Developer Magic™: Performance Analyzer User's Guide

Document Number 007-2581-002

CONTRIBUTORS

Written and Illustrated by John C. Stearns
Edited by Christina Cary
Production by Laura Cooper
Engineering contributions by Lia Adams, Jim Ambras, Trevor Bechtel, Alan Foster, Christine Hanna, David Henke, Marty Itzkowitz, Mahadevan Iyer, Lisa Kvarda, Allan McNaughton, Ashok Mouli, Sudhir Mohan, Anil Pal, Andrew Palay, Kim Rachmeler, Jack Repenning, Paul Sanville, Ravi Shankar, Shankar Unni, Mike Yang, Jun Yu, and Doug Young.

Developer Magic™: Performance Analyzer User’s Guide
Document Number 007-2581-002

Contents

Figures

Tables

Examples

About This Guide

This manual is a user's guide for the ProDev WorkShop Performance Analyzer and Tester, Release 2.5.1. It contains the following chapters:

- Chapter 1, "Introduction to the Performance Analyzer" describes the WorkShop Performance Analyzer, which helps you gain an understanding of your program's use of resources and determine if performance can be improved.
- Chapter 2, "Performance Analyzer Tutorial" provides a short tutorial that introduces you to the major features of the Performance Analyzer.
- Chapter 3, "Setting Up Performance Analysis Experiments" describes the process of setting up a performance analysis experiment, including sample trap strategy and specifying the type of performance task to capture the relevant performance data.
- Chapter 4, "Performance Analyzer Reference" provides detailed information on the Performance Analyzer and its associated facilities.
- Chapter 5, "Using Tester" provides an overview of Tester, which is used for coverage analysis, and provides a model of the different ways in which Tester can be applied.
- Chapter 6, "Tester Command Line Interface Tutorial" provides a tutorial which demonstrates how the command line version of Tester can be applied.
- Chapter 7, "Tester Command Line Reference" describes in detail each of the commands available in the Tester command line interface.
- Chapter 8, "Tester Graphical User Interface Tutorial" provides a tutorial which demonstrates how the graphical user interface version of Tester can be applied.
- Chapter 9, "Tester Graphical User Interface Reference" describes in detail each of the windows and their associated features available in the Tester graphical user interface.

Part I
The Performance Analyzer

Chapter 1

Introduction to the Performance Analyzer

This chapter describes the ProDev Performance Analyzer, which helps you gain an understanding of your program’s use of resources and determine if performance can be improved.

Chapter 1

Introduction to the Performance Analyzer

The Performance Analyzer helps you understand your program in terms of performance, determine if there are problems, and correct them. 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”
- “The Performance Analyzer Tools”
- “Sources of Performance Problems”
- “Interpreting Performance Analyzer Results”

Performance Analyzer Overview

To conduct performance analysis, you run a series of experiments to collect performance data. Prior to running an experiment, you specify the objective of your experiment through a task menu. The Performance Analyzer collects the required data and provides charts, tables, and annotated code to help you analyze the results.

The Performance Analyzer has three general techniques for collecting performance data:

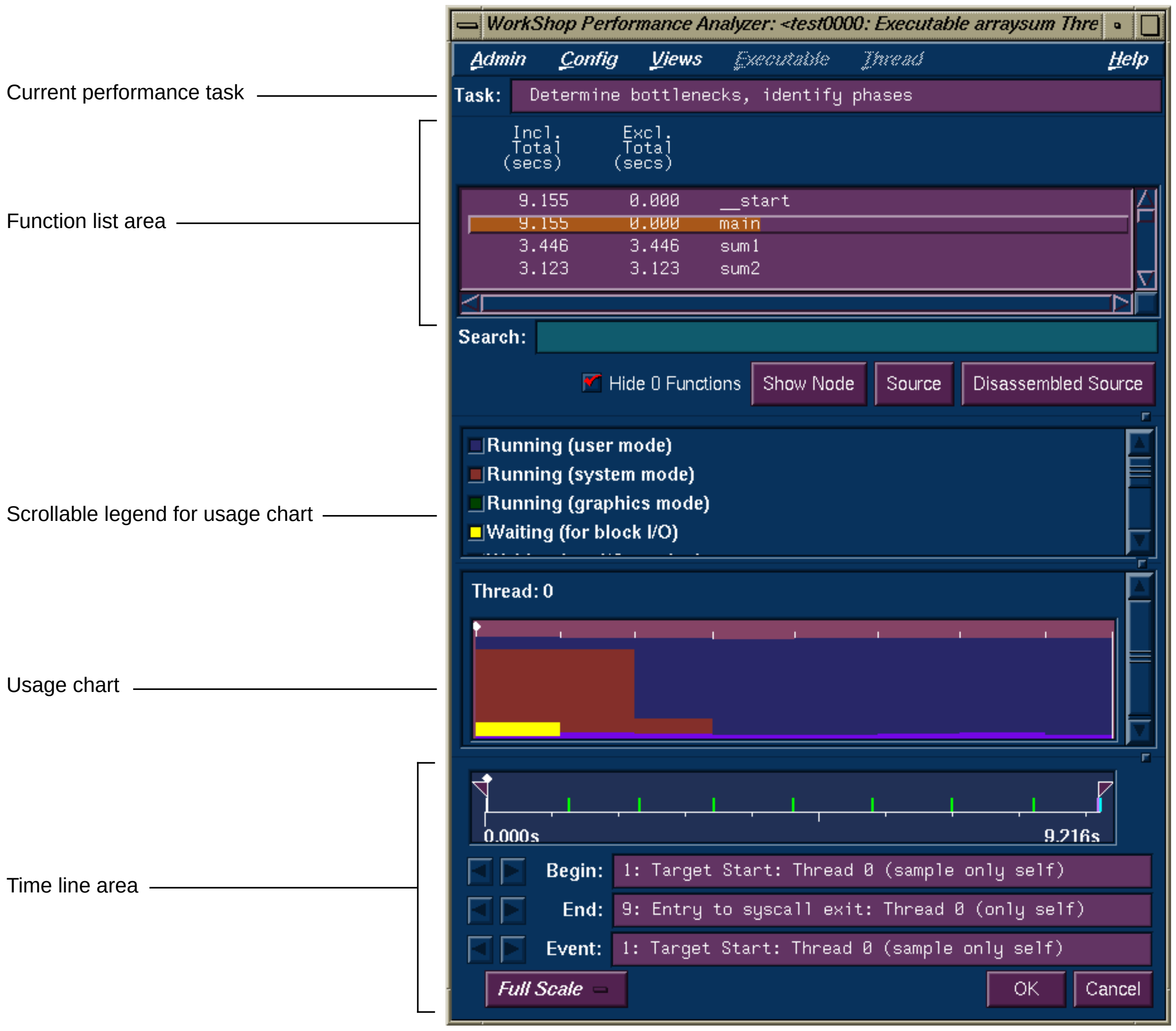
- Counting—It can count the exact number of times each function and / or basic block has been executed. This requires *instrumenting* the program, that is, inserting code into the executable to collect counts.
- Profiling—It can periodically examine and record the program's PC (program counter), call stack, and resource consumption.
- Tracing—It can trace events that impact performance, such as *reads* and *writes*, system calls, page faults, floating point exceptions, and *mallocs*, *reallocs*, and *frees*.

The Performance Analyzer processes the data to provide 28 different types of performance metrics.

The Performance Analyzer Tools

These are the major windows in the Performance Analyzer toolset:

- Performance Analyzer main window (see Figure 1-1)—contains:
  - the function list, which shows functions with their performance metrics
  - the system resource usage chart
  - the time line, which shows when sample events occurred in the experiment and controls the scope of analysis for the Performance Analyzer views



Task: Determine bottlenecks, identify phases

Incl. Total (secs)	Excl. Total (secs)	
9.155	0.000	__start
9.155	0.000	main
3.446	3.446	sum1
3.123	3.123	sum2

▨ Running (user mode)
▨ Running (system mode)
▨ Running (graphics mode)
▨ Waiting (for block I/O)

Begin: 1: Target Start: Thread 0 (sample only self)
End: 9: Entry to syscall exit: Thread 0 (only self)
Event: 1: Target Start: Thread 0 (sample only self)

Figure 1-1 Performance Analyzer Main Window

- Usage View (Graphical)—contains charts that indicate resource usage and the occurrence of sample corresponding to time intervals set by the time line calipers (see Figure 1-3)

- Usage View (Textual)—provides the actual resource usage values corresponding to time intervals set by the time line calipers (see Figure 1-4)
- Call Graph View—displays the functions (with their metrics) in a graphical format showing where the calls were made (see Figure 1-6)
- Call Stack—displays the contents of the call stack at the selected event (see Figure 1-10)
- Malloc Error View—displays each *malloc* error (leaks and bad *frees*) that occurred in the experiment, the number of times the *malloc* occurred (a count is kept of *mallocs* with identical call stacks), and the call stack corresponding to the selected *malloc* error.
- Leak View—displays 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—displays 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 *mallocs* with identical call stacks), and the call stack corresponding to the selected *malloc*.
- Heap View—displays a map of memory indicating how blocks of memory were used in the time interval set by the time line calipers (see Figure 1-9)
- I/O View—displays a chart devoted to I/O system calls. Identifies up to 10 files involved in I/O (see Figure 1-11)
- Working Set View—measures the coverage of 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 1-12).
- Cord Analyzer (accessed from *cvcord*)—works in conjunction with Working Set View to let you try out different working set configurations to improve performance (see Figure 1-13).

- Source View with performance annotations—displays performance metrics adjacent to the corresponding line of source code (see Figure 1-7)
- Disassembly View with performance annotations—displays the performance metrics adjacent to the corresponding machine code. For the "Get Ideal Time (pixie) per function & source line" experiment, Source View can show where and why a clock may have stalled during an instruction.

Sources of Performance Problems

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

- CPU speed and availability
- I/O processing
- memory size and availability
- bugs
- instruction and data cache size
- any of the above in different phases

CPU-bound Processes

A *CPU-bound* process spends its time in the CPU and is limited by CPU speed and availability. To improve its performance on CPU-bound processes, you may need to streamline your code. This can entail modifying algorithms, reordering code to avoid interlocks, removing nonessential steps, blocking to keep data in cache and registers, or using alternative algorithms.

I/O-bound Processes

An *I/O-bound* process has to wait for I/O to complete and may be limited by disk access speeds or memory caching. To improve the performance of I/O-bound processes, you can try one of the following techniques:

- improve overlap of I/O with computation
- optimize data usage to minimize disk access
- use data compression

Memory-bound Processes

A 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. One telltale indication of a page-thrashing condition is noise due to disk accesses. To fix a memory-bound process, you can try to improve the memory reference patterns or, if possible, decrease the memory used by the program.

Bugs

You may find that a bug is causing the performance problem. For example, you may find that you are reading in the same file twice in different parts of the program, that floating point exceptions are slowing down your program, that old code has not been completely removed, or that you are leaking memory (making *malloc* calls without the corresponding calls to *free*).

Performance Phases in Programs

Since programs exhibit different behavior during different phases of operation, you need to identify the limiting resource during each phase. A program can be I/O-bound while it reads in data, CPU-bound while it performs computation, and I/O-bound again in its final stage while it writes out data. Once you've identified the limiting resource in a phase, you can perform an in-depth analysis to find the problem. And after you have solved that problem, you can check for other problems within the phase—performance analysis is an iterative process.

Interpreting Performance Analyzer Results

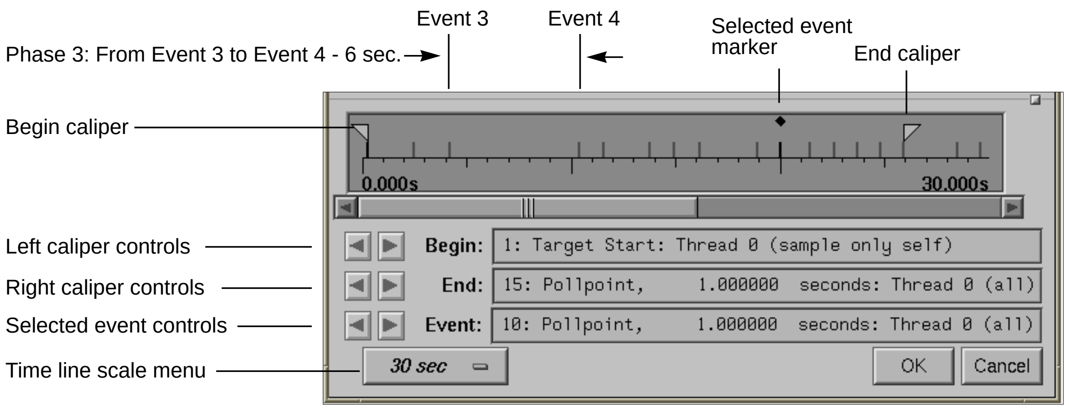
Before we discuss the mechanics of using the Performance Analyzer, let’s look at these features that help you understand the behavior of your processes:

- “The Time Line Display”
- “Resource Usage Graphs”
- “Textual Usage View”
- “The Function List Area”
- “Call Graph View”
- “Source View with Performance Annotations”
- “Malloc Error View, Leak View, Malloc View, and Heap View”
- “Call Stack View”
- “I/O View”
- “Working Set View”

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 1-2 shows the time line portion of the Performance Analyzer window with typical results. Events #3 and #4 are labeled. By looking at the distance between them and counting tick marks on the scale, you can see that this phase lasted for approximately 6 seconds.



```
Begin:  1: Target Start: Thread 0 (sample only self)
End:    15: Pollpoint,     1.000000  seconds: Thread 0 (all)
Event:  10: Pollpoint,     1.000000  seconds: Thread 0 (all)
```

Figure 1-2 Typical Performance Analyzer Time Line

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 1-3, 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
- process size

Resource usage data is always recorded (written to file) at each sample point. In addition, setting a time in the *Fine Grained Usage* field enables you to record resource usage data at regular intervals. Fine-grained usage allows you to see fluctuations at a finer gradation than the phases defined by sample points. 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 Graphical Usage View and can view the numerical values in the Textual Usage View.

Fine grained usage has little effect on the execution of the target process during data collection. It is of limited use if the program is divided into phases of uniform behavior by the placement of the sample points.

Textual Usage View

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 1-4 shows a typical Textual Usage View window.

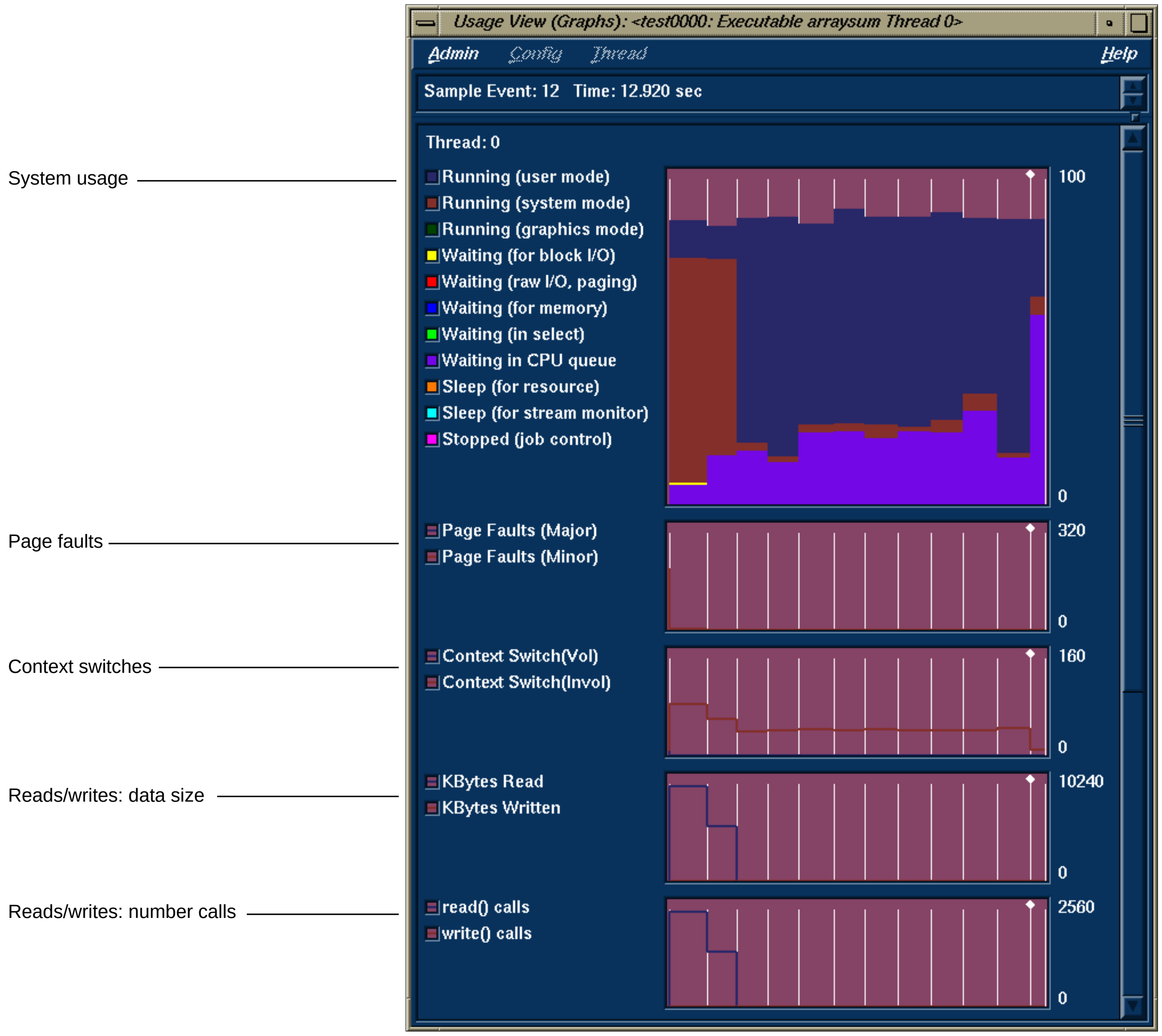
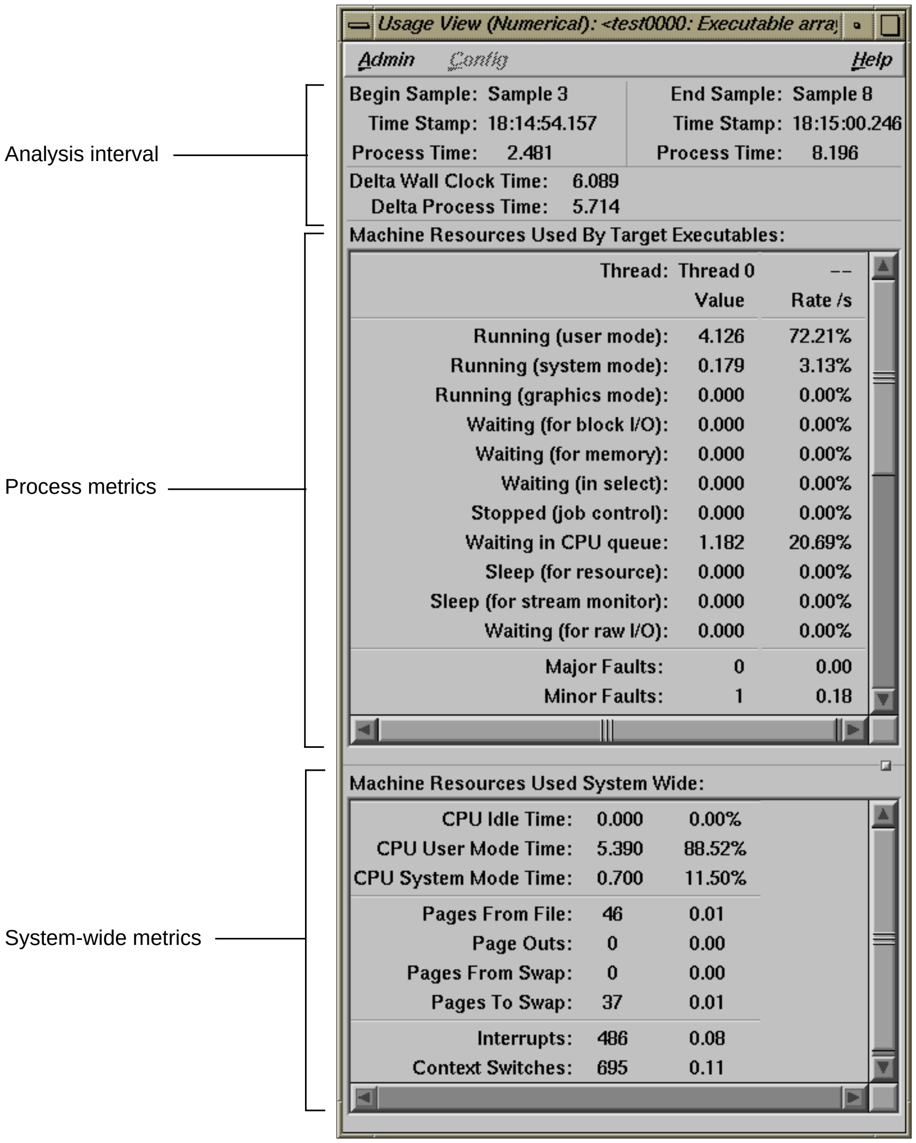


Figure 1-3 Typical Resource Usage Graphs

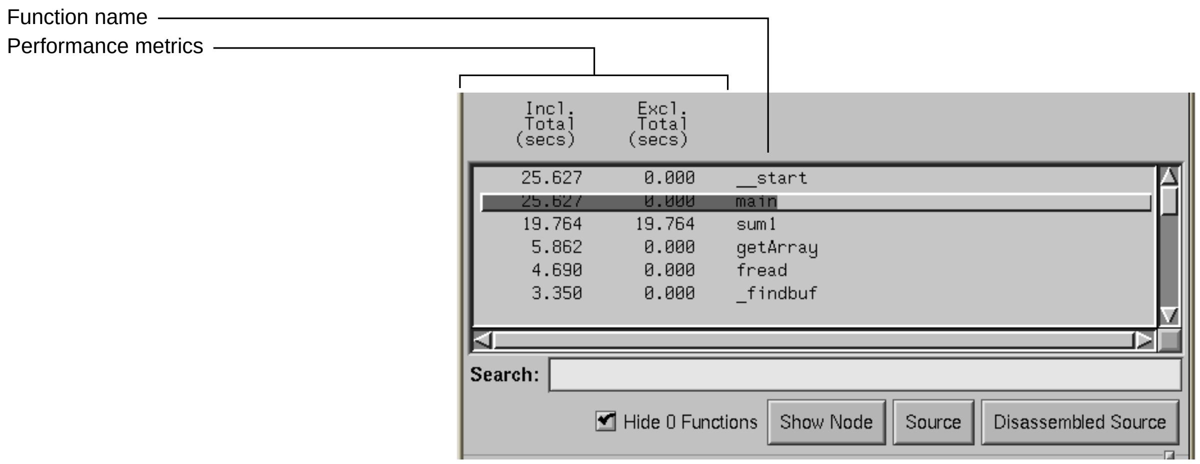


```
Begin Sample:  Sample 3          End Sample:  Sample 8
  Time Stamp:  18:14:54.157      Time Stamp:  18:15:00.246
Process Time:     2.481        Process Time:     8.196
Delta Wall Clock Time:    6.089
  Delta Process Time:     5.714
Machine Resources Used By Target Executables:
                          Thread:  Thread 0       --
                                   Value      Rate /s
           Running (user mode):    4.126      72.21%
         Running (system mode):    0.179       3.13%
       Running (graphics mode):    0.000       0.00%
         Waiting (for block I/O):  0.000       0.00%
           Waiting (for memory):   0.000       0.00%
              Waiting (in select): 0.000       0.00%
            Stopped (job control): 0.000       0.00%
            Waiting in CPU queue:  1.182      20.69%
             Sleep (for resource): 0.000       0.00%
       Sleep (for stream monitor): 0.000       0.00%
            Waiting (for raw I/O): 0.000       0.00%
                   Major Faults:       0       0.00
                   Minor Faults:       1       0.18
Machine Resources Used System Wide:
        CPU Idle Time:   0.000    0.00%
   CPU User Mode Time:   5.390   88.52%
 CPU System Mode Time:   0.700   11.50%
      Pages From File:   46       0.01
           Page Outs:    0        0.00
      Pages From Swap:   0        0.00
        Pages To Swap:   37       0.01
           Interrupts:   486      0.08
     Context Switches:   695      0.11
```

Figure 1-4 Typical Textual Usage View

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 1-5 is an example of the function list, ranked by exclusive CPU time (defined as the time this function spent in the CPU, excluding jumps to other blocks).



```
  Incl.     Excl.
  Total     Total
 (secs)    (secs)
 25.627     0.000    __start
 25.627     0.000    main
 19.764    19.764    sum1
  5.862     0.000    getArray
  4.690     0.000    fread
  3.350     0.000    _findbuf
```

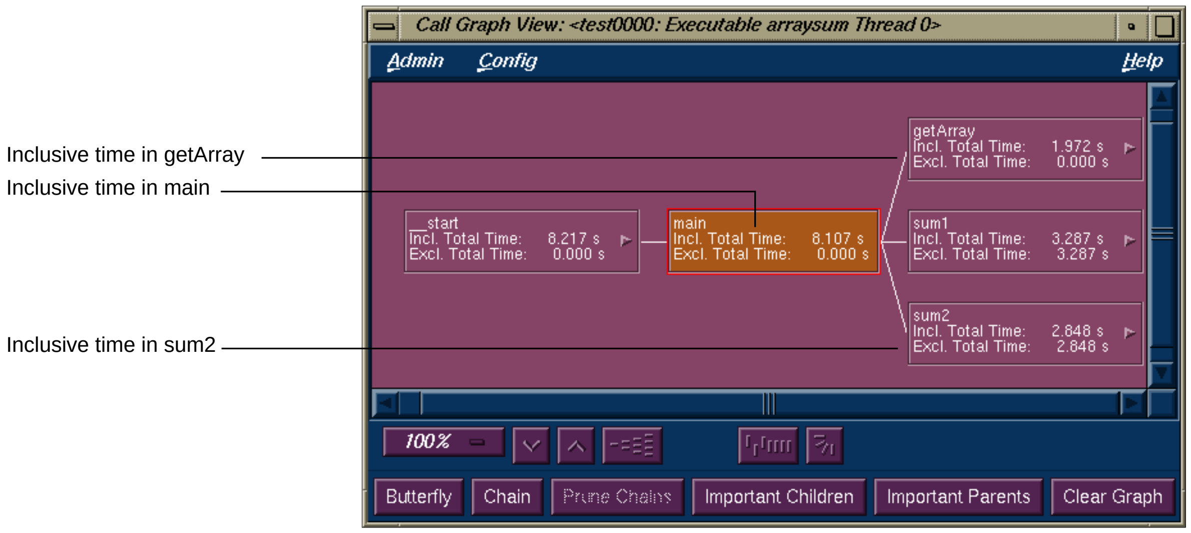
Figure 1-5 Typical Performance Analyzer Function List Area

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.

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 where the calls are made. The call graph displays functions as nodes and calls as arcs. The nodes are annotated with the performance metrics; the arcs come with counts by default and can include other metrics as well.

In Figure 1-6, 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. **main** can potentially call three functions. 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.



__start
Incl. Total Time: 8.217 s
Excl. Total Time: 0.000 s

main
Incl. Total Time: 8.107 s
Excl. Total Time: 0.000 s

getArray
Incl. Total Time: 1.972 s
Excl. Total Time: 0.000 s

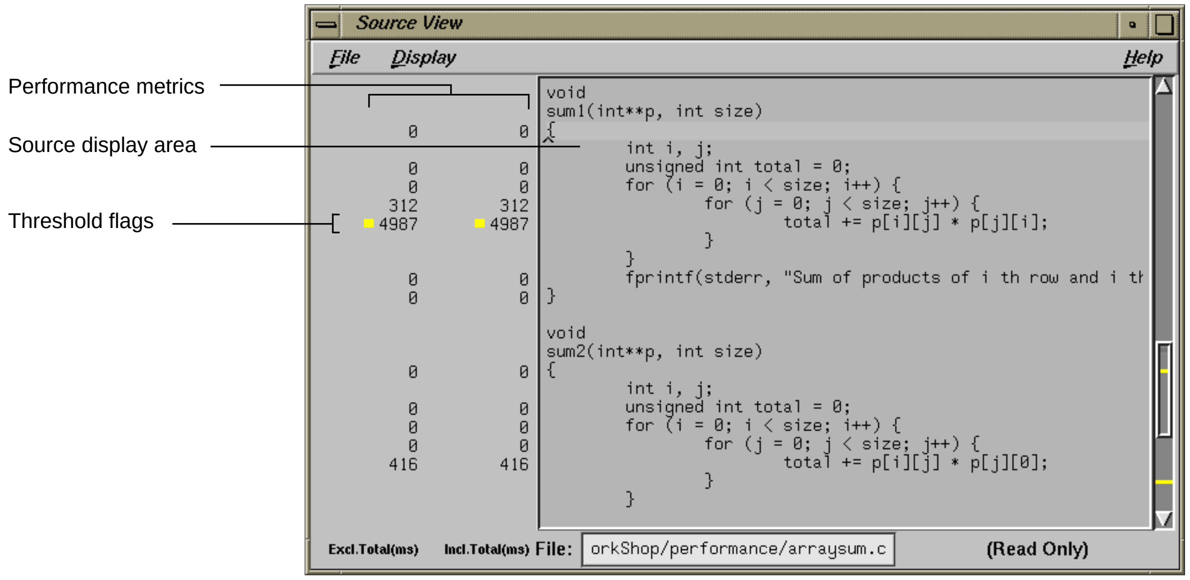
sum1
Incl. Total Time: 3.287 s
Excl. Total Time: 3.287 s

sum2
Incl. Total Time: 2.848 s
Excl. Total Time: 2.848 s

Figure 1-6 Typical Performance Analyzer Call Graph

Source View with Performance Annotations

The Performance Analyzer lets you view performance metrics by source line in Source View (see Figure 1-7) or by machine instruction in Disassembly View. Displaying performance metrics is set in the Preferences dialog box, accessed from the Display menu in Source View and Disassembly 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.



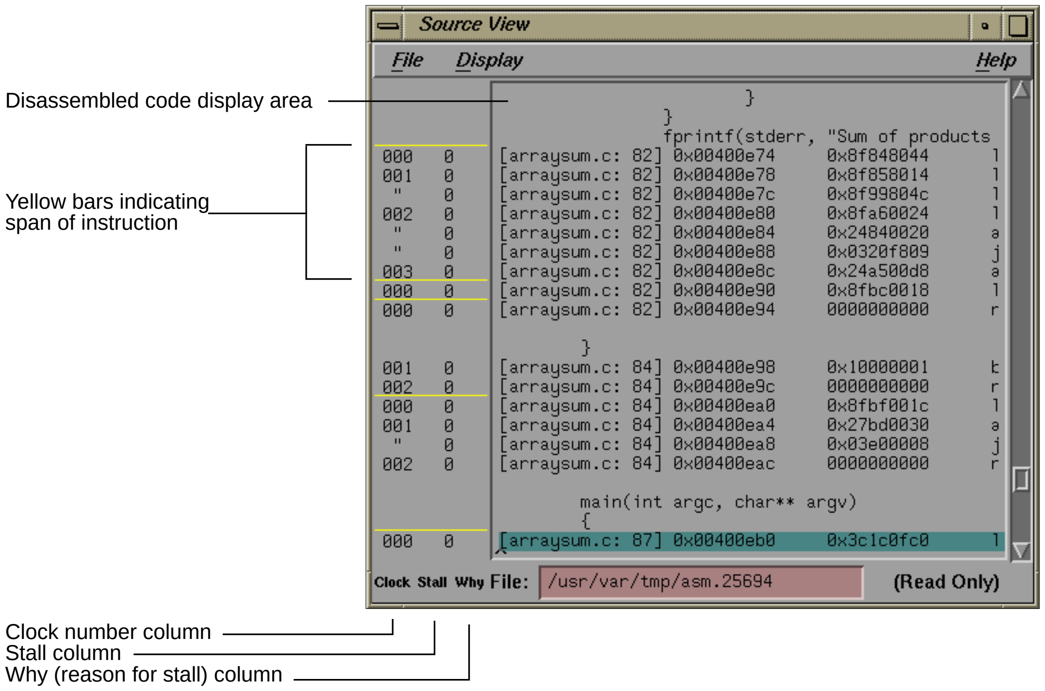
```
                 void
                 sum1(int**p, int size)
   0        0    {
                         int i, j;
   0        0            unsigned int total = 0;
   0        0            for (i = 0; i < size; i++) {
 312      312                    for (j = 0; j < size; j++) {
4987     4987                            total += p[i][j] * p[j][i];
                                 }
                         }
   0        0            fprintf(stderr, "Sum of products of i th row and i th
   0        0    }

                 void
                 sum2(int**p, int size)
   0        0    {
                         int i, j;
   0        0            unsigned int total = 0;
   0        0            for (i = 0; i < size; i++) {
   0        0                    for (j = 0; j < size; j++) {
 416      416                            total += p[i][j] * p[j][0];
                                 }
                         }
```

Figure 1-7 Detailed Performance Metrics by Source Line

Disassembled Code with Performance Annotations

The Performance Analyzer also lets you view performance metrics by machine instruction. You can view any of the performance metrics that were measured in your experiment. If you ran a “Get Ideal Time (pixie) per function & source line” experiment, you can get a special three-part annotation that providing information about on stalled instructions (see Figure 1-8). The yellow bar spanning the top of three columns in this annotation indicates the first instruction in each basic block. The first column labelled *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 ditto marks (“), it means that multiple instructions were issued in the same cycle. The second column is labelled *Stall* and shows how many clocks elapsed during the stall before the instruction was issued. The third column labelled *Why* shows the reason for the stall. There are three possibilities:

- B - branch delay
- F - function unit delay
- O - operand hasn’t arrived yet


Source View
File
Display
Help
Disassembled code display area
Yellow bars indicating span of instruction
Clock number column
Stall column
Why (reason for stall) column
Clock
Stall
Why
File:
(Read Only)

```
                                   }
                          }
                          fprintf(stderr, "Sum of products
000   0   [arraysum.c: 82] 0x00400e74     0x8f848044      l
001   0   [arraysum.c: 82] 0x00400e78     0x8f858014      l
 "    0   [arraysum.c: 82] 0x00400e7c     0x8f99804c      l
002   0   [arraysum.c: 82] 0x00400e80     0x8fa60024      l
 "    0   [arraysum.c: 82] 0x00400e84     0x24840020      a
 "    0   [arraysum.c: 82] 0x00400e88     0x0320f809      j
003   0   [arraysum.c: 82] 0x00400e8c     0x24a500d8      a
000   0   [arraysum.c: 82] 0x00400e90     0x8fbc0018      l
000   0   [arraysum.c: 82] 0x00400e94     0000000000      r

                    }
001   0   [arraysum.c: 84] 0x00400e98     0x10000001      b
002   0   [arraysum.c: 84] 0x00400e9c     0000000000      r
000   0   [arraysum.c: 84] 0x00400ea0     0x8fbf001c      l
001   0   [arraysum.c: 84] 0x00400ea4     0x27bd0030      a
 "    0   [arraysum.c: 84] 0x00400ea8     0x03e00008      j
002   0   [arraysum.c: 84] 0x00400eac     0000000000      r

                    main(int argc, char** argv)
                    {
000   0   [arraysum.c: 87] 0x00400eb0     0x3c1c0fc0      l
```

File: /usr/var/tmp/asm.25694

Figure 1-8 Disassembled Code with Stalled Clock Annotations

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"
- "Bad Frees"

The difference between these windows lies in the set of data that they collect. Malloc Error View displays all *malloc* errors: both memory leaks and bad *frees*. 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 indicating 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.

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 Process Size stripchart in Process View. 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.

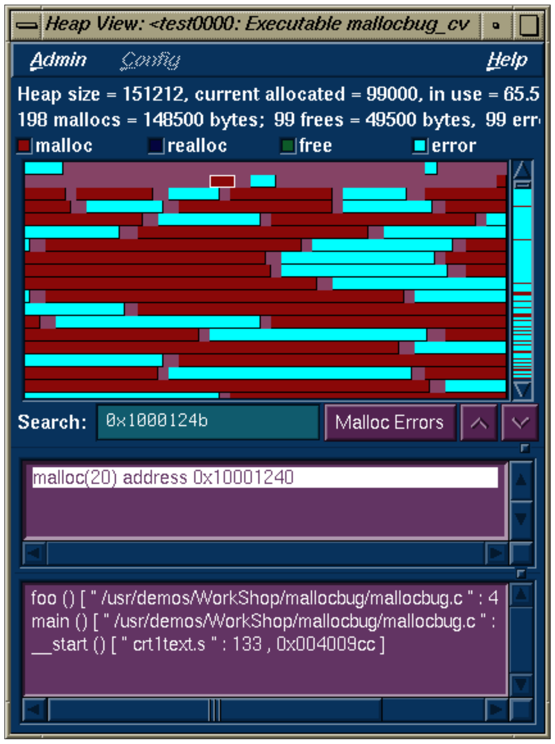
Bad Frees

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 *frees* are indicated in both Malloc Error View and in Heap View. Heap View identifies bad *frees* in its memory map display. It helps you find the address of the *freed* structure, search for the *malloc* event that created it, and 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 *frees* 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 *frees* between selected sample events in your experiment. Heap View displays a memory map that indicates *mallocs*, *reallocs*, bad *frees*, and valid *frees* during the selected period, as shown in Figure 1-9. Clicking an area in the memory map displays the address.



Heap size = 151212, current allocated = 99000, in use = 65.5
198 mallocs = 148500 bytes; 99 frees = 49500 bytes, 99 err

```
0x1000124b
```

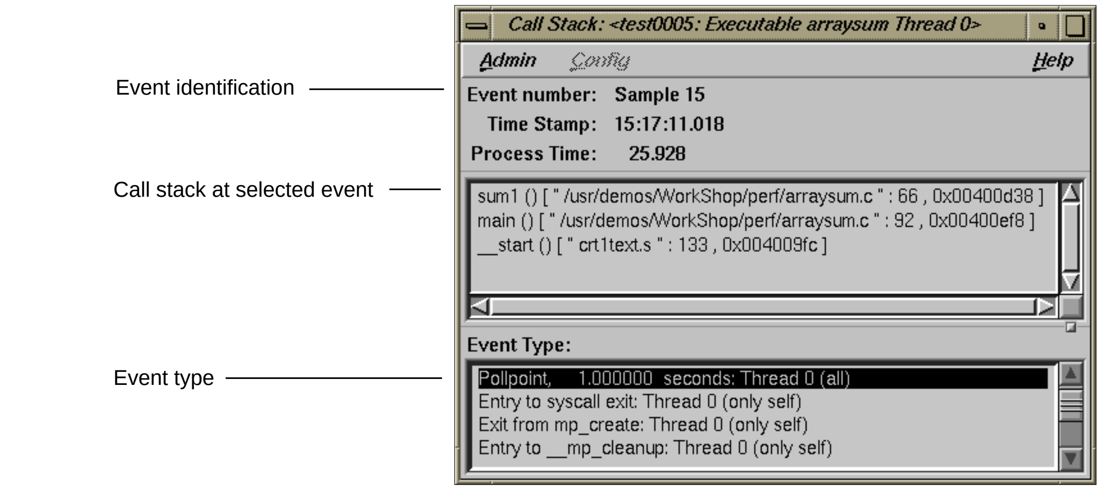
```
malloc(20) address 0x10001240
```

```
foo () [ " /usr/demos/WorkShop/mallocbug/mallocbug.c " : 4
main () [ " /usr/demos/WorkShop/mallocbug/mallocbug.c " :
__start () [ " crt1text.s " : 133 , 0x004009cc ]
```

Figure 1-9 Typical Heap View Display Area

Call Stack View

The Performance Analyzer enables 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 1-10 shows a typical call stack. It corresponds to sample event #2 in an experiment.


Event identification
Call stack at selected event
Event type
Call Stack: <test0005: Executable arraysum Thread 0>
Admin
Config
Help

Event number: Sample 15
Time Stamp: 15:17:11.018
Process Time: 25.928

```
sum1 () [ " /usr/demos/WorkShop/perf/arraysum.c " : 66 , 0x00400d38 ]
main () [ " /usr/demos/WorkShop/perf/arraysum.c " : 92 , 0x00400ef8 ]
__start () [ " crt1text.s " : 133 , 0x004009fc ]
```

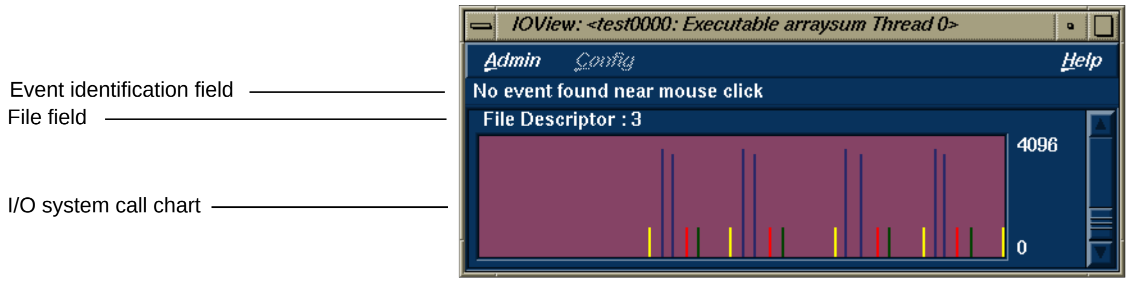
Event Type:

```
Pollpoint,     1.000000  seconds: Thread 0 (all)
Entry to syscall exit: Thread 0 (only self)
Exit from mp_create: Thread 0 (only self)
Entry to __mp_cleanup: Thread 0 (only self)
```

Figure 1-10 Typical Call Stack

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. See Figure 1-11.


