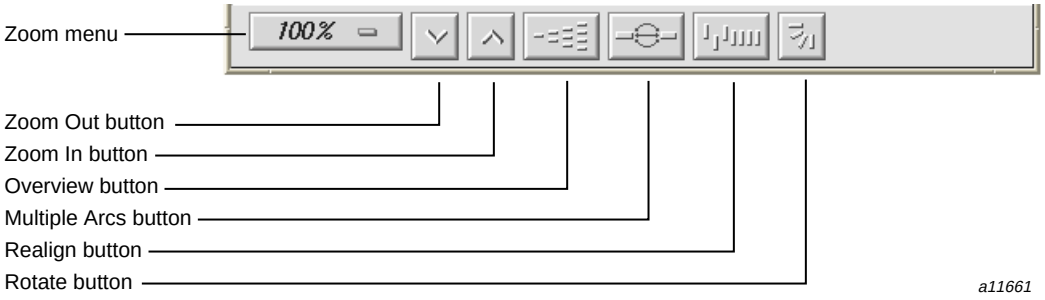

Figure 70. `Call Graph` Display Controls

These facilities are:

`Zoom menu` icon	Shows the current scale of the graph. If clicked on, a popup menu appears displaying other available scales. The scaling range is between 15% and 300% of the nominal (100%) size.
`Zoom Out` icon	Resets the scale of the graph to the next (available) smaller size in the range.
`Zoom In` icon	Resets the scale of the graph to the next (available) larger size in the range.
`Overview` icon	Invokes an overview popup display that shows a scaled-down representation of the graph. The nodes appear in the analogous places on the overview popup, and a white outline may be used to position the main graph relative to the popup. Alternatively, the main graph may be repositioned with its scroll bars.
`Multiple Arcs` icon	Toggles between single and multiple arc mode. Multiple arc mode is extremely useful for the `List Arcs` query, because it indicates

	graphically how many of the paths between two functions were actually used.
`Realign` icon	Redraws the graph, restoring the positions of any nodes that were repositioned.
`Rotate` icon	Flips the orientation of the graph between horizontal (calling nodes at the left) and vertical (calling nodes at the top).

Entering a function in the `Search Node` field scrolls the display to the portion of the graph in which the function is located.

There are two buttons controlling the type of graph. Entering a node in the `Func Name` field and clicking `Butterfly` displays the calling and called functions for that node only (`Butterfly` mode is the default). Selecting `Full` displays the entire call graph (although not all portions may be visible in the display area).

4. Select `List Arcs` from the `Queries` menu.

   The `List Arcs` query displays coverage data for calls made in the test. Because we were just in call graph mode for the `List Functions` query, `List Arcs` comes up in call graph rather than text mode.

   See Figure 71, page 186. To improve legibility, this figure has been scaled up to 150% and the nodes moved by middle-click-dragging the outlines. Arcs with 0 counts are highlighted in color. Notice that in `List Arcs`, the arcs rather than the nodes are annotated.

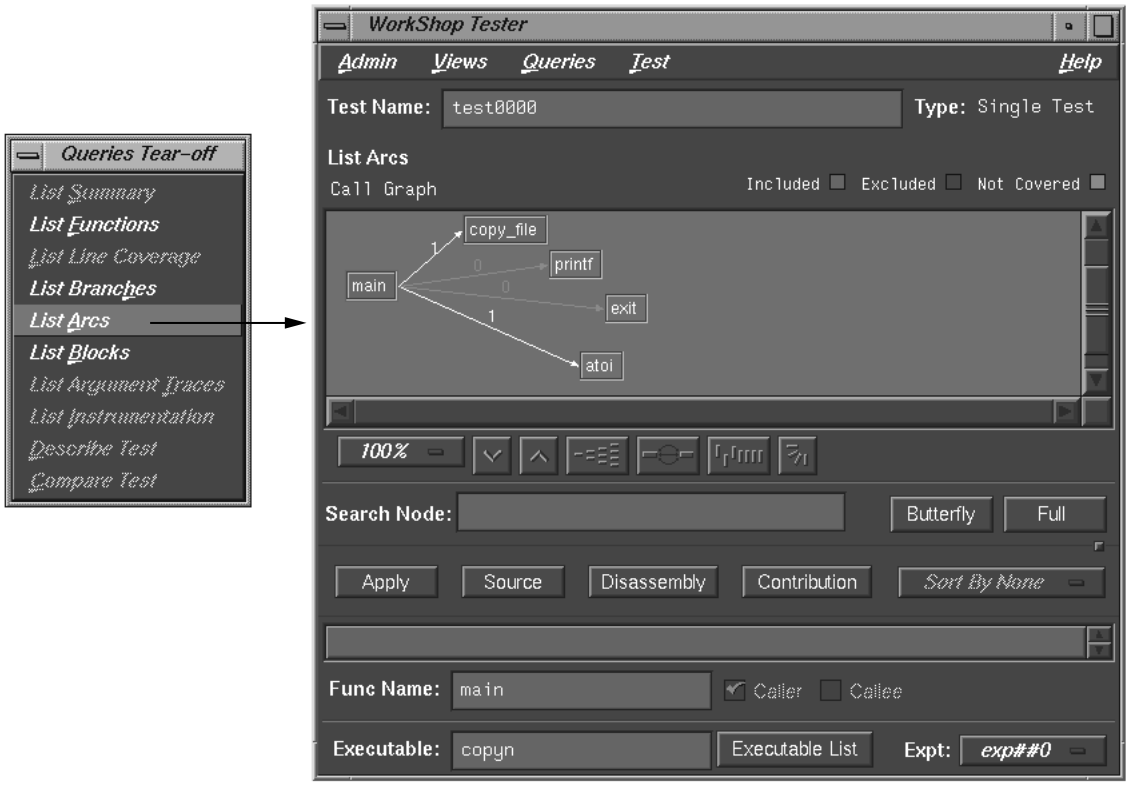

Figure 71. `Call Graph` for `List Arcs` Query

5. Click the `Multiple Arcs` button (the third button from the right in the row of display controls).

   This displays each of the potential arcs between the nodes. See Figure 72, page 187. Arcs labeled N/A connect excluded functions and do not have call counts.

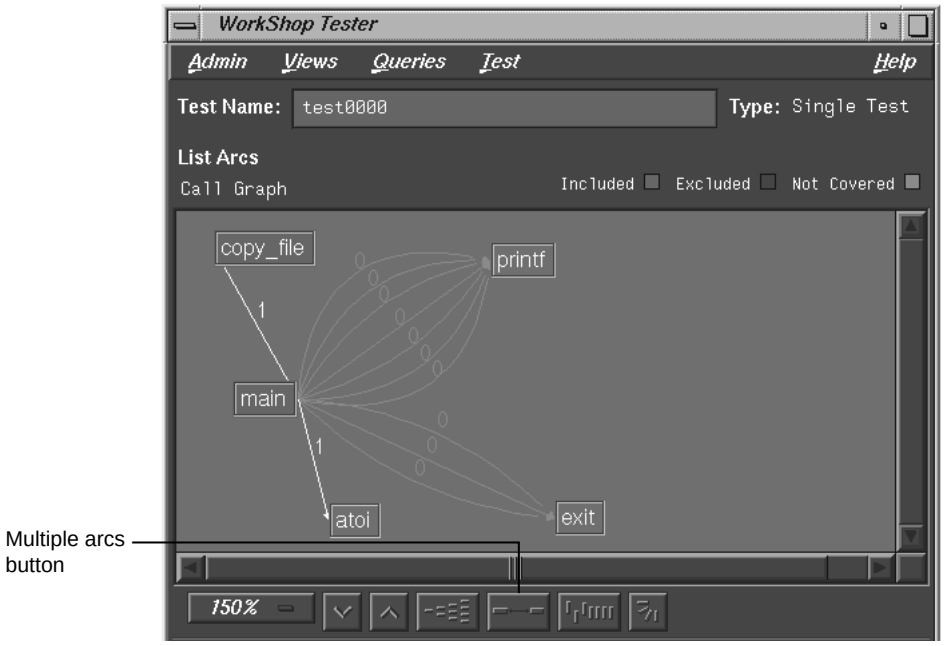

Figure 72. `Call Graph` for `List Arcs` Query — Multiple Arcs

6. Select `Text View` from the `Views` menu.

   This returns the display area to text mode from call graph mode. See Figure 73, page 188.

   The `Callers` column lists the calling functions. The `Callees` column lists the functions called. `Line` provides the line number where the call occurred; this is particularly useful if there are multiple arcs between the caller and callee. The `Files` column identifies the source code file. `Counts` shows the number of times the call was made.

   You can sort the data in the `List Arcs` query by count, file, caller, or callee.

Figure 73. Test Analyzer Queries: `List Arcs`

7. Select `List Blocks` from the `Queries` menu.

   The window should be similar to Figure 74, page 189. The data displays in order of blocks, with the starting and ending line numbers of the block indicated. Blocks that span multiple lines are labeled sequentially in parentheses. The count for each block is shown with 0-count blocks highlighted.

   Caution: Listing all blocks in a program may be very slow for large programs. To avoid this problem, limit your `List Blocks` operation to a single function.

Figure 74. Test Analyzer Queries: `List Blocks`

You can sort the data for `List Blocks` by count, file, or function.

8. Select `List Branches` from the `Queries` menu.

   The `List Branches` query displays a window similar to Figure 75, page 190.

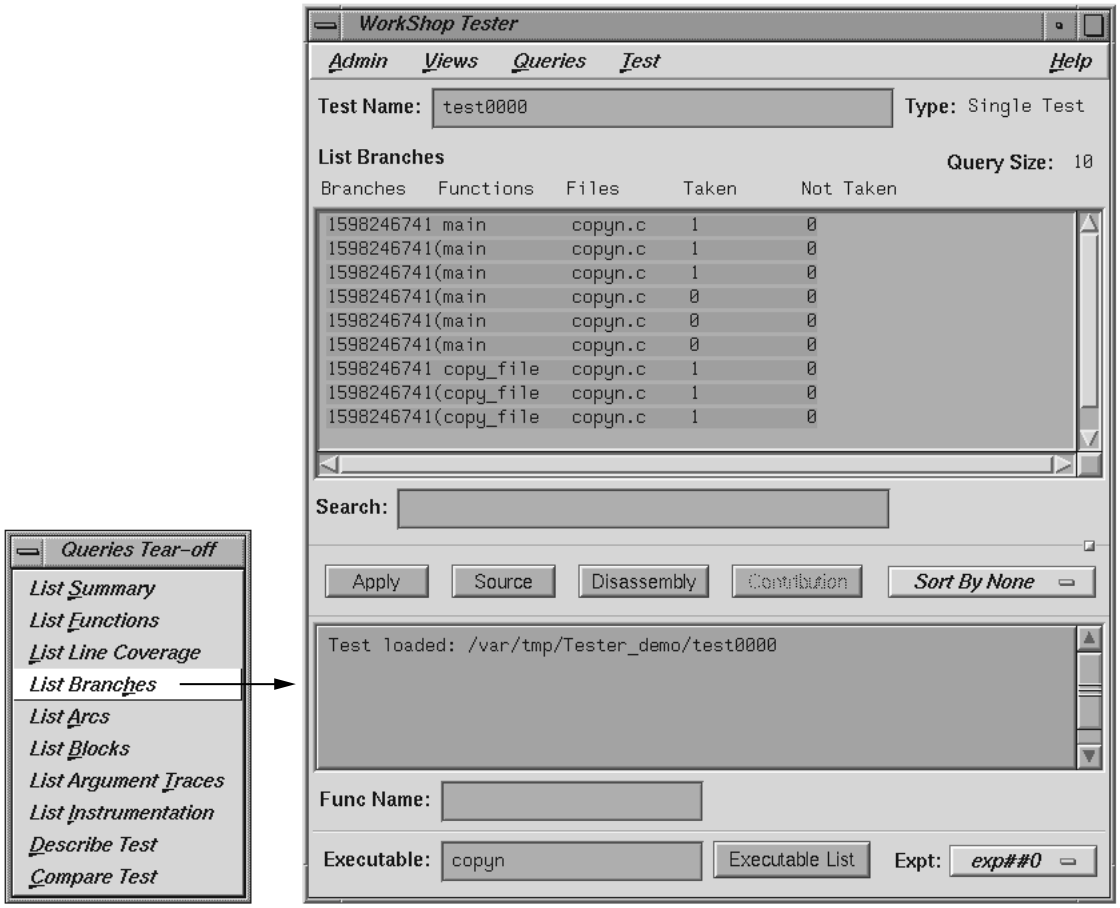

Figure 75. Test Analyzer Queries: `List Branches`

The first column shows the line number in which the branch occurs. If there are multiple branches in a line, they are labeled by order of appearance within trailing parentheses. The next two columns indicate the function containing the branch and the file. A branch is considered covered if it has been executed under both true and false conditions. The `Taken column` indicates the number of branches that were executed only under the true condition. The `Not Taken` column indicates the number of branches that were executed only under the false condition.

The `List Branches` query permits sorting by function or file.

Tester Graphical User Interface Reference [9]

This chapter describes the Tester graphical user interface. It contains these sections:

When you run `cvxcov`, the main Tester window opens and an iconized version of the Execution View appears on your screen. It displays the output and status of a running program and accepts input. To open a closed Execution View, see "Clone Execution View" in Section 9.6, page 228.

9.1 Accessing the Tester Graphical Interface

There are two methods of accessing the Tester graphical user interface:

- Type **`cvxcov`** at the command line with these optional arguments: `-testname` *test* to load the test; `-ver` to show the Tester release version; and `-scheme` *schemename* to set a predefined color scheme.
- Select `Tester` from the `Launch Tool` submenu in a WorkShop `Admin` menu (see Figure 76, page 192). The major WorkShop tools, the Debugger, Static Analyzer, and Build Manager provide `Admin` menus from which you can access Tester.

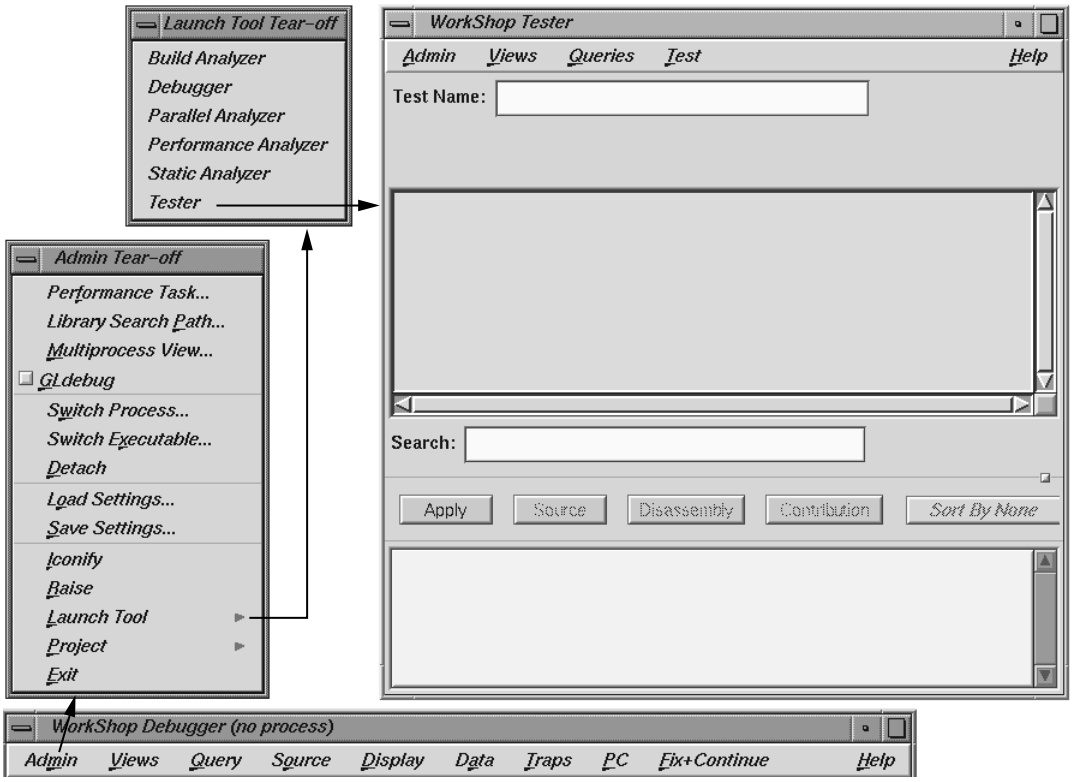

Figure 76. Accessing Tester from the WorkShop Debugger

9.2 Main Window and Menus

The main window and its menus are shown in Figure 77, page 193.

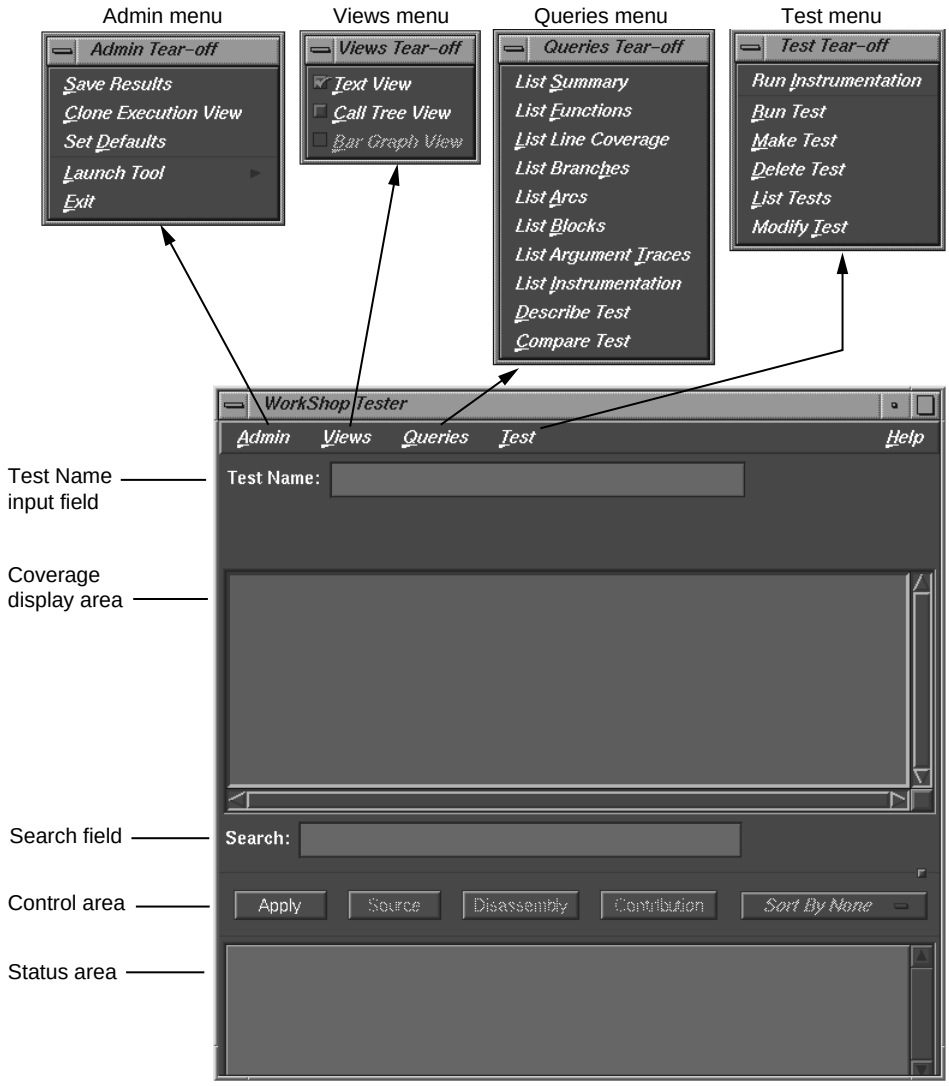

Figure 77. Main Test Analyzer Window

9.2.1 `Test Name` Input Field

The current test is entered (and displayed) in the `Test Name` field. You can switch to a different test, test set, or test group through this field. To the right, the `Type` field indicates whether it is a Single Test, Test Set, or Test Group. You

can select a test (test set or test group) from the List Tests dialog box under the Test menu, to appear in the Test Name field in the main window.

9.2.2 Coverage Display Area

Test results display in the coverage display area. You select the results by choosing an item from the Queries menu. You can select the format of the data—text, call tree, or bar chart— from the Views menu. (Note that the Text View format is available for all queries, whereas the other two views are limited.)

The Query Type displays under the Test Name field, just over the display. It is followed on the far right of the window by the Query Size (number of items in the list). Headings above the display are specific to each query.

9.2.3 Search Field

The Search field lets you look for strings in the coverage data. It uses an incremental search, that is, as you enter characters, the highlight moves to the first matching target. When you press the Return key, the highlight moves to the next occurrence.

9.2.4 Control Area Buttons

The Apply button is a general-purpose button for terminating data entry in text fields; you can use the Return key equivalently. Both start the query.

The Source button lets you bring up the standard Source View window with Tester annotations. Source View shows the counts for each line and highlights lines with 0 counts. By default, Source View is shared with other applications. For example, if cvstatic performs a search for function A, the results of the query overwrite Tester query results that are in the shared Source View. To stop sharing Source View with other applications, set the following resource:

```
cvsourceNoShare:   True
```

The Disassembly button brings up the Disassembly View window, called Assembly Source Coverage, which operates at the machine level in a similar fashion to the Source View. This view is not shared with other applications.

Note: If a test has very large counts, there may not be enough space in the `Source View` and `Disassembly View` windows to display them. To make more room, increase the *canvasWidth* resource in the `Cvxcov` app-defaults file, `Cvxcov*test*testdata*canvasWidth`.

The `Contribution` button brings up the `Test Contribution` window with the contributions made by each test so that you can compare the results. It is available for the queries `List Functions`, `List Arcs`, and `List Blocks`. When the tests do not fit on one page, multiple pages are used. Use the `Previous Page` and `Next Page` buttons to display all the tests.

The `Sort` button lets you sort the test results by criteria such as function, count, file, type, difference, caller, or callee. The criteria available depend on the current query.

9.2.5 Status Area and Query-Specific Fields

The status area displays status messages that confirm commands, issue warnings, and indicate error conditions. When you enter a test name in the `Test Name` field, the `Func Name` field appears (along with other items) in the status area for use with queries. Entering a function in this field displays the coverage results limited to that function only.