Event identification field
File field
I/O system call chart
IOView: <test0000: Executable arraysum Thread 0>
Admin
Config
Help
4096
0

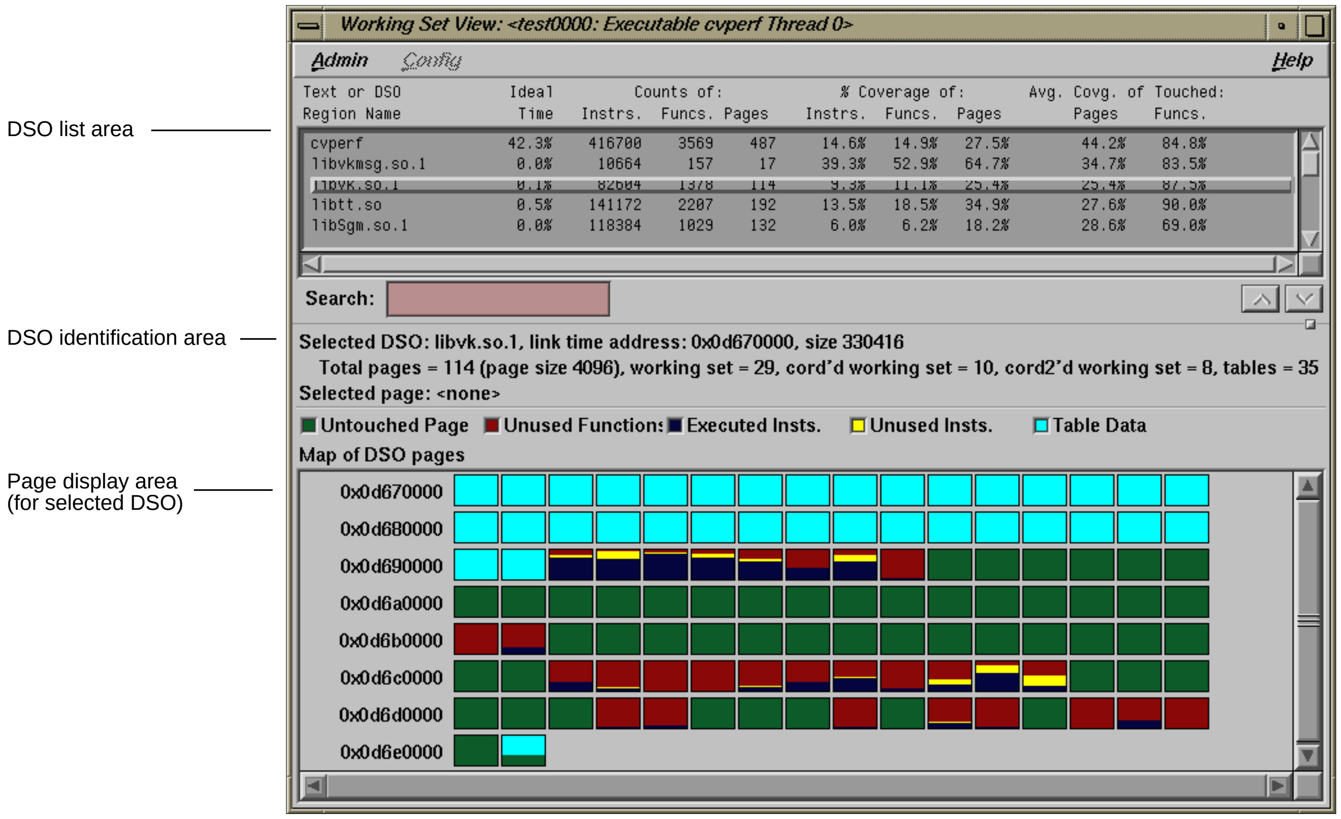
No event found near mouse click

File Descriptor : 3

Figure 1-11 I/O View

Working Set View

Working Set View measures the coverage of the dynamic shared objects (DSOs) that make up your executable (see Figure 1-12). 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.



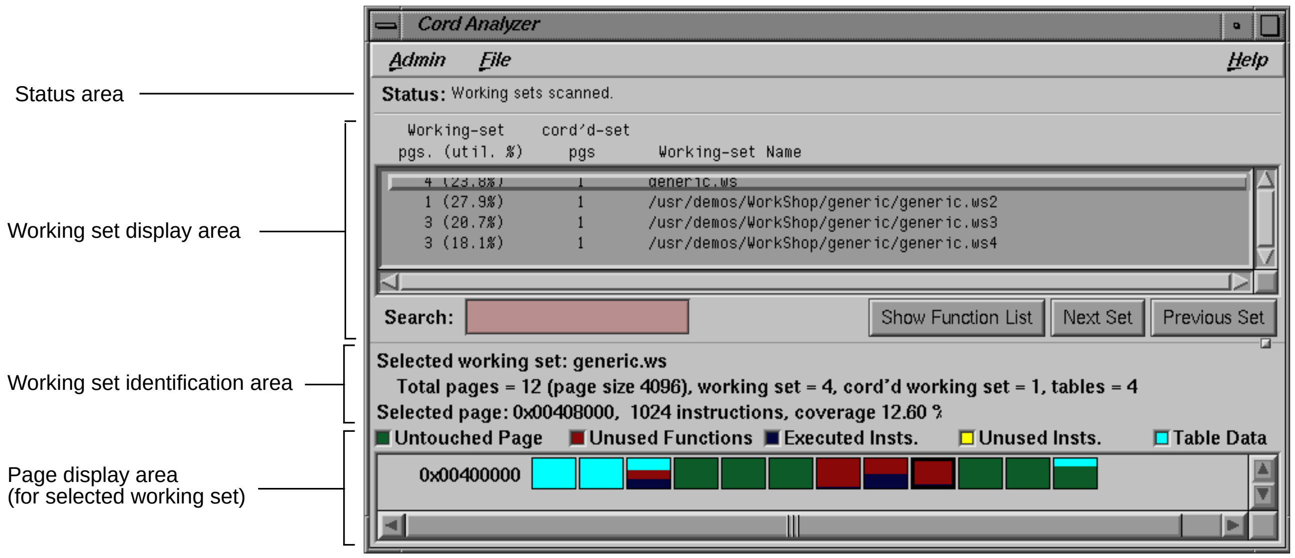
```
Text or DSO            Ideal     Counts of:              % Coverage of:          Avg. Covg. of Touched:
Region Name             Time   Instrs.  Funcs. Pages   Instrs.  Funcs.  Pages      Pages     Funcs.
 cvperf                42.3%   416700    3569   487     14.6%   14.9%   27.5%      44.2%     84.8%
 libvkmsg.so.1          0.0%    10664     157    17     39.3%   52.9%   64.7%      34.7%     83.5%
 libvk.so.1             0.1%    82604    1378   114      9.3%   11.1%   25.4%      25.4%     87.5%
 libtt.so               0.5%   141172    2207   192     13.5%   18.5%   34.9%      27.6%     90.0%
 libSgm.so.1            0.0%   118384    1029   132      6.0%    6.2%   18.2%      28.6%     69.0%
```

Selected DSO: libvk.so.1, link time address: 0x0d670000, size 330416
  Total pages = 114 (page size 4096), working set = 29, cord'd working set = 10, cord2'd working set = 8, tables = 35
Selected page: <none>

Figure 1-12 Working Set View

Cord Analyzer

The Cord Analyzer is not actually part of the Performance Analyzer and is invoked by typing **`cvcord`** at the command line. The Cord Analyzer (see Figure 1-13) 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.



Status: Working sets scanned.

```
  Working-set    cord'd-set
 pgs. (util. %)     pgs        Working-set Name
   4 (23.8%)         1        generic.ws
   1 (27.9%)         1        /usr/demos/WorkShop/generic/generic.ws2
   3 (20.7%)         1        /usr/demos/WorkShop/generic/generic.ws3
   3 (18.1%)         1        /usr/demos/WorkShop/generic/generic.ws4
```

Selected working set: generic.ws
  Total pages = 12 (page size 4096), working set = 4, cord'd working set = 1, tables = 4
Selected page: 0x00408000, 1024 instructions, coverage 12.60 %

🟩 Untouched Page 🟥 Unused Functions ■ Executed Insts. 🟨 Unused Insts. 🟦 Table Data

0x00400000

Figure 1-13 Cord Analyzer

Chapter 2

Performance Analyzer Tutorial

This chapter provides a short tutorial to introduce you to the major features of the Performance Analyzer.

Chapter 2

Performance Analyzer Tutorial

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

- "Tutorial Overview"
- "Tutorial Setup"
- "Analyzing the Performance Data"

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

Tutorial Overview

This tutorial is based on a 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

As you probably can already guess, 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 context switches, 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/tutorial* if you wish to examine it.

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. You'll need the *arraysum.c* file in either case.
2. Compile the *arraysum.c* file by typing **`make arraysum`**

   This will provide you with an executable for the experiment.
3. From the command line, type **`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 "Identify bottleneck resources & phases" from the "Select Task..." submenu in the Perf menu.

This is a general-purpose performance task that will help us determine the phases of the program and view basic resource usage.

5. 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 and the main Performance Analyzer window is displayed automatically. The experiment may take one to three minutes, depending on your system.

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.

1. Examine the main Performance Analyzer window.

   The Performance Analyzer window now displays the information from the new experiment (see Figure 2-1).

2. Look at the Usage Chart in the Performance Analyzer window.

   There are three general phases. The first phase is I/O-intensive, as evidenced by the high system time. The middle phase takes up most of the experiment. We do not have enough information yet, however, to characterize it. The third phase shows high user time and is CPU-intensive.

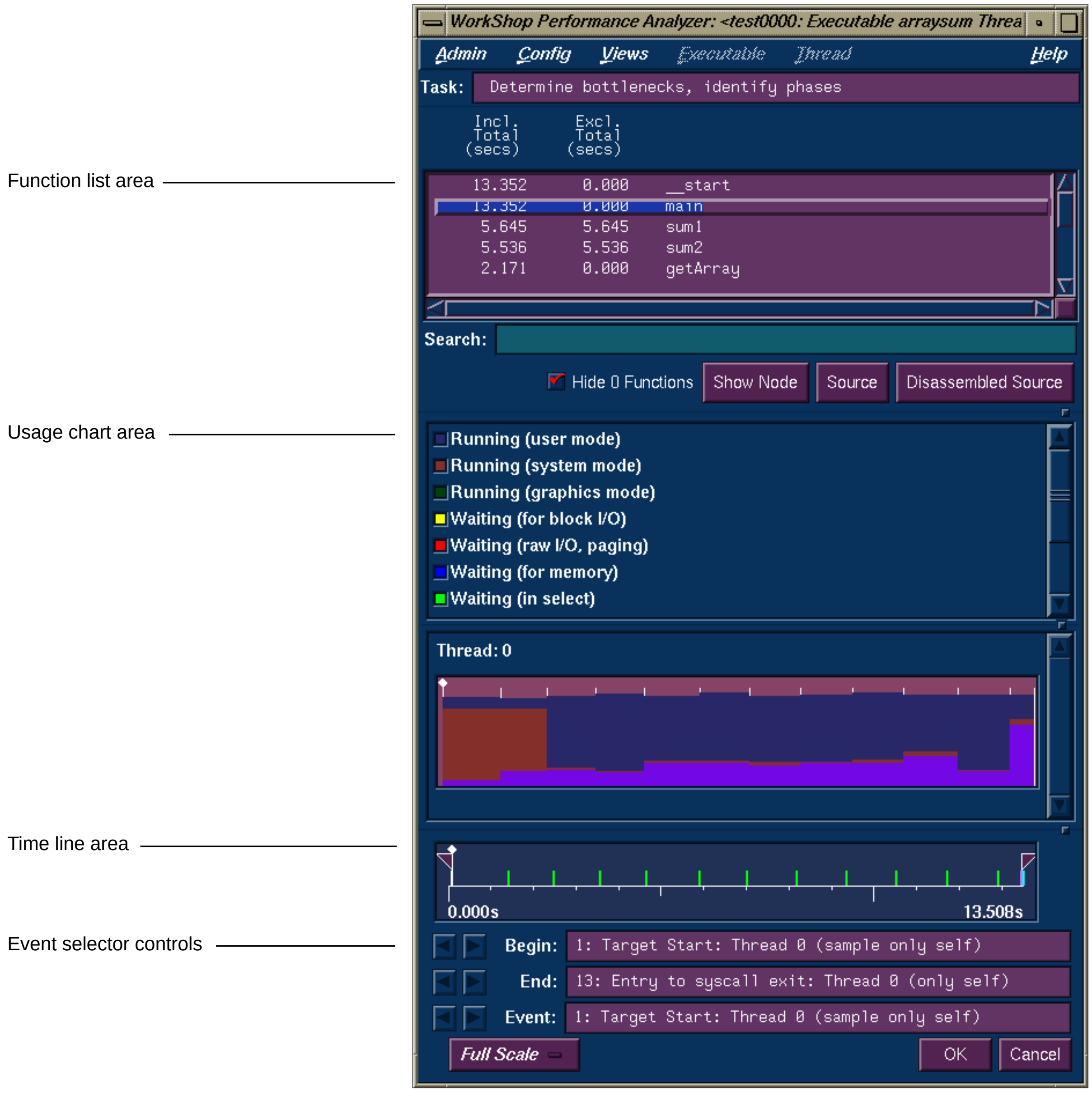
3. Select “Usage View (Graphs)” from the Views menu.

   The Usage View (Graphs) window displays as in Figure 2-2. This indicates that there are significant page faults and context switches in the middle phase. It also shows high read activity and system calls in the first phase, confirming our hypothesis that it is I/O -intensive.

   As a side note, notice that the last chart 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” 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).



Task: Determine bottlenecks, identify phases

```
  Incl.     Excl.
  Total     Total
 (secs)    (secs)

 13.352     0.000     __start
 13.352     0.000     main
  5.645     5.645     sum1
  5.536     5.536     sum2
  2.171     0.000     getArray
```

- 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)

Begin: 1: Target Start: Thread 0 (sample only self)

End: 13: Entry to syscall exit: Thread 0 (only self)

Event: 1: Target Start: Thread 0 (sample only self)

Figure 2-1 Performance Analyzer Main Window—*arraysum* Experiment

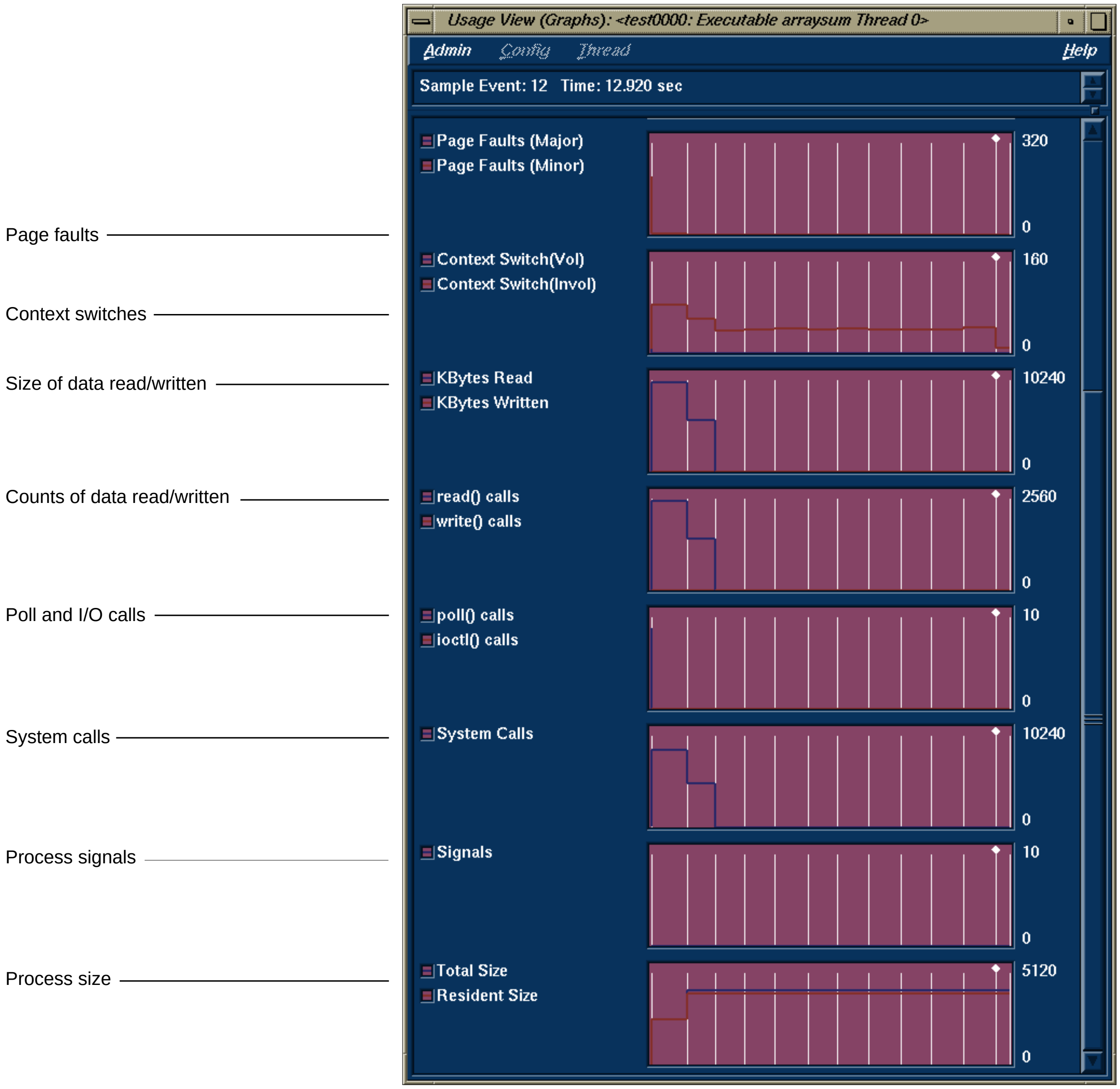


Figure 2-2 Usage View (Graphs)—*arraysum* Experiment

At this point, no events have been selected so the call stack is empty. To select events, you can click in the time line or usage chart. You can also click the event selector controls to make one event at a time (see Figure 2-1).

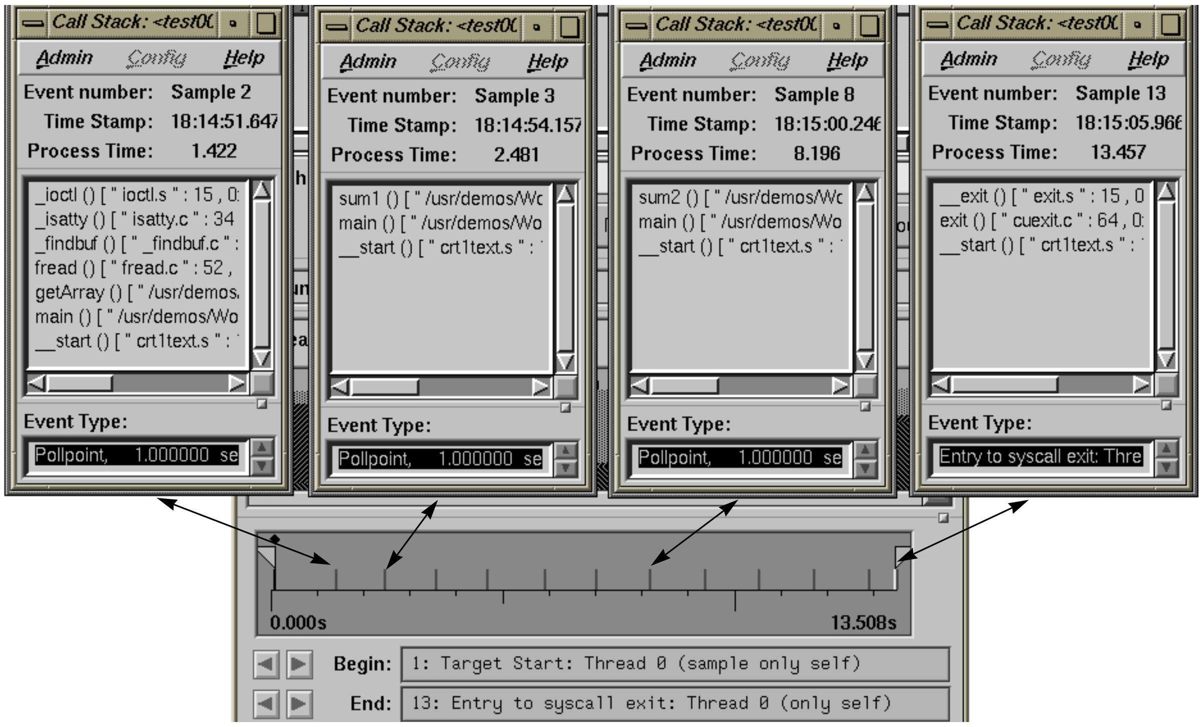
The call stack window indicates the state of the call stack when the event occurred. The significance of the call stack is that it lets you map events to the functions in which they occurred.

5. Select some random events and watch the call stack.

   This exercise helps you see the connection between events and call stacks.

   The important call stacks are the ones that occur at the beginning and end of phases. The general approach is to click in the vicinity of a usage chart where you think a phase boundary may occur and then check the call stack at that point. In this example, events #2, #3, #8, and #13 are important. The call stacks for these events are shown in Figure 2-3, which is drawn to illustrate the relationships, although you can't actually display multiple call stacks at the same time. Remember that your results will be different.

   Event #2 is the last event in the first phase. Events #3 and #7 are the first and last events in the *sum1* function. Event #8 shows the switch from *sum1* to *sum2* and represents the beginning of the last phase. The length of time in *sum1* indicates potential problems.



Event number: Sample 2
Time Stamp: 18:14:51.647
Process Time: 1.422

```
_ioctl () [ " ioctl.s " : 15 , 0:
_isatty () [ " isatty.c " : 34
_findbuf () [ " _findbuf.c " :
fread () [ " fread.c " : 52 ,
getArray () [ " /usr/demos.
main () [ " /usr/demos/Wo
__start () [ " crt1text.s " :
```

Event Type: Pollpoint, 1.000000 se

Event number: Sample 3
Time Stamp: 18:14:54.157
Process Time: 2.481

```
sum1 () [ " /usr/demos/Wc
main () [ " /usr/demos/Wo
__start () [ " crt1text.s " :
```

Event Type: Pollpoint, 1.000000 se

Event number: Sample 8
Time Stamp: 18:15:00.246
Process Time: 8.196

```
sum2 () [ " /usr/demos/Wc
main () [ " /usr/demos/Wo
__start () [ " crt1text.s " :
```

Event Type: Pollpoint, 1.000000 se

Event number: Sample 13
Time Stamp: 18:15:05.966
Process Time: 13.457

```
__exit () [ " exit.s " : 15 , 0
exit () [ " cuexit.c " : 64 , 0:
__start () [ " crt1text.s " :
```

Event Type: Entry to syscall exit: Thre

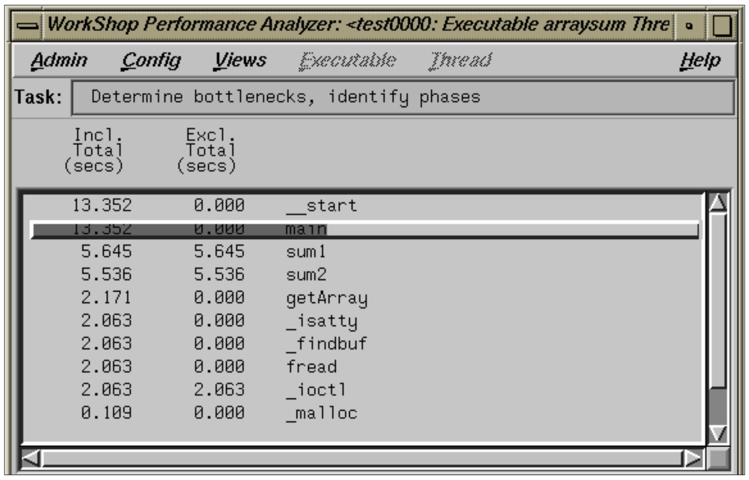
0.000s 13.508s

Begin: 1: Target Start: Thread 0 (sample only self)
End: 13: Entry to syscall exit: Thread 0 (only self)

Figure 2-3 Significant Call Stacks in the *arraysum* Experiment

6. 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, 5.645 seconds are spent in *sum1* and 5.536 seconds in *sum2*.


WorkShop Performance Analyzer: <test0000: Executable arraysum Thre
Admin
Config
Views
Executable
Thread
Help

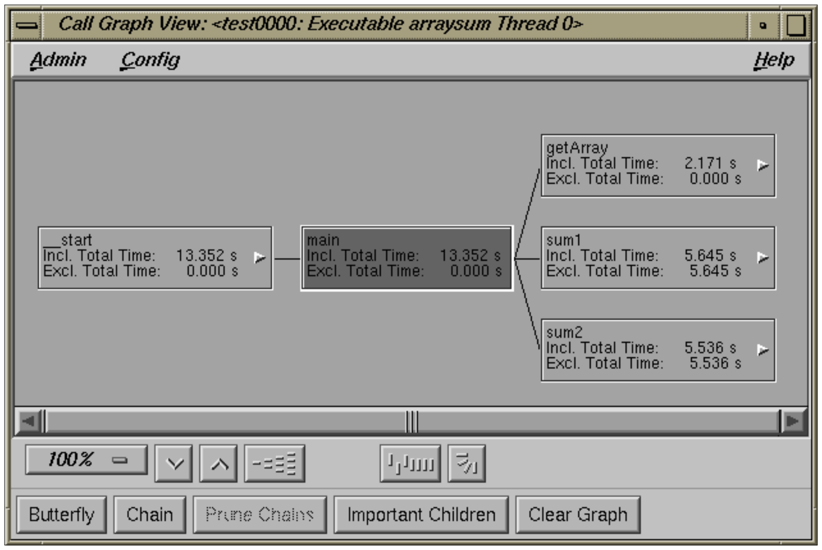
Task: Determine bottlenecks, identify phases

```
 Incl.     Excl.
 Total     Total
 (secs)    (secs)

 13.352    0.000    __start
 13.352    0.000    main
  5.645    5.645    sum1
  5.536    5.536    sum2
  2.171    0.000    getArray
  2.063    0.000    _isatty
  2.063    0.000    _findbuf
  2.063    0.000    fread
  2.063    2.063    _ioctl
  0.109    0.000    _malloc
```

Figure 2-4 Function List Portion of Performance Analyzer Window

7. Select “Call Graph View” from the Views menu and click the *Butterfly* button.

   The call graph provides an alternate means of viewing function performance data. It also shows the relationships, that is, which functions call which functions. After the *Butterfly* button is clicked, Call Graph View displays as in Figure 2-5. 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.

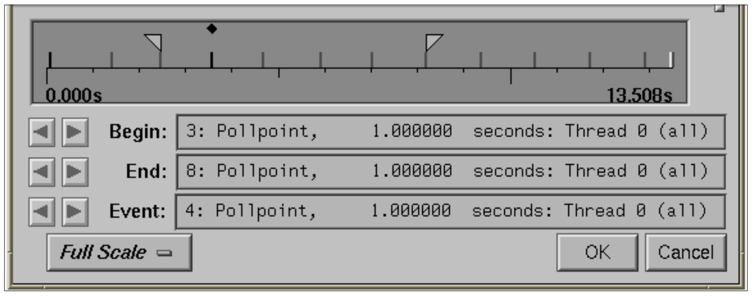


```
getArray
Incl. Total Time:   2.171 s
Excl. Total Time:   0.000 s

__start
Incl. Total Time:   13.352 s
Excl. Total Time:   0.000 s

main
Incl. Total Time:   13.352 s
Excl. Total Time:   0.000 s

sum1
Incl. Total Time:   5.645 s
Excl. Total Time:   5.645 s

sum2
Incl. Total Time:   5.536 s
Excl. Total Time:   5.536 s
```

Figure 2-5 Call Graph View—*arraysum* Experiment

8. Select “Close” from the Admin menu in the Call Graph View to close it. Return to the main Performance Analyzer window and move the left caliper (*Begin*) to event #3 and the right caliper (*End*) to event #8.

   This is shown in Figure 2-6. Moving the calipers like this lets us focus on the data between event #3 and event #8.

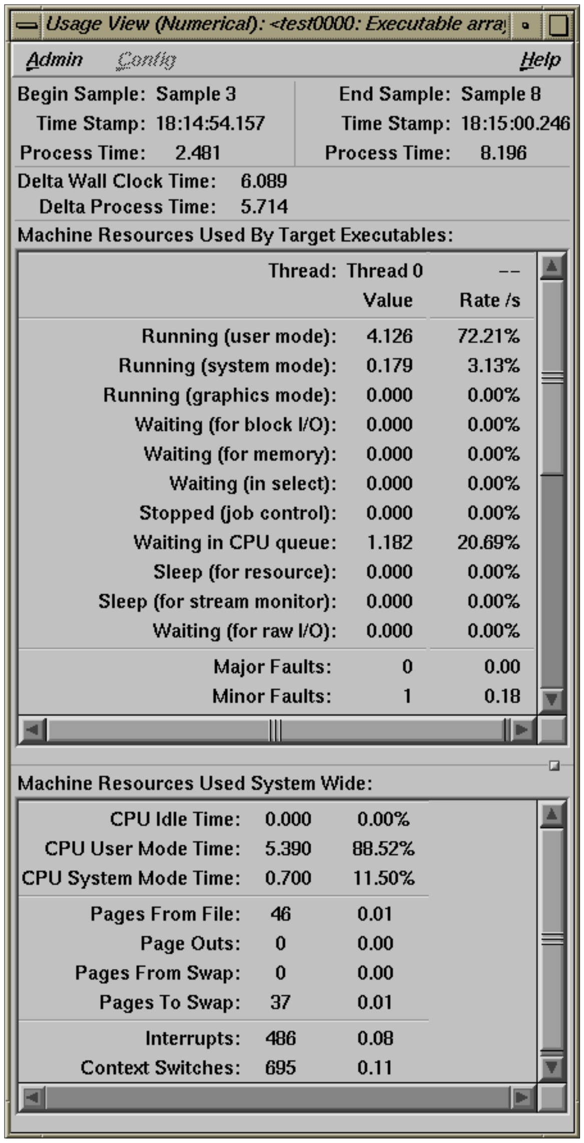


```
Begin: 3: Pollpoint,    1.000000  seconds: Thread 0 (all)
  End: 8: Pollpoint,    1.000000  seconds: Thread 0 (all)
Event: 4: Pollpoint,    1.000000  seconds: Thread 0 (all)
```

Figure 2-6 Defining a Phase with Calipers—*arraysum* Experiment

9. Select “Usage View (Numerical)” from the Views menu.

   The Usage View (Numerical) window displays as shown in Figure 2-7.



Begin Sample:	Sample 3	End Sample:	Sample 8
Time Stamp:	18:14:54.157	Time Stamp:	18:15:00.246
Process Time:	2.481	Process Time:	8.196

Delta Wall Clock Time: 6.089
Delta Process Time: 5.714

Machine Resources Used By Target Executables:

Thread:	Thread 0	--
	Value	Rate /s
Running (user mode):	4.126	72.21%
Running (system mode):	0.179	3.13%
Running (graphics mode):	0.000	0.00%
Waiting (for block I/O):	0.000	0.00%
Waiting (for memory):	0.000	0.00%
Waiting (in select):	0.000	0.00%
Stopped (job control):	0.000	0.00%
Waiting in CPU queue:	1.182	20.69%
Sleep (for resource):	0.000	0.00%
Sleep (for stream monitor):	0.000	0.00%
Waiting (for raw I/O):	0.000	0.00%
Major Faults:	0	0.00
Minor Faults:	1	0.18

Machine Resources Used System Wide:

CPU Idle Time:	0.000	0.00%
CPU User Mode Time:	5.390	88.52%
CPU System Mode Time:	0.700	11.50%
Pages From File:	46	0.01
Page Outs:	0	0.00
Pages From Swap:	0	0.00
Pages To Swap:	37	0.01
Interrupts:	486	0.08
Context Switches:	695	0.11

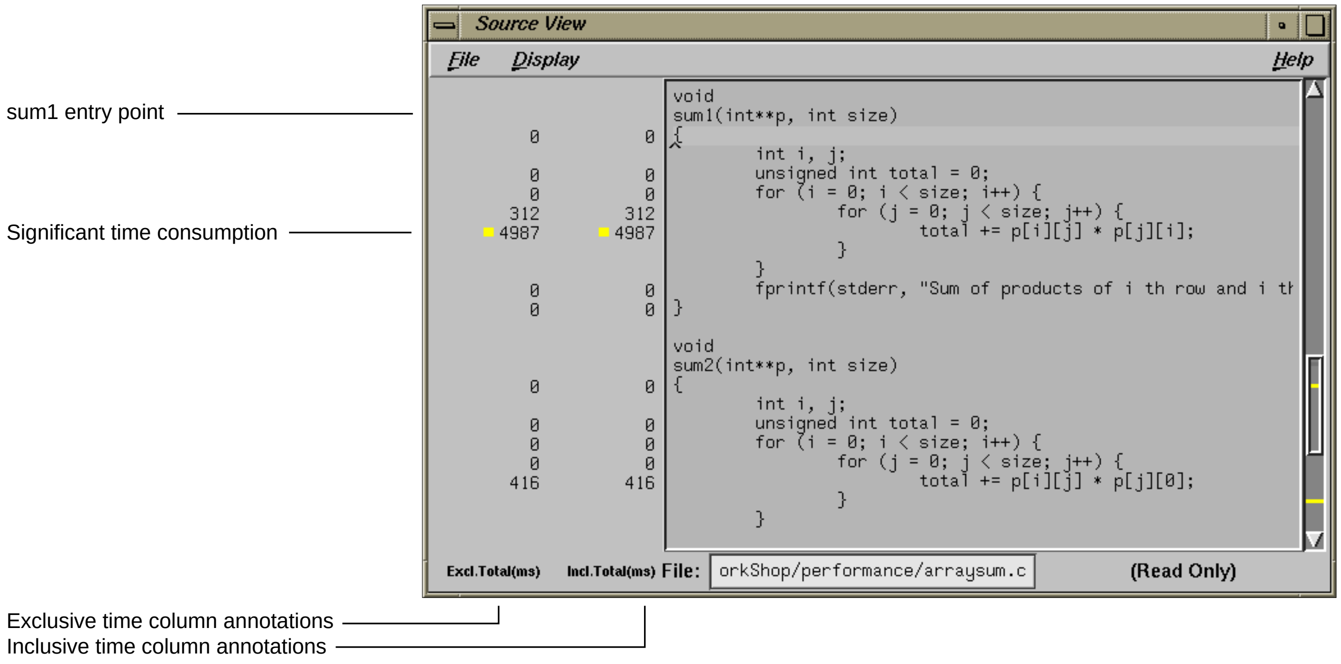
Figure 2-7 Viewing a Phase in the Usage View (Numerical)

This view provides the performance metrics for the interval defined by the calipers, in this case the *sum1* phase.

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

    The Source View window displays as in Figure 2-8, 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 4,987 milliseconds. As in the other WorkShop tools, you can make corrections in Source View, recompile and try out your changes.


Source View
File
Display
Help
sum1 entry point
Significant time consumption
Exclusive time column annotations
Inclusive time column annotations
Excl.Total(ms)
Incl.Total(ms)
File:
orkShop/performance/arraysum.c
(Read Only)

```
                       void
                       sum1(int**p, int size)
     0          0      {
                               int i, j;
     0          0              unsigned int total = 0;
     0          0              for (i = 0; i < size; i++) {
   312        312                      for (j = 0; j < size; j++) {
■ 4987     ■ 4987                              total += p[i][j] * p[j][i];
                                       }
                               }
     0          0              fprintf(stderr, "Sum of products of i th row and i th
     0          0      }

                       void
                       sum2(int**p, int size)
     0          0      {
                               int i, j;
     0          0              unsigned int total = 0;
     0          0              for (i = 0; i < size; i++) {
     0          0                      for (j = 0; j < size; j++) {
   416        416                              total += p[i][j] * p[j][0];
                                       }
                               }
```

Figure 2-8 Source View with Performance Metrics—*arraysum* Experiment

Note: At this point, we have uncovered one performance problem, that the *sum1* algorithm is inefficient. As a side exercise, you may wish 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*. Disassembly View displays, with the performance metrics in the annotation column.

11. Close any windows that are still open.

This concludes the tutorial.

Chapter 3

Setting Up Performance Analysis Experiments

This chapter describes the process of setting up a performance analysis experiment, including sample trap strategy and specifying the type of performance task to capture the relevant performance data.

Chapter 3

Setting Up Performance Analysis Experiments

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:

- “Experiment Setup Overview”
- “Selecting a Performance Task”
- “Setting Sample Traps”
- “Understanding Predefined Tasks”

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.

   You should have an objective in mind when you start an experiment. 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 Task" from the "Select Task..." submenu, which lets you enable any combination of the data collection options.

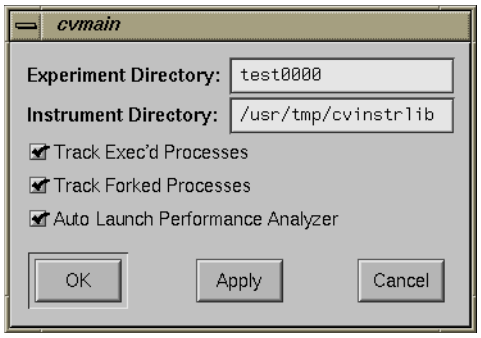
2. Specify where to capture the data.

   If you have selected the "Identify bottleneck resources & phases" task, which automatically polls for performance data, this step is not needed. If you want data at specific points in the process, you need to set sample traps. See "Setting Sample Traps" for a brief description of traps or Chapter 4, "Setting Traps," in *ProDev WorkShop Debugger User's Guide* for an in-depth discussion.

   Performance Analyzer sets sample traps at the beginning and end of the process automatically. If you want to analyze data within phases, then you should set sample traps at the beginning of each phase and at intermediate points, if desired.

3. Specify the experiment configuration parameters.

   This is an optional step if you use the defaults; otherwise you need to select “Configs...” from the Perf menu. This displays the dialog box shown in Figure 3-1.



cvmain

Experiment Directory: test0000

Instrument Directory: /usr/tmp/cvinstrlib

[✓] Track Exec’d Processes

[✓] Track Forked Processes

[✓] Auto Launch Performance Analyzer

Figure 3-1 Performance Experiment Configuration Dialog Box

   The dialog box lets you specify

   - the experiment directory where the data is to be stored
   - the instrument directory where the instrumented executable is to be stored
   - tracking *exec*’d processes
   - tracking *fork*ed processes
   - launching the Performance Analyzer automatically when the experiment finishes

4. Run the program to collect the data.

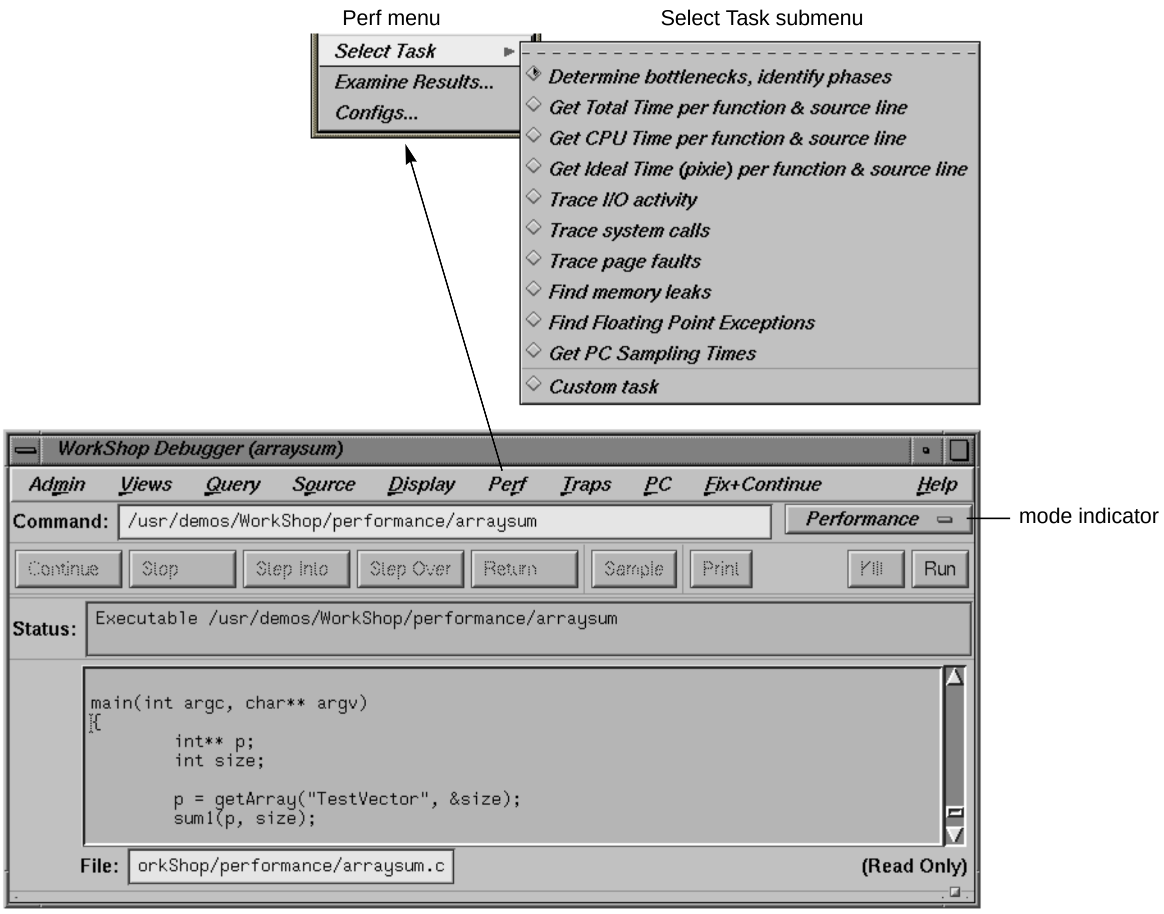
   You run the experiment from the Debugger Main View window. If you are running a small experiment to capture resource usage, you may be able to watch the experiment in real time in Process Meter. Performance Analyzer stores the results in the designated experiment subdirectory.

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.

Selecting a Performance Task

To set up a Performance Analyzer experiment, you need to choose a task from the Select Task submenu in the Perf menu in the Debugger Main View (see Figure 3-2).



Command: /usr/demos/WorkShop/performance/arraysum

Status: Executable /usr/demos/WorkShop/performance/arraysum

```
main(int argc, char** argv)
{
        int** p;
        int size;

        p = getArray("TestVector", &size);
        sum1(p, size);
```

File: orkShop/performance/arraysum.c

Figure 3-2 Perf Menu with Select Task Submenu

The Select Task submenu provides these tasks:

- Determine bottlenecks, identify phases
- Get Total Time per function & source line
- Get CPU Time per function & source line
- Get Ideal Time (pixie) per function & source line
- Trace I/O activity
- Trace system calls
- Trace page faults
- Find memory leaks
- Find Floating Point Exceptions
- Custom task

Selecting a task enables data collection. The mode indicator in the upper right corner of the Main View changes to show that performance analysis is enabled.