Additional items display in the area below the status area that change when you select commands from the `Queries` menu. These items are specific to the query selected. Some of these items can be used as defaults (see Section 9.5, page 209).

9.2.6 Main Window Menus

The `Admin` menu lets you perform general housekeeping concerning saving files, setting defaults, changing directories, launching other WorkShop applications, and exiting.

The `Test` menu lets you create, modify, and run tests, test sets, and test groups.

The `Views` menu lets you choose one of the following modes:

- Text mode, which displays results numerically in columns
- Graphical mode, which displays the following:
  - Functions as nodes (rectangles) annotated by results
  - Calls as arcs (connecting arrows)

- Bar graph mode, which displays the summary of a test as a bar graph.

The `Queries` menu lets you analyze the results of tests. The `Help` menu is standard in all tools.

9.3 `Test` Menu Operations

All operations for running tests are accessed from the `Test` menu in the main Tester window. Figure 78, page 197, shows the dialog boxes used to perform test operations.

The `Test` menu provides the following selections:

`Run Instrumentation`	Instruments the target executable. Instrumentation adds code to the executable to collect coverage data. For a more detailed discussion of instrumentation and instrument files, see Section 5.2.1, page 115.

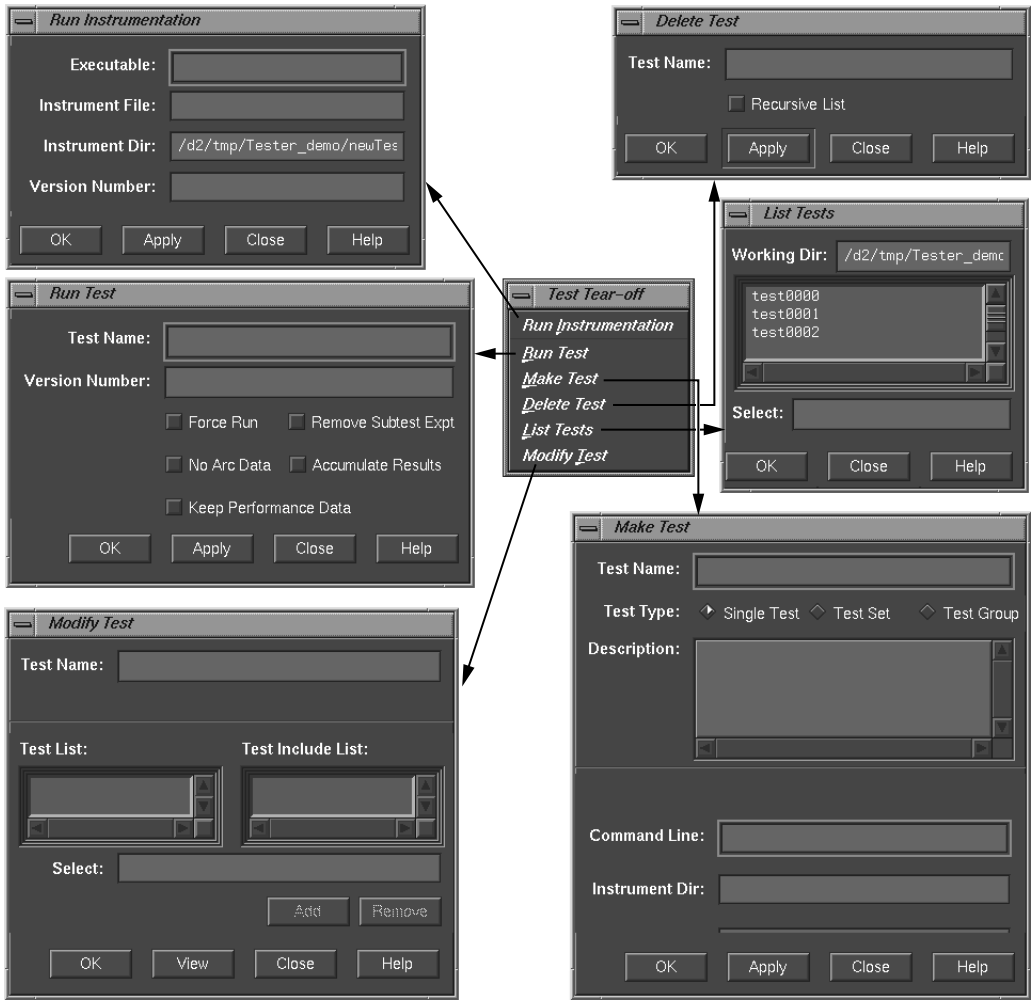

Figure 78. `Test` Menu Commands

The `Run Instrumentation` dialog box (see Figure 79, page 198) provides these fields:

- `Executable` lets you enter the name of the target.
- `Instrumentation File` is for entering the instrumentation file, which is an ASCII

description of the instrumentation criteria for the experiment.

- `Instrumentation Dir` lets you enter the directory in which the instrumentation file is stored (not necessary if you are using the current working directory).
- `Version Number` lets you specify the version number of the instrumentation directory (`ver##<versionnumber>`). If this field is left blank, the version number increments automatically.

  If you are testing multiple executables (that is, testing coverage of an executable that `fork`s, `exec`s, or `sproc`s other processes), then you need to store these in the same instrumentation directory. You do this by entering the same number in the `Version Number` field.

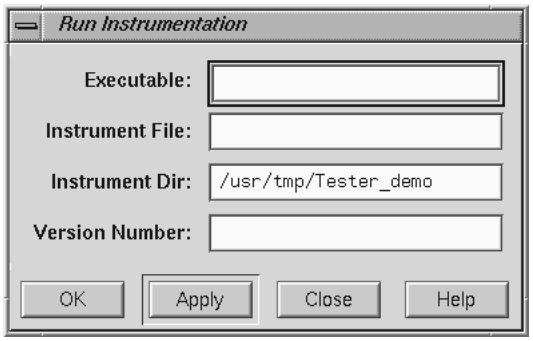

Figure 79. `Run Instrumentation` Dialog Box

`Run Test`

Invokes the executable with selected arguments and collects the coverage data. The `Run Test` dialog box (see Figure 80, page 200) provides these fields and buttons:

- `Test Name` is for entering the test name.

- `Version Number` is for entering the version number of the directory (`ver## <number>`) containing the instrumented executable. If you are using the most current (highest) version number, then you can leave the field blank; otherwise, you need to enter the desired number.
- `Force Run` is a toggle that when turned on causes the test to be run even if results already exist.
- `Keep Performance Data` is a toggle that when turned on retains all the performance data collected in the experiment.
- `Accumulate Results` is a toggle that when turned on accumulates (sums over) the coverage data into the existing experiment results.
- `No Arc Data` prevents arc information from being collected in the experiment. It cannot be used with `List Arcs` or a `Call Tree View`. `List Summary` and `Compare Test` will have 0% coverage on arc items. Use it to save space if you do not need arc data.
- `Remove Subtest Expt` removes results for individual subtests for test sets or test groups, letting you see the top level and taking less space. There will be no data to query if you are querying a subtest.

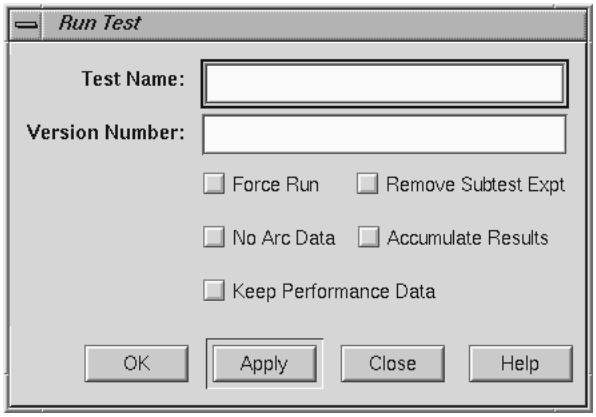

Figure 80. `Run Test` Dialog Box

`Make Test`

Creates a test directory where the coverage data is to be stored and stores a TDF (test description file).

The `Make Test` dialog box (see Figure 81, page 201) provides these fields for tests, test sets, and test groups:

- `Test Name` is for entering the test name.
- `Test Type` is a toggle for indicating the type of test: single, test set, or test group (for dynamically shared objects).
- `Description` lets you enter a description to document the test.

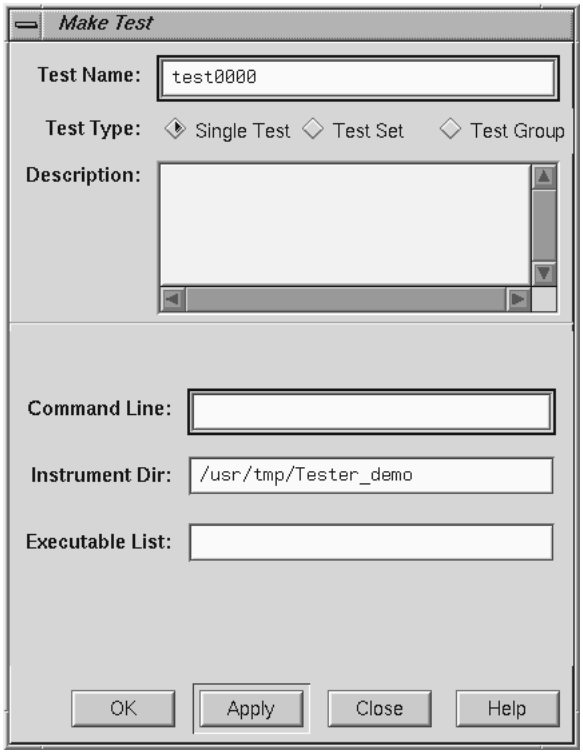

Figure 81. `Make Test` Dialog Box

If you select `Single Test`, the following fields are provided:

- `Command Line` lets you enter the target and any arguments to be used in the test.
- `Instrument Dir` is the directory in which the instrumentation file and related data are stored (not necessary if current working directory).
- `Executable List` is used if you are testing coverage of an executable that `fork`s, `exec`s, or `sproc`s other processes and want to include those processes. You must specify these executables in the `Executable List` field.

If you select `Test Set`, the following fields and buttons are provided:

- `Test List` contains all the tests in the working directory.
- `Test Include List` (to the right) displays tests included in the test set or test group.
- `Add` looks at the selected item in the `Test List` or `Select field` and adds it to the `Test Include List`.
- `Remove` looks at the selected item in the `Test Include List` and removes it.
- `Select` displays the currently selected test.

For a test group (see Figure 82, page 203), the following field is added to the same fields and buttons used for a test set:

- `Targets` lets you enter a list of target DSOs or shared libraries, separated by spaces.