Setting Sample Traps

For a thorough discussion of setting traps, refer to Chapter 4, “Setting Traps,”in the *ProDev WorkShop Debugger User’s Guide*. Sample traps enable you to record data when a specified condition occurs. You set them from the Debugger Main View, Trap Manager, or Source View. 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.

Understanding Predefined Tasks

If you are unfamiliar with performance analysis, it is very easy to request more data collection than you actually need—this can degrade performance of the Performance Analyzer and skew results. To help you record data appropriate to your current objective, WorkShop provides predefined combinations of options (or tasks), which are available in the Selact Task submenu in the Perf menu. When you select a task, the required data collection is automatically enabled.

"Determine bottlenecks, identify phases"

"Determine bottlenecks, identify phases" measures machine resource usage and takes pollpoint samples at 1-second intervals.

Call stack data is captured at each pollpoint sample to compute the total time for each function and source line. In call stack profiling, the time spent at a PC (program counter) is determined by multiplying the number of times the PC appears in any call 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 of 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.

Gathering machine resource usage data lets you observe resource consumption over time. With it, you can break your program down into phases with similar resource consumption. You can analyze individual phases in detail in subsequent experiments. You can view resource usage in Usage View (Graphical), Usage View (Numerical), and in the Usage Chart in the Performance Analyzer main window.

Figure 3-3 shows a typical example of the resource usage graph and time line portion of the main Performance Analyzer window for a "Determine bottlenecks, identify phases" task. The resource usage graph shows the user vs. system time. The legend indicates the use of color in the graph. The time line is below the resource graph; it has a time scale so that you can correlate the resource usage with experiment time and with specific events.

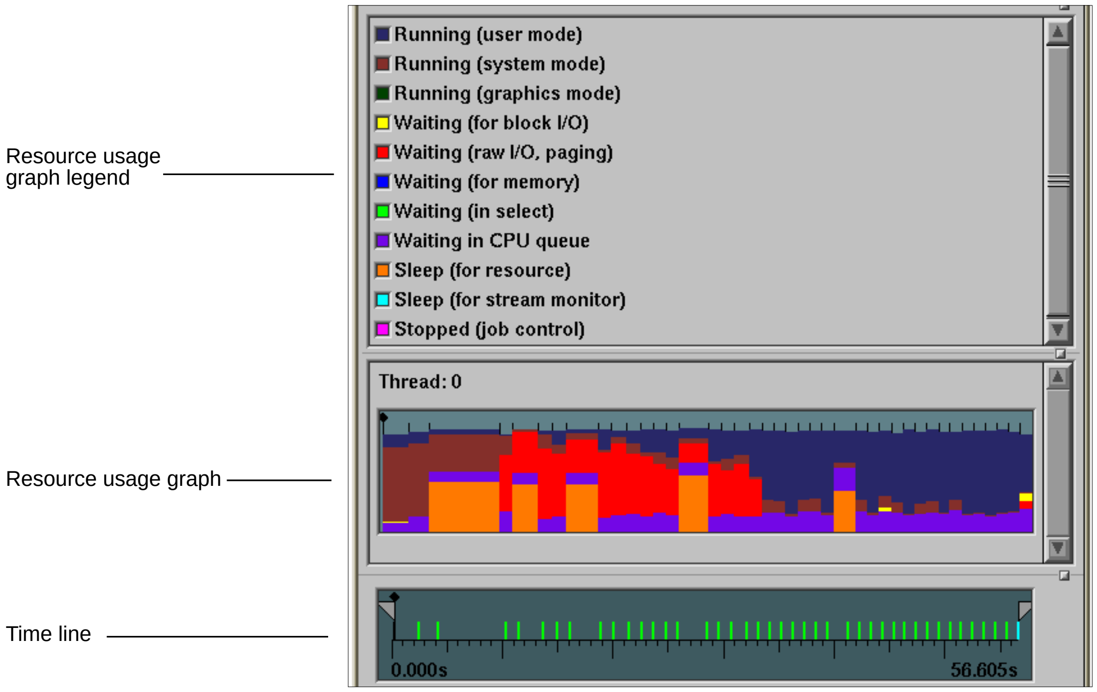


Figure 3-3 Machine Resource Usage in Performance Analyzer Window

"Get Total Time per function & source line"

Use "Get Total Time per function & source line" to tune a phase that has been determined not to be CPU-bound. This task records:

- call stacks every 100 ms, whether the target program is running or blocked
- machine resource usage data at 1-second pollpoints and at sample points

The Total Time values for the PCs are summed up and displayed:

- by function in the function list
- by source line in Source View
- by instruction in Disassembly View

"Get CPU Time per function & source line"

Use "Get CPU Time per function & source line" to tune a CPU-bound phase. It enables you to display the time spent in the CPU by function, source line, and instruction. This task records:

- PC every 10 ms
- function counts
- machine resource usage data at 1-second intervals and at sample points

The CPU time is calculated by multiplying the number of times a PC appears in the profile by 10 ms. PCs are profiled only when the program is running in the CPU; hence, the time computed is the time spent within the CPU, or the CPU time.

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

The CPU time values for the PCs are summed up and displayed:

- by function in the function list
- by source line in Source View
- by instruction in Disassembly View

Function count data is computed by inserting machine code that increments a counter at the start of the function (this is called *instrumentation*). This data is used in the function list to show how many times the function was called and also in the call graph to show how many times one function called another function, that is, arc counts.

PC profiling is done by the kernel and is only minimally intrusive. Gathering function counts intrudes substantially due to:

- instrumentation code consuming CPU cycles
- instrumentation code increasing the target executable size
- PC profiling of the instrumented code itself, which distorts the metrics

However, function counts are useful when combined with PC profiling, because they help in the computation of Inclusive CPU times. (Inclusive CPU time is the total time spent in a function and all the functions it calls; exclusive CPU time is the time spent in the function only.) The arc counts indicate what percentage of a function's CPU time can be attributed to each of its callers.

If you only need Exclusive CPU times and are willing to forgo Inclusive CPU times, arc counts, and function count information, you should select “Custom task” and enable *PC Profile Counts* and set *Fine-Grained Usage* to 1 second.

Also look at the task “Get Ideal Time (pixie) per function & source line”.

“Get Ideal Time (pixie) per function & source line”

Use “Get Ideal Time (pixie) per function & source line” to tune a CPU-bound phase. This task provides exact counts with theoretical times. It is very useful when used in conjunction with the “Get CPU Time per function & source line” task. This approach lets you examine actual versus ideal time. The difference is the time spent as a result of:

- *load* operations, which take a minimum of two cycles if the data is available in the cache and a lot longer if the data has to be accessed from the swap area or second-level cache
- *store* operations, which cause the CPU to stall if the write buffer in the CPU gets filled
- floating point operations, which consume more than one cycle
- time spent with the CPU stalled as a result of data dependencies

This task records:

- basic block counts
- machine resource usage data at 1-second intervals and at sample points

The following results are shown in the function list, Source View, and Disassembly View:

- 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 interlocks)
- the ideal time, that is, the product of the number of the machine instructions executed and the cycle time of the machine (The assumption made in the computation of ideal time is that each instruction takes exactly one cycle to execute.)

This 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 specific kinds of instructions like loads or stores).

Note that the execution of the instrumentation code will skew the behavior of the target executable making it almost entirely CPU-bound; so pay no attention to the User vs Sys Time stripcharts.

“Trace I/O activity”

Use “Trace I/O activity” 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:

- by function in the function list
- by source line in Source View
- by instruction in Disassembly View

The I/O View window displays a graph of the number of bytes read and written for each file descriptor over time, and displays the files involved in the I/O. You can also see the *read* and *write* system calls.

"Trace system calls"

Use "Trace system calls" when you suspect that system calls are slowing down performance and you wish to determine the responsible code.

The number of system calls made is presented:

- by function in the function list
- by source line in Source View
- by instruction in Disassembly View
- To observe the pattern of system calls over time, look in the syscall event chart of the Usage View (Graphical).

"Trace page faults"

The "Trace page faults" task indicates areas of high page faulting activity and identifies the code responsible. The task records call stacks at every page fault.

The number of page faults is presented:

- by function in the function list
- by source line in Source View
- by instruction in Disassembly View

To observe the pattern of page faulting over time, look in the page fault event chart of the Usage View (Graphical).

"Find memory leaks"

Use "Find memory leaks" to determine where memory leaks and bad *frees* may occur in a process. The task records the call stacks, address, and number of bytes at every *malloc*, *realloc*, and *free*. The currently *malloc*ed bytes (that might represent leaks), and the list of double *frees* are presented in Malloc Error View and the other memory analysis views. The number of bytes *malloc*ed is presented:

- by function in the function list
- by source line in Source View
- by instruction in Disassembly View

"Find Floating Point Exceptions"

Use "Find Floating Point Exceptions" when you suspect that large, unaccountable 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:

- by function in the function list
- by source line in 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).

"Custom task"

Use the "Custom task" selection when you need a combination of performance data collected that is not available through the predefined tasks. Selecting "Custom Task" displays the dialog box shown in Figure 3-4.

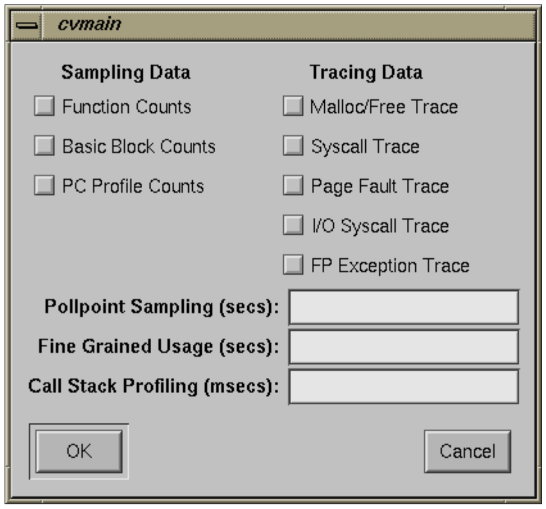


Figure 3-4 Custom Task Dialog Box

The Custom Task dialog box lets you specify

- sampling data—function counts, basic block counts, and PC profile counts
- tracing data—malloc / free trace, syscall trace, page fault trace, I / O syscall trace, FP exception race,
- recording intervals—the frequency of data recording for pollpoint sampling, fine-grained usage, and call stack profiling

Remember the basic warnings in this chapter about collecting data:

- Too much data can bog down the experiment.
- Combining PC profiling and basic block counting will cause the instrumented code to be profiled, including the count code.
- 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.

Chapter 4

Performance Analyzer Reference

This chapter provides detailed information on the Performance Analyzer and its associated facilities.

Chapter 4

Performance Analyzer Reference

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

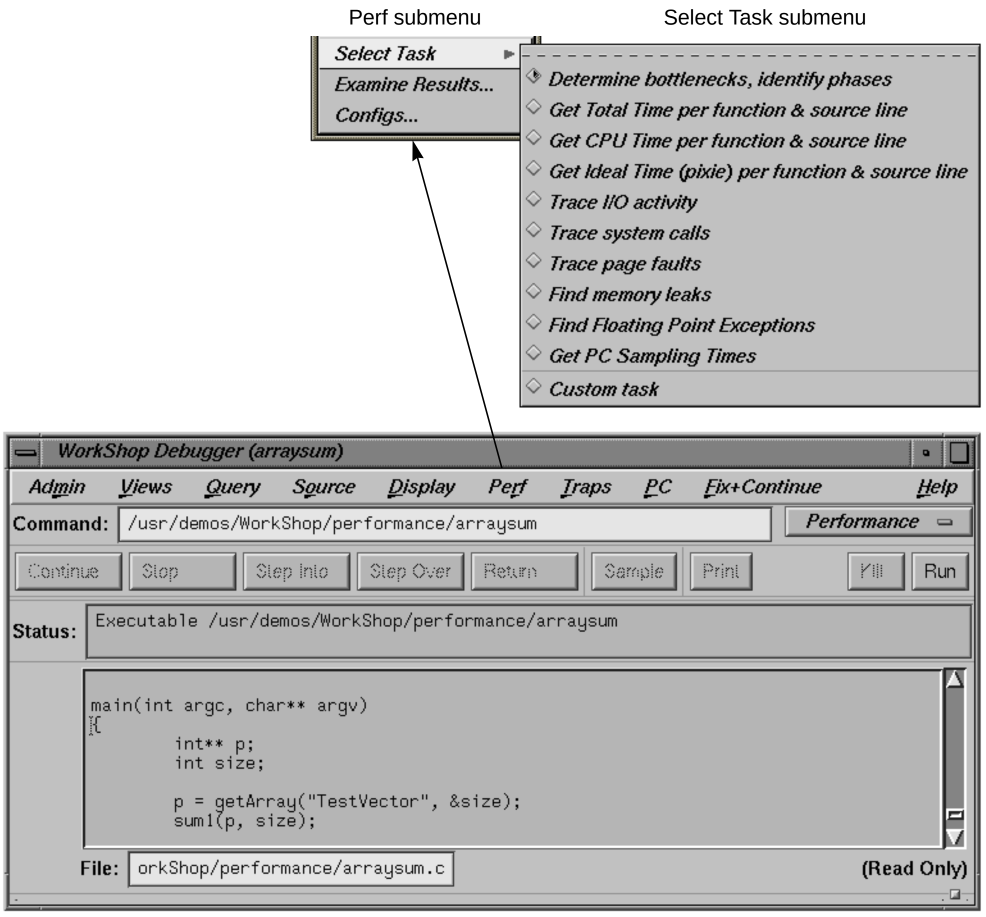
- “Selecting Performance Tasks”
- “Specifying a Custom Task”
- “Specifying the Experiment Configuration”
- “The Performance Analyzer Main Window”
- “Usage View (Graphs)”
- “Process Meter”
- “Usage View (Numerical)”
- “I/O View”
- “Call Graph View”
- “Analyzing Memory Problems”
- “Call Stack”
- “Analyzing Working Sets”

Selecting Performance Tasks

You choose performance tasks from the Select Task submenu in the Perf menu in Main View (see Figure 4-1). 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 bog down the experiment and skew the data for collection.

Task Summary

The tasks are summarized in Table 4-1. The Task column identifies the task as it appears in the Performance Task menu in the Performance Panel window. 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, pollpoints, and process events. The Description column describes the technique used.



Command: /usr/demos/WorkShop/performance/arraysum

Status: Executable /usr/demos/WorkShop/performance/arraysum

```
main(int argc, char** argv)
{
        int** p;
        int size;

        p = getArray("TestVector", &size);
        sum1(p, size);
```

File: orkShop/performance/arraysum.c

Figure 4-1 Performance Panel Window with Task Menu

Table 4-1 Summary of Performance Analyzer Tasks

Task	Clues	Data Collected	Description
Determine bottlenecks, identify phases	Slow program, nothing else known	• Pollpoint Sampling (1 sec.) • Call Stack Profiling (10 msec.) • call stacks at sample points	Captures resource usage at the pollpoint sample and displays it in resource usage graphs. Minimal intrusion. Tracks the total time spent by function, source code line, and instruction.
Get Total Time per function & source line	Not CPU-bound	• Fine-Grained Usage (1 sec.) • Call Stack Profiling (10 msec.) • call stacks at sample points	Tracks the total time spent by function, source code line, and instruction. Useful for non-CPU-bound conditions. Total time metrics are displayed.
Get CPU Time per function & source line	CPU-bound	• Function Counts • 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.
Get Ideal Time (pixie) per function & source line	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.
Trace I/O activity	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.
Trace system calls	Resource usage chart shows high system calls.	• System call Trace • FP Exception Trace • Fine-Grained Usage (1 sec.) • call stacks at sample points	Records all system calls and corresponding call stacks. Gives system call counts for the functions, source code lines, and instructions making the system call. Also provides a stripchart showing the chronological sequence of system calls.
Trace page faults	"Noisy disk" due to accesses	• Page Fault Trace • Fine-Grained Usage (1 sec.) • call stacks at sample points	Captures all page faults and corresponding call stacks. Produces event chart showing the page fault pattern. Lists page faults caused by function, source code line, and instruction.

Table 4-1 (continued) Summary of Performance Analyzer Tasks

Task	Clues	Data Collected	Description
Find memory leaks	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 *mallocs*, *reallocs*, and *frees* and displays them in a memory map. Also indicates double *frees*.
Find Floating Point Exceptions	High sys 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.
Custom task		• 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.

Specifying a Custom Task

When you choose "Custom Task" from the Select Task submenu in the Perf menu in Main View, the dialog box shown in Figure 4-2 appears. This section provides an explanation of the performance data.

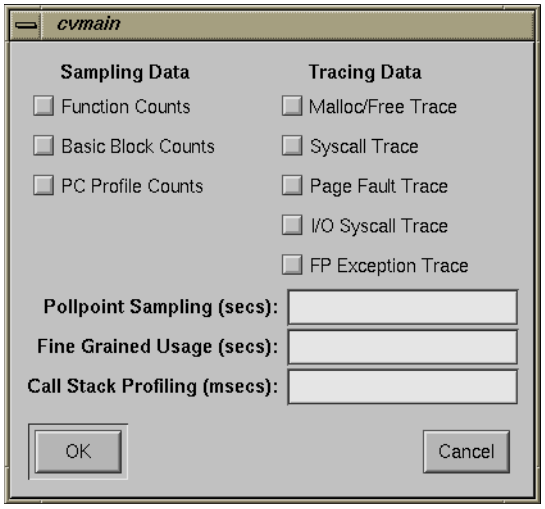


Figure 4-2 Custom Task Dialog Box

Specifying Sampling Data

Sampling data is collected and recorded at every sample point. The collection of sampling data requires instrumentation, that is, adding special code to the target executable. You can request four kinds of sampling data:

- call stack
- function counts
- basic block counts
- PC profile counts

Call Stack Profiling

The Performance Analyzer performs call stack data collection automatically, capturing data at every sample point, pollpoint, and process event. There is no instrumentation involved.

Function Count Collection

Function count collection provides this information:

- execution count of each function
- execution count of each call site

This data is a subset of the information provided by basic block counts. However, gathering function count data does not slow down the instrumented executable as much as gathering basic block data.

Note: It is not possible to collect function count data simultaneously with call stack profiling data.

Basic Block Count Sampling

In addition to the data provided by function counts, basic block counting provides you with the execution count of each line of machine code.

Basic block counts are translated to ideal CPU time displayed at the function, source line and machine line levels. The assumption made in calculating ideal CPU time is that each instruction takes exactly one cycle, and ignores potential floating point interlocks and memory latency time (cache misses and memory bus contention). Each system call is also assumed to take one cycle. The end result might be better described as *ideal user CPU time*.

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 lines of code 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 behavior.

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

Note: It is not possible to collect basic block count data simultaneously with call stack profiling data.

PC Profile Counts

Enabling PC profile counts causes the Program Counter (PC) of the target executable to be sampled every 10 ms when it is in the CPU. PC profiling is a lightweight, high-speed operation done with kernel support. Every 10 ms, 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.

PC Profile Counts is 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 ms.

A major discrepancy between actual CPU time and ideal CPU Time indicates:

- cache misses and floating point interlocks 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 rough approximation is to divide the Actual CPU Time by three.

A comparison between basic block counts and PC profile counts is shown in Table 4-2.

Table 4-2 Basic Block Counts and PC Profile Counts Compared

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

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/Free Tracing"
- "System Call Tracing"
- "Page Fault Tracing"
- "I/O Syscall Tracing"
- "Floating Point Exception Tracing"

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.

Malloc/Free Tracing

Malloc / free tracing enables you to study your program's use of dynamic storage and to quickly detect memory leaks (*mallocs* without corresponding *frees*) and bad *frees* (freeing a previously freed pointer). For this kind of tracing, you must create the target executable by linking with **`-lmalloc_cv`** instead of the usual **`-lmalloc`**. This data can be analyzed in Malloc Error View, Leak View, Malloc View, and Heap View (see "Analyzing Memory Problems" on page 104).

Note that linking with **`-lmalloc_cv`** is not compatible with MP analysis so that using **`-lmpc -lmalloc_cv`** will not work.

System Call Tracing

Enabling system call tracing causes the call stack to be recorded whenever your program makes a system call. This data can be viewed in the system call event chart in Usage View (Graphs) which indicates where the system calls took place and in the Call Stack window which displays the call stack for a selected system call.

Page Fault Tracing

Enabling page fault tracing causes the call stack and the faulting address to be recorded every time your program makes a memory reference that causes a page fault.

The Page Fault event chart displays where the page faults took place in Process View. The Call Stack Information window displays the call stack for a selected page fault event.

I/O Syscall Tracing

I / O syscall 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.

Floating Point Exception Tracing

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.

Note: To use the floating point exception feature, you have to link your program with the library *libfpe.a*.

The floating point exceptions are:

- overflow
- underflow
- divide-by-zero
- inexact result
- invalid operand, e.g., infinity

Specifying Polling Data

There are three categories of polling data:

- “Pollpoint Sampling”
- “Fine Grained Usage”
- “Call Stack Profiling”

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

Pollpoint Sampling

Setting pollpoint sampling 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.

Fine Grained Usage

Resource usage data is always collected at each sample point. Setting a time in the *Fine Grained Usage* field records resource usage data more frequently, at the specified time intervals. Fine grained usage helps you see fluctuations in usage between sample points.

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

Fine grained usage has little effect on the execution of the target process during data collection. It is of limited use if the program is divided into phases of uniform behavior by the placement of the sample points.

Call Stack Profiling

Enabling call stack profiling causes the call stack of the target executable to be sampled at the specified time interval (minimum of 10 ms) and saved. The call stack continues to be sampled when the program is not running, while it is internally or externally blocked. Call stack profiling is used in the "Identify high total time functions" task to calculate total times.

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.

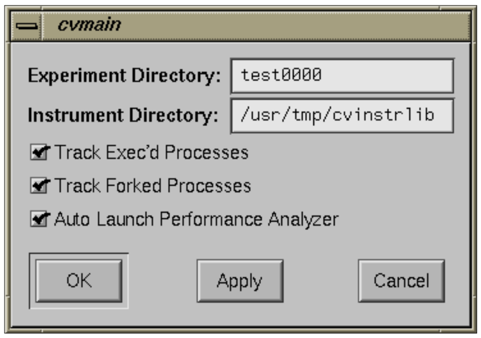
Table 4-3 compares call stack profiling and PC profiling.

Table 4-3 Call Stack Profiling and PC Profiling Compared

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

Specifying the Experiment Configuration

To specify the experiment configuration, you choose “Configs...” from the Perf menu. This displays the dialog box shown in Figure 4-3.


cvmain

Experiment Directory: test0000

Instrument Directory: /usr/tmp/cvinstrlib

✓ Track Exec'd Processes

✓ Track Forked Processes

✓ Auto Launch Performance Analyzer

Figure 4-3 Experiment Configuration Dialog Box

Specifying the Experiment Directory

The *Experiment Directory* field lets you specify the directory where the data captured during the next experiment is stored. The Performance Analyzer provides a default directory named *test0000*. 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 experiment. Note that the Performance Analyzer does not remove (or overwrite) experiment directories. You need to remove directories yourself.

Specifying the Instrument Directory

The *Instrument Directory* lets re-use a previously instrumented executable. This technique avoids the processing necessary for a new instrumentation. Often in a series of experiments, you collect the same type of data while stressing the target executable in different ways. Reusing the instrumented executable lets you do this conveniently.

To reuse an executable from a previous experiment, simply enter the old experiment directory.

Other Options

The *Track Exec'd Processes* toggle allows you to specify whether or not you want the Performance Analyzer to gather performance data for any programs that are launched by an *exec* in any of the target processes. If this feature is enabled and there are *exec*s in the course of the experiment, then you can view the performance data for any of these other executables by using the Executable menu in the Performance Analyzer main window. The *Track Forked Processes* toggle acts analogously for forked processes.

The *Auto Launch Performance Analyzer* toggle provides the convenience of launching the Performance Analyzer automatically when an experiment finishes.

The Performance Analyzer Main Window

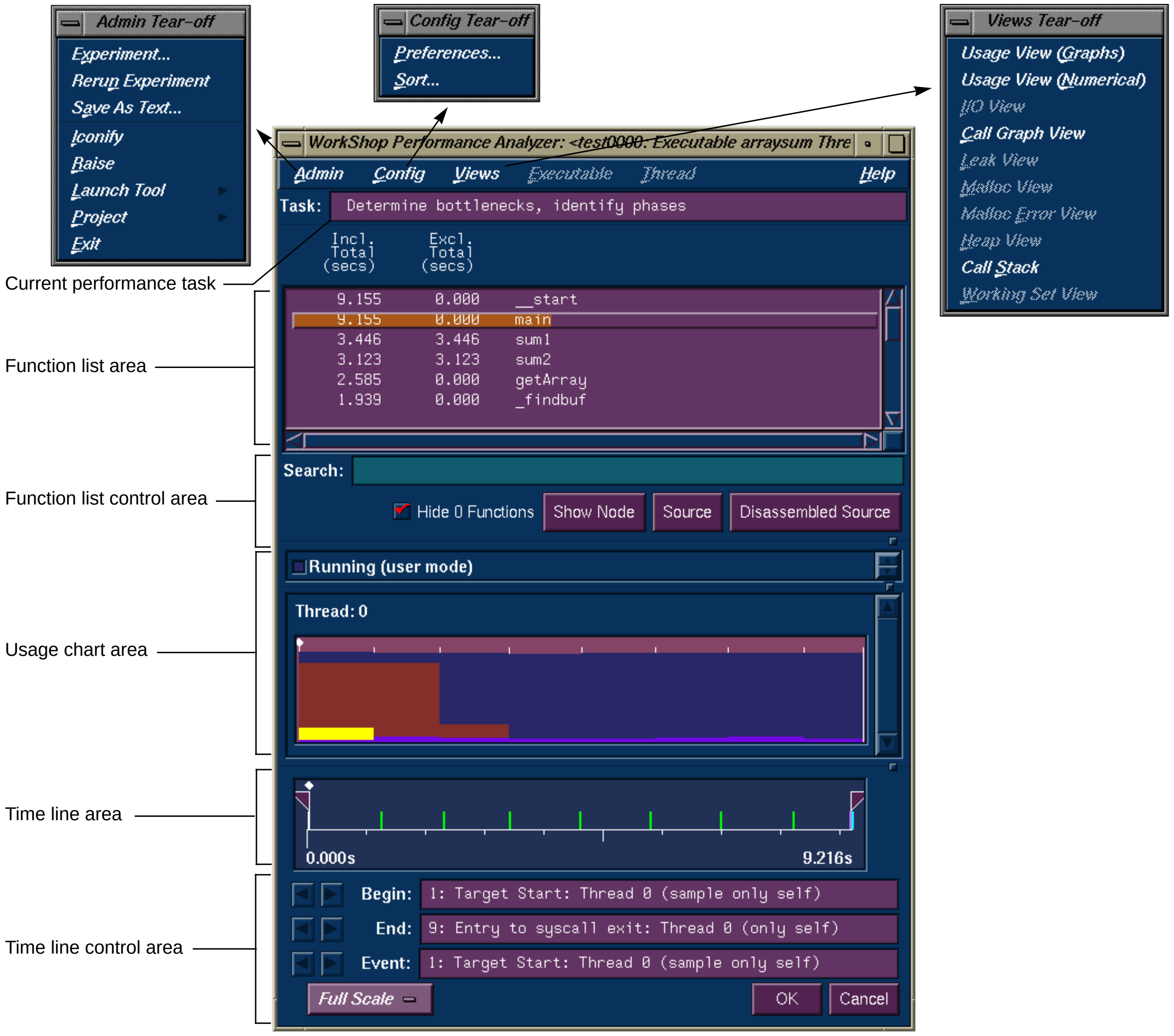
The Performance Analyzer main window is used for analysis after the performance data has been captured (see Figure 4-4). It contains a time line area indicating when events took place over the span of the experiment, a list of functions with their performance data, and a resource usage chart. This section covers these topics:

- "Task Field"
- "Function List Display and Controls"
- "Usage Chart Area"
- "Time Line Area and Controls"
- "Admin Menu"
- "Config Menu"
- "Views Menu"
- "Executable Menu"
- "Thread Menu"

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:

`cvperf -exp` ***`experimentdirectory`***

where ***`experimentdirectory`*** is the directory containing the performance data from the experiment.



Task: Determine bottlenecks, identify phases

```
   Incl.      Excl.
   Total      Total
   (secs)     (secs)
   9.155      0.000      __start
   9.155      0.000      main
   3.446      3.446      sum1
   3.123      3.123      sum2
   2.585      0.000      getArray
   1.939      0.000      _findbuf
```

Begin: 1: Target Start: Thread 0 (sample only self)

End: 9: Entry to syscall exit: Thread 0 (only self)

Event: 1: Target Start: Thread 0 (sample only self)

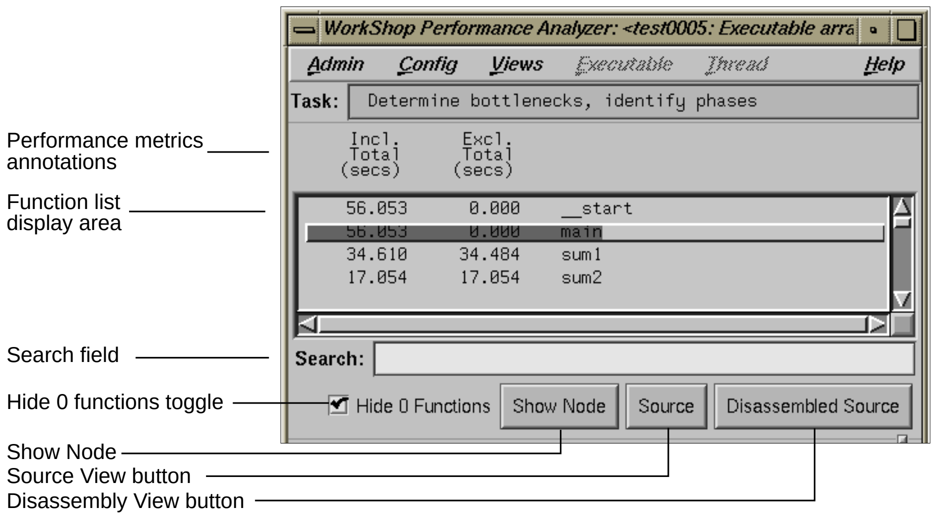
Figure 4-4 Performance Analyzer Main Window with Menus

Task Field

The *Task* field identifies the task for the current experiment and is read-only. See “Selecting Performance Tasks” on page 60 for a summary of the performance tasks. For an in-depth explanation of each task, refer to Chapter 3, “Setting Up Performance Analysis Experiments.”

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 4-5.



```
Task: Determine bottlenecks, identify phases

  Incl.     Excl.
  Total     Total
  (secs)    (secs)
  56.053     0.000    __start
  56.053     0.000    main
  34.610    34.484    sum1
  17.054    17.054    sum2
```

Figure 4-5 Typical Function List Area

The main features of the function list are:

Function list display area
: shows all functions in the source code 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
lets you filter functions with 0 counts from the list.

Show Node
causes the specified node to display in the call graph.

Source
lets you display the Source View window corresponding to the selected function. The Source View window displays, with 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
lets you display the Disassembly View window corresponding to the selected function. Disassembly View displays, annotated with the performance metrics for total (CPU) time.

Usage Chart Area

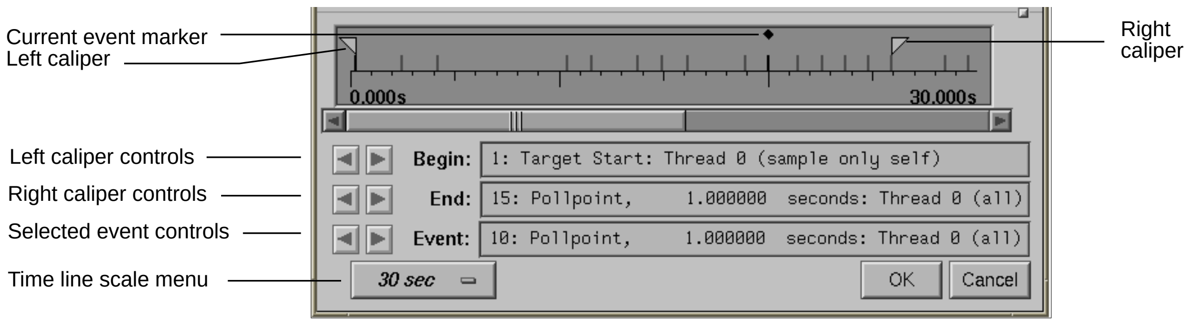
The usage chart area in the Performance Analyzer main window (see Figure 4-4) displays the stripchart most relevant to the current task. The upper subwindow displays the legend for the stripchart and the lower subwindow displays the stripchart itself. This lets you obtain some useful information without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

Table 4-4 Task Display in Usage Chart Area

Task	Data in Usage Chart Area
Determine bottlenecks, identify phases	User versus system time
Get total time per function & source line	User versus system time
Get CPU time per function & source line	User versus system time
Get ideal time per function & source line	User versus system time
Trace I/O activity	*read()*, *write()* system calls
Trace system calls	System call event chart
Trace page faults	Page fault event chart
Find memory leaks	Process Size stripchart
Find floating point exceptions	Floating point exception event chart
Custom task	User versus system time unless tracing data has been selected (see Trace tasks above)

Time Line Area and Controls

The time line shows when each sample event in the experiment occurred. Figure 4-6 shows the time line portion of the Performance Analyzer window with typical results.



```
1: Target Start: Thread 0 (sample only self)
15: Pollpoint,     1.000000  seconds: Thread 0 (all)
10: Pollpoint,     1.000000  seconds: Thread 0 (all)
```

Figure 4-6 Typical Performance Analyzer Time Line

The Time Line Calipers

The 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. (Note that it actually doesn't 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.

Current Event Selection

If you want to get more information on an event in the time line or in the charts in Usage View (Graphs), you can click an event with the left button. The *Event* field (see Figure 4-6) displays

- event number
- description of the trap that triggered the event
- the thread in which it was defined
- whether the sample was taken in all threads or the indicated thread only, in parentheses

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.

Time Line Scale Menu

The 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.

Admin Menu

The Admin menu and its options are shown in Figure 4-7. The Admin menu has selections common to the other WorkShop tools. There are three selections different in the Performance Analyzer:

“Experiment...”
: lets you change the experiment directory and displays the dialog box shown in Figure 4-7.

“Rerun Experiment”
: lets you run another experiment with or without the same Performance Panel settings. A dialog box displays

requesting confirmation (see Figure 4-7). The Debugger Main View window then displays so that you can start the experiment.

“Save As Text...”

records a text file with preference information selected in the view and displays the dialog box shown in Figure 4-7. You can use the default file name or replace it with another name in the File 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.

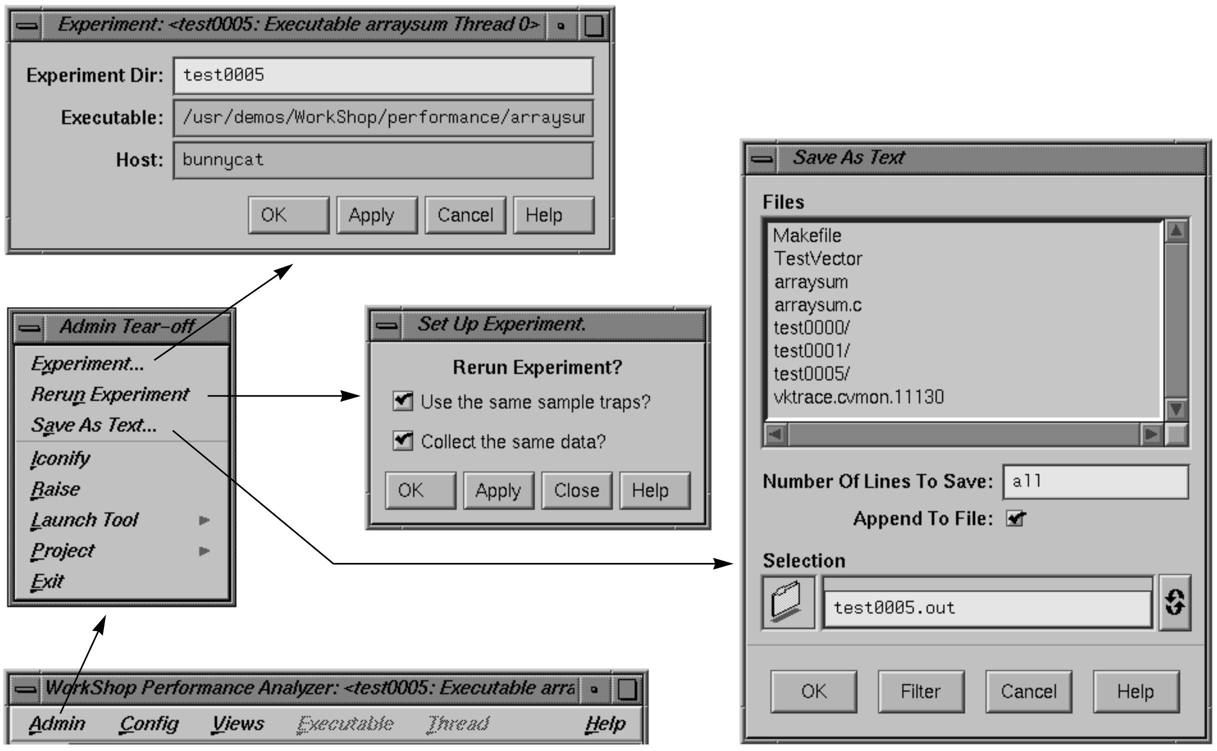


Figure 4-7 Performance Analyzer Admin Menu Options

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.

The selections in the Config menu are:

"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 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.

See Figure 4-8.

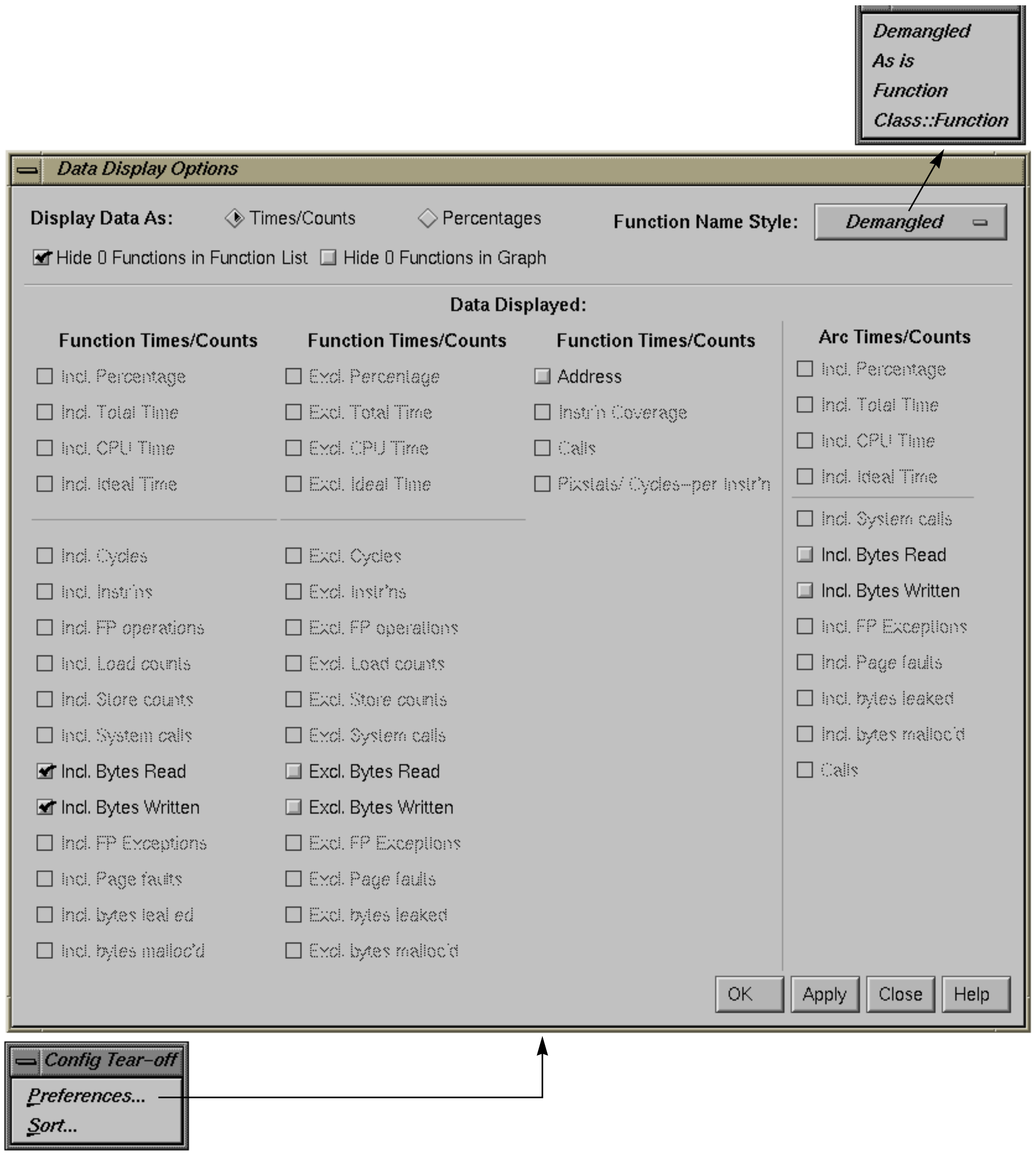
"Sort..."

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 Times or counts
2. Exclusive Time or counts
3. Counts

See Figure 4-8.