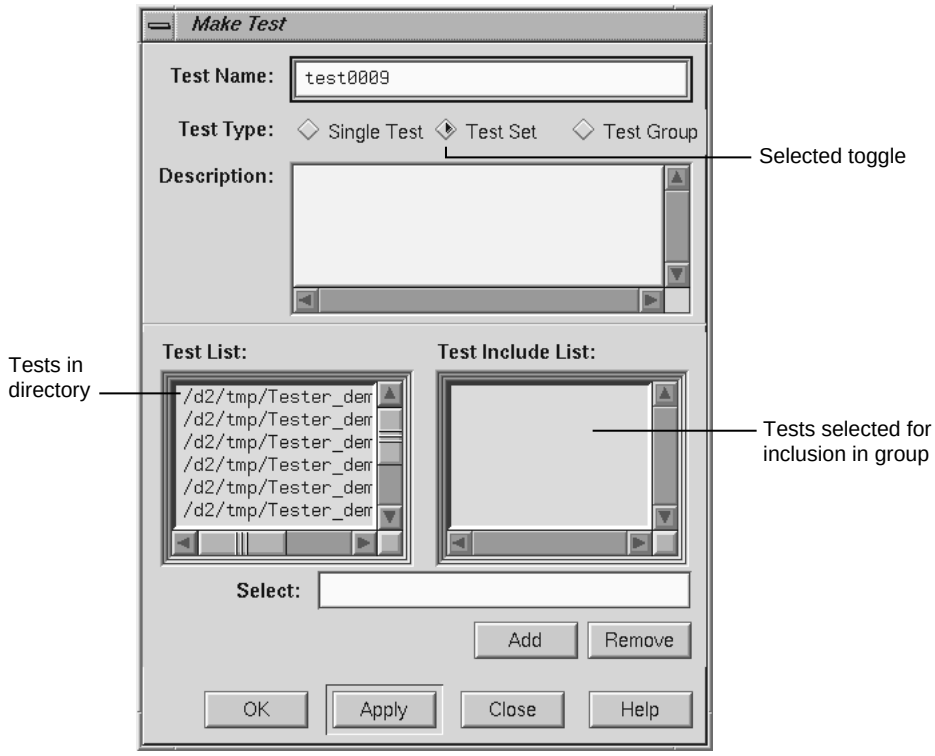

Figure 82. `Make Test` Dialog Box with Test Group Selected

`Delete Test`

Removes the specified test directory and its contents. The `Delete Test` dialog box (see Figure 83, page 204) provides these fields:

- `Test Name` is for entering the test name.
- `Recursive List` is a toggle that when turned on includes all subtests in the removal of test sets and test groups.

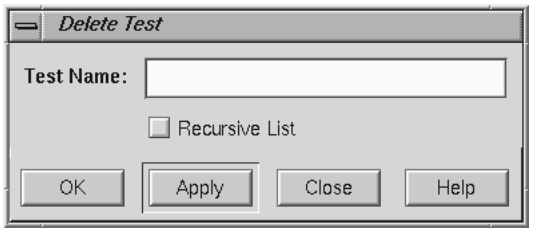

Figure 83. `Delete Test` Dialog Box

`List Tests`

Shows you the tests in the current working directory. The `List Tests` dialog box (see Figure 84, page 205) provides these fields:

- `Working Dir` shows the directory containing the tests.
- A scrollable list field displays the tests present in the specified directory. The scroll bars let you navigate through the tests if they do not fit completely in the field. Clicking an item places it in the `Select` field. Double-clicking on a test selects and loads it.
- `Select` displays the test name you type in or that you clicked in the list. Click `OK` to load your selection into the `Test Name` field of the main Tester window.
- `Close` lets you exit without loading a selection.

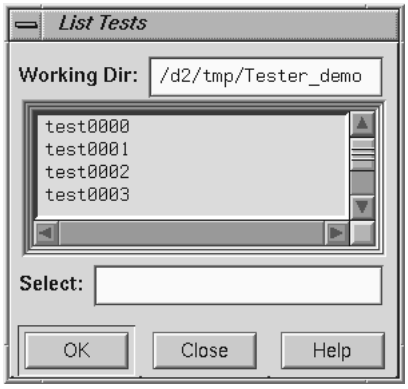

Figure 84. `List Tests` Dialog Box

`Modify Test`

Lets you modify a test set or test group. You enter the test name in the `Test Name` field and press the `Return` key or click the `View` button to load it. The `View` button changes to `Apply`, the `Test List` field displays tests in the current working directory, and the `Test Include List` field displays the contents of the test set or test group. You can then add or delete tests, test sets, or test groups in the current test set or test group, respectively. The `Modify Test` dialog box (see Figure 85, page 206) has these fields:

- `Test Name` is for entering the test name.
- `Test List` displays the tests in the current directory.
- `Test Include List` displays the subtests for the test specified in the `Test Name` field.
- `Select` displays the test currently selected for adding or removing. You can enter the test directly in this field instead of selecting it from the `Test List` or `Test Include List`.
- The `Add` button lets you add the selected test to the `Test Include List`.

- The `Remove` button lets you delete the selected test from the `Test Include List`.
- The `Apply` button applies the changes you have selected. (The button name is `View` until you load something.)

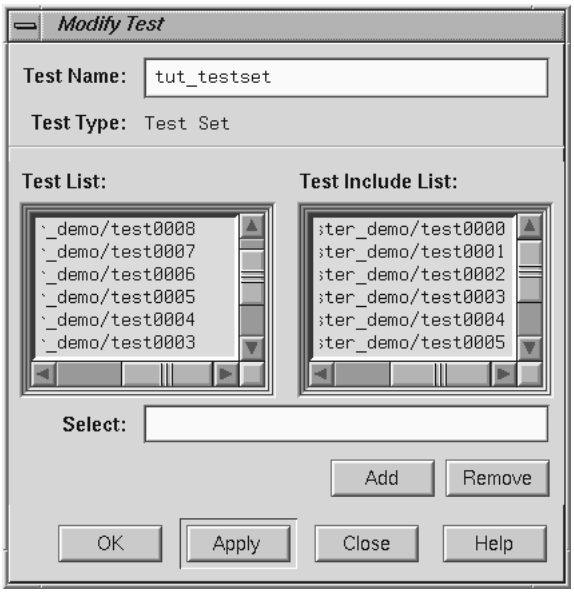

Figure 85. `Modify Test` Dialog Box after Loading Tests

9.4 `Views` Menu Operations

The `Views` menu has three selections that let you view coverage data in different forms. The selections are:

`Text View` — Displays the coverage data in text form. The information displayed depends on which query you have selected. See Figure 86, page 207.

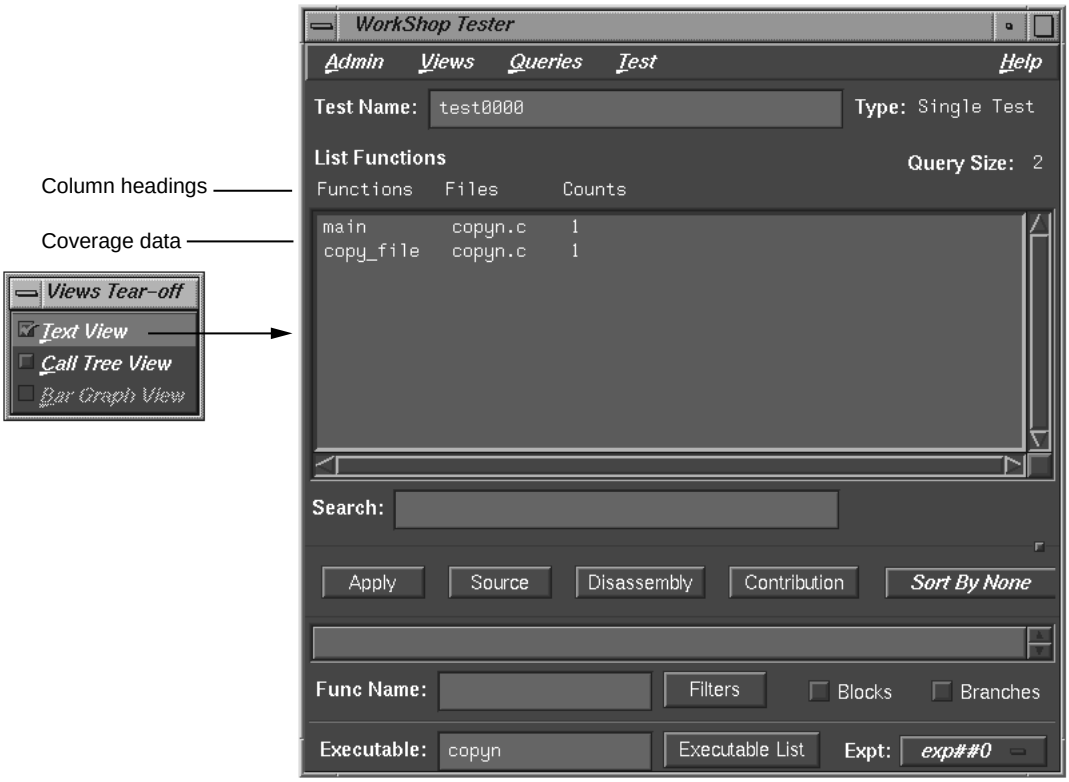

Figure 86. `List Functions` Query in `Text View` Format

`Call Tree View`

Displays coverage data graphically, with functions as nodes (rectangles) and calls as arcs (connecting arrows). This view is only valid for `List Functions`, `List Blocks`, `List Branches`, and `List Arcs`. See Figure 87, page 208. It is not available if you run a test with No Arc Data on.

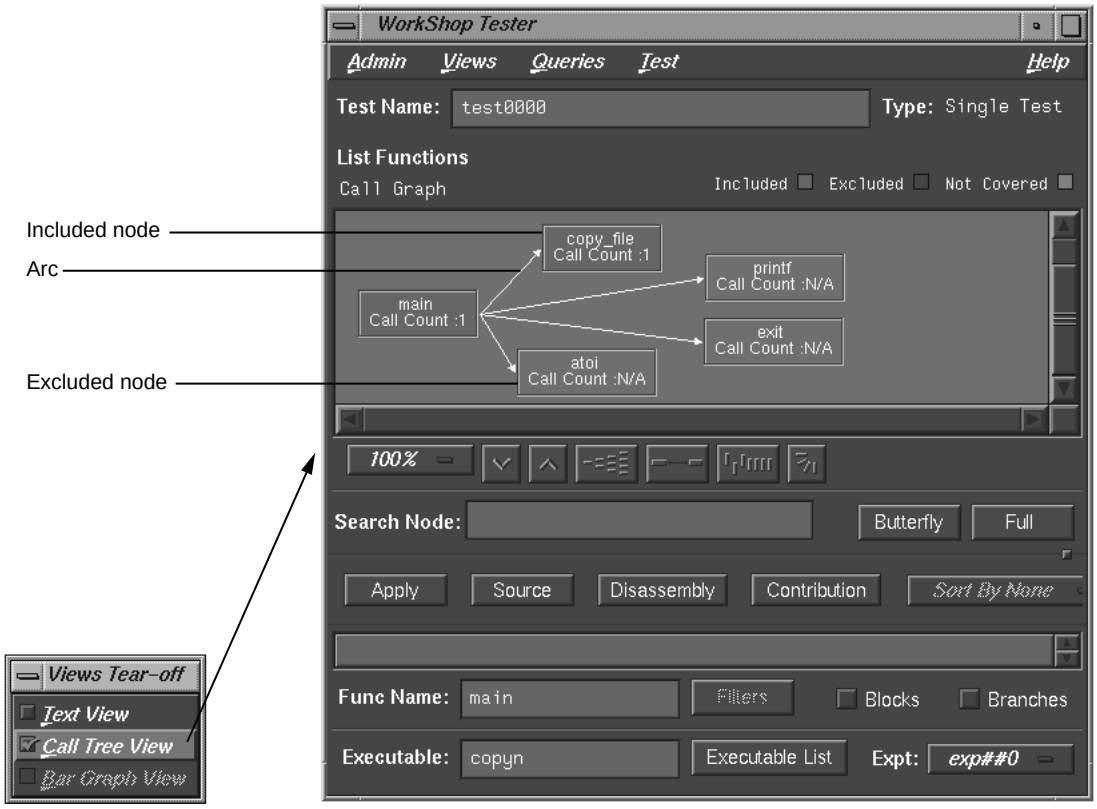

Figure 87. `List Functions` Query in `Call Tree View` Format

`Bar Graph View` Displays a bar chart showing the percentage covered for functions, lines, blocks, branches, and arcs. See Figure 88, page 209. This view is only valid for `List Summary`, which is described in detail in Section 9.5, page 209.

Figure 88. `List Summary` Query in `Bar Graph View` Format

9.5 `Queries` Menu Operations

The `Queries` menu provides different methods for analyzing the results of coverage tests. Each type of query displays the coverage data in the coverage display area in the main Tester window and displays items that are specific to the query in the area below the status area. When you set these items for a query, the same values are used by default for subsequent queries until you change them. You can set these defaults before the first query or as part of any query. For a single test or test set, all queries except `Describe Test` have the fields shown in Figure 90, page 210.

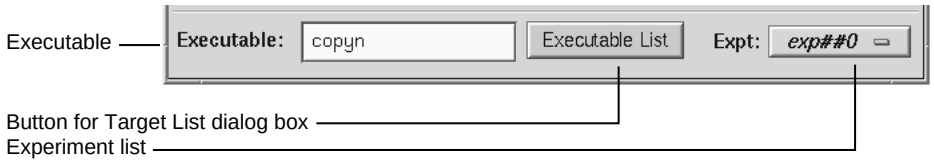

Figure 89. Query-Specific Default Fields for a Test or Test Set

The `Executable` field displays the executable associated with the current coverage data. You can switch to a different executable by entering it directly in this field. You can also switch executables by clicking the `Executable List` button, selecting from the list in the `Target List` dialog box and clicking `Apply` in the dialog box.

The experiment menu (`Expt`) lets you see the results for a different experiment that uses the same test criteria.

Note: When you are performing queries on a test group, the `Executable` field changes to `Object` field and the `Executable List` button changes to `Object List` as shown in Figure 90, page 210. These items act analogously except that they operate on dynamically shared objects (DSOs). Refer to Section 6.5, page 142, for more information on test groups.

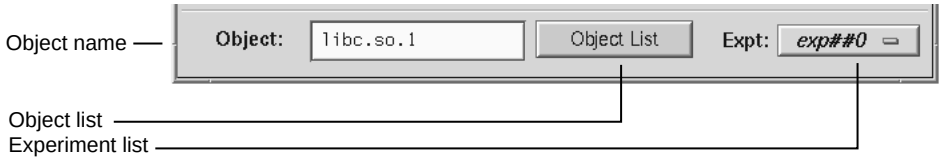

Figure 90. Query-Specific Default Fields for a DSO Test Group

The `Queries` menu (see Figure 91, page 211) provides these selections:

Figure 91. `Queries` Menu

`List Summary` Shows the overall coverage based on the user-defined weighted average over function, source line, branch, arc, and block coverage. The coverage data appears in the coverage display area. A typical summary appears in Figure 92, page 212.

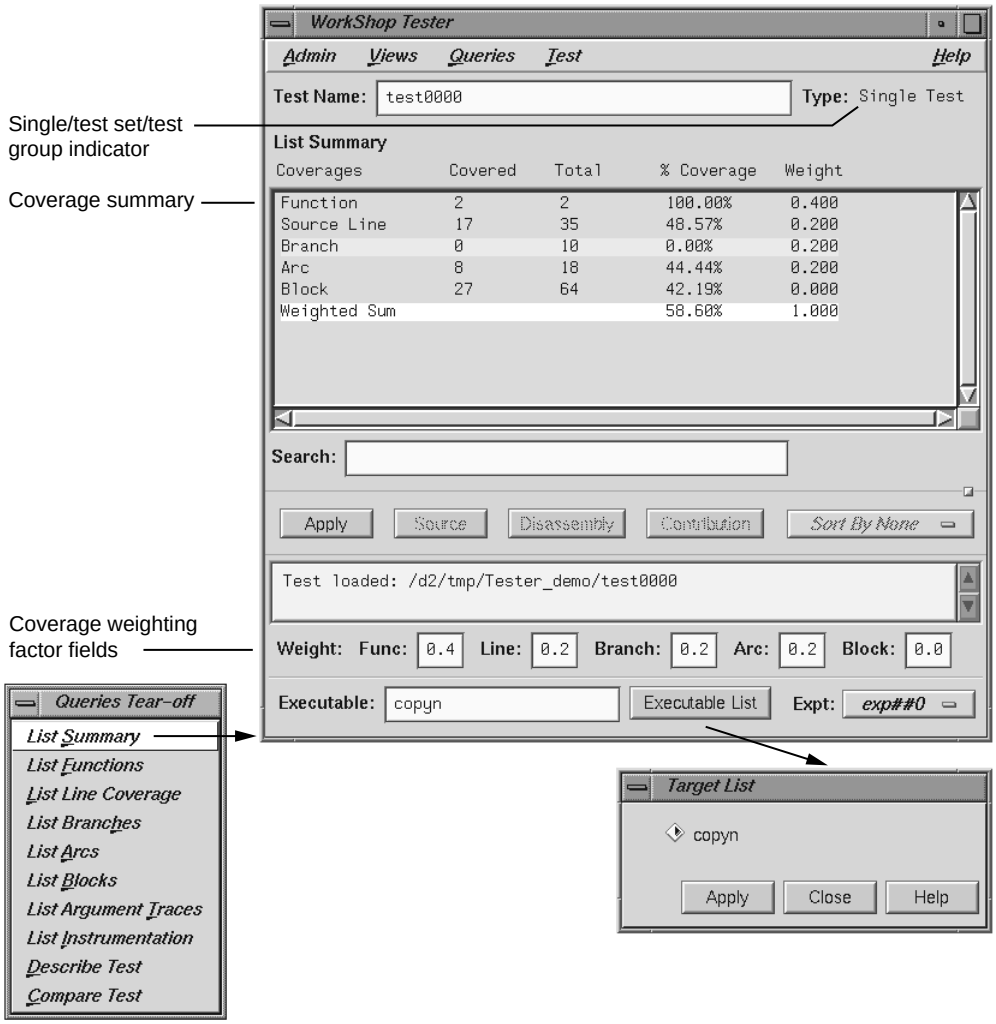

Figure 92. `List Summary` Query

The `Coverages` column indicates the type of coverage. The `Covered` column shows the number of functions, source lines, branches, arcs, and blocks that were executed in this test (or test set or test group). The `Total` column indicates the total number of items that could be executed for each type of coverage. The `% Coverage`

column is simply the Covered value divided by the Total value in each category. The Weight column indicates the weighting assigned to each type of coverage. It is used to compute the Weighted Sum, a user-defined factor that can be used to judge the effectiveness of the test. The Weighted Sum is obtained by first multiplying the individual coverage percentages by the weighting factors and then summing the products.

The List Summary command causes the coverage weighting factor fields to display below the status area. Use these to adjust the factor values as desired. They should add up to 1.0.

If you select Bar Graph View from the Views menu, the summary will be shown in bar graph format as shown in Figure 88, page 209. The percentage covered is shown along the vertical axis; the types of coverage are indicated along the horizontal axis.

List Functions

Displays the coverage data for functions in the specified test. The Functions column heading identifies the function, Files shows the source file containing the function, and Counts displays the number of times the function was executed in the test.

List Functions enables the sort menu that lets you determine the order in which the functions display. Only the sort criteria appropriate for the current query are enabled, in this case, Sort By Func, Sort By Count, and Sort By File as shown in Figure 93, page 215.

The Search field scrolls the list to the string entered. The string may occur in any of the columns. This is an incremental search and is activated as you enter characters, scrolling to the first matching occurrence.

Entering a function in the `Func Name` field displays the coverage results limited to that function only in the display area.

The `Filters` button displays the `Filters` dialog box, which lets you enter filter criteria to display a subset of the coverage results. There are three types of filters: `Function Count`, `Block Count (%)`, and `Branch Count (%)`. For blocks or *branch coverage*, use the toggles described below. Following each label is an operator menu to define the relationship to the limit quantity entered. Each filter type has a text field for entering the desired limit. The limits for `Block Count` and `Branch Count` are percentages (of coverage) and can also be entered using sliders.

Two toggles are available for including branch and block counts. Both appear as actual counts followed by parentheses containing the ratio of counts to total possible.

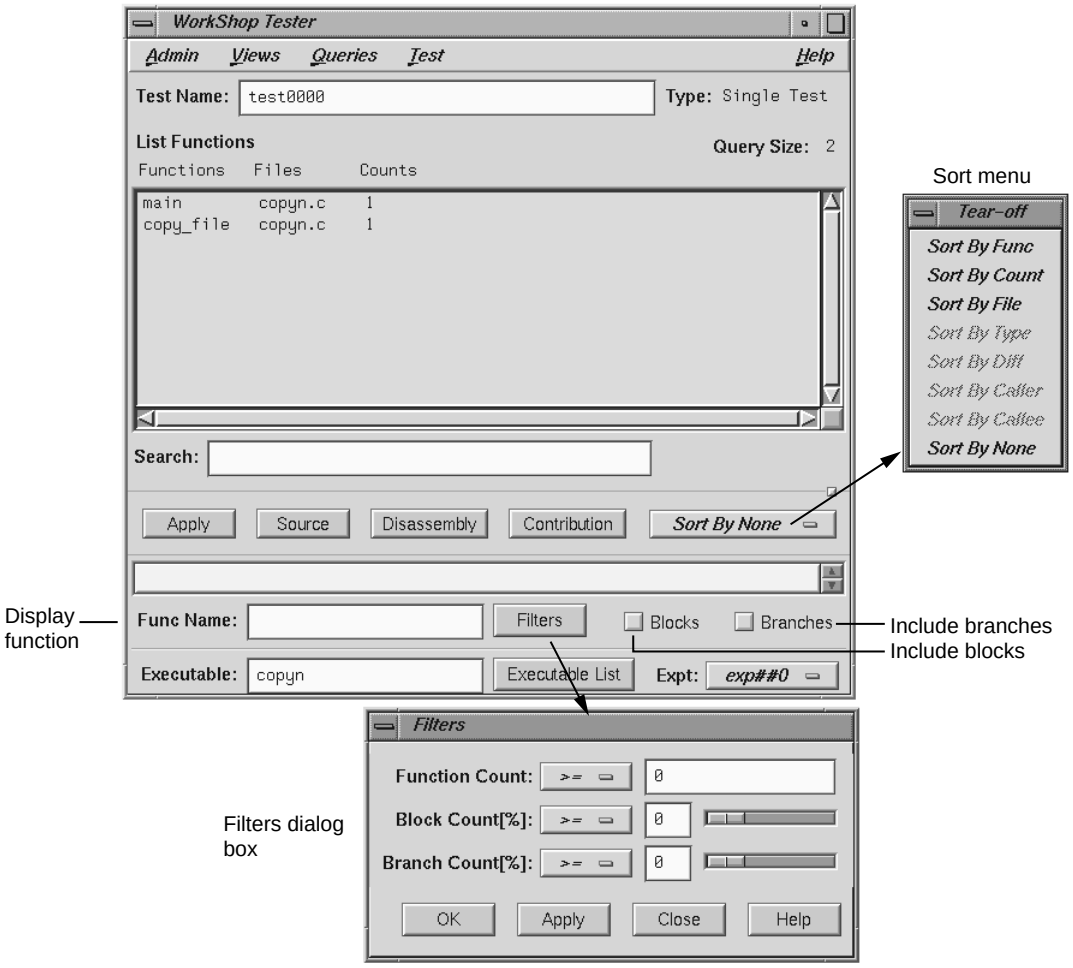

Figure 93. `List Functions` Query with Options

If you select `Call Tree View` from the `Views` menu with a `List Functions` query, a call graph displays (see Figure 94, page 216). The call graph displays coverage data graphically, with functions as nodes (rectangles) and calls as arcs (connecting arrows). The nodes are color-coded according to whether the function was included and covered in the test, included and not covered,

or excluded from the test. Arcs labeled N/A connect excluded functions and do not have call counts.