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



Data Display Options

Display Data As: [✓ Times/Counts] Percentages

Function Name Style: Demangled

[✓ Hide 0 Functions in Function List] Hide 0 Functions in Graph

Data Displayed:

Function Times/Counts	Function Times/Counts	Function Times/Counts	Arc Times/Counts
Incl. Percentage	Excl. Percentage	Address	Incl. Percentage
Incl. Total Time	Excl. Total Time	Instr'n Coverage	Incl. Total Time
Incl. CPU Time	Excl. CPU Time	Calls	Incl. CPU Time
Incl. Ideal Time	Excl. Ideal Time	Pixstats/ Cycles–per instr'n	Incl. Ideal Time
Incl. Cycles	Excl. Cycles		Incl. System calls
Incl. Instr'ns	Excl. Instr'ns		Incl. Bytes Read
Incl. FP operations	Excl. FP operations		Incl. Bytes Written
Incl. Load counts	Excl. Load counts		Incl. FP Exceptions
Incl. Store counts	Excl. Store counts		Incl. Page faults
Incl. System calls	Excl. System calls		Incl. bytes leaked
✓ Incl. Bytes Read	Excl. Bytes Read		Incl. bytes malloc'd
✓ Incl. Bytes Written	Excl. Bytes Written		Calls
Incl. FP Exceptions	Excl. FP Exceptions		
Incl. Page faults	Excl. Page faults		
Incl. bytes leaked	Excl. bytes leaked		
Incl. bytes malloc'd	Excl. bytes malloc'd		

Figure 4-8 Performance Analyzer Data Display Options

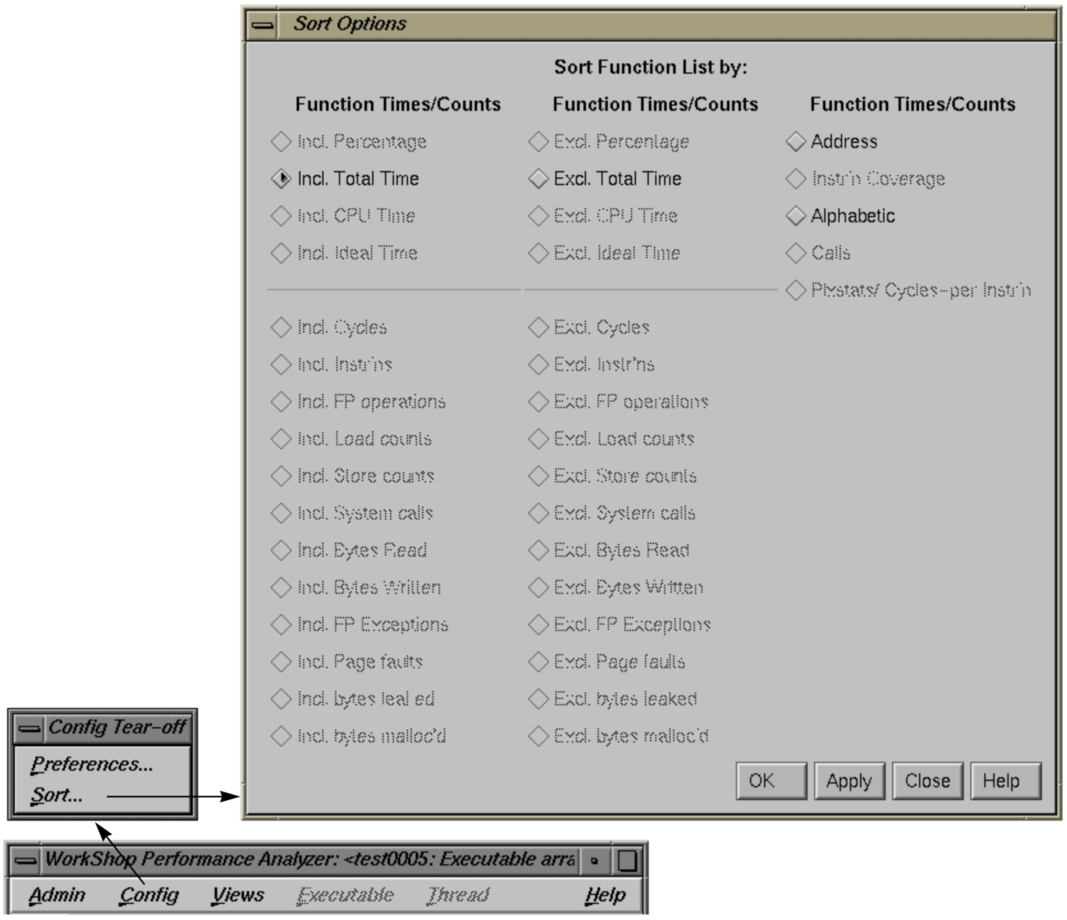


Figure 4-9 Performance Analyzer Sort Options

The toggles in the Data Display Options and Sort Options are:

Address
: is the address of the function.

Calls
: refers to the number of times a function is called.

Incl. Total Time, Excl. Total Time
refers to the 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
refers to the 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
refers to the 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. Malloc counts, Excl. Malloc counts
refers to the number of *malloc*, *realloc*, and *free* operations.

Incl. System calls, Excl. System calls
refers to system calls.

Incl. Page faults, Excl. Page faults
refers to page faults.

Incl. FP operations, Excl. FP operations
refers to floating point operations.

Incl. Load counts, Excl. Load counts
refers to the number of *load* operations.

Incl. Store counts, Excl. Store counts
refers to the number of *store* operations.

Incl. Bytes Read, Excl. Bytes Read
refers to the number of bytes in a *read* operation.

Incl. Bytes Written, Excl. Bytes Written
refers to the number of bytes in a *write* operation.

Incl. FP Exceptions, Excl. FP Exceptions
refers to the number of floating point exceptions.

Incl. Instructions, Excl. Instructions
refers to the number of instructions.

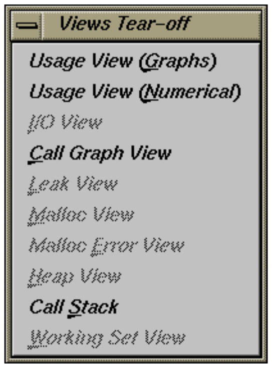


Figure 4-10
Performance Analyzer Views Menu

Views Menu

The Views menu in Performance Analyzer (see Figure 4-10) provides these 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. Refer to “Usage View (Graphs)”.

“Usage View (Numerical)”
: displays the aggregate values of resources used. Refer to “Usage View (Numerical)”.

“I/O View”
: displays I/O events. Refer to “I/O View”.

“Call Graph View”
: displays a call graph that shows functions and calls and their associated performance metrics. Refer to “Call Graph View”.

“Leak View”
: displays individual leaks and their associated call stacks.

“Malloc View”
: displays individual *malloc*s and their associated call stacks.

“Heap View”
: displays a map of heap memory showing *malloc*, *realloc*, *free*, and bad *free* operations. Refer to “Analyzing the Memory Map with Heap View”.

“Call Stack”
: displays the call stack for the selected event and the corresponding event type. Refer to “Call Stack”.

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*’d processes. These selections let you see the performance results for the other executables.

Thread Menu

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

Usage View (Graphs)

Usage View (Graphs) 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 4-11 shows the User vs system time and Page faults graphs; Figure 4-12 shows the other graphs.

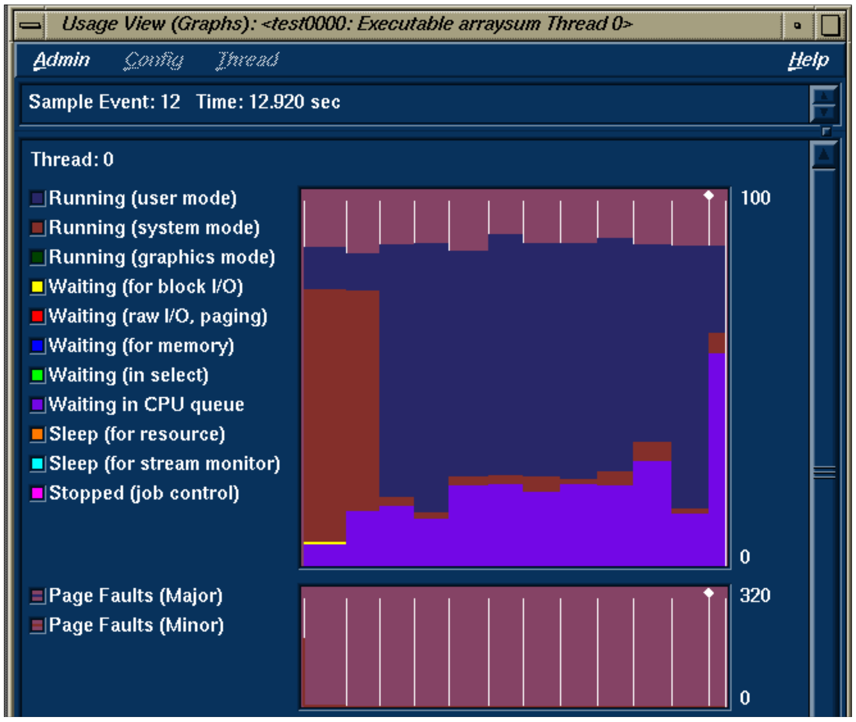


Figure 4-11 Usage View (Graphs) Window: Top Graphs

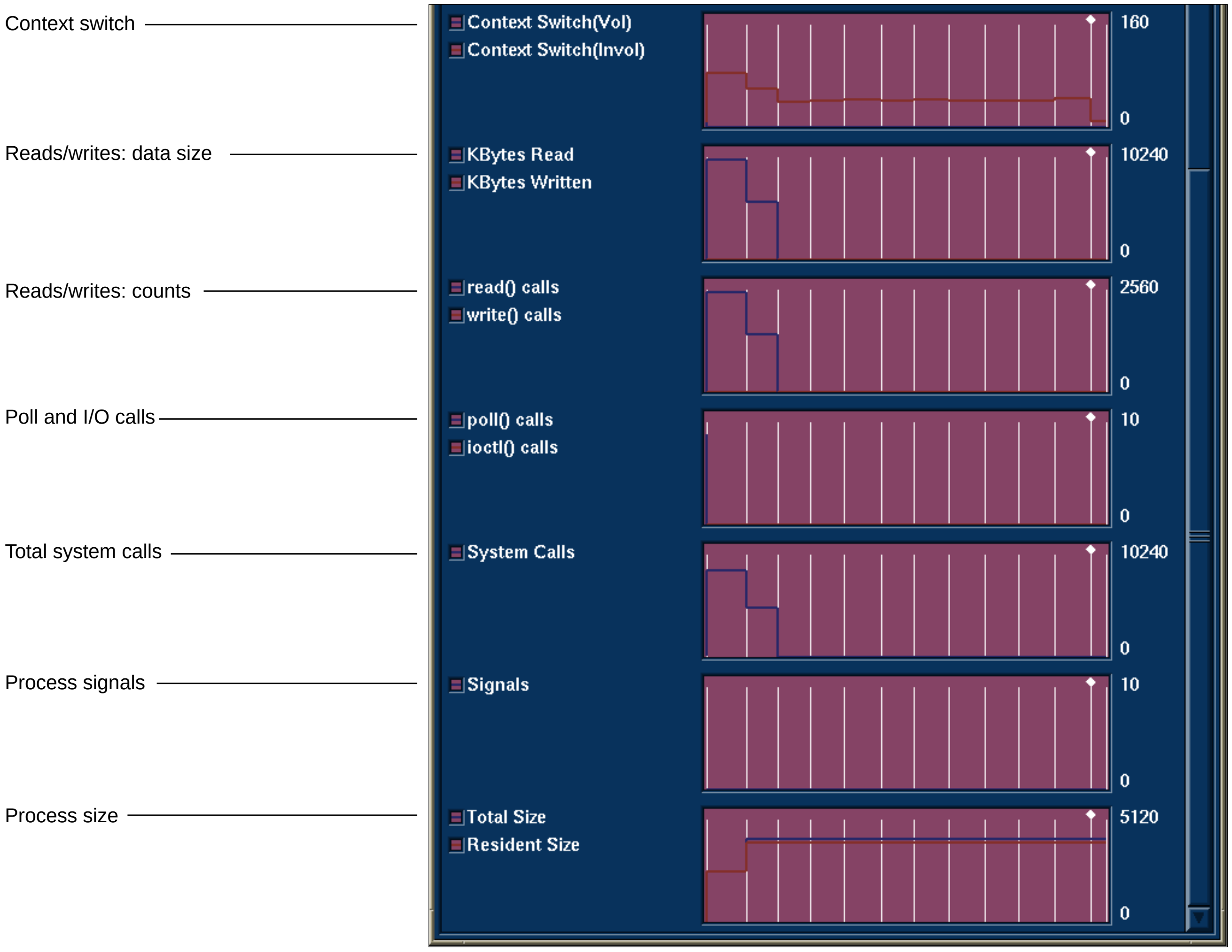


Figure 4-12 Usage View (Graphs) Window: Lower Graphs

Charts in Usage View (Graphs)

The available charts are:

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's address space.

Each major fault in a process takes approximately 10-50 ms. 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 decides to give 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.

Read/write: data size
: 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/write: counts
: shows the number of read and write system calls made by the process.

Poll and I/O 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.

Total 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.

Process signals
: shows the total number of signals received by the process.

Process 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.

Getting Event Information from Usage View (Graphs)

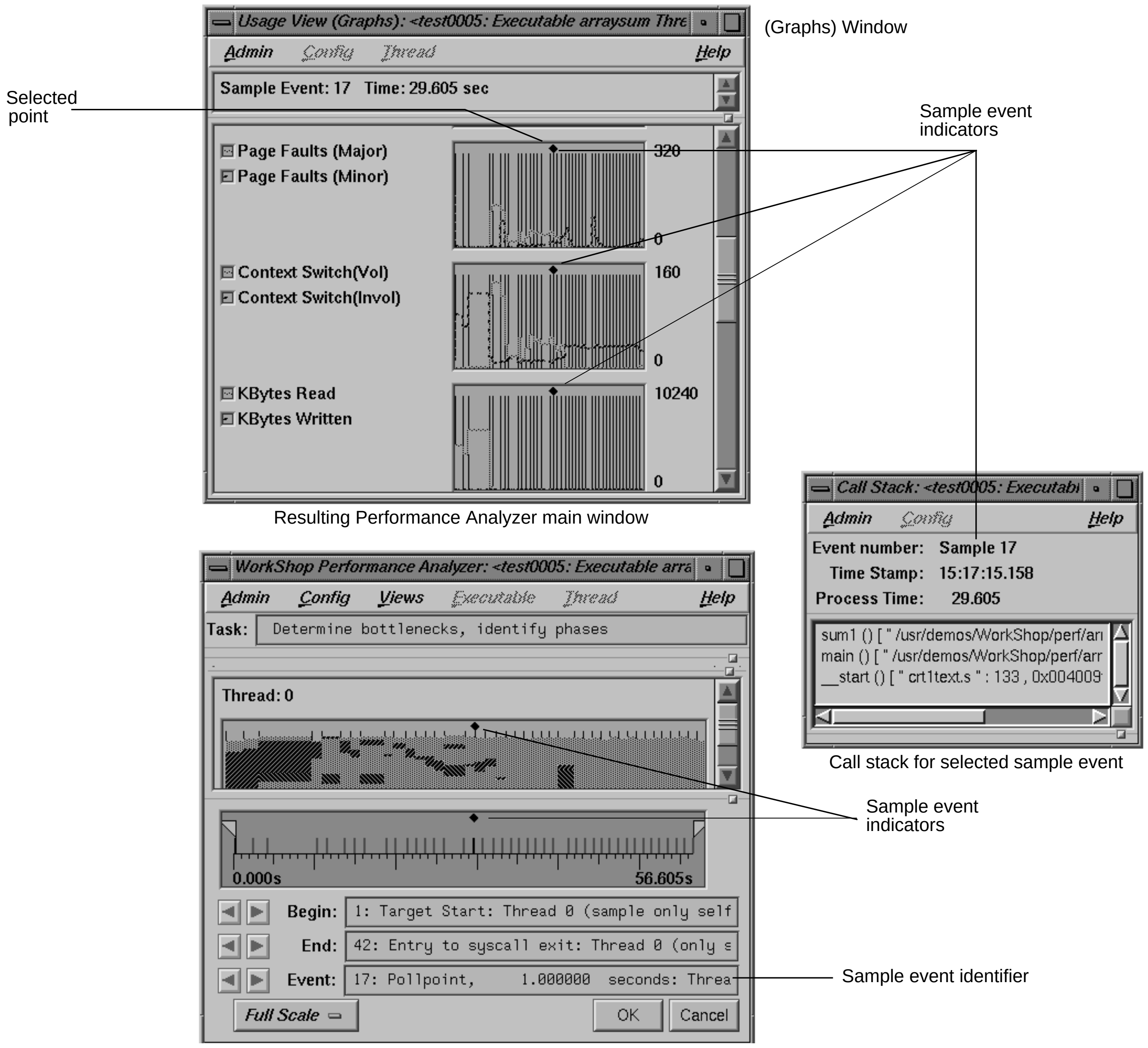
The charts indicate trends; to get detailed data, you click the relevant area on the chart and the data displays in the current event line. The left mouse button displays event data; the right displays interval data.

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 identifies the event and displays its time.
- The call stack corresponding to this sample point gets displayed in the Call Stack window (see “Call Stack”).

Figure 4-13 illustrates the process of selecting a sample event.

Clicking a graph with the right button displays the values for the fine-grained interval (if collection was specified) or if not, the interval bracketed by the nearest sample events.



Sample Event: 17 Time: 29.605 sec

Event number: Sample 17
Time Stamp: 15:17:15.158
Process Time: 29.605

```
sum1 () [ " /usr/demos/WorkShop/perf/arı
main () [ " /usr/demos/WorkShop/perf/arr
__start () [ " crt1text.s " : 133 , 0x004009
```

Task: Determine bottlenecks, identify phases

Begin: 1: Target Start: Thread 0 (sample only self
End: 42: Entry to syscall exit: Thread 0 (only s
Event: 17: Pollpoint, 1.000000 seconds: Threa

Figure 4-13 Effects of Selecting a Sample Event

Process Meter

Process Meter lets you observe resource usage for a running process without conducting an experiment. To call 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 4-14. Process Meter uses the same Admin menu as the WorkShop Debugger tools.

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

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.

Usage View (Numerical)

The Usage View (Numerical) window (see Figure 4-15) 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.

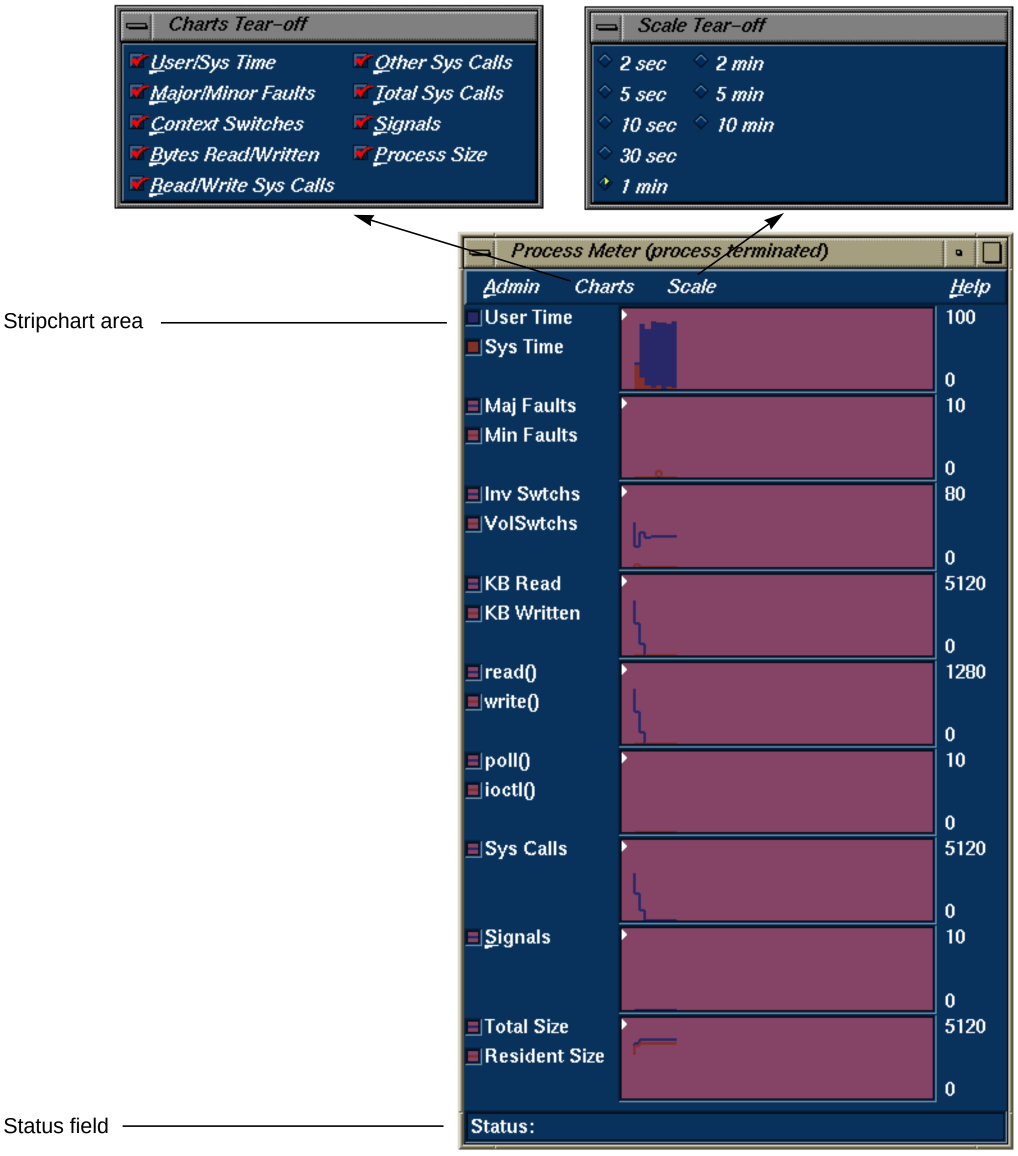
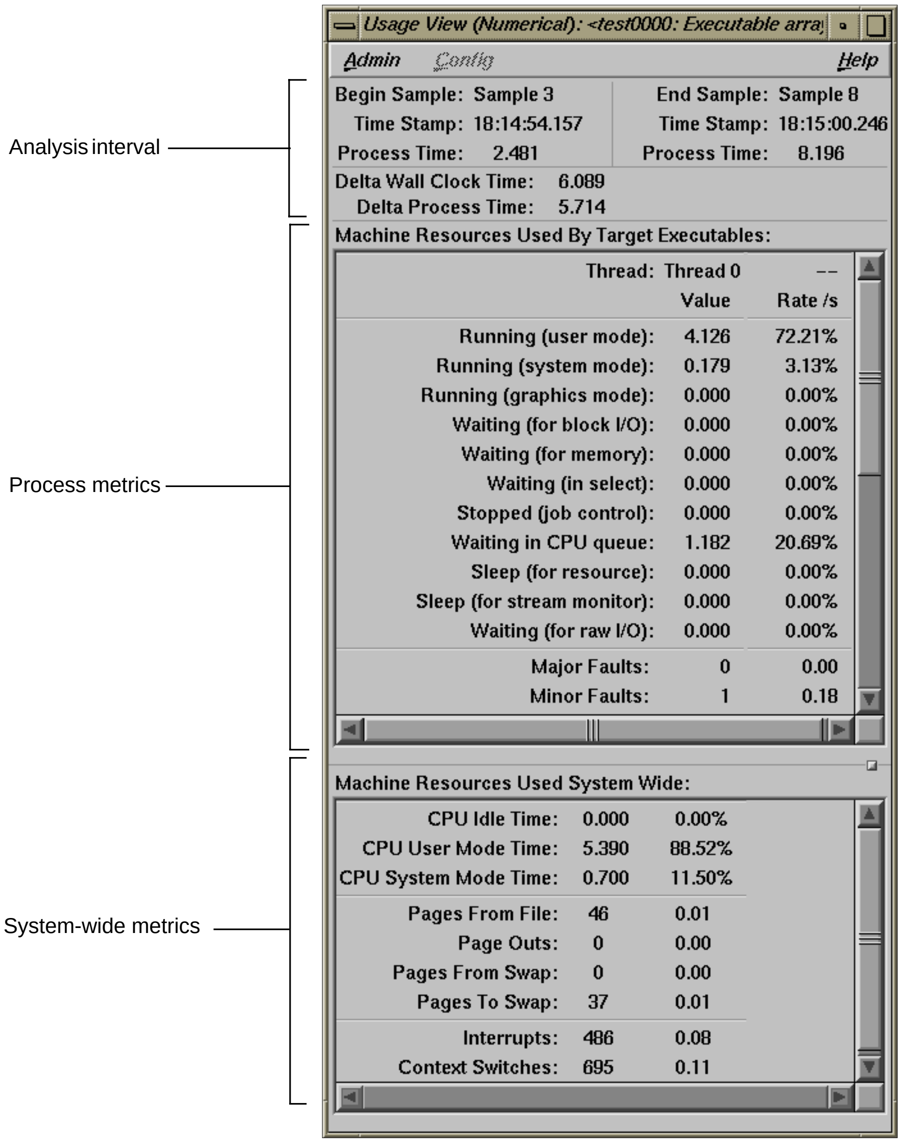


Figure 4-14 The Process Meter with Major Menus Displayed

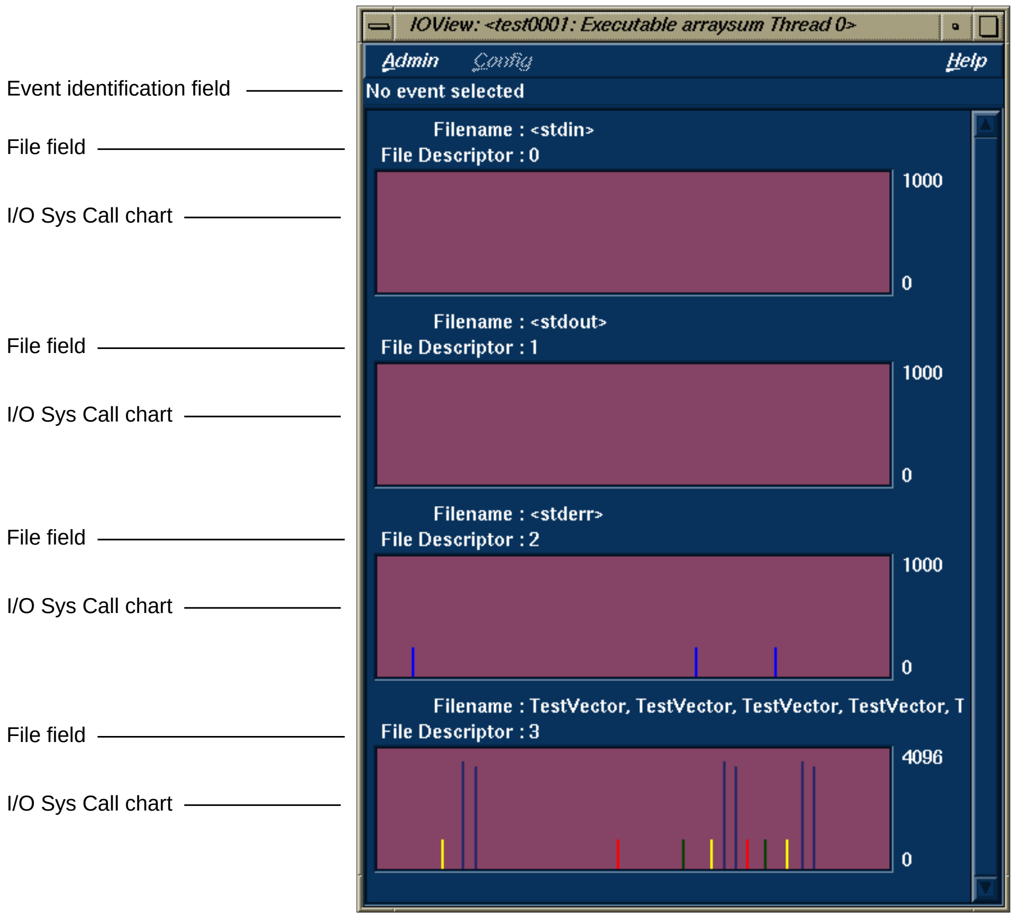


```
Begin Sample:  Sample 3              End Sample:  Sample 8
  Time Stamp:  18:14:54.157           Time Stamp:  18:15:00.246
Process Time:     2.481             Process Time:     8.196
Delta Wall Clock Time:    6.089
  Delta Process Time:    5.714
Machine Resources Used By Target Executables:
                          Thread:  Thread 0       --
                                      Value    Rate /s
          Running (user mode):       4.126    72.21%
        Running (system mode):       0.179     3.13%
      Running (graphics mode):       0.000     0.00%
        Waiting (for block I/O):     0.000     0.00%
          Waiting (for memory):      0.000     0.00%
             Waiting (in select):    0.000     0.00%
          Stopped (job control):     0.000     0.00%
          Waiting in CPU queue:      1.182    20.69%
            Sleep (for resource):    0.000     0.00%
      Sleep (for stream monitor):    0.000     0.00%
           Waiting (for raw I/O):    0.000     0.00%
                  Major Faults:          0      0.00
                  Minor Faults:          1      0.18
Machine Resources Used System Wide:
          CPU Idle Time:    0.000    0.00%
     CPU User Mode Time:    5.390   88.52%
   CPU System Mode Time:    0.700   11.50%
        Pages From File:     46      0.01
             Page Outs:      0       0.00
        Pages From Swap:     0       0.00
          Pages To Swap:     37      0.01
             Interrupts:    486      0.08
       Context Switches:    695      0.11
```

Figure 4-15 Usage View (Numerical)

I/O View

I/O View 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 window. See Figure 4-16.



IOView: <test0001: Executable arraysum Thread 0>

No event selected

Filename : <stdin>
File Descriptor : 0

Filename : <stdout>
File Descriptor : 1

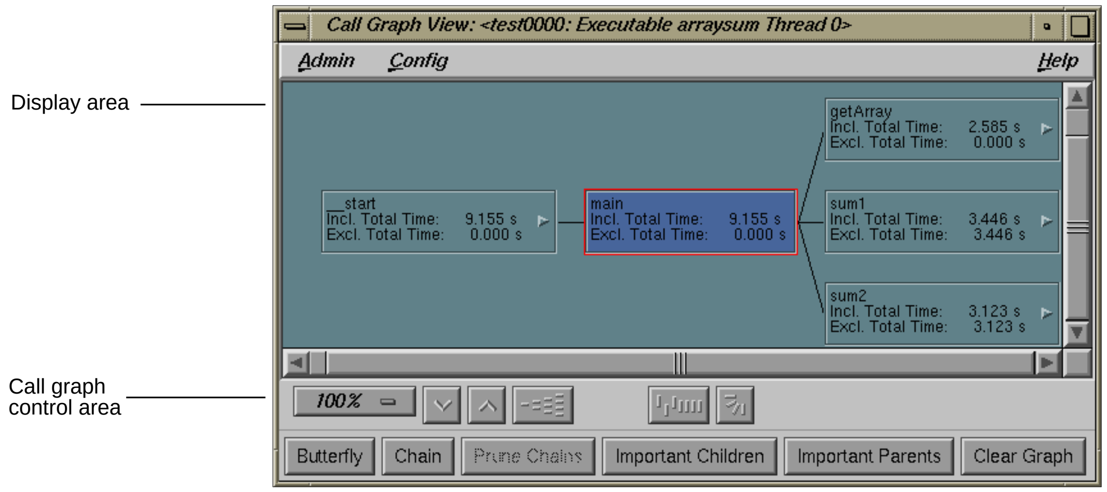
Filename : <stderr>
File Descriptor : 2

Filename : TestVector, TestVector, TestVector, TestVector, T
File Descriptor : 3

Figure 4-16 I/O View

Call Graph View

The Call Graph View window displays a call graph showing the functions as nodes and their calls as connecting arcs, both annotated with performance metrics (see Figure 4-17). You bring up Call Graph View by selecting “Call Graph View” from the Views menu.



```
__start
Incl. Total Time:   9.155 s
Excl. Total Time:   0.000 s

main
Incl. Total Time:   9.155 s
Excl. Total Time:   0.000 s

getArray
Incl. Total Time:   2.585 s
Excl. Total Time:   0.000 s

sum1
Incl. Total Time:   3.446 s
Excl. Total Time:   3.446 s

sum2
Incl. Total Time:   3.123 s
Excl. Total Time:   3.123 s
```

Figure 4-17 Call Graph View with Display Controls

Since a call graph can get quite complicated, 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 and let you change scale, alignment, and orientation, or see an overview (see Appendix A, “Using Graphical Views,”) in the *ProDev WorkShop Overview*. The bottom row of controls lets you define the form of the 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.

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.

Annotating Nodes and Arcs

You can specify which performance metrics appear in the call graph as follows.

Node Annotations

To specify the performance metrics that display inside a node, you need the Preferences dialog box in the Config menu from the Performance Analyzer main view (see Figure 4-8).

Arc Annotations

Arc annotations are specified by selecting "Preferences..." from the Config menu in Call Graph View (see Figure 4-8). You can display the counts on the arcs. You can also display the percentage of calls to a function broken down by incoming arc. For an explanation of the performance metric items, refer to "Config Menu".

Filtering Nodes and Arcs

You can specify which nodes and arcs appear in the call graph as follows.

Call Graph Preferences Filtering Options

The Call Graph Display Options dialog box accessed from the “Preferences” selection in the Call Graph View Config menu also lets you hide functions and arcs that have 0 (zero) calls. See Figure 4-8.

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 4-18.

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

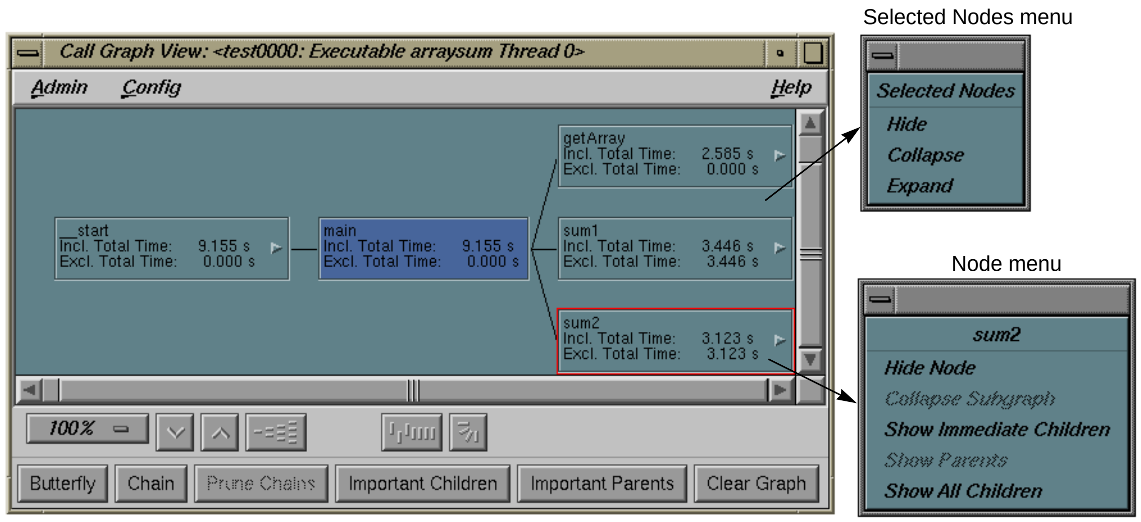

Selected Nodes menu
Call Graph View: <test0000: Executable arraysum Thread 0>
Admin
Config
Help
getArray
Incl. Total Time: 2.585 s
Excl. Total Time: 0.000 s
__start
Incl. Total Time: 9.155 s
Excl. Total Time: 0.000 s
main
Incl. Total Time: 9.155 s
Excl. Total Time: 0.000 s
sum1
Incl. Total Time: 3.446 s
Excl. Total Time: 3.446 s
sum2
Incl. Total Time: 3.123 s
Excl. Total Time: 3.123 s
100%
Butterfly
Chain
Prune Chains
Important Children
Important Parents
Clear Graph
Selected Nodes
Hide
Collapse
Expand
Node menu
sum2
Hide Node
Collapse Subgraph
Show Immediate Children
Show Parents
Show All Children

Figure 4-18 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 (descendants) called by the selected node.

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:

“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.

Filtering Nodes through the Display Controls

The lower row of controls in the panel helps you reduce the complexity of a busy call graph (see Figure 4-19).

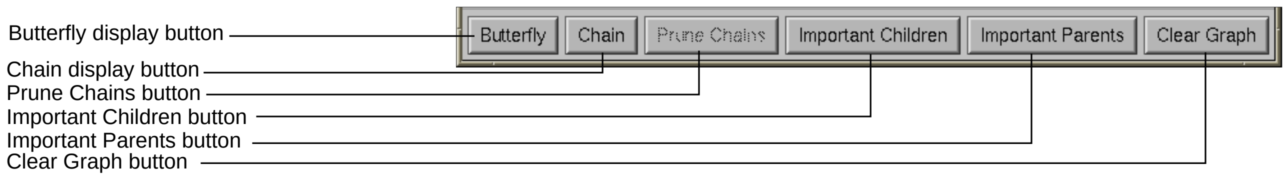
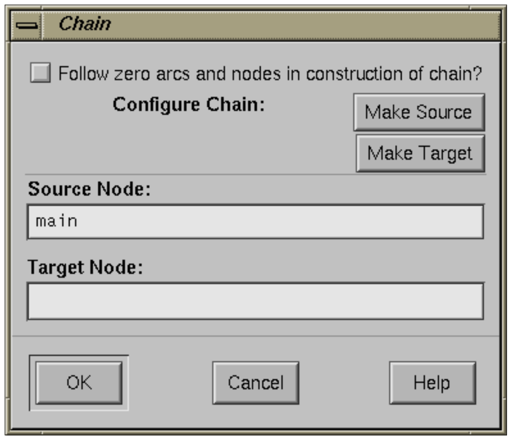


Figure 4-19 Call Graph View Controls for Content Manipulation

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* causes the graph to be redrawn with the selected node as 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 4-20. 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 wish to filter out paths that go through nodes and arcs with 0 counts, click the toggle. After these selections are made, click *OK*.



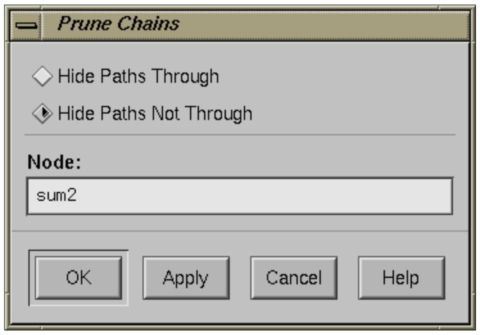
Source Node: main

Target Node:

Figure 4-20 Chain Dialog Box

Prune Chains

displays a dialog box that provides two selections for filtering paths from the call graph (see Figure 4-21).



Hide Paths Through
[✓ Hide Paths Not Through]

Node: sum2

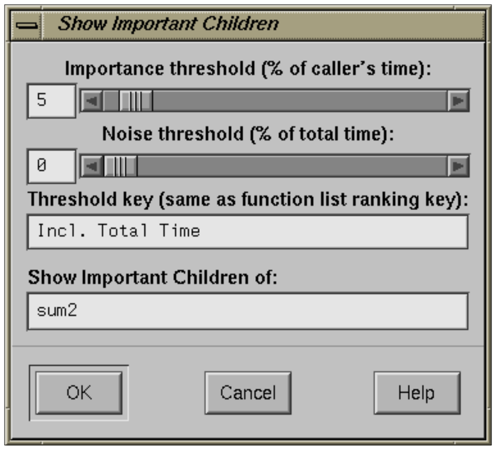
Figure 4-21 Prune Chains Dialog Box

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 re-display the hidden paths unless you perform the chain command 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.

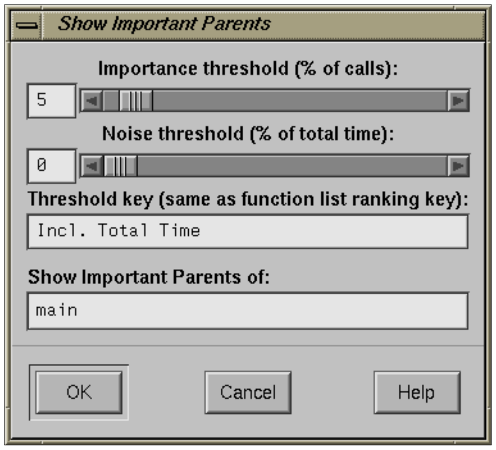
Important Children
lets you focus on a function and its descendants and set thresholds to filter the descendants. You can filter the descendants either by percentage of the caller's time or by percentage of the total time. The *Threshold key* field identifies the type of performance time data used as the threshold. See Figure 4-22.



Importance threshold (% of caller's time): 5
Noise threshold (% of total time): 0
Threshold key (same as function list ranking key): Incl. Total Time
Show Important Children of: sum2

Figure 4-22 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 4-23.



Importance threshold (% of calls): 5

Noise threshold (% of total time): 0

Threshold key (same as function list ranking key): Incl. Total Time

Show Important Parents of: main

Figure 4-23 Show Important Parents Dialog Box

Clear Graph
removes all nodes and arcs from the call graph.

Other Manipulation of the Call Graph

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

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 4-24).

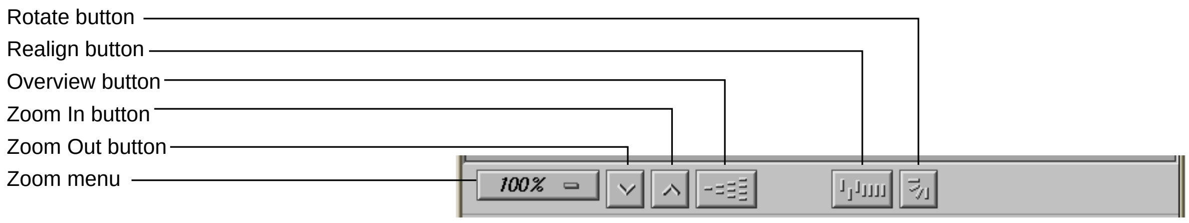


Figure 4-24 Call Graph View Controls for Geometric Manipulation

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 300% 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 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.

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 Appendix A, "Using Graphical Views," in the *ProDev WorkShop Overview.*

Using the Mouse in 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. You can also shift-click a node and it will be selected along with all the nodes that it calls.

Selecting Nodes from the Function List

You can also select functions from the function list to be highlighted in the call graph. You 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.

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.

Conducting Memory Leak Experiments

To look for memory leaks or bad frees, or perform other analysis of memory allocation, you need to run a Performance Analyzer experiment with "Find memory leaks" 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:

1. Link your executable with one of the special WorkShop *malloc* libraries (*-lmalloc_cv* or *-lmalloc_cv_d*).

   Note: For a detailed discussion of the *malloc* libraries, see "Compiling With the Malloc Library" on page 114 in the *ProDev WorkShop Debugger User's Guide*. This tutorial assumes that library *-lmalloc_cv* is used.

   Before you even run a memory experiment, you need to relink your executable with the WorkShop *malloc* library (*libmalloc_cv*) instead of the *malloc* library (*libmalloc*). You can compile it from scratch as follows:

   ```
   cc -g -o targetprogram targetprogram.c -lmalloc_cv
   ```

   or you can relink it by using:

   ```
   ld -o targetprogram targetprogram.o -lmalloc_cv
   ```

2. Display the Performance Panel.

   You can bring up the Performance Panel by selecting “Performance Task...” from the Admin menu in the Debugger Main View or by typing **cvspeed** at the command line.

3. Specify “Find memory leaks” as the experiment task.

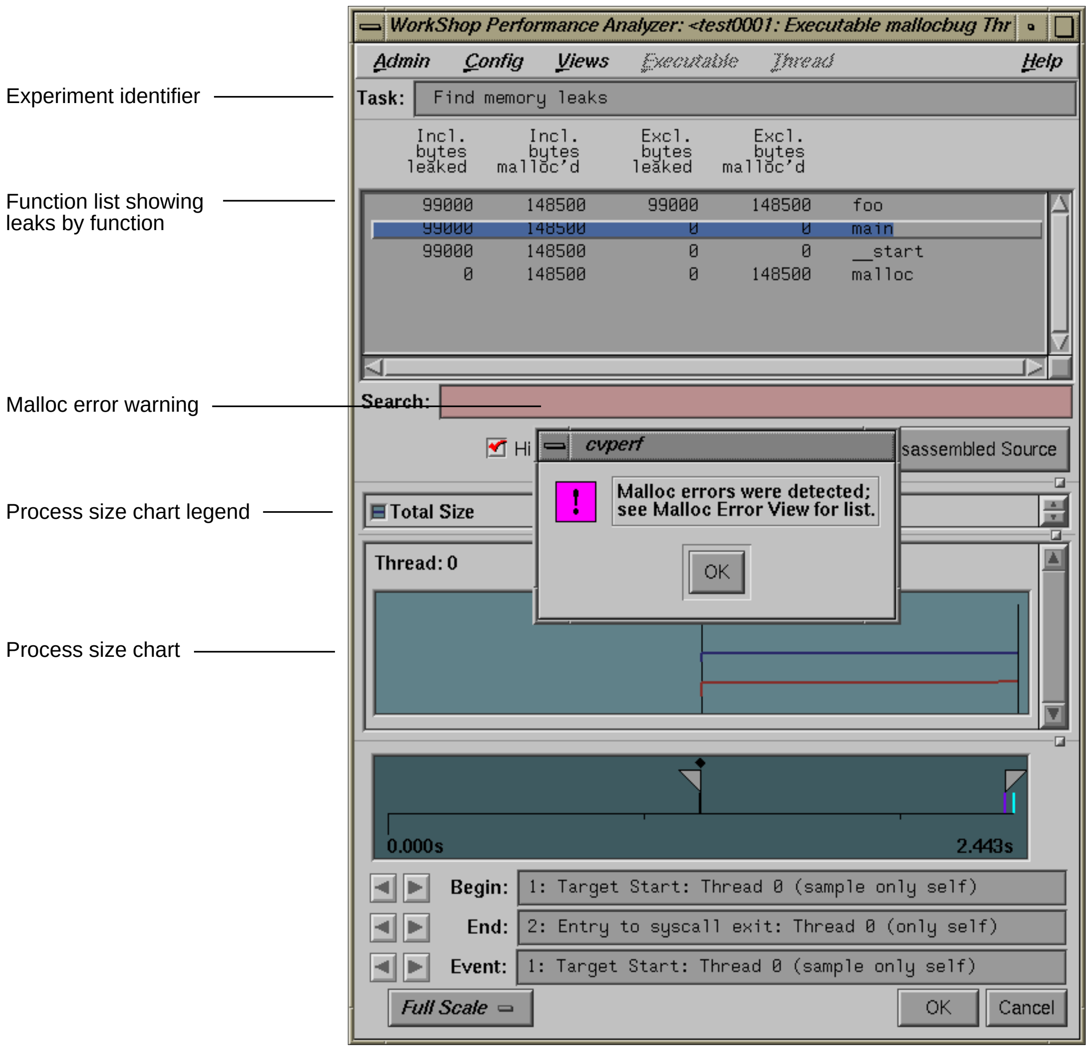
   “Find memory leaks” is a selection on the Performance Task menu in the Performance Panel. It ensures that the appropriate performance data is collected. (For more information, see “Find memory leaks” on page 301).

4. Run the memory leak experiment.

   You run experiments by clicking the *Run* button in the Debugger Main View window.

5. Display the Performance Analyzer.

   The Performance Analyzer displays results appropriate to the task selected, in this case, “Find memory leaks”. Figure 4-25 shows the Performance Analyzer window after a memory experiment (after resizing). Note the dialog box that appears when memory problems are found.



Task: Find memory leaks

```
 Incl.     Incl.      Excl.     Excl.
 bytes     bytes      bytes     bytes
 leaked    malloc'd   leaked    malloc'd
  99000    148500      99000    148500    foo
  99000    148500          0         0    main
  99000    148500          0         0    __start
      0    148500          0    148500    malloc
```

Search:

Malloc errors were detected;
see Malloc Error View for list.

Begin: 1: Target Start: Thread 0 (sample only self)

End: 2: Entry to syscall exit: Thread 0 (only self)

Event: 1: Target Start: Thread 0 (sample only self)

Figure 4-25 Performance Analyzer Displaying Results of a Memory Experiment

Notice that the Function List displays inclusive and exclusive bytes leaked and *malloc*ed per function. Double-clicking a function brings up Source View displaying the function's source code annotated with bytes leaked and *malloc*ed. (To pinpoint the location of a memory problem more exactly, however, it is better to use Malloc Error View or

Leak View and bring up Source View pointing to the exact location of the problem.) You can set other annotations in Source View and the Function List by choosing "Preferences..." from the Config menu in the Performance Analyzer and selecting the desired items.

The total process size chart displays in the usage chart portion of the window. Leakage and other memory problems cause a process's size to increase over time.

6. 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. To look at the source code annotated with memory problems, bring up Source View from one of the other three tools.

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 4-26), Leak View (see Figure 4-27), or Malloc View (see Figure 4-28). Malloc View is the most general showing all memory 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).

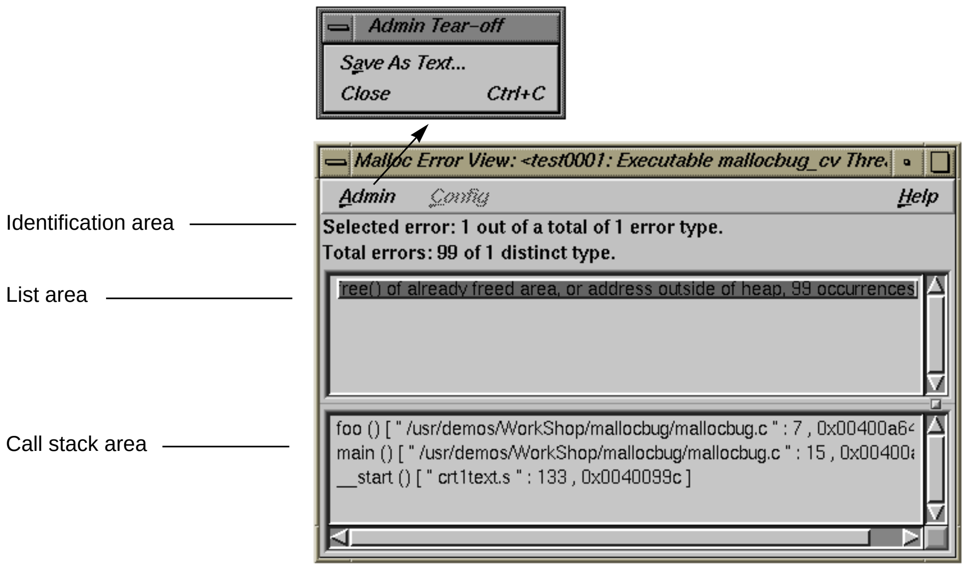
Each of these views has three major areas:

- identification area—This indicates which operation has been selected from the list. Malloc View identifies *mallocs*, indicating the number of *malloc* locations and the size of all *malloc* operations in bytes. Malloc Error View identifies leaks and bad *free*s, 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. You can double-click a frame in the call stack to see the source code that caused the leak. Figure 4-29 shows a typical Source View window with leak annotations (you can change the annotations through the “Preferences...” selection in the Performance Analyzer Config menu). Notice that high counts display tagged.

Note: As an alternative to viewing leaks in Leak View, you can select one or more memory operations, choose “Save As Text...” from the Admin menu, 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.

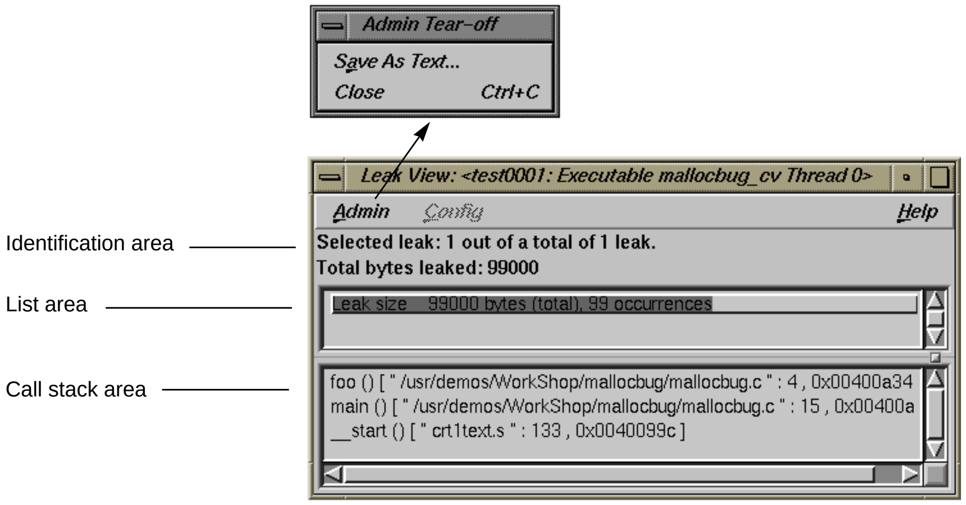


Selected error: 1 out of a total of 1 error type.
Total errors: 99 of 1 distinct type.

ree() of already freed area, or address outside of heap, 99 occurrences

```
foo () [ " /usr/demos/WorkShop/mallocbug/mallocbug.c " : 7 , 0x00400a6
main () [ " /usr/demos/WorkShop/mallocbug/mallocbug.c " : 15 , 0x00400
__start () [ " crt1text.s " : 133 , 0x0040099c ]
```

Figure 4-26 Malloc Error View Window with Admin Menu

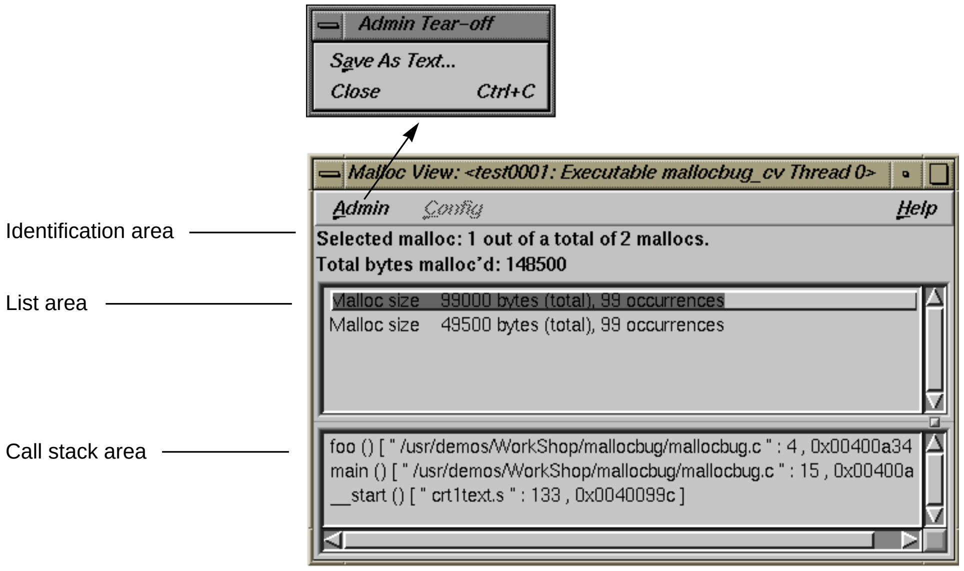


Selected leak: 1 out of a total of 1 leak.
Total bytes leaked: 99000

```
Leak size    99000 bytes (total), 99 occurrences
```

```
foo () [ " /usr/demos/WorkShop/mallocbug/mallocbug.c " : 4 , 0x00400a34
main () [ " /usr/demos/WorkShop/mallocbug/mallocbug.c " : 15 , 0x00400a
__start () [ " crt1text.s " : 133 , 0x0040099c ]
```

Figure 4-27 Leak View Window with Admin Menu

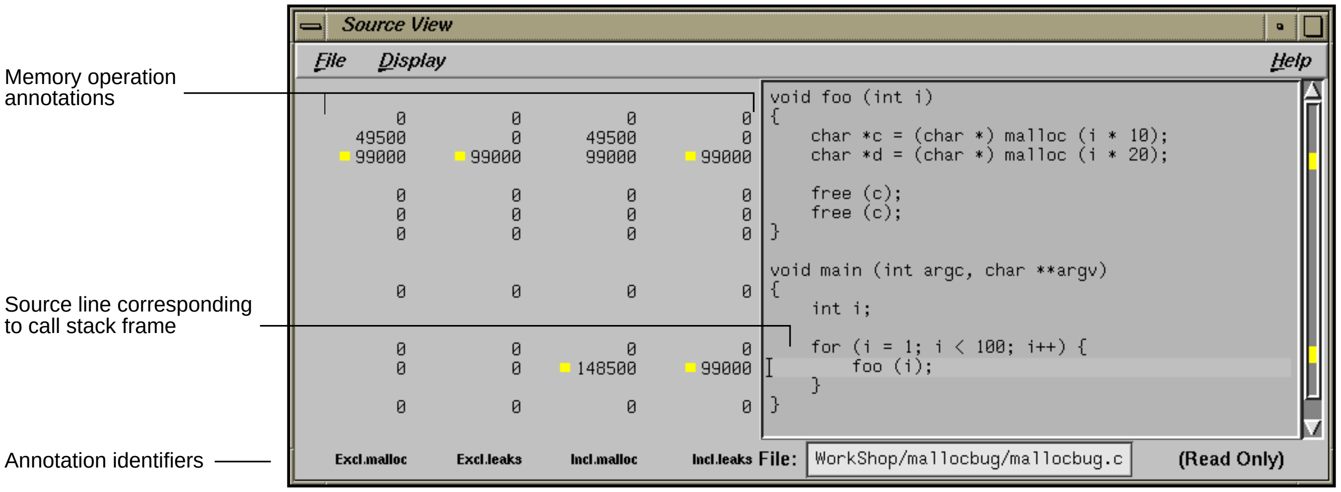


Selected malloc: 1 out of a total of 2 mallocs.
Total bytes malloc'd: 148500

```
Malloc size    99000 bytes (total), 99 occurrences
Malloc size    49500 bytes (total), 99 occurrences
```

```
foo () [ " /usr/demos/WorkShop/mallocbug/mallocbug.c " : 4 , 0x00400a34
main () [ " /usr/demos/WorkShop/mallocbug/mallocbug.c " : 15 , 0x00400a
__start () [ " crt1text.s " : 133 , 0x0040099c ]
```

Figure 4-28 Malloc View Window with Admin Menu



```
       0          0          0          0  void foo (int i)
   49500          0      49500          0  {
  ■99000     ■99000      99000     ■99000      char *c = (char *) malloc (i * 10);
                                               char *d = (char *) malloc (i * 20);
       0          0          0          0
       0          0          0          0      free (c);
       0          0          0          0      free (c);
                                           }

       0          0          0          0  void main (int argc, char **argv)
                                           {
                                               int i;

       0          0          0          0      for (i = 1; i < 100; i++) {
       0          0    ■148500     ■99000          foo (i);
                                               }
       0          0          0          0  }
```

Figure 4-29 Source View with Memory Analysis Annotations

Analyzing the Memory Map with Heap View

Heap View lets you analyze data from experiments based on the “Find Memory Leaks” 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 these memory block conditions:

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

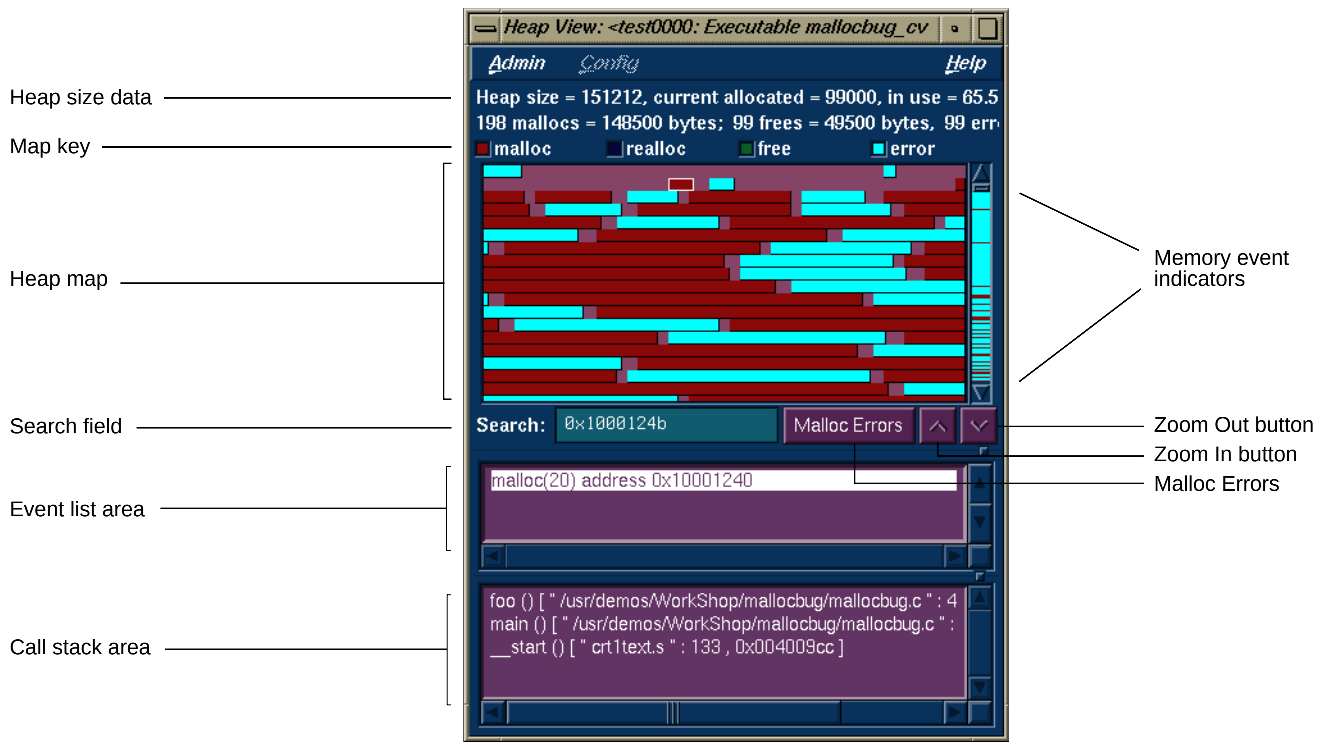
In addition to the Heap View memory map, you can analyze memory leak data using these other tools:

- 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 function list in the Performance Analyzer main window shows inclusive *mallocs* and *frees* with bytes used by function for memory leak experiments.
- The Source View window shows inclusive *mallocs* and *frees* and the number of bytes used by source line.

Heap View Window

A typical Heap View window with its parts labeled appears in Figure 4-30.



```
Heap size = 151212, current allocated = 99000, in use = 65.5
198 mallocs = 148500 bytes;  99 frees = 49500 bytes,  99 err
```

```
Search: 0x1000124b
```

```
malloc(20) address 0x10001240
```

```
foo () [ " /usr/demos/WorkShop/mallocbug/mallocbug.c " : 4
main () [ " /usr/demos/WorkShop/mallocbug/mallocbug.c " :
__start () [ " crt1text.s " : 133 , 0x004009cc ]
```

Figure 4-30 Heap View Window

The major features of a Heap View window are:

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 View map area
: 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, and its size in the event list area and the associated call stack in the call stack display area.

  Note in Figure 4-30 that there are only a few problems in the memory at the lower addresses and many more at the higher addresses.

Memory event indicators
: 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 was no problem, the address at the beginning of the memory block and its size display.

  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 call stack 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 a list of 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, press **`<Enter>`** to see the error's *malloc* information and its associated call stack.

Zoom in button
: (the upward-pointing arrow) redisplays the heap area at twice the current size of the display. If you reach the limit, an error message displays.

Zoom out button
: (the downward-pointing arrow) 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.

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, which is highlighted and indicated by a caret (^) with the number of bytes used by *malloc* in the annotation column. See Figure 4-29.

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 4-31). This lets you select the file name for saving the Heap View data. The default file name suggested is *<experiment-directory>.out*. When you click *OK*, the data for the current caliper setting and the list of unmatched *frees*, if any, is 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 filename default is provided, and the file that you name will be overwritten.

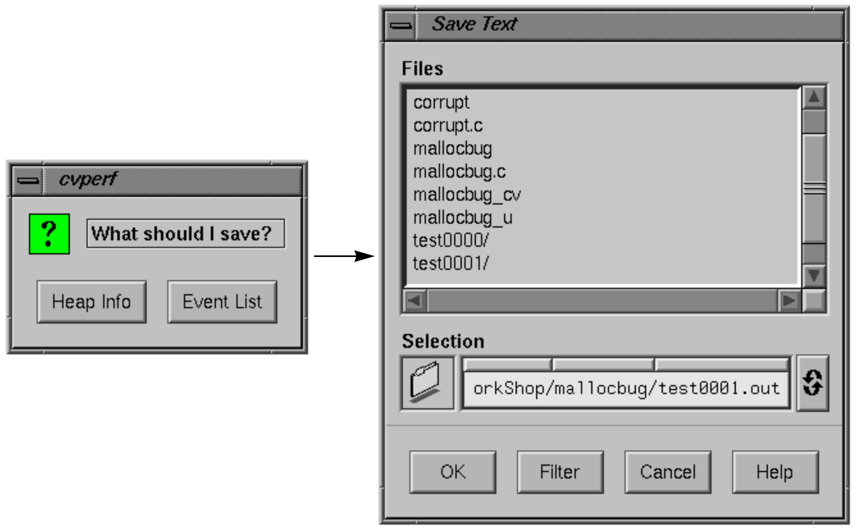


Figure 4-31 Heap View Save Text Dialog Boxes

Memory Experiment Tutorial

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

1. Go to the */usr/demos/WorkShop/mallocbug* directory. Note the executable *mallocbug_cv*. This was compiled as follows:

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

2. Invoke the Debugger by typing

   ```
   cvd mallocbug_cv
   ```

3. Bring up the Performance Panel by selecting “Performance Task...” from the Admin menu in Main View.
4. Select “Find memory leaks” from the Task menu and click the *OK* button. Then click *Run* to begin the experiment.

   The program runs quickly and terminates.

5. Select “Performance Analyzer” from the “Launch Tool” submenu in the Debugger Admin menu.

   The Performance Analyzer window appears. 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 at 1,000 bytes each occurrence.

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

   Source View displays showing the function’s code, annotated by the exclusive and inclusive leaks.

10. Select “Heap View...” from the Performance Analyzer Views menu.

    The Heap View window displays. The heap size and percentage used is shown 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 bad *free*. Notice also that color-coded indicators showing *mallocs*, *reallocs*, and bad *frees* 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 control (upwards pointing arrow) and a zoom-out control (downwards arrow).

    The event display area and the call stack are at the bottom of the window. Clicking any event in the heap 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 field. 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.

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

14. Close the Source View.
15. Click the *Malloc Errors* button.

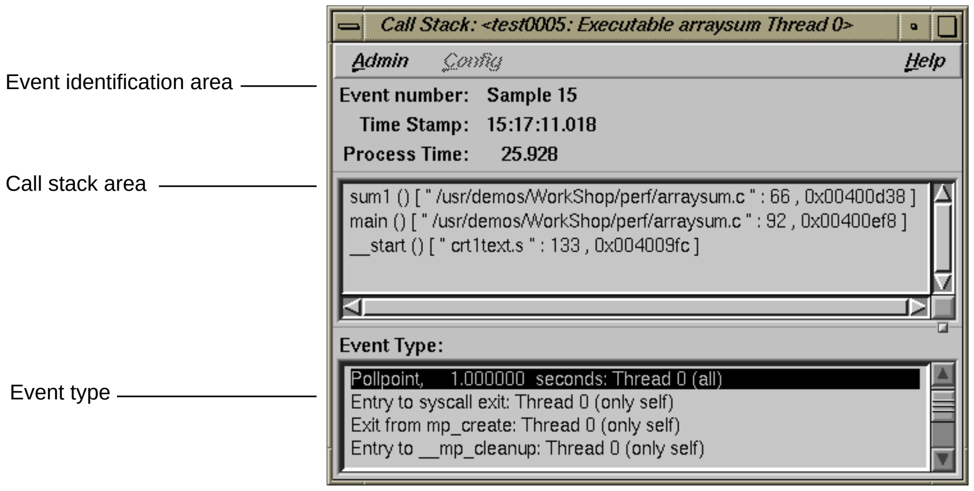
    The data in the Heap View information window changes to display memory problems. Note that a *free* may be unmatched within the analysis interval, yet it may have a corresponding *free* outside of the interval.

16. Click *Close* to leave the Heap View information window.
17. Select "Exit" from the Admin menu in any open window to end the experiment.

This ends the tutorial.

Call Stack

The Call Stack window 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 4-26.


Event identification area
Call stack area
Event type
Call Stack: <test0005: Executable arraysum Thread 0>
Admin
Config
Help
Event Type:

Event number: Sample 15
Time Stamp: 15:17:11.018
Process Time: 25.928

```
sum1 () [ " /usr/demos/WorkShop/perf/arraysum.c " : 66 , 0x00400d38 ]
main () [ " /usr/demos/WorkShop/perf/arraysum.c " : 92 , 0x00400ef8 ]
__start () [ " crt1text.s " : 133 , 0x004009fc ]
```

```
Pollpoint,     1.000000  seconds: Thread 0 (all)
Entry to syscall exit: Thread 0 (only self)
Exit from mp_create: Thread 0 (only self)
Entry to __mp_cleanup: Thread 0 (only self)
```

Figure 4-32 Performance Analyzer Call Stack

There are three main areas in the window:

- The event identification area displays the number of the event, its time stamp, and the time within the experiment. If you have a multi-process experiment, the thread will be indicated here.
- The call stack area displays the contents of the call stack when the sample event took place.
- The event type area highlights the type of event and shows the thread in which it was defined and whether the sample was taken in all threads or the indicated thread only, in parentheses.

Analyzing Working Sets

If you suspect a problem with high page faulting or instruction cache misses, you should conduct 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, then page thrashing, that is, swapping in and out of pages, may result slowing your program down. 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 reference (man) page for *cord*.

Working set analysis is appropriate for:

- any program that runs for a long time
- programs whose operation comes in distinct phases
- distributed shared objects (DSOs) that are shared among several programs

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 (*cvcord*) is separate from the Performance Analyzer and is invoked by typing **`cvcord`** 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, can compute an optimized ordering to reduce working sets.

Figure 4-33 presents an overview of the process of conducting working set analysis.

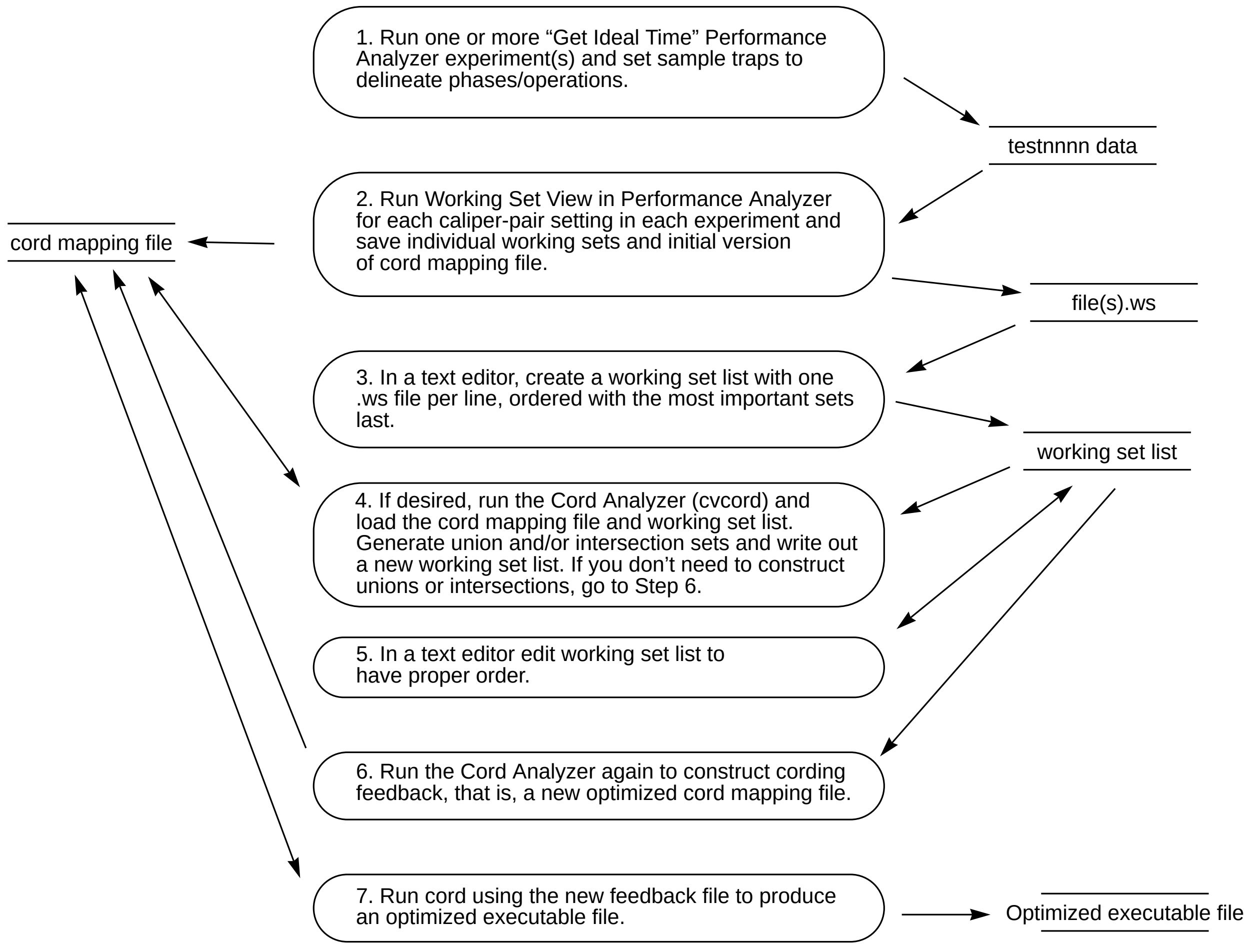

1. Run one or more “Get Ideal Time” Performance Analyzer experiment(s) and set sample traps to delineate phases/operations.
testnnnn data
2. Run Working Set View in Performance Analyzer for each caliper-pair setting in each experiment and save individual working sets and initial version of cord mapping file.
cord mapping file
file(s).ws
3. In a text editor, create a working set list with one .ws file per line, ordered with the most important sets last.
working set list
4. If desired, run the Cord Analyzer (cvcord) and load the cord mapping file and working set list. Generate union and/or intersection sets and write out a new working set list. If you don’t need to construct unions or intersections, go to Step 6.
5. In a text editor edit working set list to have proper order.
6. Run the Cord Analyzer again to construct cording feedback, that is, a new optimized cord mapping file.
7. Run cord using the new feedback file to produce an optimized executable file.
Optimized executable file

Figure 4-33 Working Set Analysis Process

First you conduct one or more Performance Analyzer experiments using the “Get Ideal Time (pixie) per function & source line” task. You need to set sample traps at the beginning and end of each operation or phase that represents a distinct task. If you want, 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, you run the Performance Analyzer and select Working Set View. You need to save the working set for each phase or operation that you wish to improve. Do this by setting the calipers to bracket each phase and selecting “Save Working Set” from the Admin menu.

You also must 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 wish to analyze using the Cord Analyzer. You create the working set list file in a text editor, specifying one line for each working set, in reverse order of priority, that is, the most important comes last.

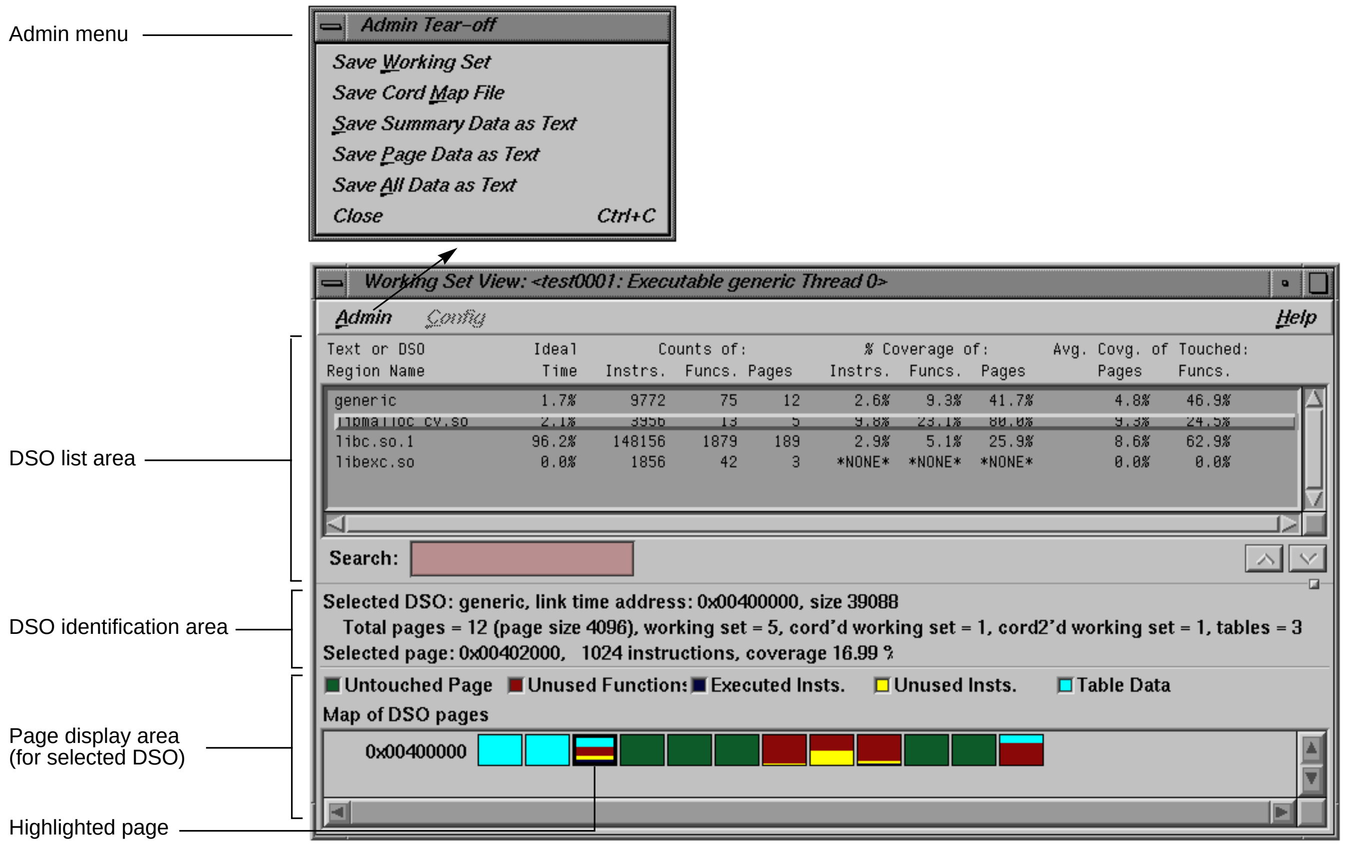
The working set list and the *cord* mapping file serve as input to the Cord Analyzer. The working set list provides the Cord Analyzer with working sets to be improved. The *cord* mapping file provides a list of all the functions in the executable. The Cord Analyzer displays the list of working sets and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

- examine the page layout and 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*.

Working Set View

Working Set View measures the coverage of the dynamic shared objects (DSOs) that make up your executable (see Figure 4-34). 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.



Working Set View: <test0001: Executable generic Thread 0>

```
Text or DSO       Ideal        Counts of:              % Coverage of:         Avg. Covg. of Touched:
Region Name        Time   Instrs.  Funcs. Pages   Instrs.   Funcs.   Pages         Pages     Funcs.
generic            1.7%      9772      75    12      2.6%     9.3%   41.7%          4.8%     46.9%
libmalloc_cv.so    2.1%      3956      13     5      9.8%    23.1%   80.0%          9.3%     24.5%
libc.so.1         96.2%    148156    1879   189      2.9%     5.1%   25.9%          8.6%     62.9%
libexc.so          0.0%      1856      42     3    *NONE*   *NONE*  *NONE*          0.0%      0.0%
```

Selected DSO: generic, link time address: 0x00400000, size 39088
Total pages = 12 (page size 4096), working set = 5, cord'd working set = 1, cord2'd working set = 1, tables = 3
Selected page: 0x00402000, 1024 instructions, coverage 16.99 %

Figure 4-34 Working Set View

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

Counts of: Instrs.
: is the number of instructions contained in the DSO.

Counts of: Funcs.
: is the number of functions contained in the DSO.

Counts of: Pages
: is the number of pages occupied by the DSO.

% Coverage of: Instrs.
: is the percentage obtained by dividing the number of instructions used by the total number of instructions in the DSO.

% Coverage of: Funcs.
: is the percentage obtained by dividing the number of functions used by the total number of functions in the DSO.

% Coverage of: Pages
: is the coverage obtained by dividing the number of pages touched by the total pages in the DSO.

Avg. Covg. of Touched: Pages
: is 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
: is 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.)

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.

Page Display Area

The page display area at the bottom of the window shows all the pages in the DSO and indicates untouched pages, unused functions, executed instructions, unused instructions, and table data (related to *rld*). 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.

Admin Menu

The Admin menu provides these 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.

Cord Analyzer

The Cord Analyzer is not actually part of the Performance Analyzer; it's discussed in this part of the manual because it works in conjunction with 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 *cord* to generate an executable with improved working-set behavior. You invoke the Cord Analyzer using this syntax at the command line:

cvcord -L *executable* **[-fb** *feedbackFile***] [-wsl** *workingsetList***]**
[-ws *workingsetFile***] [-scheme** *schemeName***]**

where

-L *executable*
: specifies a single executable file name as input.

-fb *feedbackFile*
: 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 *workingsetList*
: specifies a single text file name as input; the working set list will consist of the working set files whose names appear in the input file. Each file name should be on a single line.

-ws *workingsetFile*
: specifies a single working set file name.

`-scheme` *schemeName*
: specifies which color scheme should be used for the Cord Analyzer.

The Cord Analyzer is shown in Figure 4-35 with its major areas and menus labeled.

Working Set Display Area

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

Working-set pgs. (util. %)
: shows the number of pages in the working set and the percentage of page space that is utilized.

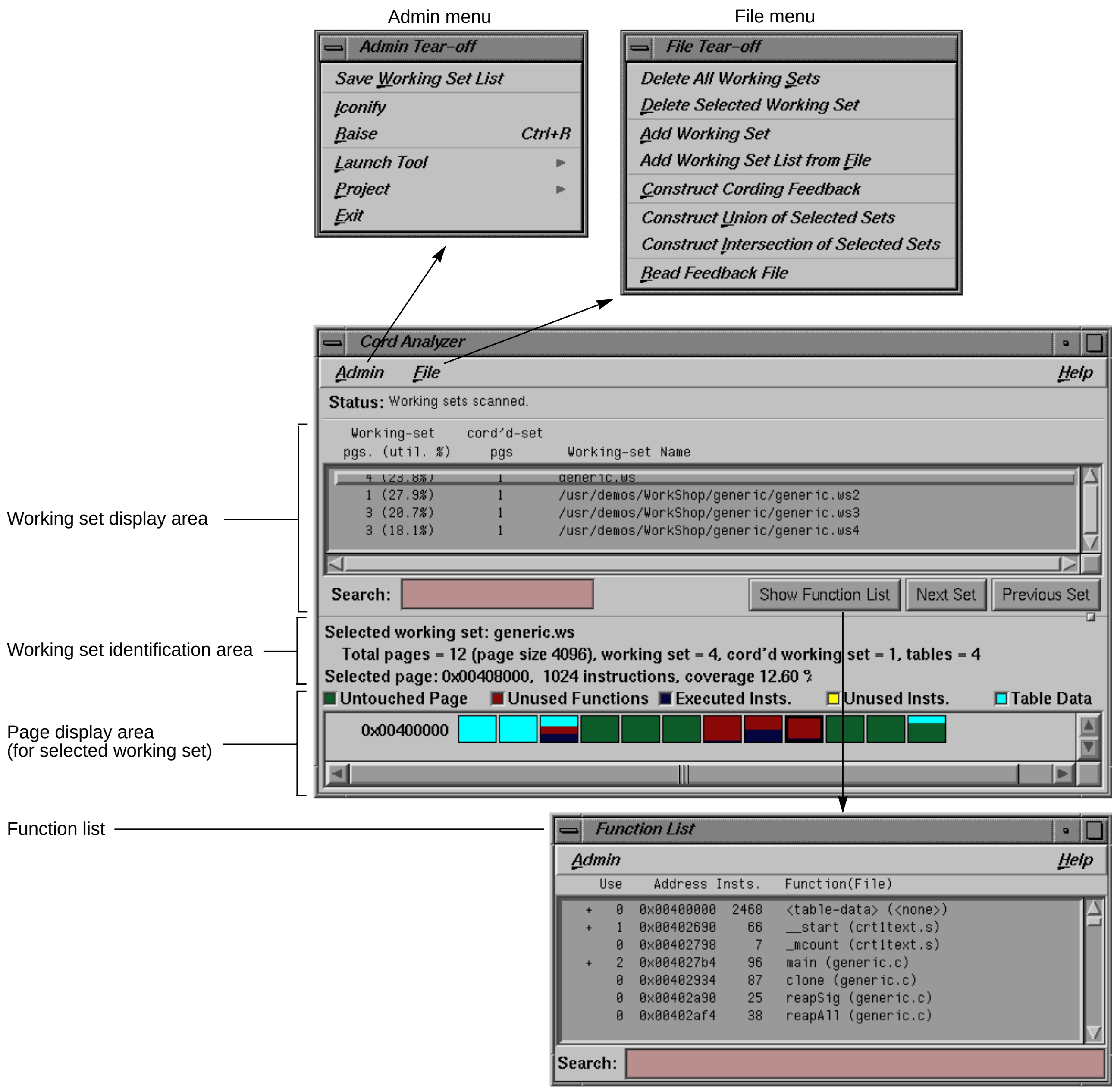
cord'd set pgs
: is the minimum number of pages for this set, that is, the number of pages the working set would occupy if the program or DSO were *cord*'d optimally for that specific working set.