If you hold down the right mouse button over a node, the node menu displays, including the function name, coverage statistics, and standard node manipulation commands. If you have a particularly large graph, you may find it useful to zoom to 15% or 40% and look at the coverage statistics through the node menu.

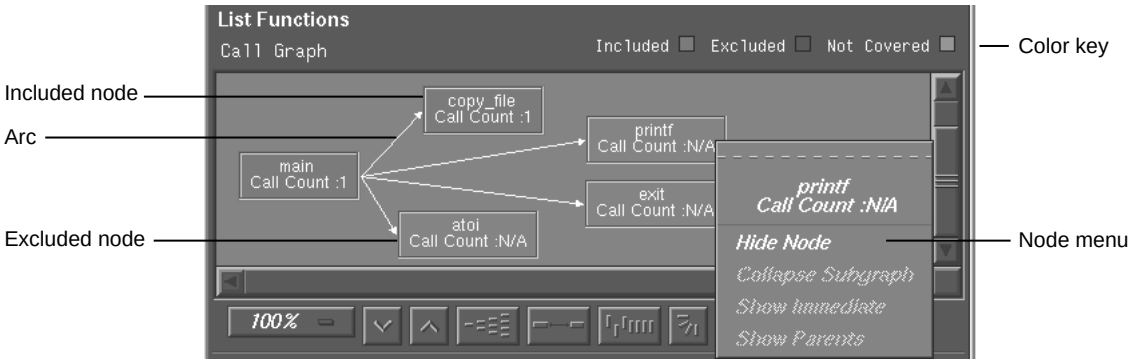

Figure 94. `List Functions` Example in `Call Tree View` Format

`List Blocks`

Displays a list of blocks for one or more functions and the count information associated with each block (see Figure 95, page 217). The `Blocks` column displays the line number in which the block occurs. If there are multiple blocks in a line, blocks subsequent to the first are shown in order with an index number in parentheses. The other three columns show the function and file containing the block and the count, that is, the number of times the block was executed in the test. Uncovered blocks (those containing 0 counts) are highlighted. Block data can be sorted by function, file, or count.

Be careful before listing all blocks in the program, since this can produce a lot of data. Entering a function in the `Func Name` field displays the coverage results limited to that function only in the display area.

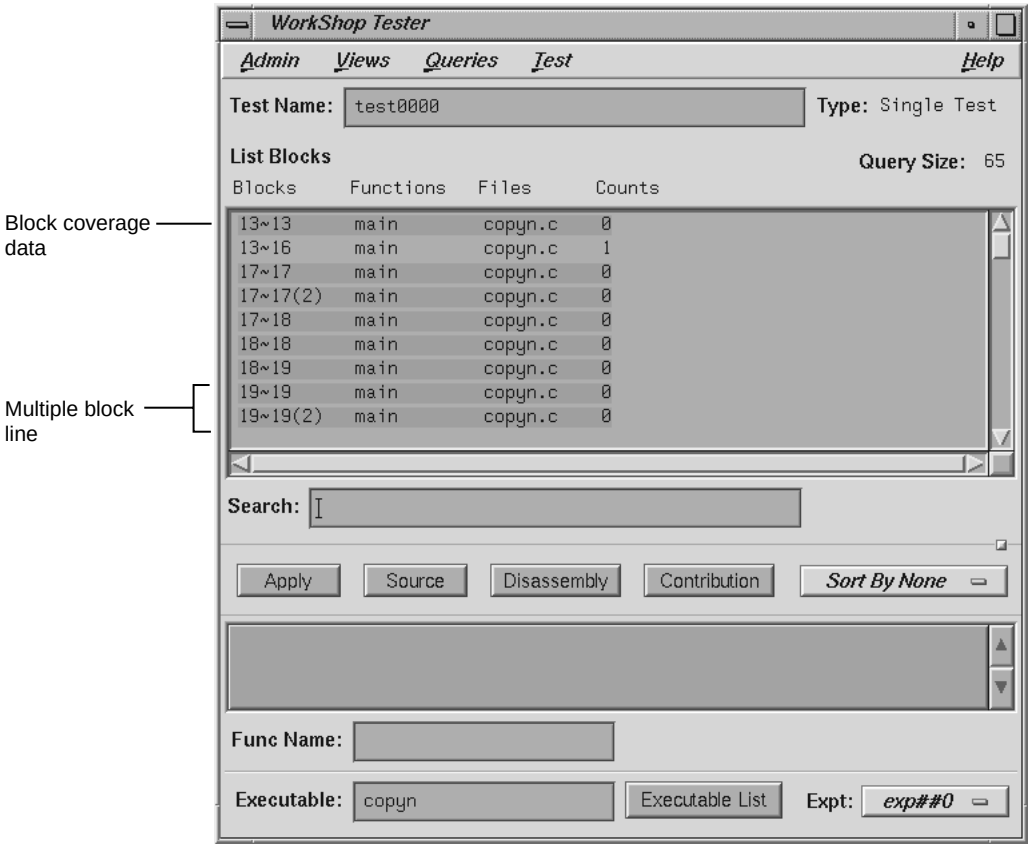

Figure 95. `List Blocks` Example

`List Branches`

Lists coverage information for branches in the program. Branch coverage counts assembly language branch instructions that are taken and not taken. See Figure 96, page 219.

The first column shows the line number in which the branch occurs. If there are multiple branches in a line, they are labeled by order of appearance within trailing parentheses. The next two columns indicate the function containing the branch and the file. A branch is considered covered if it has been executed under both true and false conditions. The `Taken` column indicates the number of branches that were executed only under the true condition. The `Not Taken` column indicates the number of branches that were executed only under the false condition. Branch coverage can be sorted only by function and file. Entering a function in the `Func Name` field displays the coverage results limited to that function only in the display area.

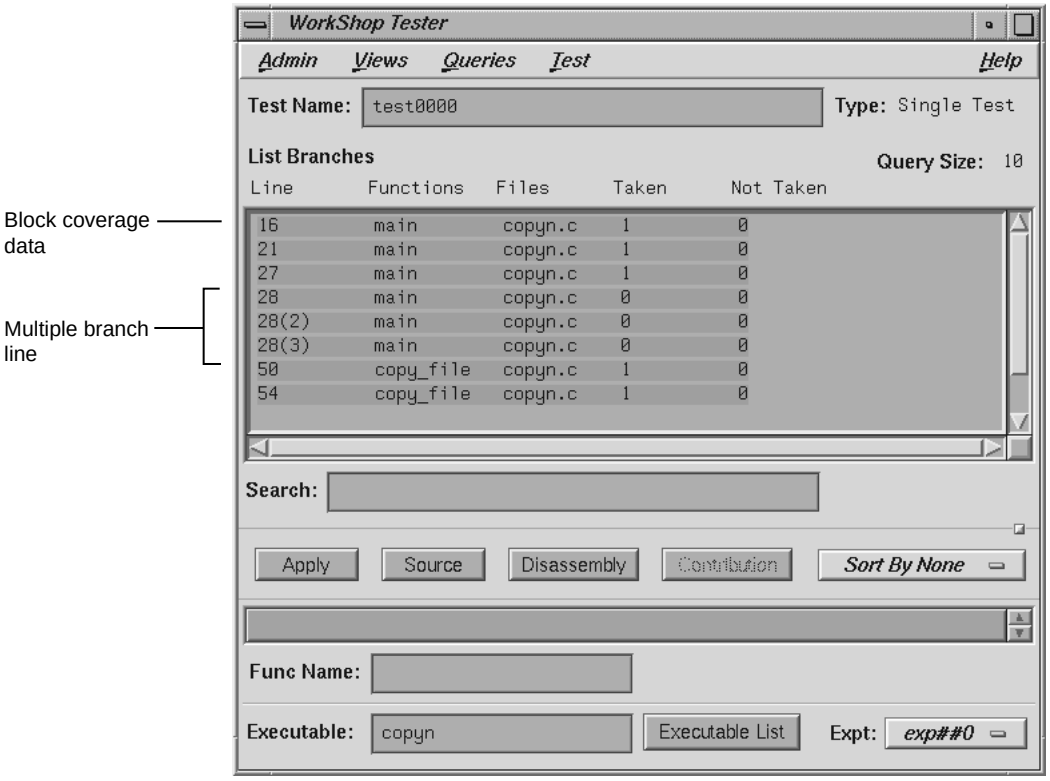

Figure 96. `List Branches` Example

`List Arcs`

Shows arc coverage, that is, the number of arcs taken out of the total possible arcs. An arc is a call from one function (caller) to another (callee). See Figure 97, page 220. The caller and callee functions are identified in the first two columns. The `Line` column identifies the line in the caller function where the call occurs. The file and arc execution count display in the last two columns.

Figure 97. `List Arcs` Example

Entering a function in the `Func Name` field displays the coverage results limited to that function only.

The `Caller` and `CalleeFunc Name` toggles let you view the arcs for a single function either as a caller or callee. You do this by entering the function name in the field and then clicking the appropriate toggle, or `CallerCallee`.

`List Argument Traces`

Shows argument tracing information (see Figure 98, page 222). Argument tracing is enabled in the instrumentation file using the TRACE command with the MAX, MIN, BOUNDS, and RETURN options. TRACE lets you monitor argument values in the functions over all experiments. The syntax in the file is:

`TRACE [RETURN] MAX|MIN|BOUNDS` *function(arg)*

where:

- MAX monitors the maximum value of an argument.
- MIN monitors the minimum value of an argument.
- BOUNDS monitors both the minimum and maximum values.
- RETURN monitors the function return values.