Working-set Name
: identifies the path for the working set.

Note also 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.

Working Set Identification Area

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



Status: Working sets scanned.

```
 Working-set    cord'd-set
pgs. (util. %)     pgs        Working-set Name
   4 (23.8%)        1        generic.ws
   1 (27.9%)        1        /usr/demos/WorkShop/generic/generic.ws2
   3 (20.7%)        1        /usr/demos/WorkShop/generic/generic.ws3
   3 (18.1%)        1        /usr/demos/WorkShop/generic/generic.ws4
```

Selected working set: generic.ws
Total pages = 12 (page size 4096), working set = 4, cord'd working set = 1, tables = 4
Selected page: 0x00408000, 1024 instructions, coverage 12.60 %

```
   Use   Address Insts.   Function(File)
 +  0  0x00400000  2468   <table-data> (<none>)
 +  1  0x00402690    66   __start (crt1text.s)
    0  0x00402798     7   _mcount (crt1text.s)
 +  2  0x004027b4    96   main (generic.c)
    0  0x00402934    87   clone (generic.c)
    0  0x00402a90    25   reapSig (generic.c)
    0  0x00402af4    38   reapAll (generic.c)
```

Figure 4-35 The Cord Analyzer

Page Display Area

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

Function List

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

Use
: is a count of the working sets containing the function.

Address
: is the starting address for the function.

Insts.
: shows the number of instructions in the function.

Function (File)
: identifies the function and the file in which it occurs.

Note also that when the Function List 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 function in the Function List.

Admin Menu

The Admin menu contains the standard Admin menu commands in WorkShop views (see "Admin Menu" on page 161 in the *ProDev WorkShop Debugger User's Guide*). It has one command specific to the Cord Analyzer:

"Save Working Set List"
: lets you save a new working set list with whatever changes you made to it in the session.

File Menu

The File menu contains these commands:

“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. It asks you if you want the file deleted as well.

“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 Cording Feedback”
: builds a cord mapping file that you can supply as input to the *cord* utility.

“Construct Union of Selected Sets”
: lets you see 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”
: lets you see 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”
: lets you load a new cord mapping file into the Cord Analyzer.

Part II
Tester

Chapter 5

Using Tester

This chapter provides an overview of Tester, which is used for coverage analysis, and provides a model of the different ways in which Tester can be applied.

Chapter 5

Using Tester

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

- "Tester Overview"
- "Usage Model"

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.

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.

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).

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.

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.

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.

Usage Model

This section is divided into three parts:

- “Single Test Analysis Process” shows the general steps in conducting a test.
- “Automated Testing” discusses using scripts to automate your testing.
- “Additional Coverage Testing” describes strategies using multiple tests.

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 5-1. An instrumentation directory is created (*.../ver##<n>*). It contains the instrumented executable and other files used in instrumentation.

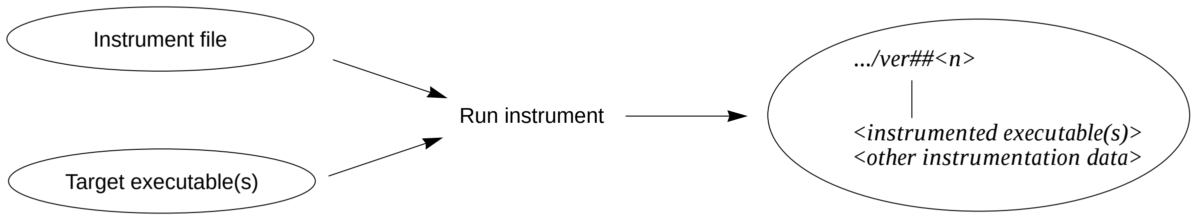


Figure 5-1 Instrumentation Process

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 5-2.

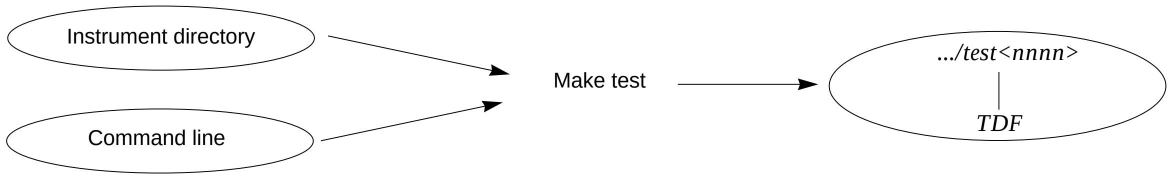


Figure 5-2 Make Test Process

   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 *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 5-3. The commands to run a test use the most recent instrumentation directory version unless you specify a different directory.

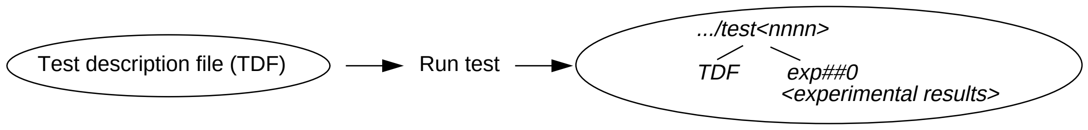


Figure 5-3 Run Test Process

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 5-1 shows query commands for a single test that are available either from the command line or the graphical user interface Queries menu.

Table 5-1 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 5-4.

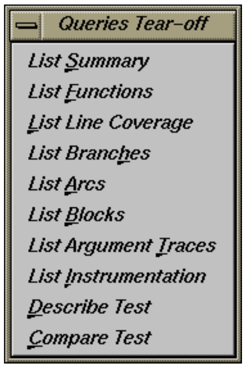


Figure 5-4 The Queries Menu from Main Tester Window

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 5-1, a script that automates a test program called *target* with different arguments:

Example 5-1 Making Tests and Then 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 5-2. Of course, you can also rewrite the whole test machinery as described in Example 5-1.

Example 5-2 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.

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 5-5.

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*.

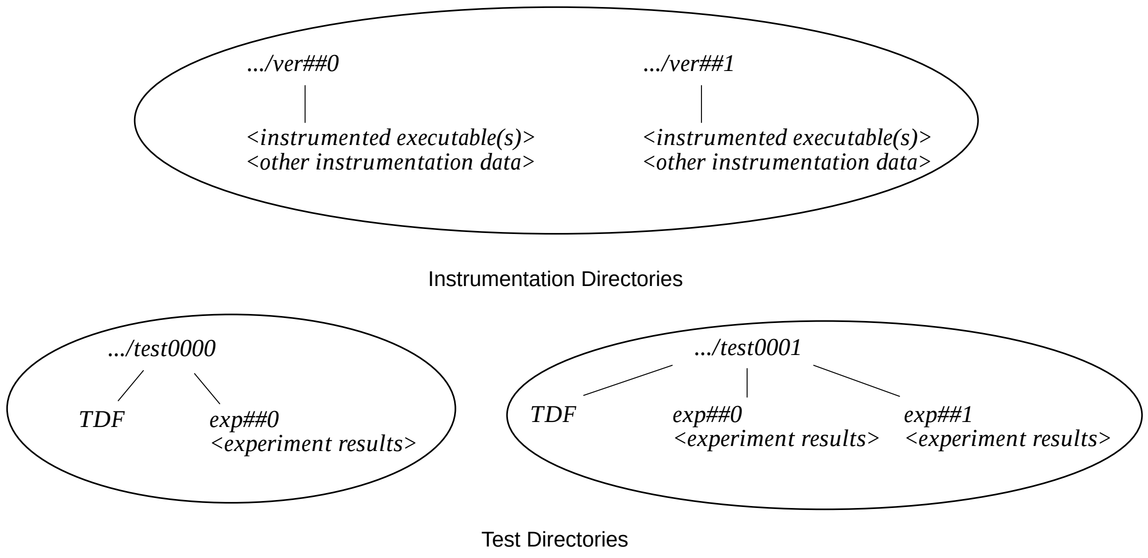


Figure 5-5 Typical Coverage Testing Hierarchy

Chapter 6

Tester Command Line Interface Tutorial

This chapter provides a tutorial which demonstrates how the command line version of Tester can be applied.

Chapter 6

Tester Command Line Interface Tutorial

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:

- "Setting Up the Tutorials" shows you how to run the script that creates the files needed for the tutorials.
- "Tutorial #1 - Analyzing a Single Test" takes you through the steps of performing coverage analysis for a single test.
- "Tutorial #2 - Analyzing a Test Set" discusses creating additional tests to achieve full coverage.
- "Tutorial #3 - Optimizing a Test Set" explains how to fine-tune a test set to eliminate redundant tests.
- "Tutorial #4 - Analyzing a Test Group" 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'd rather have the test data built automatically, run the script:

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

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

```
% cvcov help < commandname >
```

Setting Up the Tutorials

1. Enter the following 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*.

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

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.

   Making a Test

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

3. Enter the following to make a test:

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

   You will see the 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*.

4. 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
```

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*).

5. Enter the following command:

```
% cvcov runtest test0000
```

You will see the 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.

Analyzing Test Coverage Data

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

6. Enter the following to get the summary:

```
% cvcov lssum test0000
```

You will see the display shown in Example 6-1.

Example 6-1 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're showing highlighting or user input in boldface.

7. 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 6-2.

Example 6-2 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.

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 6-2, 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 6-3 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 6-3 *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 6-4 shows the portion of the script in which the test set is created and the individual tests are added to the test set.

Example 6-4 *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 6-5. 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 6-5 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 6-6. 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 6-6 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 6-7 shows the results of the tests in the new test set with *lssum*.

Example 6-7 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 6-8.

Example 6-8 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.

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 wish 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. Let's 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 6-9 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 6-9 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 6-10 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 6-10 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'll 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 6-11. 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 6-11 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
```

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 6-12 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.

   mktgroup is the only command that we haven't used previously in the tutorials. *mktgroup* creates the test group. As a final part of the preparation, the script performs a *cattest* to show the contents of the test group.

Example 6-12 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 6-13.

Example 6-13 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've used in the 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 covered, 3 times by *printtest* (Test [0]) and twice by *copyn* (Test [1]). On the other hand, checking *sscanf* coverage shows that it was covered 3 times by Test [0] but not at all by Test [1].

Chapter 7

Tester Command Line Reference

This chapter describes in detail each of the commands available in the Tester command line interface.

Chapter 7

Tester Command Line Reference

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

- "Common cvcov Options" — the command arguments that are common to more than one command
- "cvcov Command Syntax and Description" — 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
```

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]`

is a mnemonic for recursion. It 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.

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 "Common cvcov Options" on page 174.

The most general command is the help command.

`cvcov help command_name`
: 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

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 7-1 shows the ASCII display for a single test.

Example 7-1 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 7-2 shows the ASCII report for a test set without recursion.

Example 7-2 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 7-3 shows the ASCII report for a nested test set.

Example 7-3 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 7-4.

Example 7-4 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 7-5 shows an example of *mktest*, followed by *cattest* to display the contents of the Test Description File (*TDF*).

Example 7-5 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 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.

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 7-6 shows a typical *lssum* report.

Example 7-6 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 7-7 shows a typical *lsfun* ASCII report.

Example 7-7 lsfun Example

```
% cvcov lsfun -pretty -sort function test0000
Functions    Files       Counts
-------------------------------------
copy_file    copyn.c     1
main         copyn.c     1
```

Note: C++ in-line 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 7-8 shows a typical *lsblock* ASCII report.

Example 7-8 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 7-9 shows a typical branch coverage ASCII report. Note that branches with incomplete or null coverage are highlighted (**boldfaced**).

Example 7-9 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 "Common cvcov Options" on page 174.

Example 7-10 shows a typical *lsarc* ASCII report.

Example 7-10 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 7-11 shows a typical *lscall* ASCII report. `N/A` means the node is excluded.

Example 7-11 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 7-11.

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 7-12.

Example 7-12 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 7-13 shows a segment of a typical *lssource* ASCII report.

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

Example 7-13 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 7-14 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 7-14 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 7-15 shows an example of the *diff* command applied to two tests (although you should make sure that the comparison is relevant). Example 7-16 shows *diff* applied to different instrumentations of the same test.

Example 7-15 *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 7-16 *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%)
```

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: Don't 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 7-17 demonstrates how test sets are optimized. In this case, optimizing is applied to all tests matching the expression *test00**.

Example 7-17 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%
```

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

Note: Don't use UNIX commands *mv* and *cp* to rename or copy test groups because they are constructed with absolute files paths.

Chapter 8

Tester Graphical User Interface Tutorial

This chapter provides a tutorial which demonstrates how the graphical user interface version of Tester can be applied.

Chapter 8

Tester Graphical User Interface Tutorial

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

- "Setting Up the Tutorial"
- "Tutorial #1 — Analyzing a Single Test"
- "Tutorial #2 — Analyzing a Test Set"
- "Tutorial #3 — Exploring the Graphical User Interface"

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'd like the test data built automatically, run the 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:

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

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

Tutorial #1 — Analyzing a Single Test

Tutorial #1 discusses the following topics:

- "Invoking the Graphical User Interface"
- "Invoking the Graphical User Interface"
- "Invoking the Graphical User Interface"
- "Invoking the Graphical User Interface"
- "Invoking the Graphical User Interface"

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 8-1 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 CASEVision 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 CASEVision 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'll find an iconized version of Execution View labeled *cvxcovExec*. It is a shell window for viewing test results as they would appear on the command line.

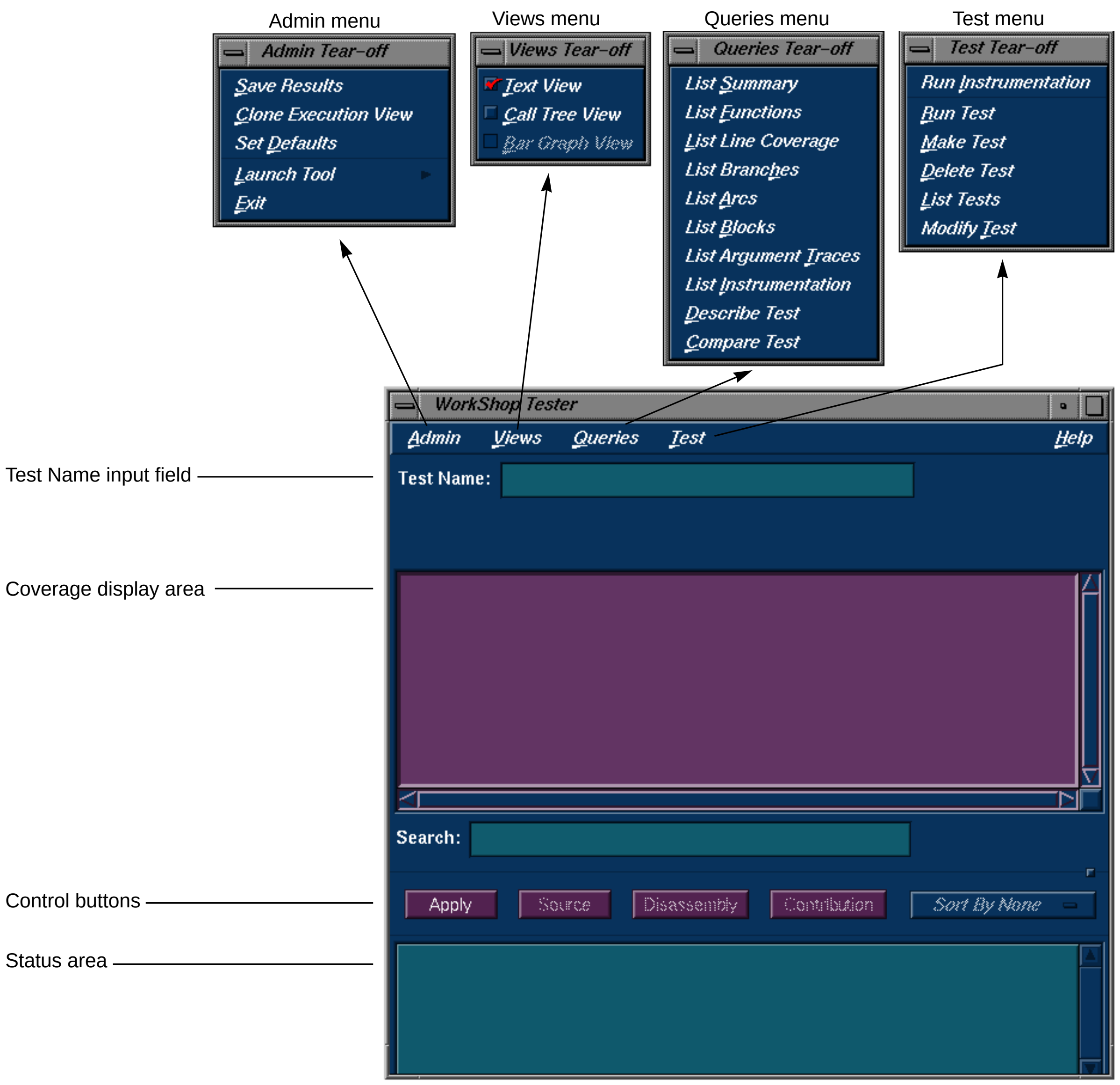


Figure 8-1 Main Tester Window

Instrumenting an Executable

The first step in providing test coverage is to define the instrumentation criteria in an instrumentation file.

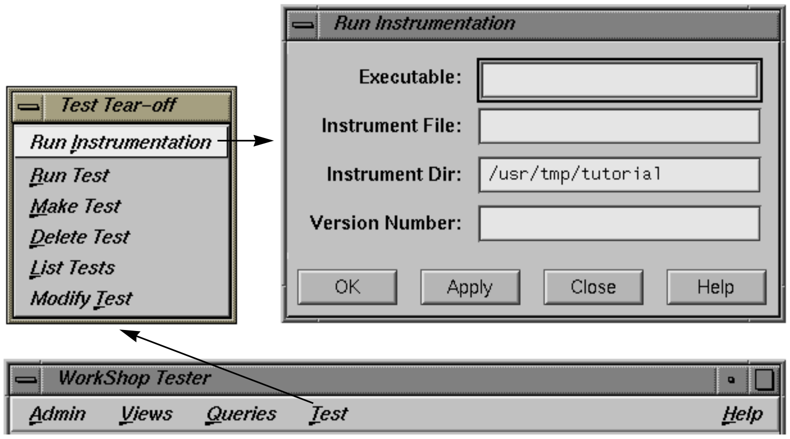
3. On the command line or from Execution View, enter the following to see the instrumentation directives in the file *tut_instr_file* used in the tutorials:

```
% 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 8-2 displays when “Run Instrumentation” is selected from the Test menu.



Instrument Dir: /usr/tmp/tutorial

Figure 8-2 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.

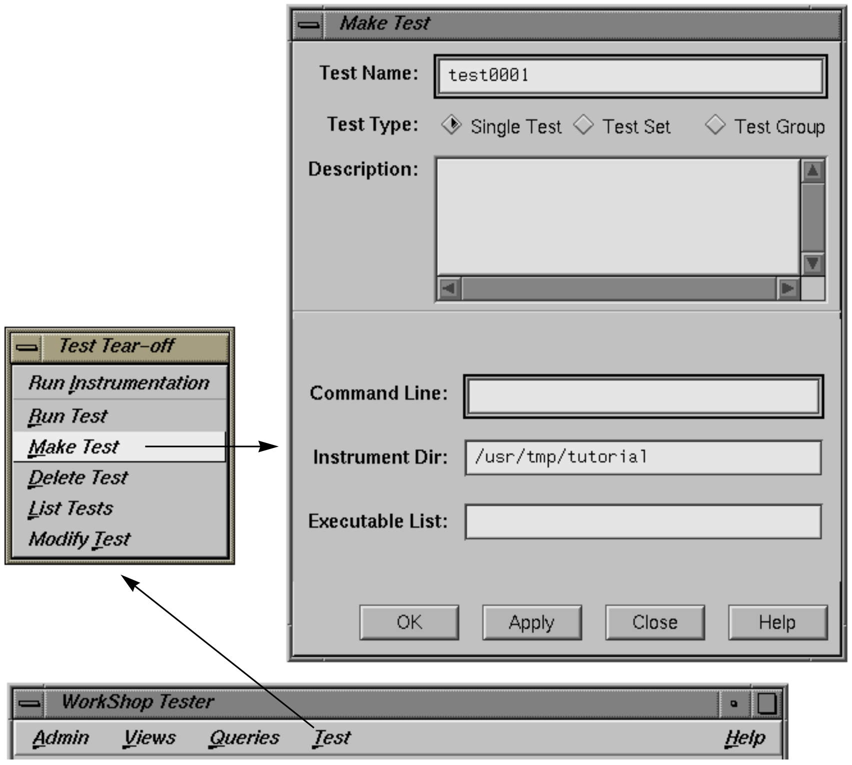
   Making a Test

   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 8-3 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.



Test Name: test0001

Instrument Dir: /usr/tmp/tutorial

Figure 8-3 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 8-4 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 don't wish 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.

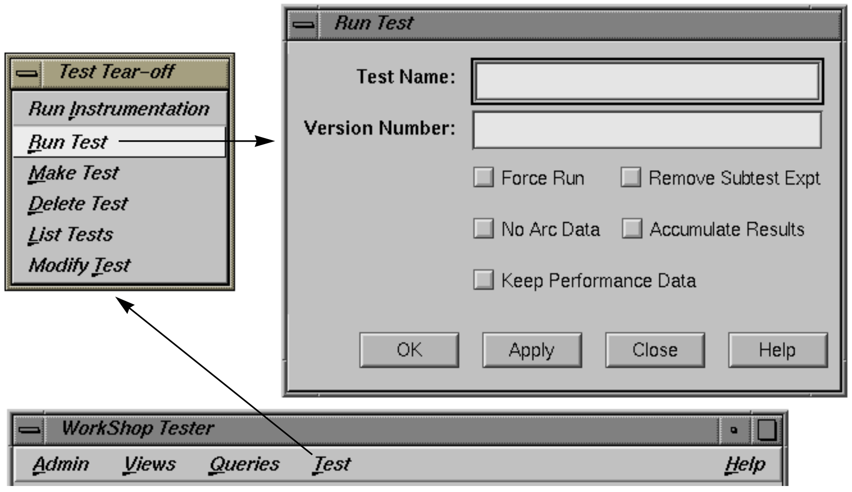


Figure 8-4 "Run Test" Dialog Box

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

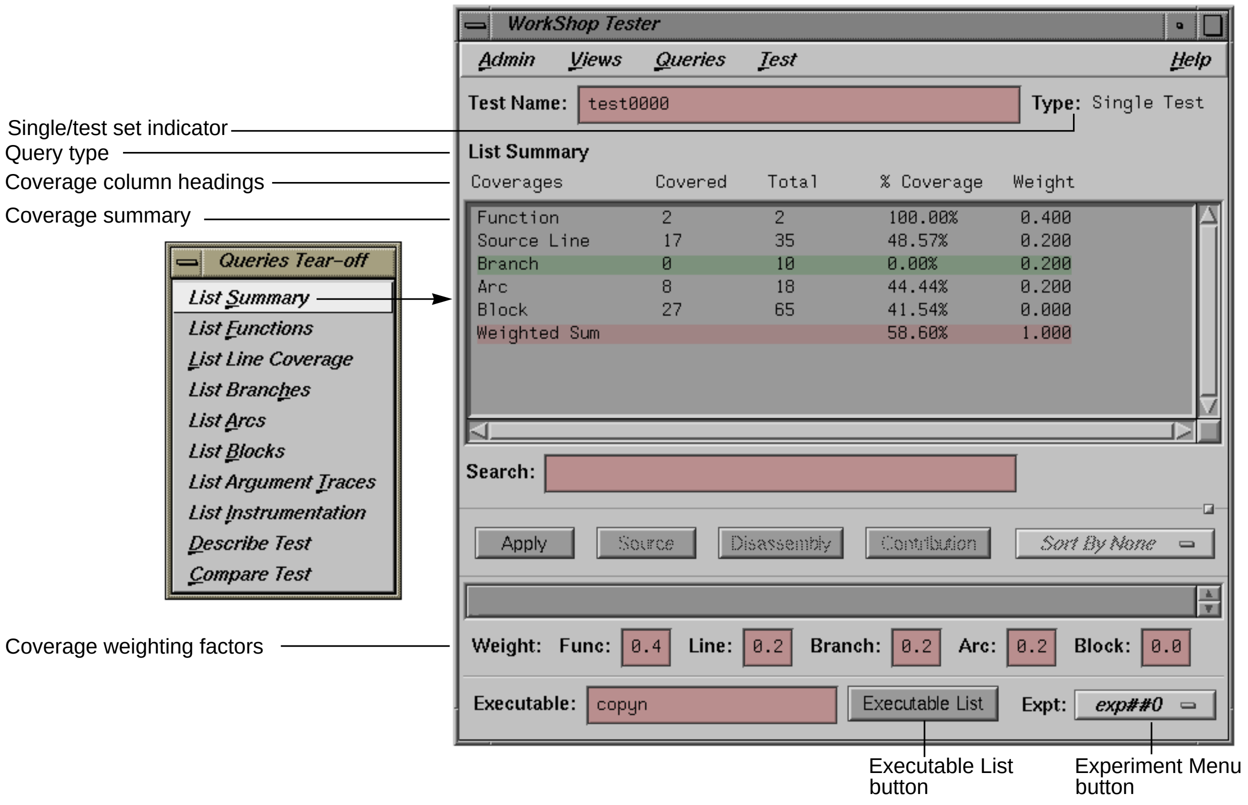
    You can analyze test coverage data in many ways. In this tutorial, we will illustrate a simple top-down approach. We will start at the top to get a summary of overall coverage, proceed to the function level, and finally go to the actual source lines.

    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 8-5. 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.



Test Name: test0000 Type: Single Test

List Summary

```
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           27         65       41.54%        0.000
Weighted Sum                        58.60%        1.000
```

Search:

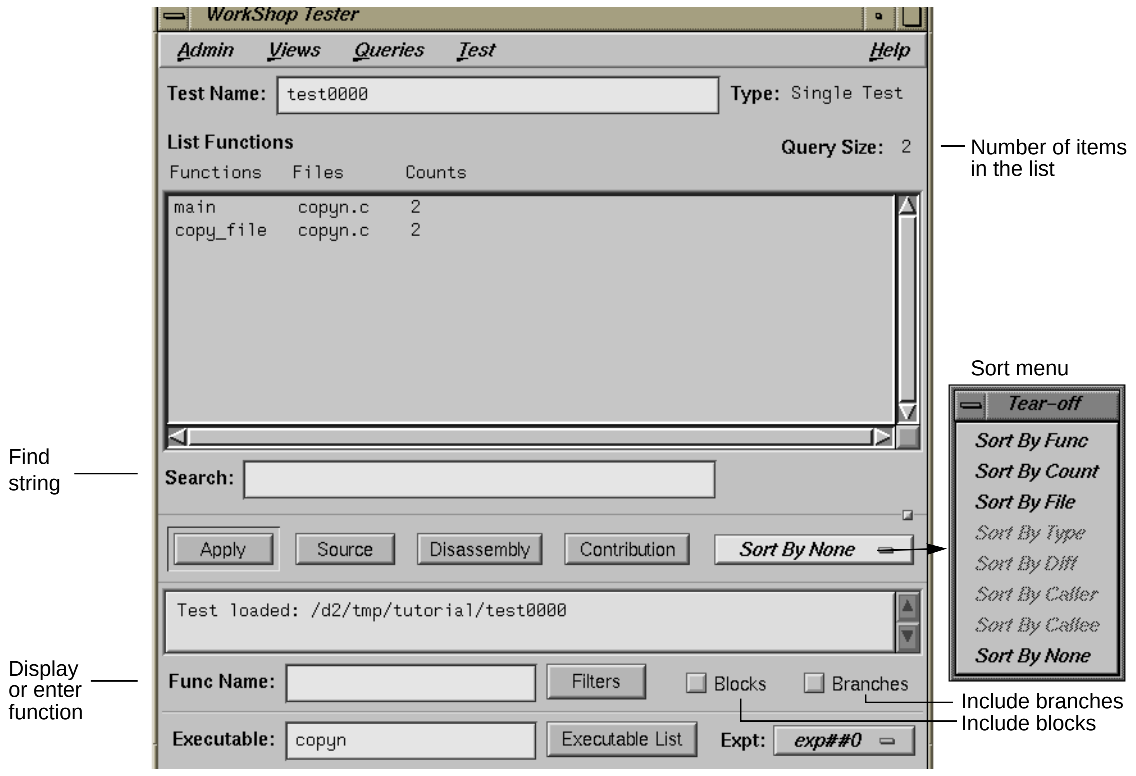
Weight: Func: 0.4 Line: 0.2 Branch: 0.2 Arc: 0.2 Block: 0.0

Executable: copyn Expt: exp##0

Figure 8-5 “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 8-6, with the available options.



Test Name: test0000

Functions Files Counts

```
main        copyn.c    2
copy_file   copyn.c    2
```

```
Test loaded: /d2/tmp/tutorial/test0000
```

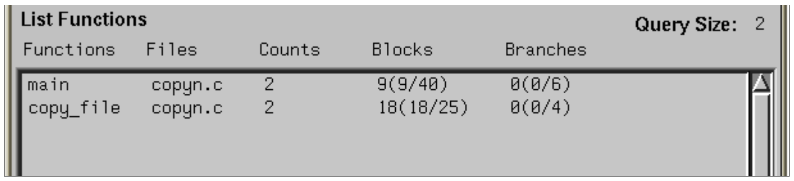
Executable: copyn

Figure 8-6 "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 8-7 shows the display area with *Blocks* and *Branches* enabled.



```
Functions   Files      Counts     Blocks       Branches
main        copyn.c    2          9(9/40)      0(0/6)
copy_file   copyn.c    2          18(18/25)    0(0/4)
```

Figure 8-7 “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’ll arrive at the percentage of coverage.

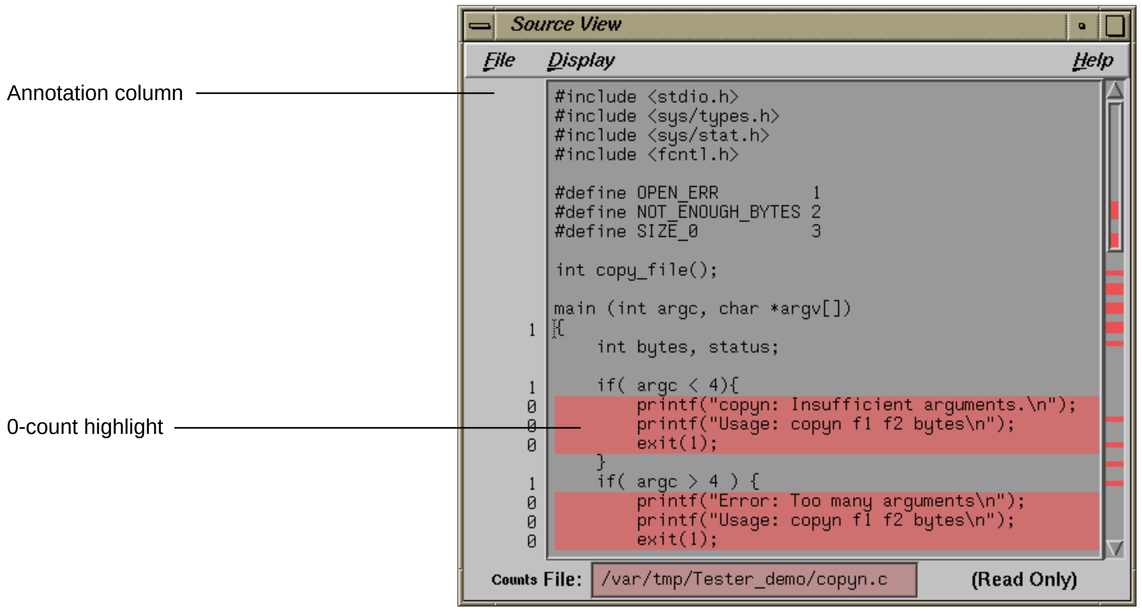
Similarly, the *Branches* column shows the number of branches covered, followed by the number covered out of the total possible branches. The term *covered* means that the branch has been executed under both true and false conditions.

20. Select the function *main* in the display area and click *Source*.

    The Source View window displays with count annotations as shown in Figure 8-8. 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.

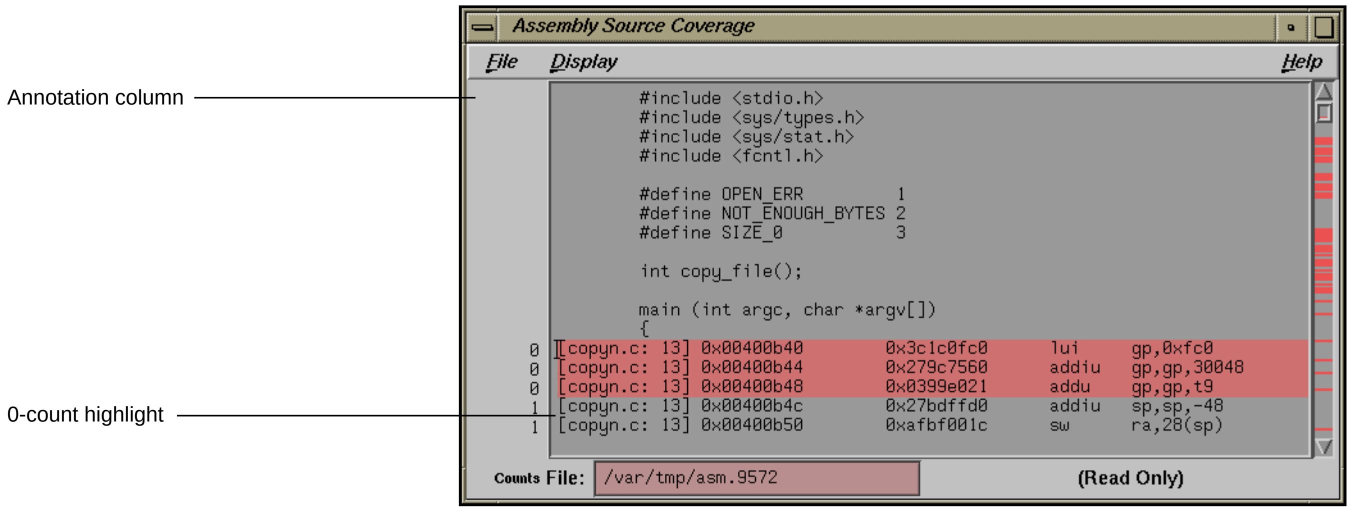
    The Disassembly View window displays with count annotations as shown in Figure 8-9. Lines with 0 counts are highlighted in the display area and in the vertical scroll bar area.



```
     #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);
```

Counts File: /var/tmp/Tester_demo/copyn.c (Read Only)

Figure 8-8 Source View with Count Annotations



```
             #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[])
             {
  0  [copyn.c: 13] 0x00400b40       0x3c1c0fc0       lui      gp,0xfc0
  0  [copyn.c: 13] 0x00400b44       0x279c7560       addiu    gp,gp,30048
  0  [copyn.c: 13] 0x00400b48       0x0399e021       addu     gp,gp,t9
  1  [copyn.c: 13] 0x00400b4c       0x27bdffd0       addiu    sp,sp,-48
  1  [copyn.c: 13] 0x00400b50       0xafbf001c       sw       ra,28(sp)
```

Counts File: /var/tmp/asm.9572 (Read Only)

Figure 8-9 Disassembly View with Count Annotations

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.

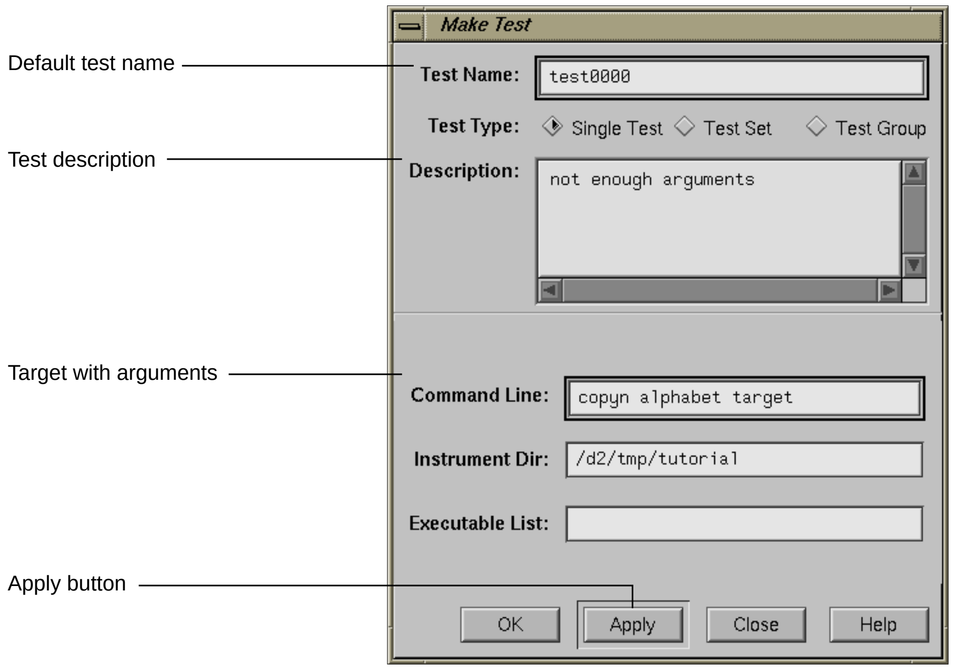
   This displays the Make Test dialog box. It is easy to enter a series of tests. As is standard in CASEVision, using the *Apply* button in the dialog box instead of the *OK* button completes the task without closing the dialog box. The *Test Name* field supplies an incremented default test name after each 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 doesn't 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 8-10 shows the features of the dialog box you'll 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 **`<return>`**) 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*.



Test Name: test0000

Test Type: [✓ Single Test] Test Set Test Group

Description: not enough arguments

Command Line: copyn alphabet target

Instrument Dir: /d2/tmp/tutorial

Executable List:

Figure 8-10 ”Make Test” Dialog Box with Features Used in Tutorial

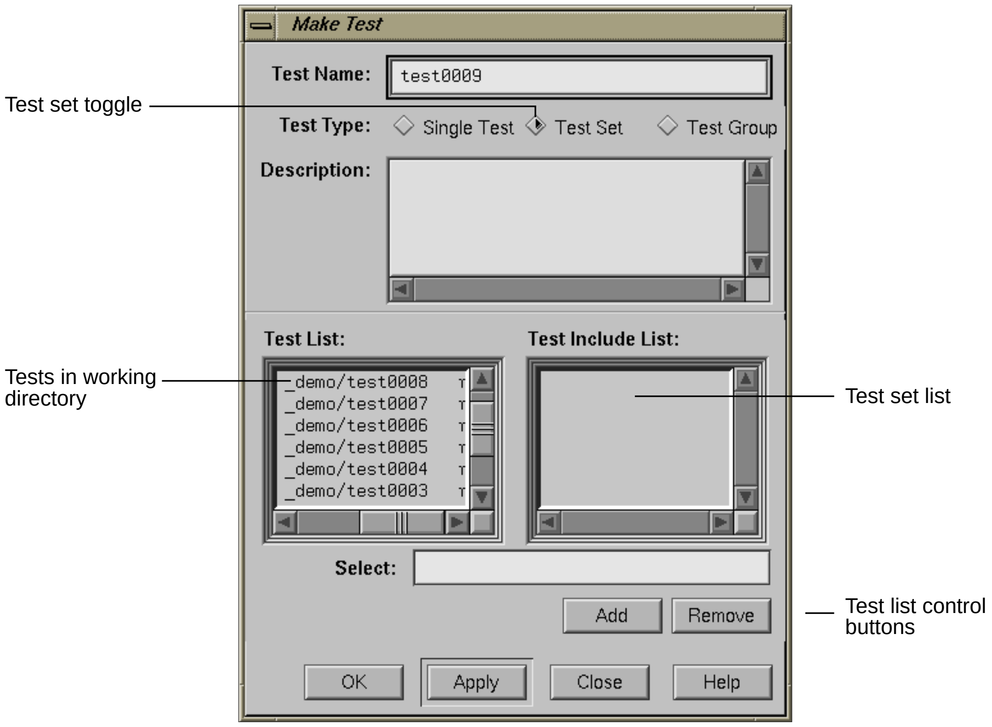
4. Enter **`copyn no_file target 20`** in the *Command Line* field, **`cannot access file`** in the *Description* field, and click *Apply* to make *test0003*.
5. Enter **`copyn alphabet target 0`** in the *Command Line* field, **`pass bad size arg`** in the *Description* field, and click *Apply* to make *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 8-11.



Test Name: test0009

Test List:
```
_demo/test0008   T
_demo/test0007   T
_demo/test0006   T
_demo/test0005   T
_demo/test0004   T
_demo/test0003   T
```

Figure 8-11 "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).

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 "Tutorial #3 — Optimizing a Test Set" on page 467. To learn more about test groups, see "Tutorial #4 — Analyzing a Test Group" on page 470. 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 **`<return>`**.