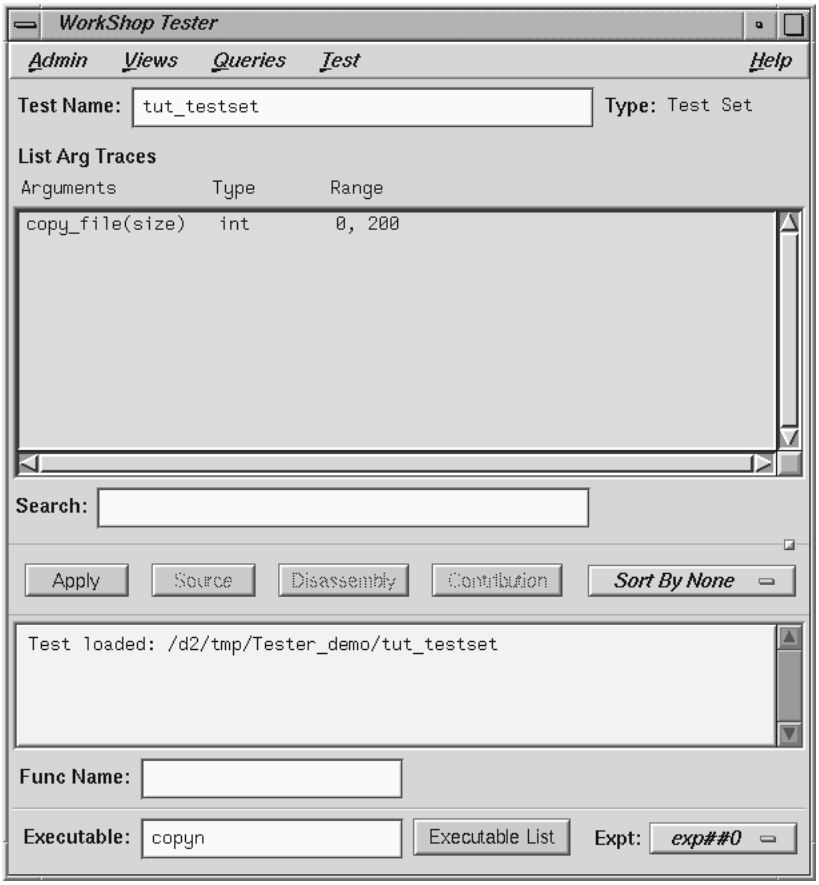

Figure 98. `List Argument Traces` Example

The `Arguments` column shows the calling function with its argument. `Type` indicates the type of the argument. `Range` shows the minimum and maximum values if TRACE bounds was selected; otherwise, it shows the end of the range selected with a short line (-) substituted for the opposite end of the range.

Entering a function in the `Func Name` field displays the coverage results limited to that function only in the display area.

`List Instrumentation`

Displays the instrumentation information for a particular test. See Figure 99, page 224.

`Function List toggle` shows the functions that are included in the coverage experiment.

`Ver` allows you to specify the version of the program that was instrumented. The latest version is used by default.

`Executable` displays the executable associated with the current coverage data. You can switch to a different executable by entering it directly in this field. You can also switch executables by clicking the `Executable List` button, selecting from the list in the dialog box, and clicking `Apply` in the dialog box.

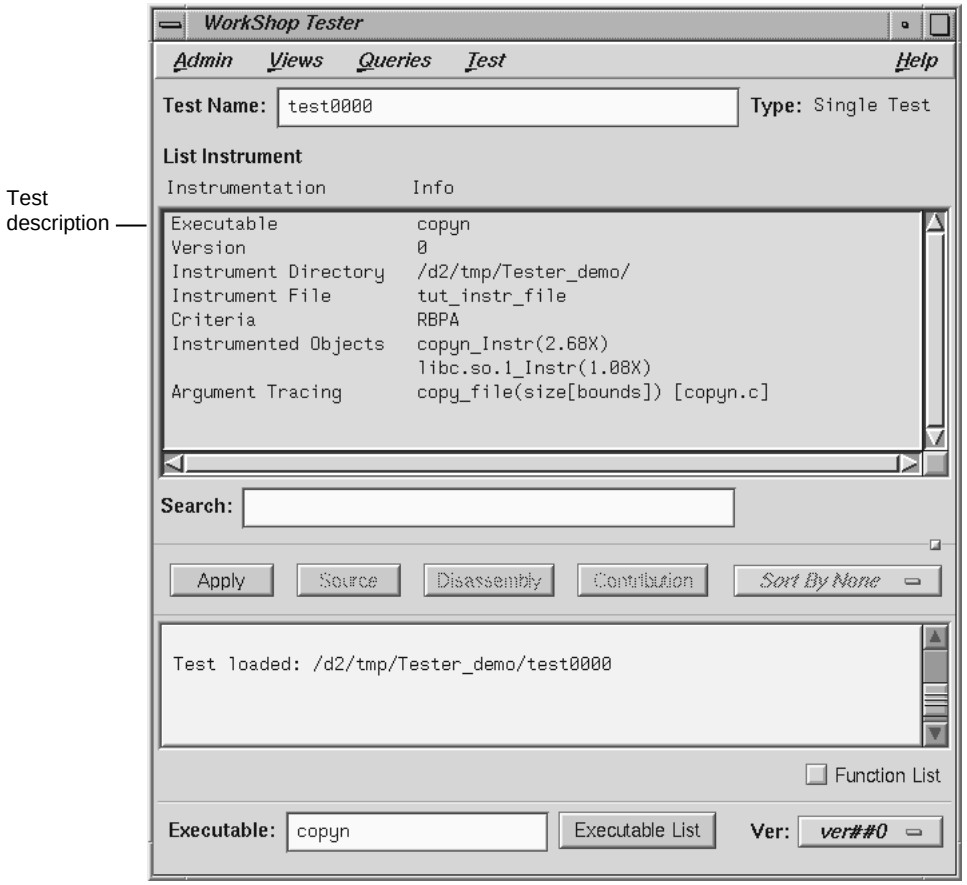

Figure 99. `List Instrumentation` Example

`List Line Coverage` Lists the coverage for each function for native source lines. Entering a function in the `Func Name` field displays the coverage results limited to that function only in the display area. See Figure 100.

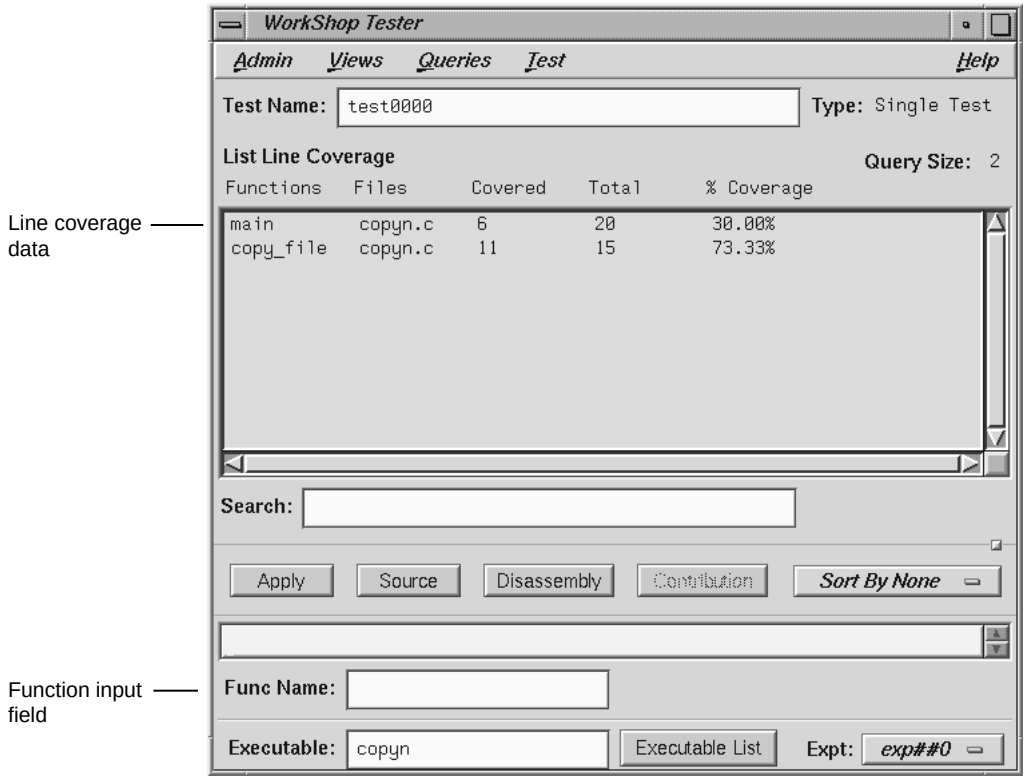

Figure 100. “List Line Coverage” Example

`Describe Test` Describes the details of the test, test set, or test group. When working with test sets and test groups, it is useful to select the `Recursive List` toggle, because it describes the details for all subtests. See Figure 101, page 226.

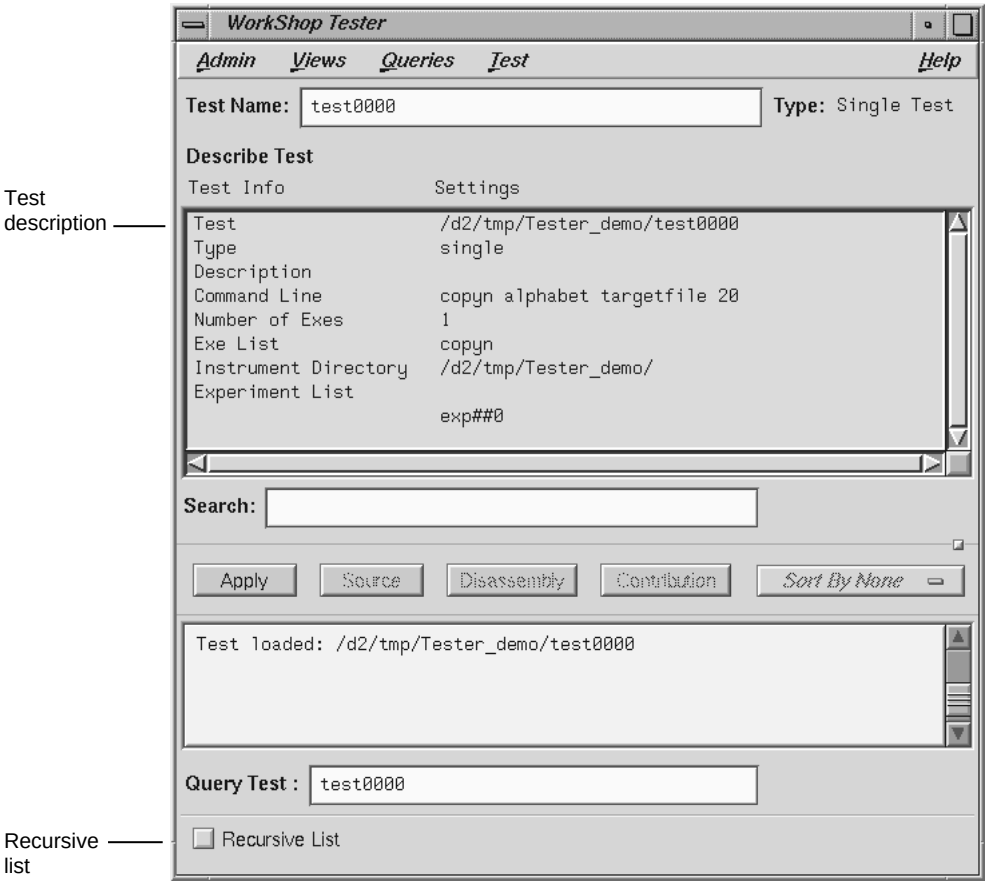

Figure 101. `Describe Test` Example

`Compare Test`

Shows the difference in coverage for the same test applied to different versions of the same program. To perform a comparison, you need to select `Compare Test` from the `Queries` menu, enter experiment directories in the experiment fields, and click `Apply` or press `Return`. The experiments are entered in the form `exp##<n>` if in the same test or in the form `test<nnnn>/exp##<n>` when comparing the results of different tests. See Figure 102, page 227.