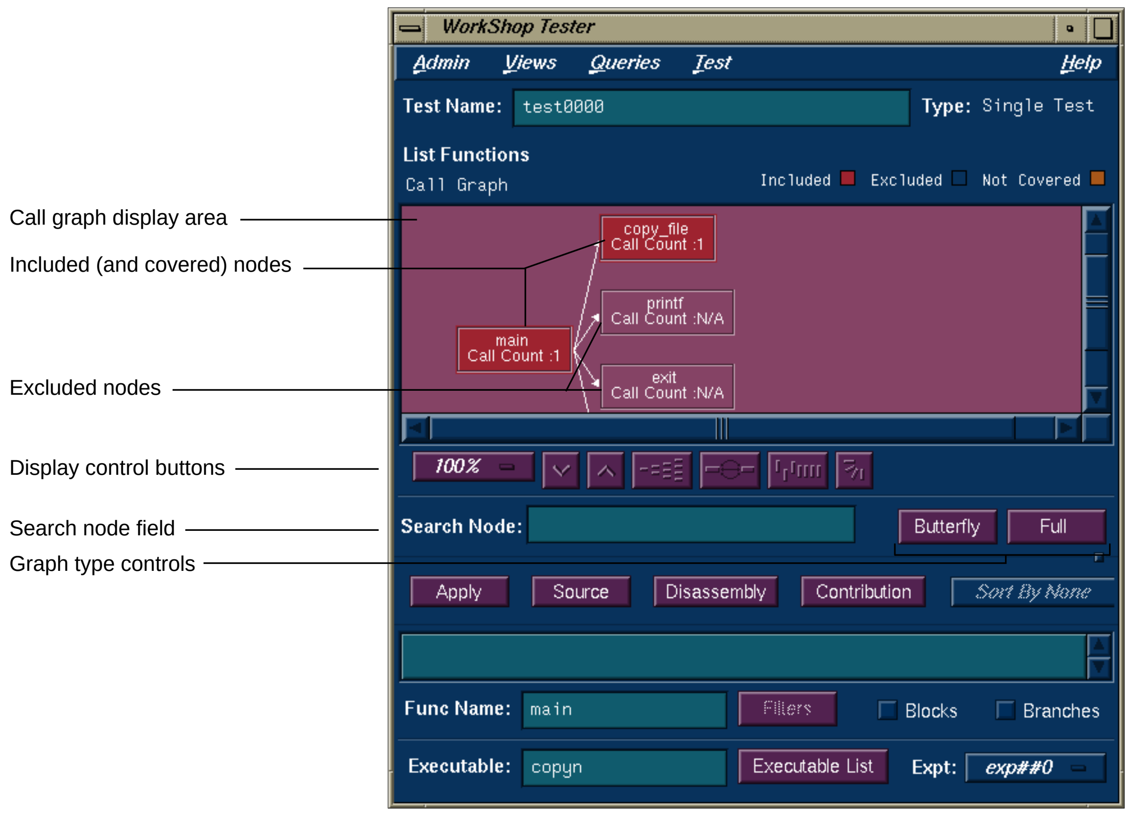
   Since *test0000* has incomplete coverage, it is more useful for illustrating how uncovered items appear.

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 8-12 shows a typical call graph. Initially, the call graph displays the *main* function and its immediate callees.



Test Name: test0000 Type: Single Test

Func Name: main

Executable: copyn Expt: exp##0

Figure 8-12 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 8-13).

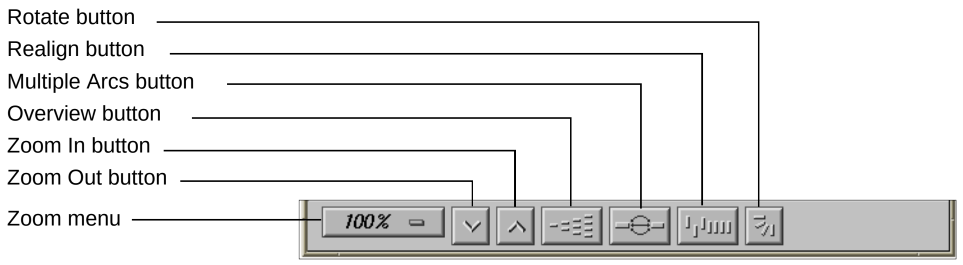


Figure 8-13 Call Graph Display Controls

These facilities are:

Zoom menu icon
: shows the current scale of the graph. If clicked on, a pop-up 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 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 may be used to position the main graph relative to the pop-up. 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).

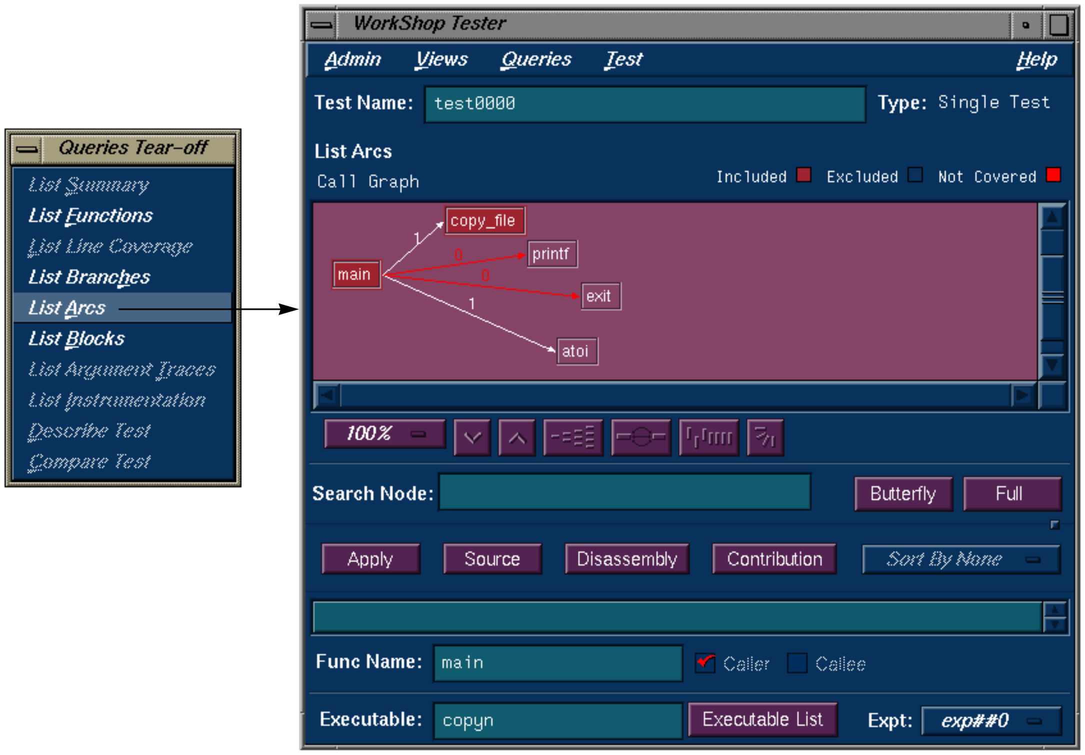
Entering a function in the *Search Node* field scrolls the display to the portion of the graph in which the function is located.

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 8-14. 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.



Test Name: test0000 Type: Single Test

Func Name: main

Executable: copyn

Figure 8-14 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 8-15. Arcs labeled N / A connect excluded functions and do not have call counts.

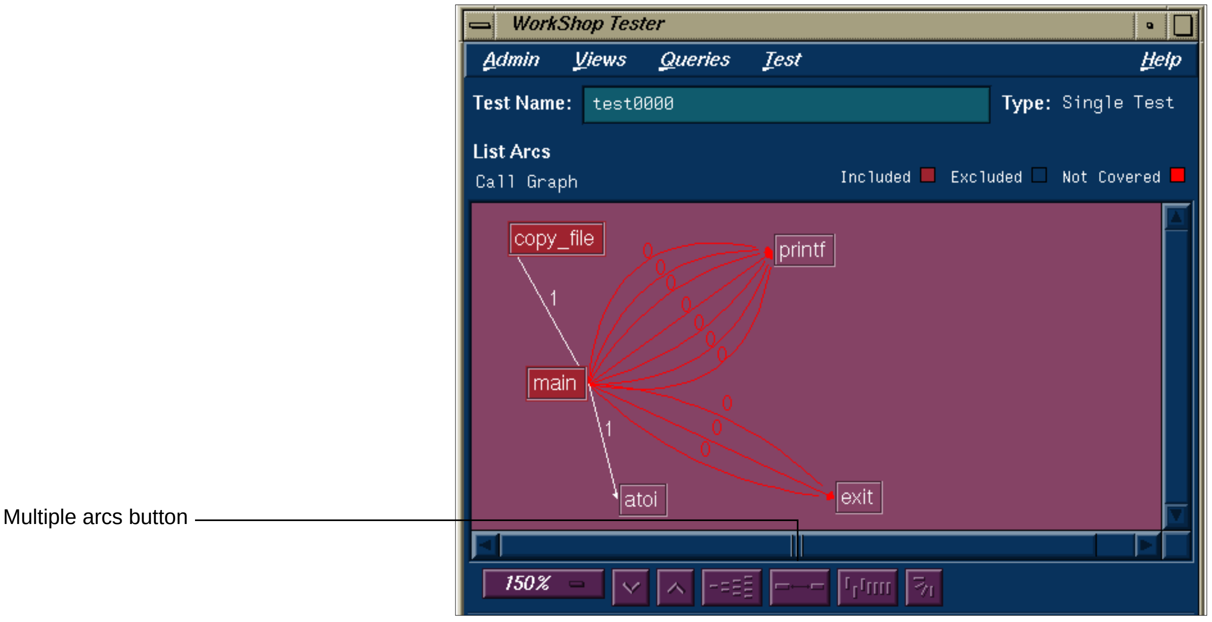


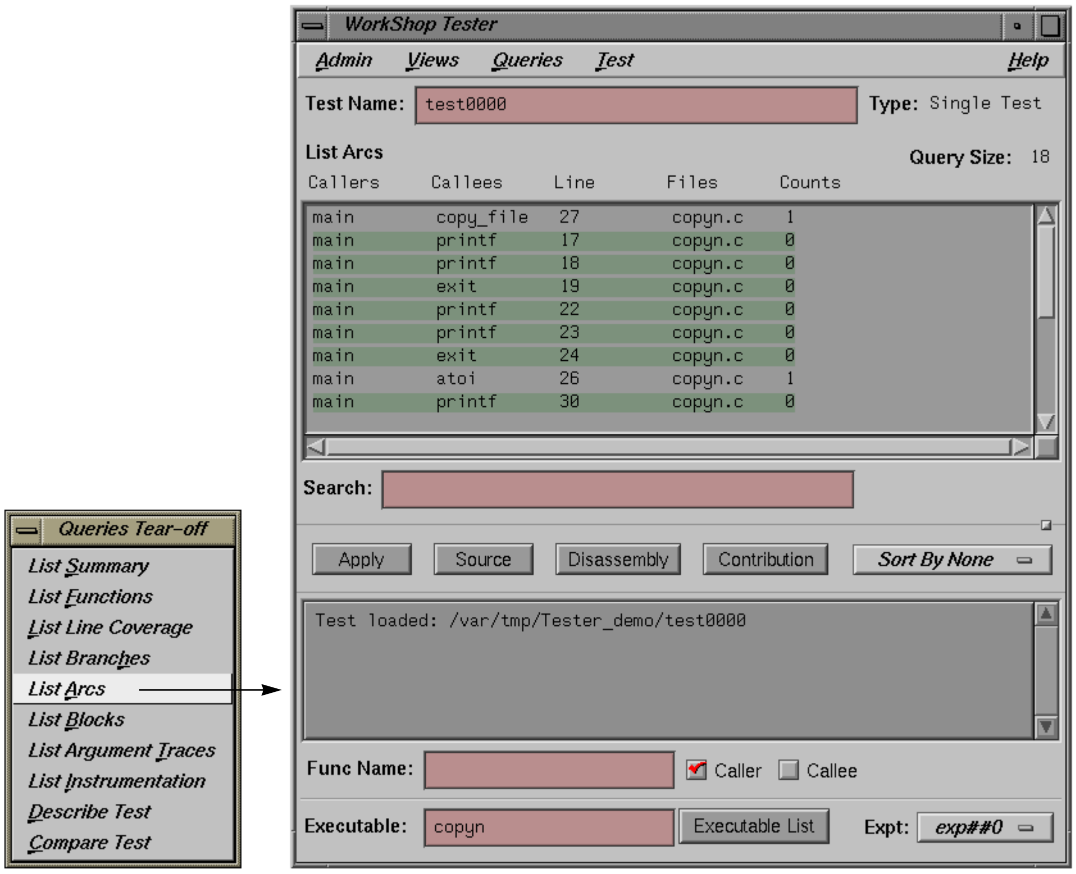
Figure 8-15 Call Graph for “List Arcs” Query — Multiple Arcs

6. Select “Text View” from the Views menu.

   This returns the display area to text mode from call graph mode. See Figure 8-16.

   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.



Test Name: test0000 Type: Single Test

List Arcs Query Size: 18

```
Callers     Callees     Line     Files      Counts
main        copy_file   27       copyn.c    1
main        printf      17       copyn.c    0
main        printf      18       copyn.c    0
main        exit        19       copyn.c    0
main        printf      22       copyn.c    0
main        printf      23       copyn.c    0
main        exit        24       copyn.c    0
main        atoi        26       copyn.c    1
main        printf      30       copyn.c    0
```

Search:

```
Test loaded: /var/tmp/Tester_demo/test0000
```

Func Name: [✓ Caller] Callee

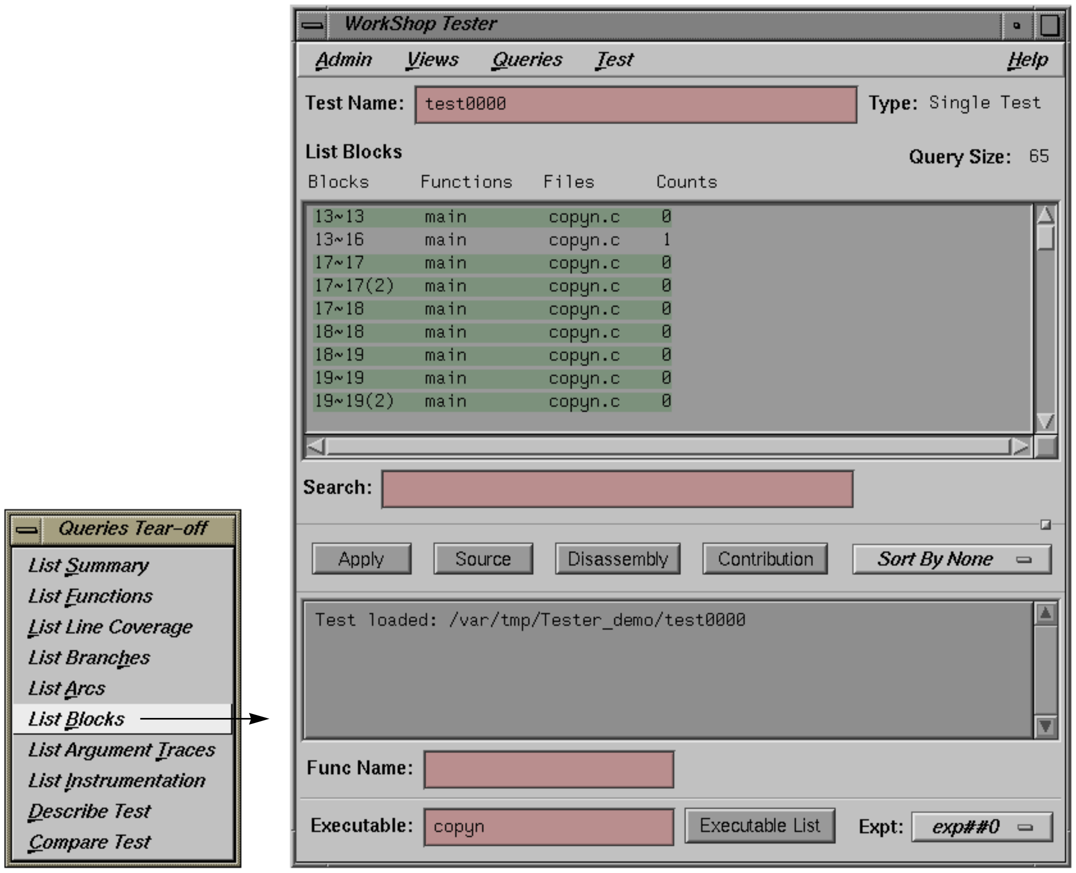
Executable: copyn Expt: exp##0

Figure 8-16 Test Analyzer Queries: “List Arcs“

7. Select “List Blocks” from the Queries menu.

   The window should be similar to Figure 8-17. 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.



Test Name: test0000 Type: Single Test

List Blocks Query Size: 65

```
Blocks      Functions   Files      Counts
13~13       main        copyn.c    0
13~16       main        copyn.c    1
17~17       main        copyn.c    0
17~17(2)    main        copyn.c    0
17~18       main        copyn.c    0
18~18       main        copyn.c    0
18~19       main        copyn.c    0
19~19       main        copyn.c    0
19~19(2)    main        copyn.c    0
```

Search:

```
Test loaded: /var/tmp/Tester_demo/test0000
```

Func Name:

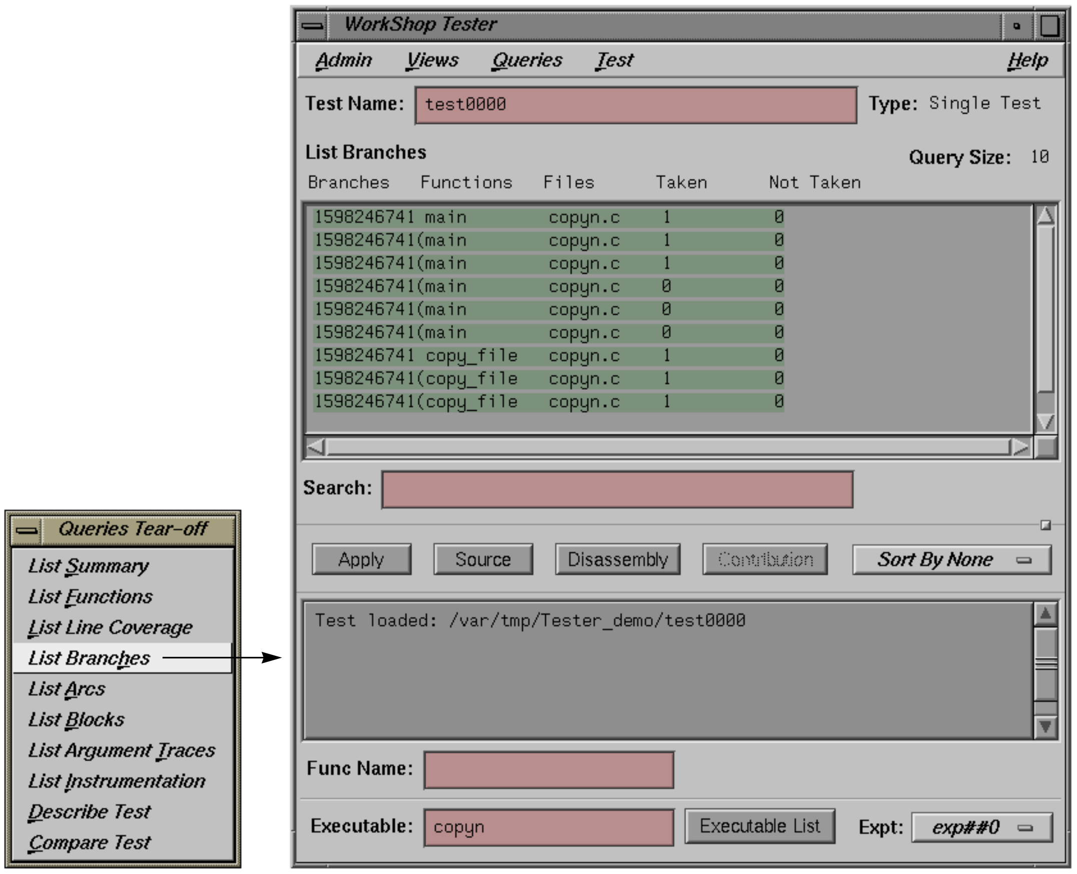
Executable: copyn Expt: exp##0

Figure 8-17 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 8-18.



Test Name: test0000 Type: Single Test

List Branches Query Size: 10

```
Branches    Functions   Files      Taken     Not Taken
1598246741 main        copyn.c    1         0
1598246741(main        copyn.c    1         0
1598246741(main        copyn.c    1         0
1598246741(main        copyn.c    0         0
1598246741(main        copyn.c    0         0
1598246741(main        copyn.c    0         0
1598246741 copy_file   copyn.c    1         0
1598246741(copy_file   copyn.c    1         0
1598246741(copy_file   copyn.c    1         0
```

Search:

```
Test loaded: /var/tmp/Tester_demo/test0000
```

Func Name:

Executable: copyn Expt: exp##0

Figure 8-18 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.

Chapter 9

Tester Graphical User Interface Reference

This chapter describes in detail each of the windows and their associated features available in the Tester graphical user interface.

Chapter 9

Tester Graphical User Interface Reference

This chapter describes the Tester graphical user interface. It contains these sections:

- "Accessing the Tester Graphical Interface"
- "Main Window and Menus"
- "Test Menu Operations"
- "Views Menu Operations"
- "Queries Menu Operations"
- "Admin Menu Operations"

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 "Admin Menu Operations" on page 257.

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 CASEVision color scheme.
- Select “Tester” from the “Launch Tool” submenu in a WorkShop Admin menu (see Figure 9-1). The major WorkShop tools, the Debugger, Static Analyzer, and Build Manager provide Admin menus from which you can access Tester.

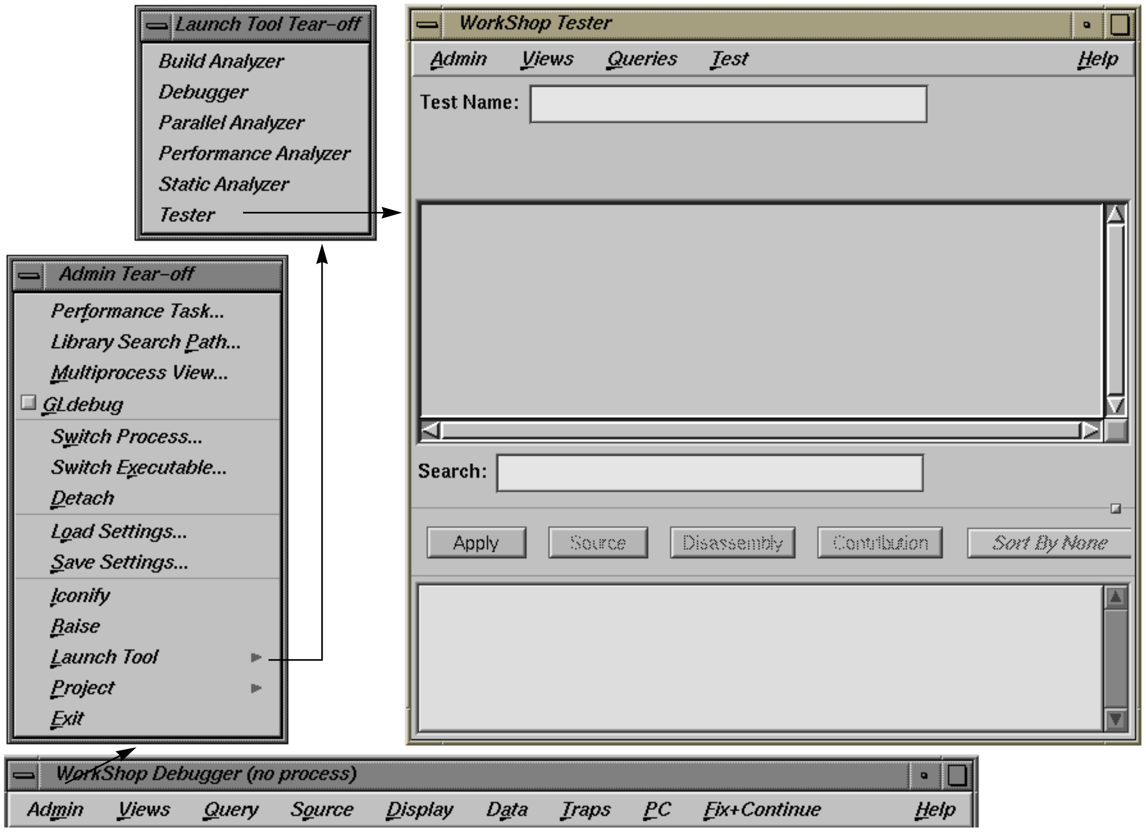


Figure 9-1 Accessing Tester from the WorkShop Debugger

Main Window and Menus

The main window and its menus are shown in Figure 9-2.

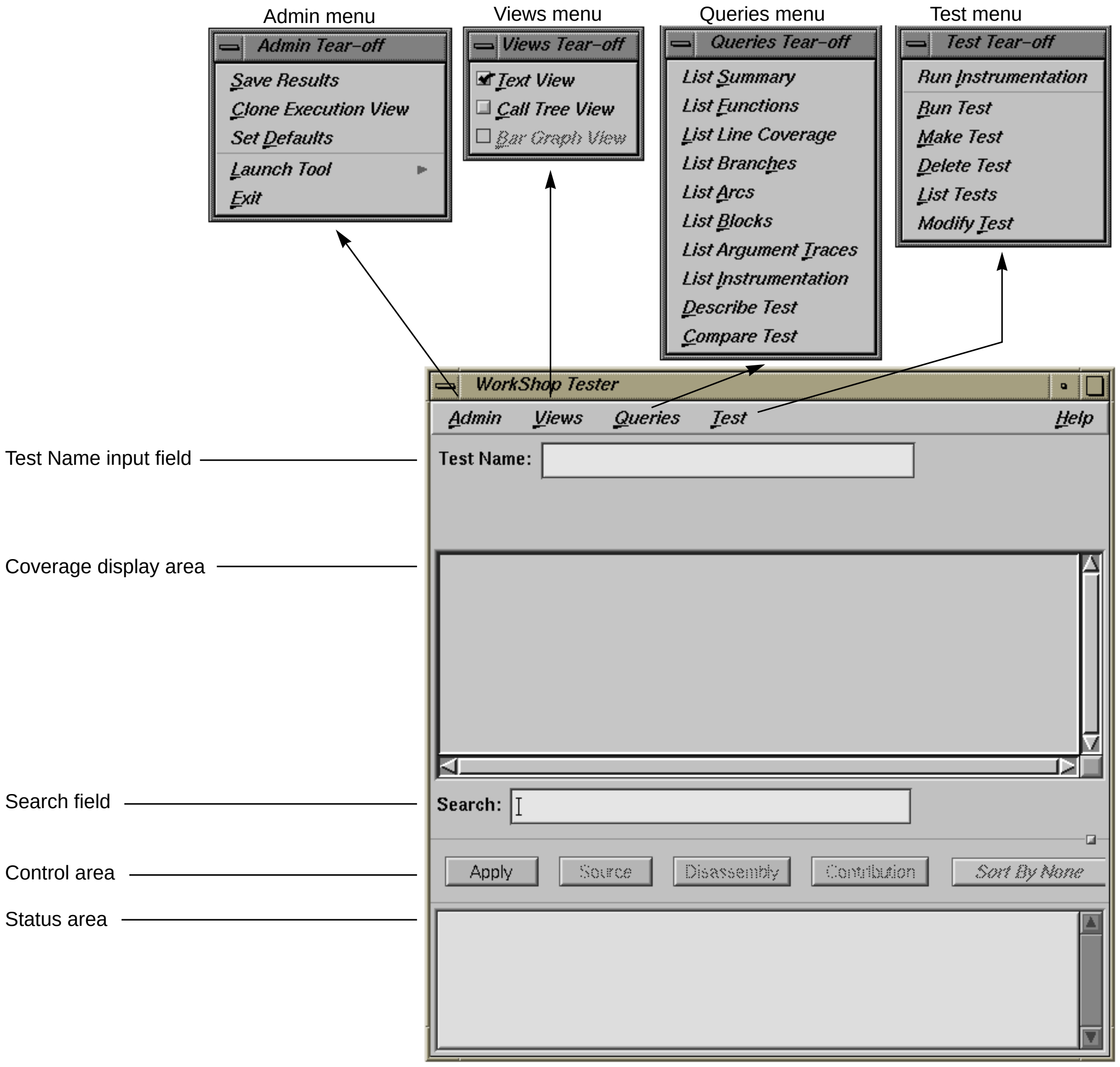


Figure 9-2 Main Test Analyzer Window

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.

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.

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 **`<return>`**, the highlight moves to the next occurrence.

Control Area Buttons

Apply is a general-purpose button for terminating data entry in text fields; you can use <**`return`**> equivalently. Both start the query.

Source lets you bring up the standard CASEVision 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
```

Disassembly brings up the CASEVision 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*.

Contribution 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.

Sort 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.

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 “Queries Menu Operations” on page 240).

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
  - 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 CASEVision tools.

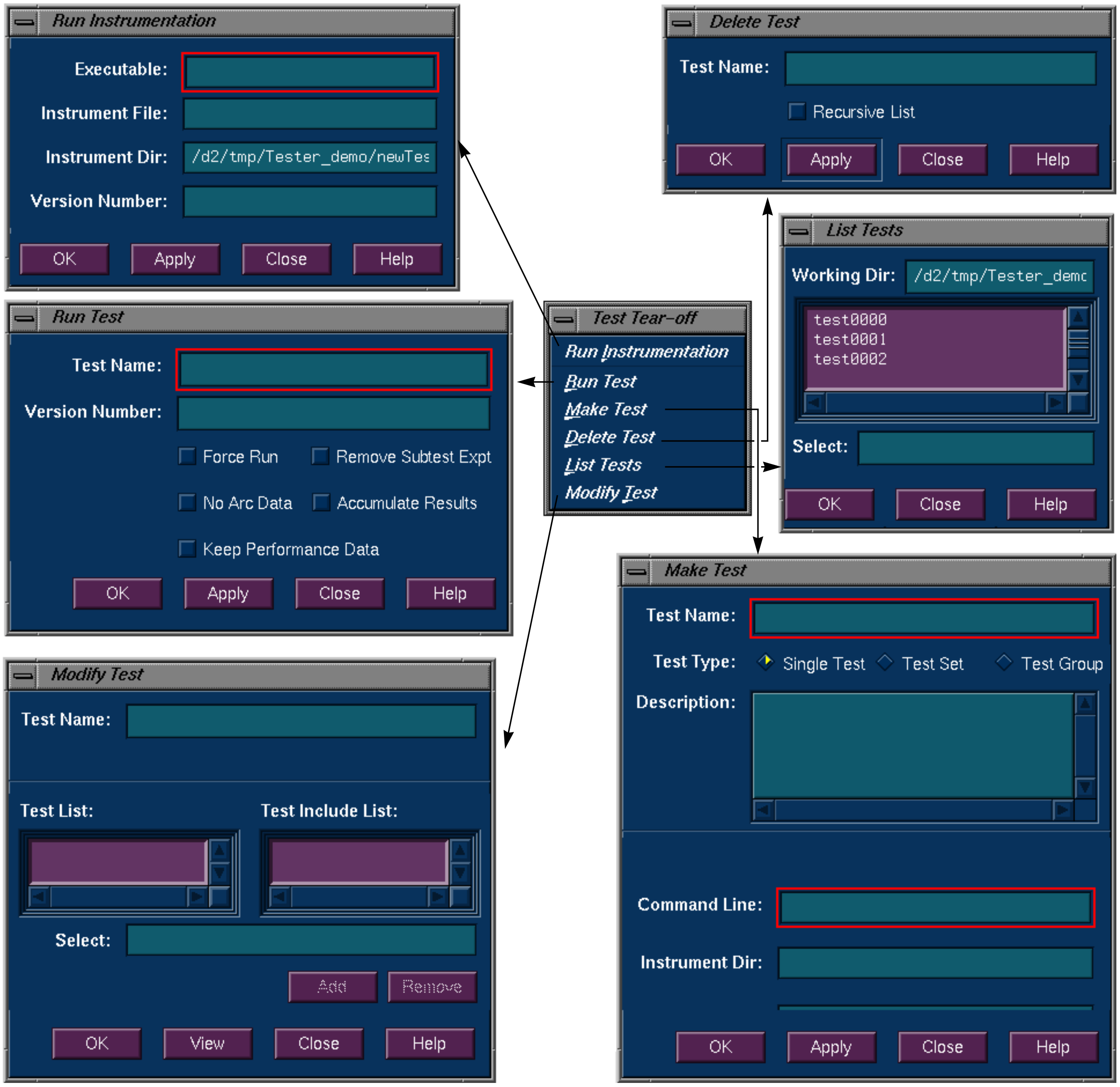
Test Menu Operations

All operations for running tests are accessed from the Test menu in the main Tester window. Figure 9-3 shows the dialog boxes used to perform test operations.

The Test menu provides these 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 “Single Test Analysis Process” on page 442.



Instrument Dir: /d2/tmp/Tester_demo/newTes

Working Dir: /d2/tmp/Tester_demc

```
test0000
test0001
test0002
```

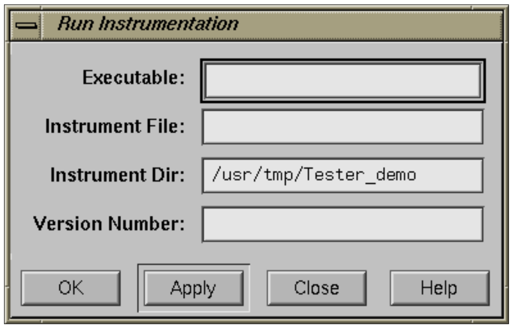
Test Type: [✓ Single Test] Test Set Test Group

Figure 9-3 Test Menu Commands

The “Run Instrumentation” dialog box (see Figure 9-4) 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’re 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 *forks*, *execs*, or *sprocs* 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.


Run Instrumentation

Executable:

Instrument File:

Instrument Dir: /usr/tmp/Tester_demo

Version Number:

Figure 9-4 “Run Instrumentation” Dialog Box

“Run Test”

invokes the executable with selected arguments and collects the coverage data. The “Run Test” dialog box (see Figure 9-5) 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 can't 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 don't 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.

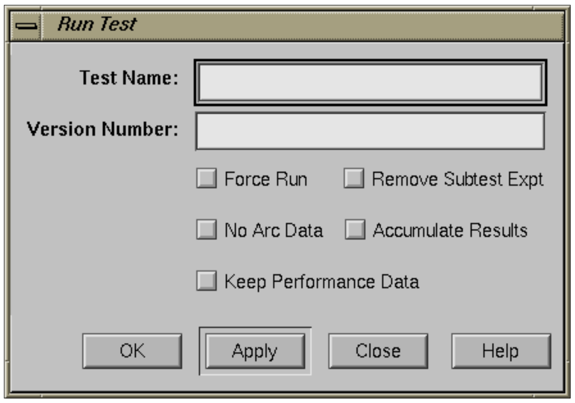


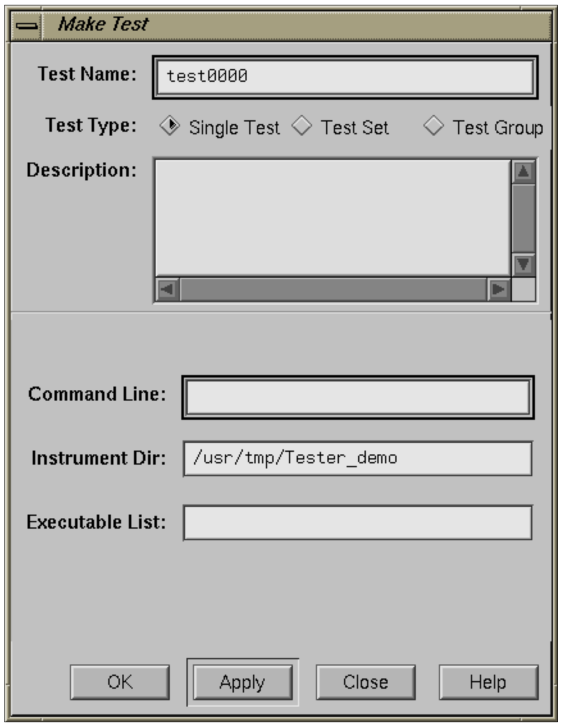
Figure 9-5 "Run Test" Dialog Box

“Make Test”

creates a test directory where the coverage data is to be stored and stores a TDF (test description file). For more information on this process, see “Single Test Analysis Process” on page 442.

The “Make Test” dialog box (see Figure 9-6) 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.



Test Name: test0000
Test Type: [✓ Single Test] Test Set Test Group
Description:
Command Line:
Instrument Dir: /usr/tmp/Tester_demo
Executable List:

Figure 9-6 “Make Test” Dialog Box

If you select *Single Test*, the following fields are provided:

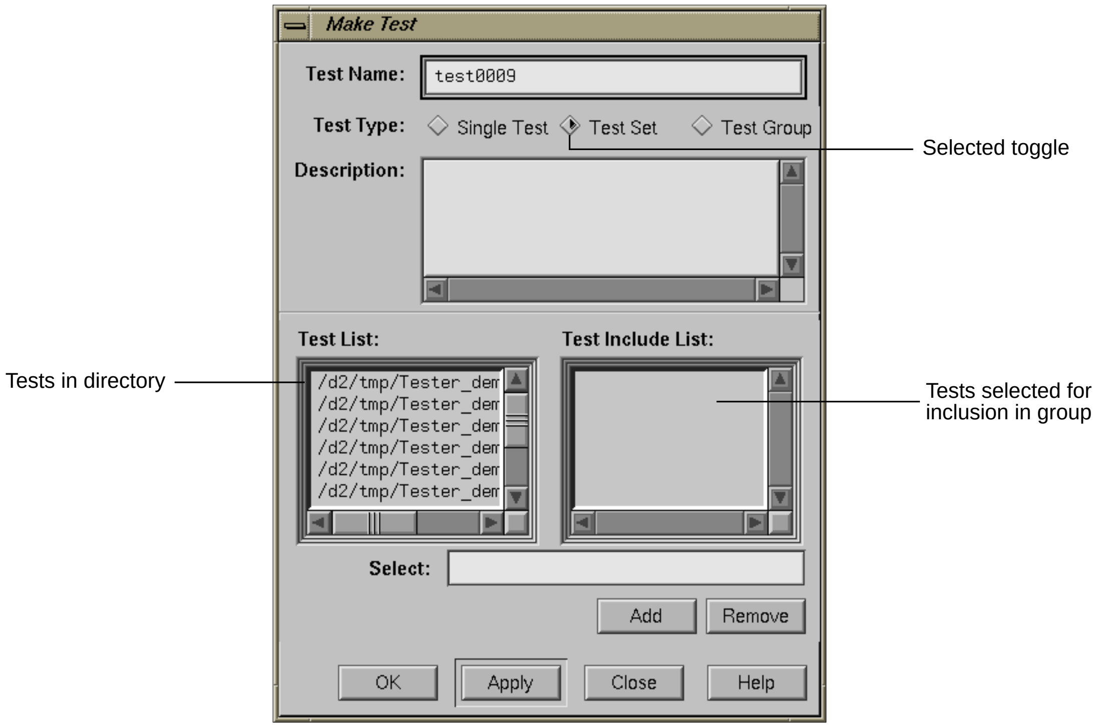
- *Command Line* lets you enter the target and any arguments to be used in the test.
- *Instrument Dir* is the directory in which the instrumentation file and related data are stored (not necessary if current working directory).
- *Executable List* is used if you are testing coverage of an executable that *forks*, *execs*, or *sprocs* 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 9-7), 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.



Test Name: test0009

Test Type: Single Test [✓ Test Set] Test Group

Description:

Test List:

/d2/tmp/Tester_dem
/d2/tmp/Tester_dem
/d2/tmp/Tester_dem
/d2/tmp/Tester_dem
/d2/tmp/Tester_dem
/d2/tmp/Tester_dem

Test Include List:

Select:

Figure 9-7 “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 9-8) 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.

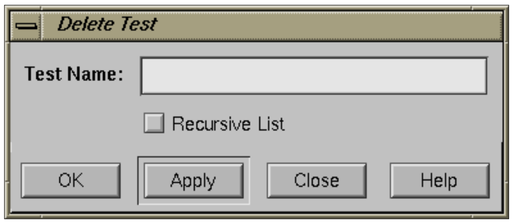
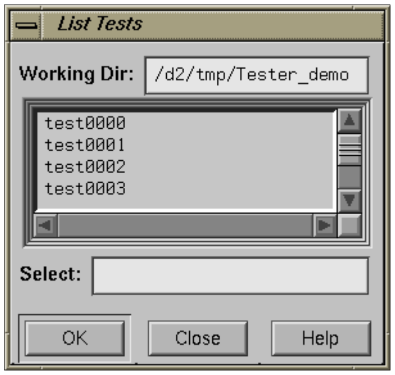


Figure 9-8 “Delete Test” Dialog Box

“List Tests”

shows you the tests in the current working directory. The “List Tests” dialog box (see Figure 9-9) 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 don’t 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.



Working Dir: /d2/tmp/Tester_demo

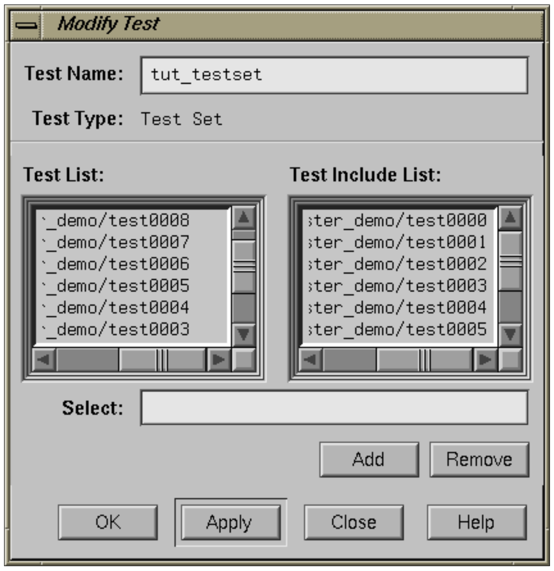
```
test0000
test0001
test0002
test0003
```

Figure 9-9 “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 **`<return>`** 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 9-10) 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 Lis*t.
- 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.)



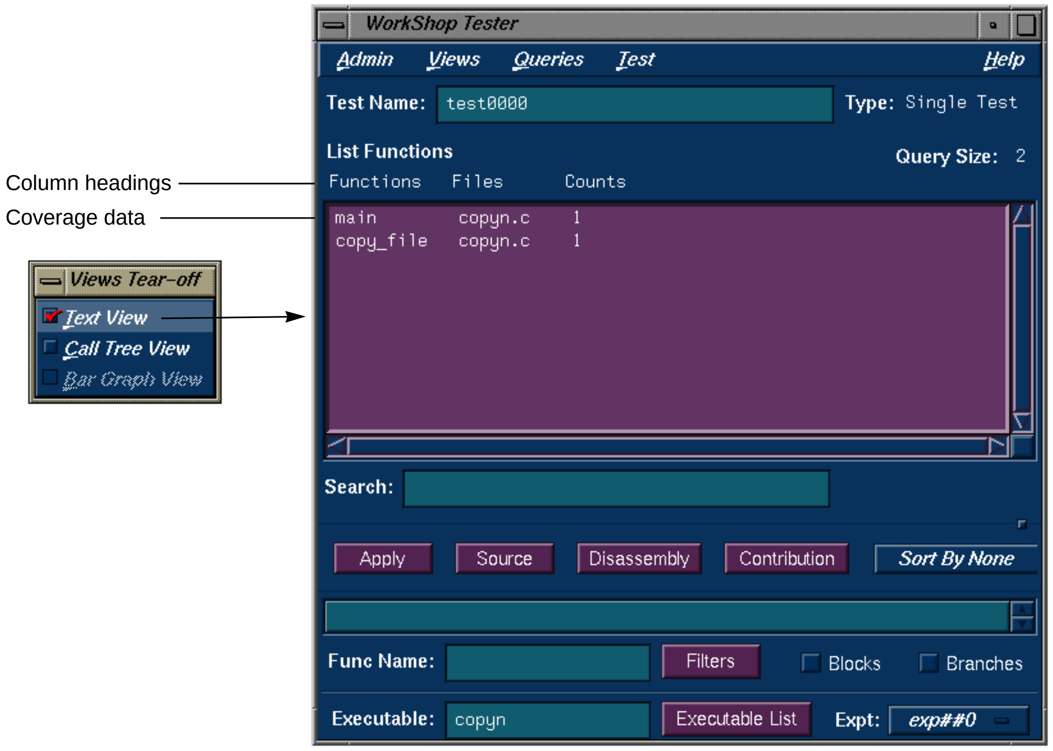
```
tut_testset
Test Set
`_demo/test0008
`_demo/test0007
`_demo/test0006
`_demo/test0005
`_demo/test0004
`_demo/test0003
ster_demo/test0000
ster_demo/test0001
ster_demo/test0002
ster_demo/test0003
ster_demo/test0004
ster_demo/test0005
```

Figure 9-10 “Modify Test” Dialog Box After Loading Tests

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 9-11.



Test Name: test0000 Type: Single Test

List Functions Query Size: 2