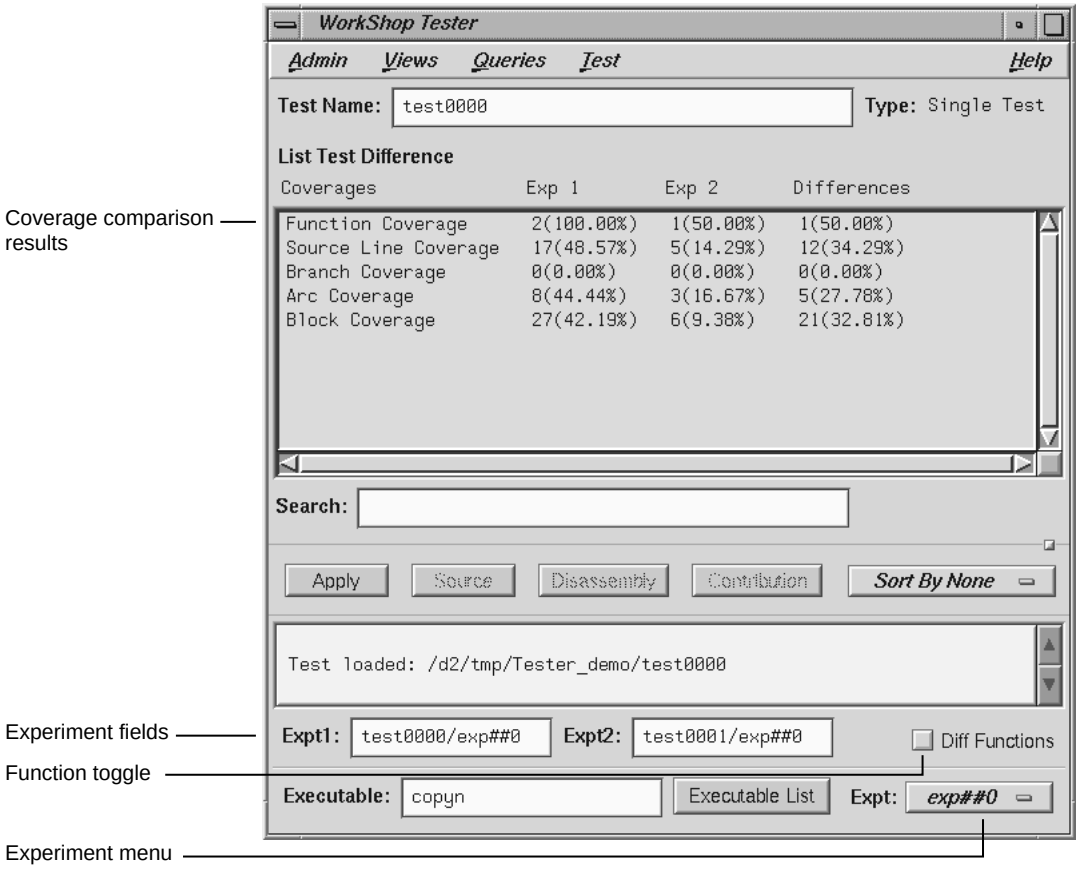

Figure 102. `Compare Test` Example — Coverage Differences

The comparison data displays in the coverage display area. The basic types of coverage display in the `Coverages` column. `Result 1` and `Result 2` display the results of the experiments specified in the `Expt1` and `Expt2` fields, respectively. Results are shown as the counts followed by the coverage percentage in parentheses. The values in the `Result 2` column are subtracted from those in `Result 1` and the differences are shown in the `Differences` column. If you want to view the available experiments, click the `Expt:  menu`.

You can also compare the differences in *function coverage* by clicking the `Diff Functions` toggle. Figure 103, page 228, shows a typical function difference example.

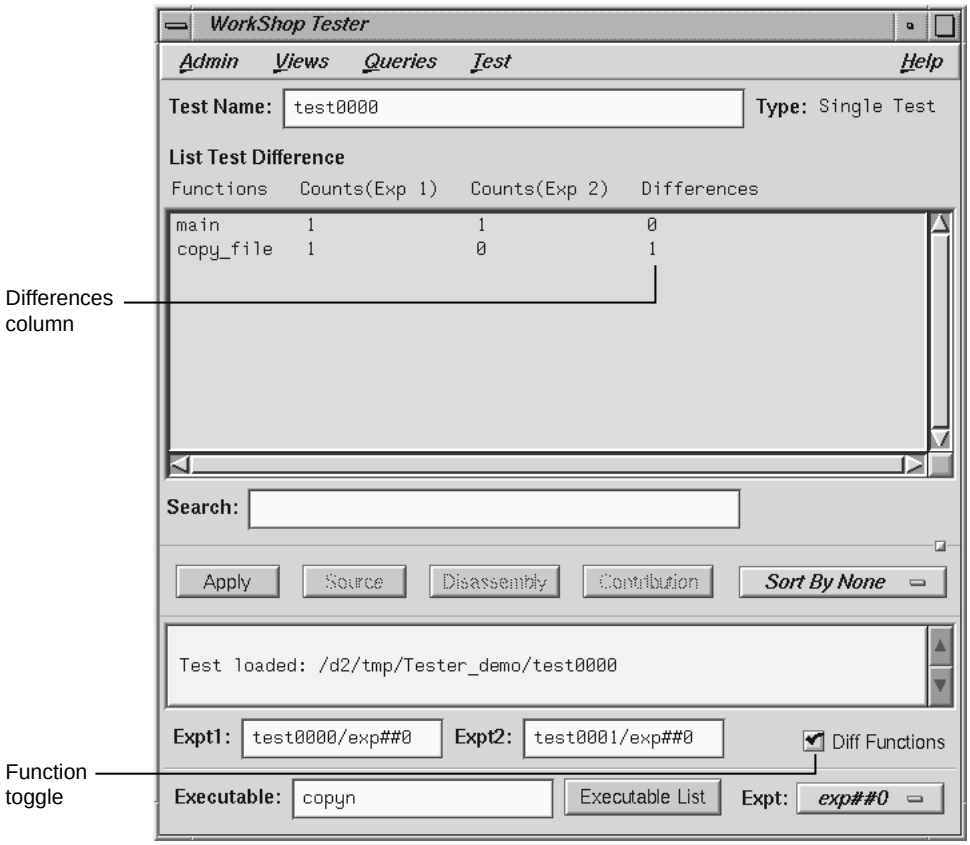

Figure 103. `Compare Test` Example — Function Differences

9.6 `Admin` Menu Operations

The `Admin` menu is shown in Figure 104, page 229.

Figure 104. Admin Menu

The `Admin` menu provides these selections:

`Save Results`	Brings up the standard `File Browser` dialog box so that you can specify a file in which to save the results.
`Clone Execution View`	Displays an `Execution View` window. Use this if you have closed the initial `Execution View` window and need a new one. (You need this window to see the results of `Run Test`.)
`Set Defaults`	Allows you to change the working directory for work on tests in other directories. Also, you can select whether or not to show function arguments. This is useful when distinguishing functions that have the same name but different arguments (for example, C++ constructors and overloaded functions). See Figure 105, page 229.

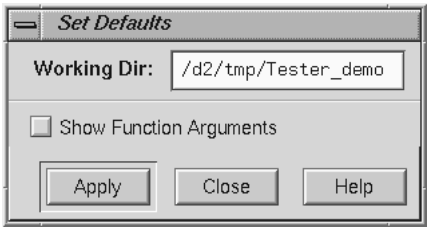

Figure 105. “Set Defaults” Dialog Box

"Launch Tool"

The Launch Tool submenu contains commands for launching other WorkShop applications (see Figure 106, page 230).

Figure 106. `Launch Tool` Submenu

If any of these tools are not installed on your system, the corresponding menu item will be grayed out.

`Exit` closes all Tester windows.

Glossary [10]

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.

blocking — Waiting in the kernel for a resource to become available.

caliper points — Markers in the time domain that can be used to delimit a performance analysis. For instance, you may want to analyze only the CPU-bound part of your code.

call stack — A software stack of functions and routines used by the running program. 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 — When system scheduler stops a job from executing and replaces it with another job.

cord mapping file — A file containing a list of functions, their sizes, and their addresses.

CPU time — Process virtual time (see the glossary entry) plus time spent when the system is running on behalf of the process, performing such tasks as executing a system call. This is the time returned in `pcsamp` and `usertime` experiments.

disassembly — Assembly language version of the program.

exclusive time — The time spent only in the function itself, not including any functions it might call.

heart beat resource data — Resource usage data (such as CPU time, wait time, I/O transfers, and so on) recorded at regular intervals (one second, by default). The `cvperf` usage view graphs are drawn using this data.

inclusive time The total time spent in a function and all the functions it calls.

instrumenting A method of collecting data by inserting code into the executable program to count events, such as the number of times a section of the program executes.

interlock A feature of the CPU that causes a stall when resources are not available.

memory leak Making `malloc` calls without the corresponding calls to `free`. The result is, the amount of heap memory used continues to increase as the process runs.

memory page The smallest unit of memory handled by the operating system. It is usually either 4K or 16K bytes.

page fault A problem resulting in the possible loss of data. A high page fault rate is an indication of a memory-bound situation.

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

phase A part of a program that concentrates on a single activity. Examples are the input phase, the computation phase, and the output phase.

pollpoint A regular time interval at which performance data is captured.

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.

profiling A method of collecting data by periodically examining and recording the program's program counter (PC), call stack, and hardware counters that measure resource consumption.

profiling time This is the same as *CPU time.*

real time The same as *wall–clock time.*

sample event A point in the program at which the PC or some resource is sampled.

system time The time during a program's execution during which the system has control. It could be performing I/O or executing a system call.

thrashing Accessing data from different parts of memory, causing frequent loads of pages of memory into cache. Using random access on an array might be an example.

threshold An upper limit. For example, in the Source View, any line of code that exceeds a threshold of resource usage is flagged in the display.

total time The same as *wall-clock time.*

user time The same as *CPU time.*

virtual address A location in memory as it appears in a program. For example, `a[10]` is the virtual address of element `10` of the array `a`. Internally, the virtual address is translated into the computer's physical address.

virtual time The same as *process virtual time.*

wall-clock time The total time a program takes to execute, including the time it takes waiting for a CPU. This is real time, not computer time.

working set Executable pages, functions, and instructions that are actually brought into memory during a phase or operation of the executable.

Index

D

E

F

G

M

N

O

P

Q

R

S

T

U

V

W

Z

Tell Us About This Manual

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

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

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

Please send the title and part number of the document with your comments. The part number for this document is 007-2581-004.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

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