```
Functions   Files     Counts
main        copyn.c    1
copy_file   copyn.c    1
```

Search:

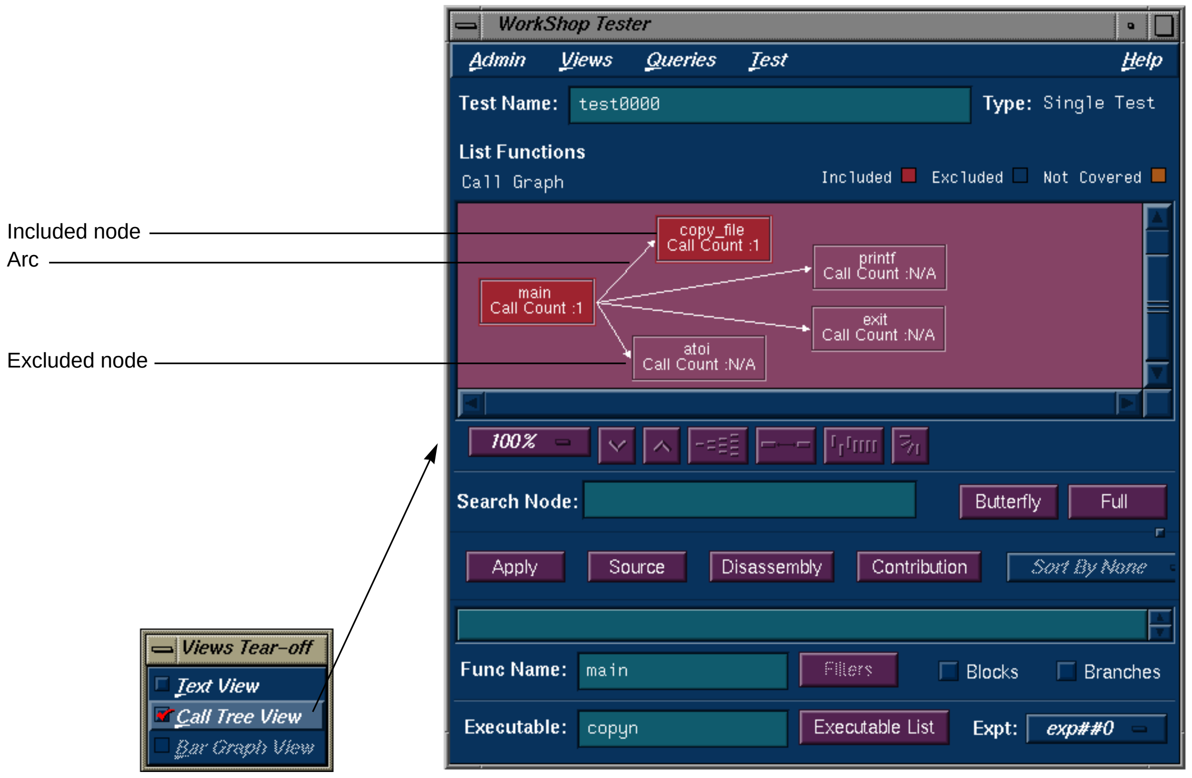
Func Name:

Executable: copyn Expt: exp##0

Figure 9-11 “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 9-12. It is not available if you run a test with No Arc Data on.



Test Name: test0000 Type: Single Test

Func Name: main

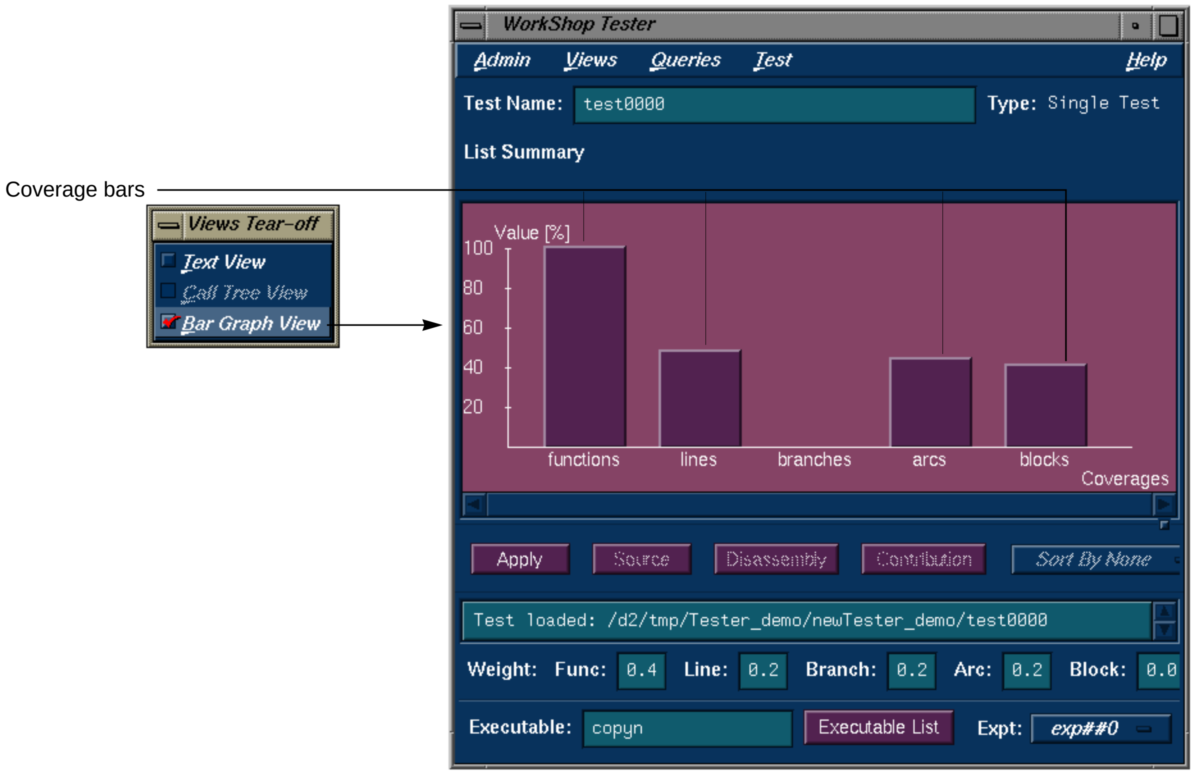
Executable: copyn

Expt: exp##0

Figure 9-12 “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 9-13. This view is only valid for “List Summary”, which is described in detail in “Queries Menu Operations” on page 240.



Test Name: test0000 Type: Single Test

```
Test loaded: /d2/tmp/Tester_demo/newTester_demo/test0000
```

Weight: Func: 0.4 Line: 0.2 Branch: 0.2 Arc: 0.2 Block: 0.0

Executable: copyn Expt: exp##0

Figure 9-13 “List Summary” Query in “Bar Graph View” Format

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 9-14.

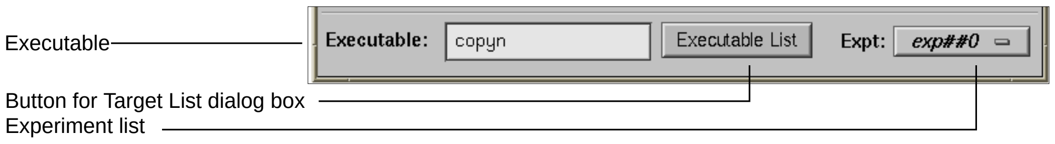


Figure 9-14 Query-Specific Default Fields for a Test or Test Set

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 9-14. These items act analogously except that they operate on dynamically shared objects (DSOs). Refer to "Tutorial #4 — Analyzing a Test Group" on page 470 for more information on test groups.

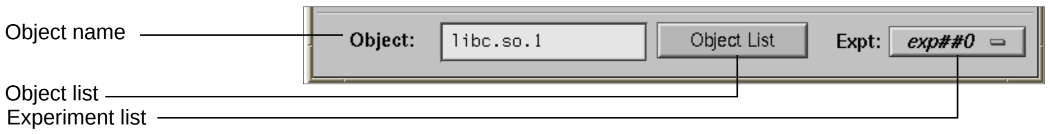


Figure 9-15 Query-Specific Default Fields for a DSO Test Group

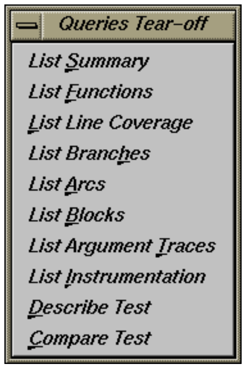
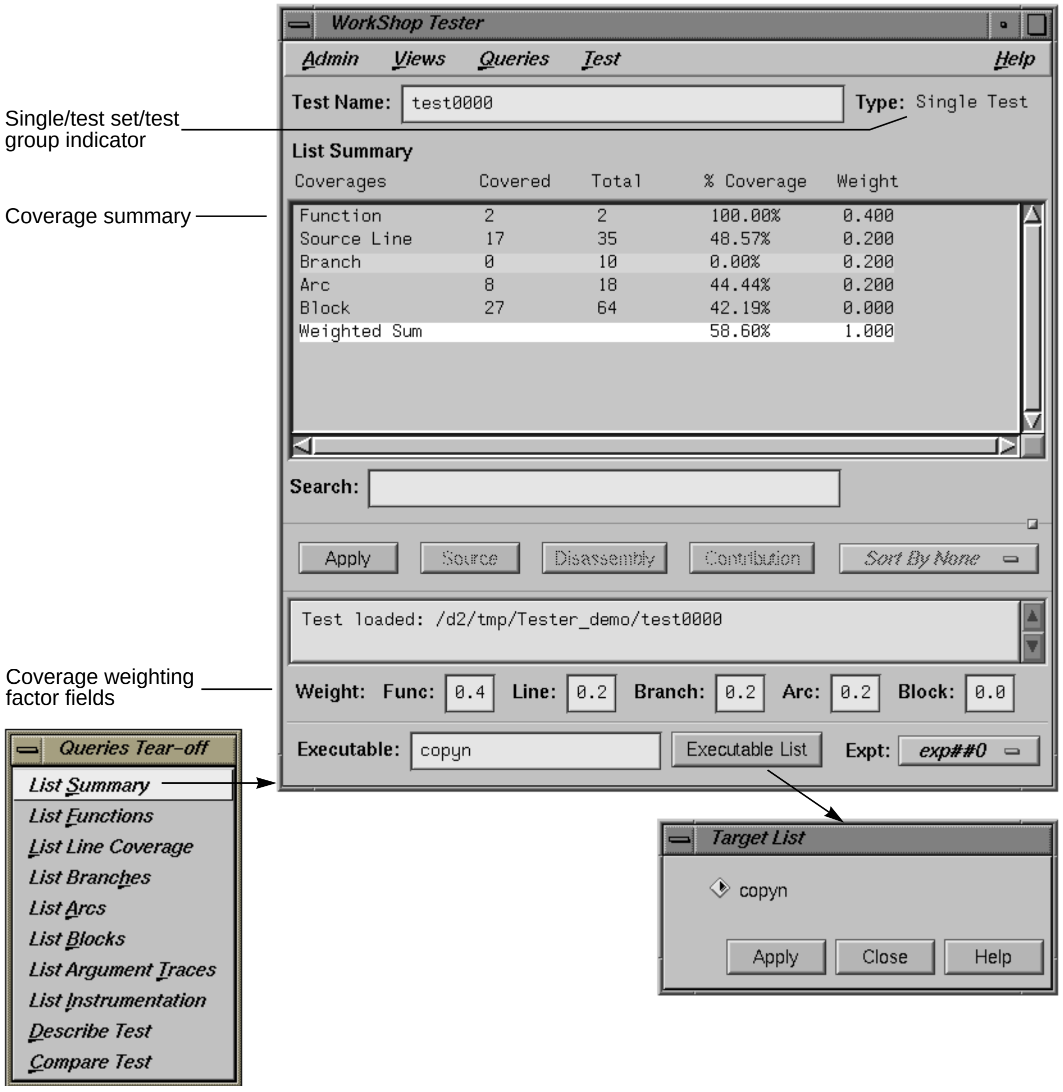


Figure 9-16 Queries Menu

The Queries menu (see Figure 9-16) provides these selections:

"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 9-17.



Test Name: test0000 Type: Single Test

```
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                27          64          42.19%        0.000
Weighted Sum                                 58.60%        1.000
```

```
Test loaded: /d2/tmp/Tester_demo/test0000
```

Weight: Func: 0.4 Line: 0.2 Branch: 0.2 Arc: 0.2 Block: 0.0

Executable: copyn Expt: exp##0

Target List: copyn

Figure 9-17 “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 9-13. 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.

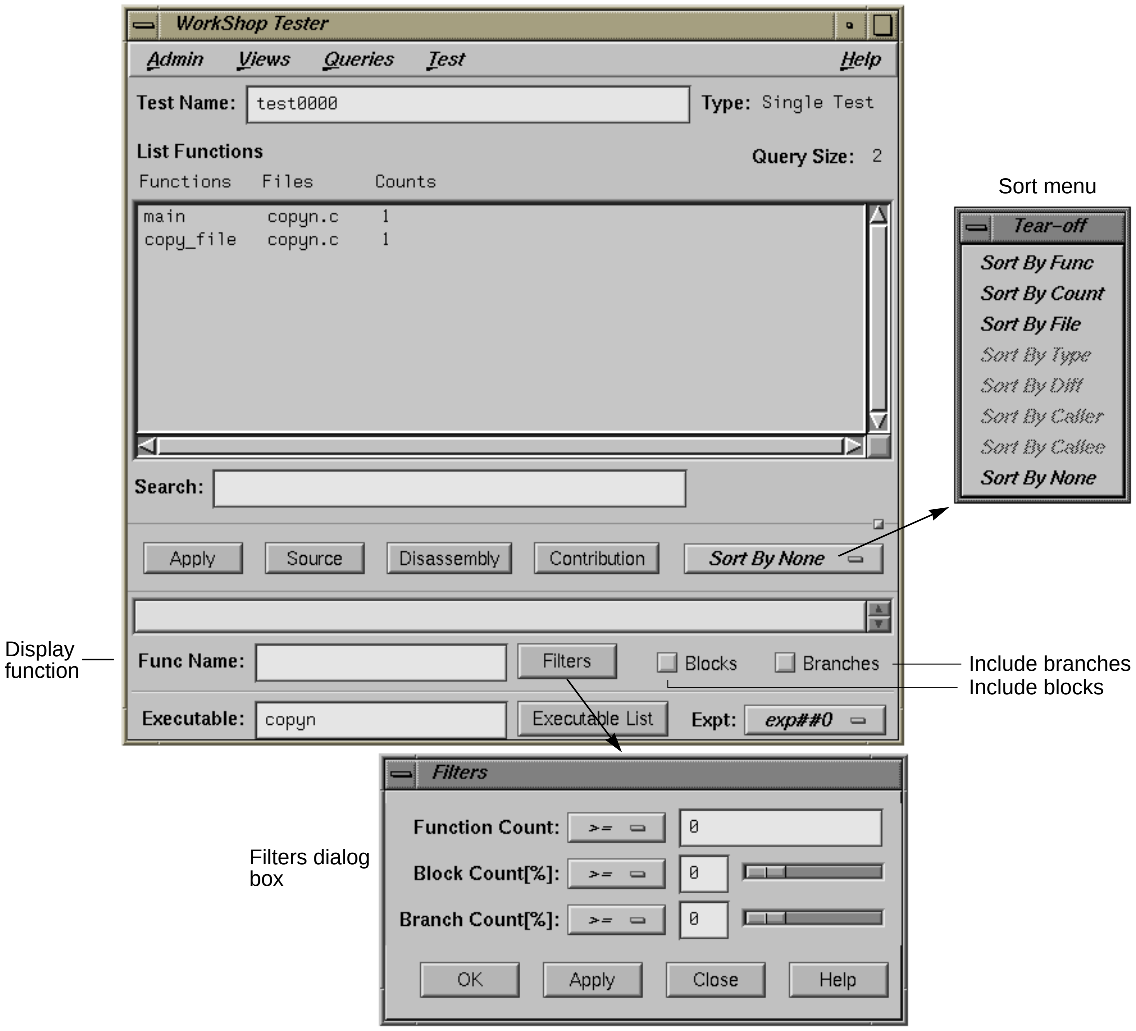
“List Functions” enables the sort menu that lets you determine the order in which the functions display. Only the sort criteria appropriate for the current query are enabled, in this case, “Sort By Func”, “Sort By Count”, and “Sort By File” as shown in Figure 9-18.

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.



Test Name: test0000 Type: Single Test

List Functions Query Size: 2

```
Functions    Files      Counts
main         copyn.c     1
copy_file    copyn.c     1
```

Executable: copyn

Function Count: >= 0
Block Count[%]: >= 0
Branch Count[%]: >= 0

Figure 9-18 “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 9-19). 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.

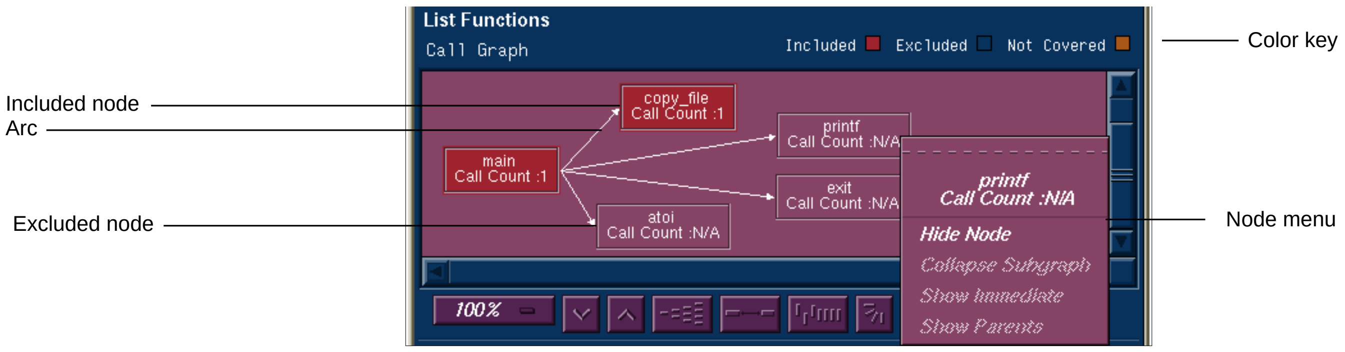
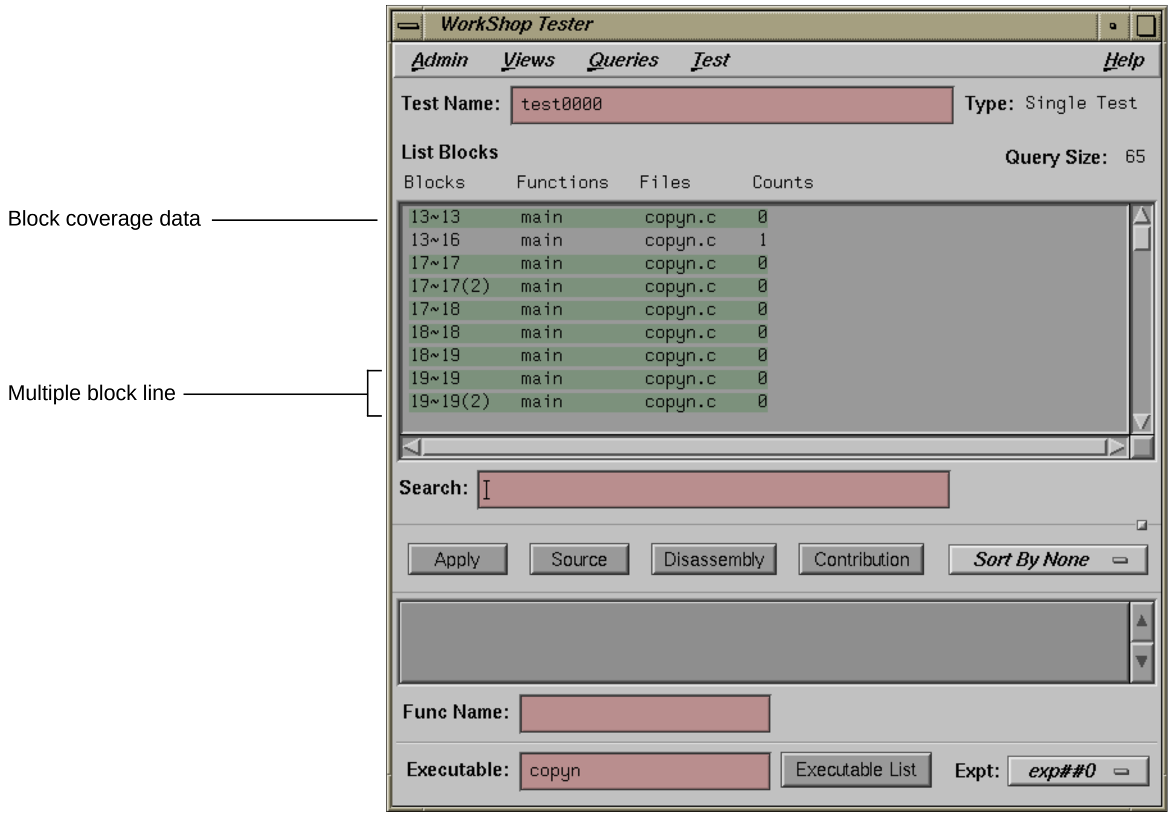


Figure 9-19 “List Functions” Example in “Call Tree View” Format

“List Blocks”

displays a list of blocks for one or more functions and the count information associated with each block (see Figure 9-20). 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.



Test Name: test0000 Type: Single Test

List Blocks Query Size: 65

```
Blocks      Functions   Files      Counts
13~13       main        copyn.c    0
13~16       main        copyn.c    1
17~17       main        copyn.c    0
17~17(2)    main        copyn.c    0
17~18       main        copyn.c    0
18~18       main        copyn.c    0
18~19       main        copyn.c    0
19~19       main        copyn.c    0
19~19(2)    main        copyn.c    0
```

Search:

Func Name:

Executable: copyn Expt: exp##0

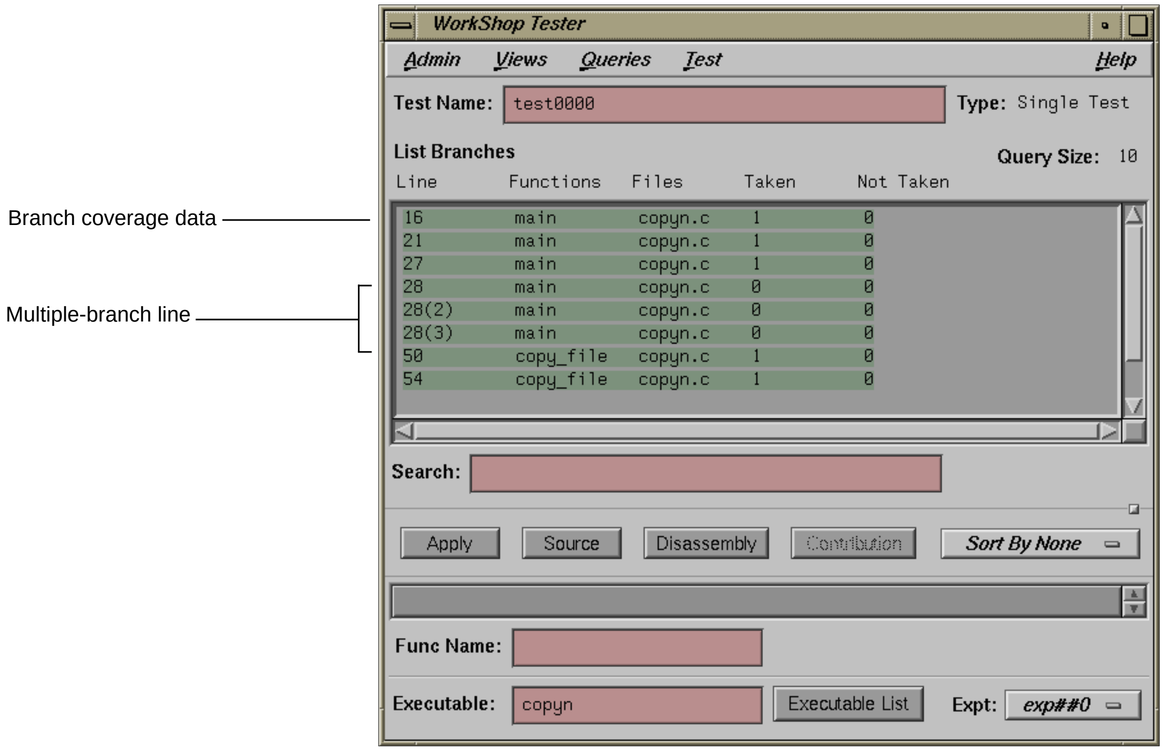
Figure 9-20 “List Blocks” Example

“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 9-21.

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.



Test Name: test0000 Type: Single Test

List Branches Query Size: 10

```
Line        Functions   Files      Taken       Not Taken
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
50          copy_file   copyn.c    1           0
54          copy_file   copyn.c    1           0
```

Search:

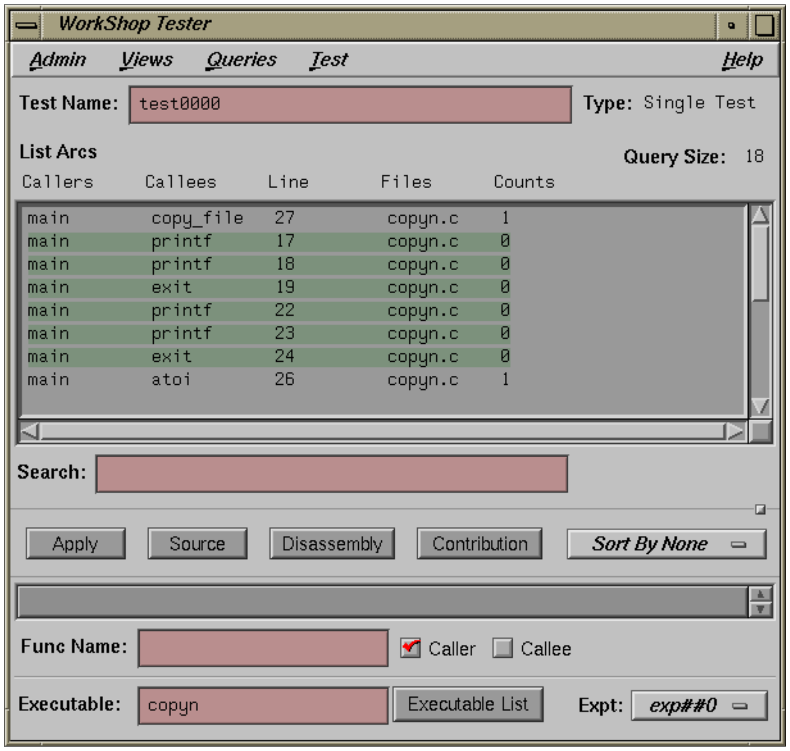
Func Name:

Executable: copyn Expt: exp##0

Figure 9-21 “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 9-22. 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.



Test Name: test0000 Type: Single Test

List Arcs Query Size: 18

```
Callers     Callees     Line      Files      Counts
main        copy_file   27        copyn.c    1
main        printf      17        copyn.c    0
main        printf      18        copyn.c    0
main        exit        19        copyn.c    0
main        printf      22        copyn.c    0
main        printf      23        copyn.c    0
main        exit        24        copyn.c    0
main        atoi        26        copyn.c    1
```

Search:

Func Name: [✓ Caller] Callee

Executable: copyn Expt: exp##0

Figure 9-22 “List Arcs” Example

Entering a function in the *Func Name* field displays the coverage results limited to that function only.

The *Caller* and *Callee* 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 *Func Name* field and then clicking the appropriate toggle, *Caller* or *Callee*.

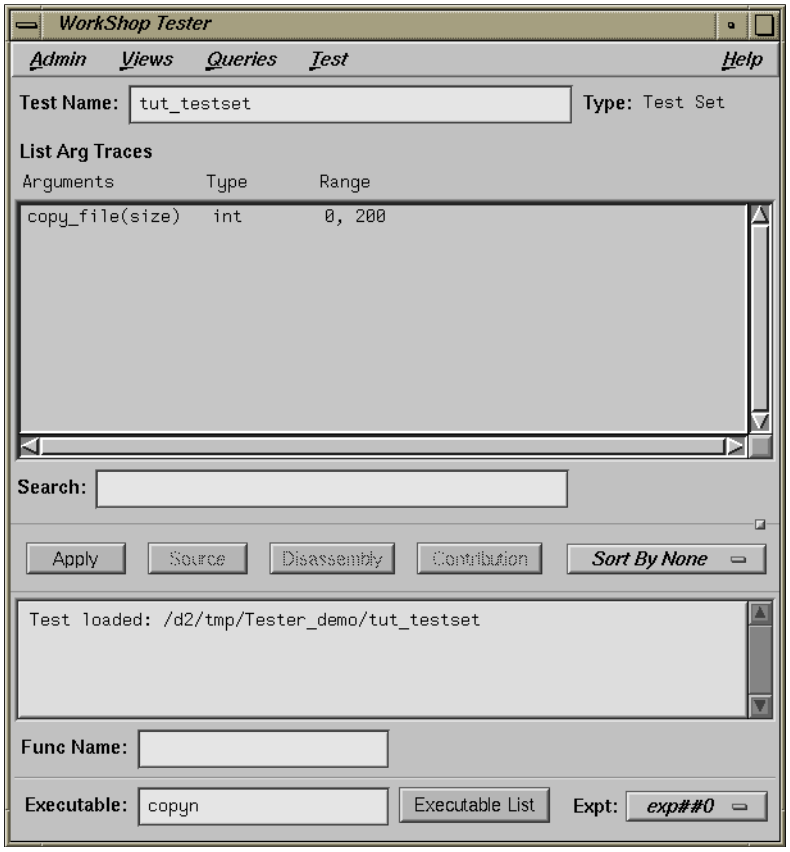
“List Argument Traces”
shows argument tracing information (see Figure 9-23). Argument tracing is enabled in the instrumentation file using the TRACE command with the MAX, MIN, BOUNDS, and RETURN options. TRACE lets you monitor argument values in the functions over all experiments. The syntax in the file 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.

For more information on the instrumentation file, see “Single Test Analysis Process” on page 442.



Test Name: tut_testset Type: Test Set

List Arg Traces

```
Arguments          Type        Range
copy_file(size)    int         0, 200
```

Search:

```
Test loaded: /d2/tmp/Tester_demo/tut_testset
```

Func Name:

Executable: copyn

Figure 9-23 “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.

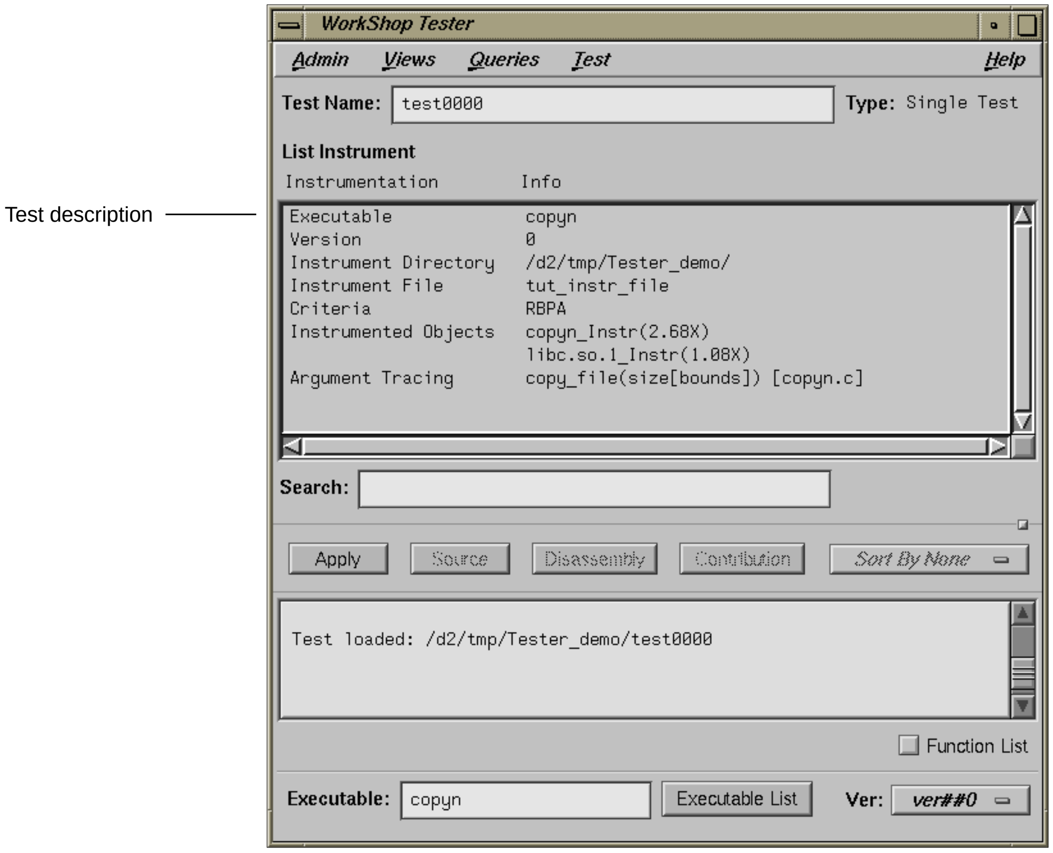
"List Instrumentation"

displays the instrumentation information for a particular test. See Figure 9-24.

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.



Test Name: test0000 Type: Single Test

List Instrument

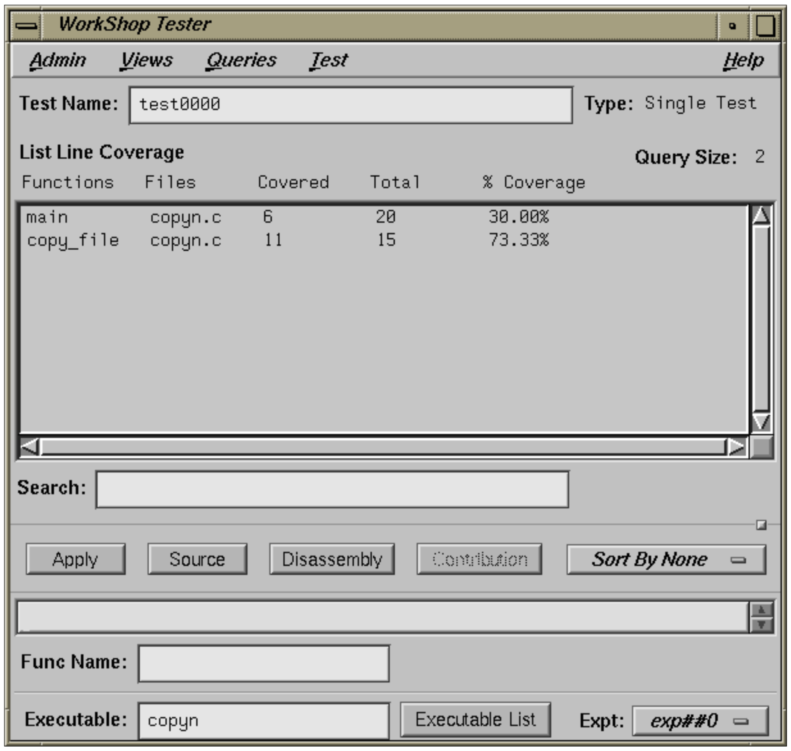
```
Instrumentation        Info
Executable             copyn
Version                0
Instrument Directory   /d2/tmp/Tester_demo/
Instrument File        tut_instr_file
Criteria               RBPA
Instrumented Objects   copyn_Instr(2.68X)
                       libc.so.1_Instr(1.08X)
Argument Tracing       copy_file(size[bounds]) [copyn.c]
```

```
Test loaded: /d2/tmp/Tester_demo/test0000
```

Executable: copyn Ver: ver##0

Figure 9-24 “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 9-25.



Test Name: test0000 Type: Single Test

List Line Coverage Query Size: 2

```
Functions   Files     Covered   Total    % Coverage
main        copyn.c   6         20       30.00%
copy_file   copyn.c   11        15       73.33%
```

Search:

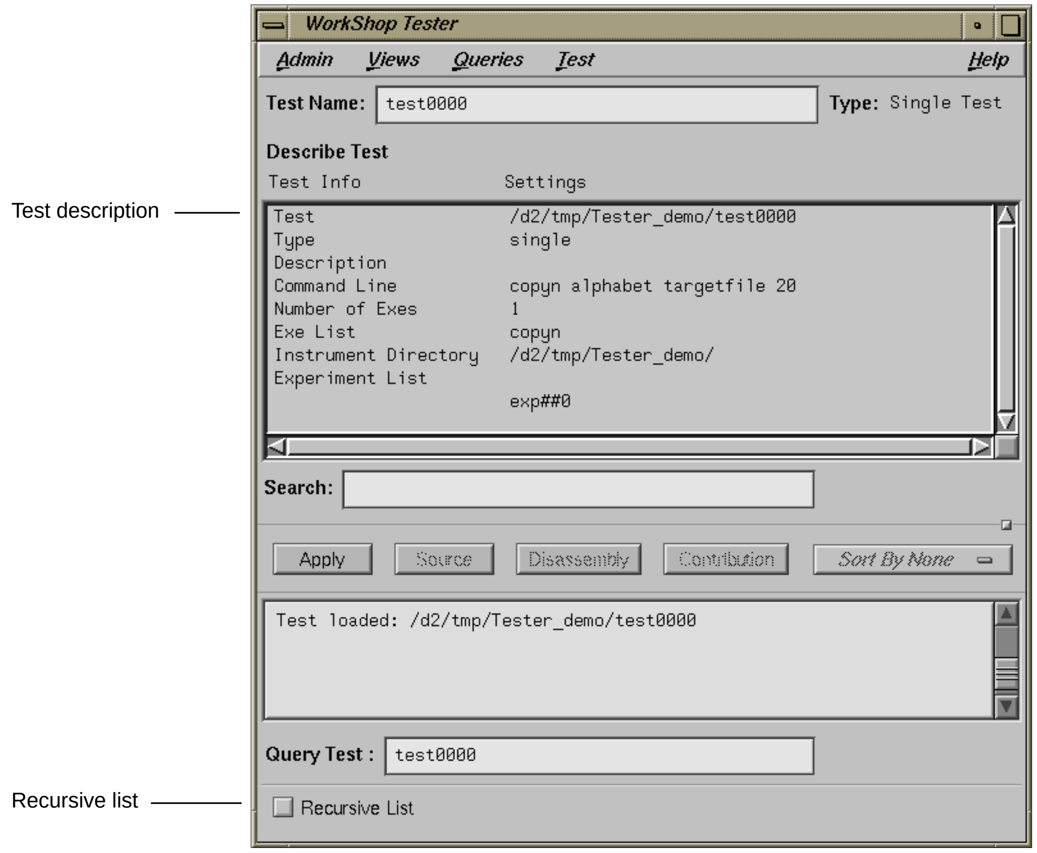
Func Name:

Executable: copyn Expt: exp##0

Figure 9-25 “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 9-26.



Test Name: test0000 Type: Single Test

Describe Test

```
Test Info               Settings
Test                    /d2/tmp/Tester_demo/test0000
Type                    single
Description
Command Line            copyn alphabet targetfile 20
Number of Exes          1
Exe List                copyn
Instrument Directory    /d2/tmp/Tester_demo/
Experiment List
                        exp##0
```

Search:

```
Test loaded: /d2/tmp/Tester_demo/test0000
```

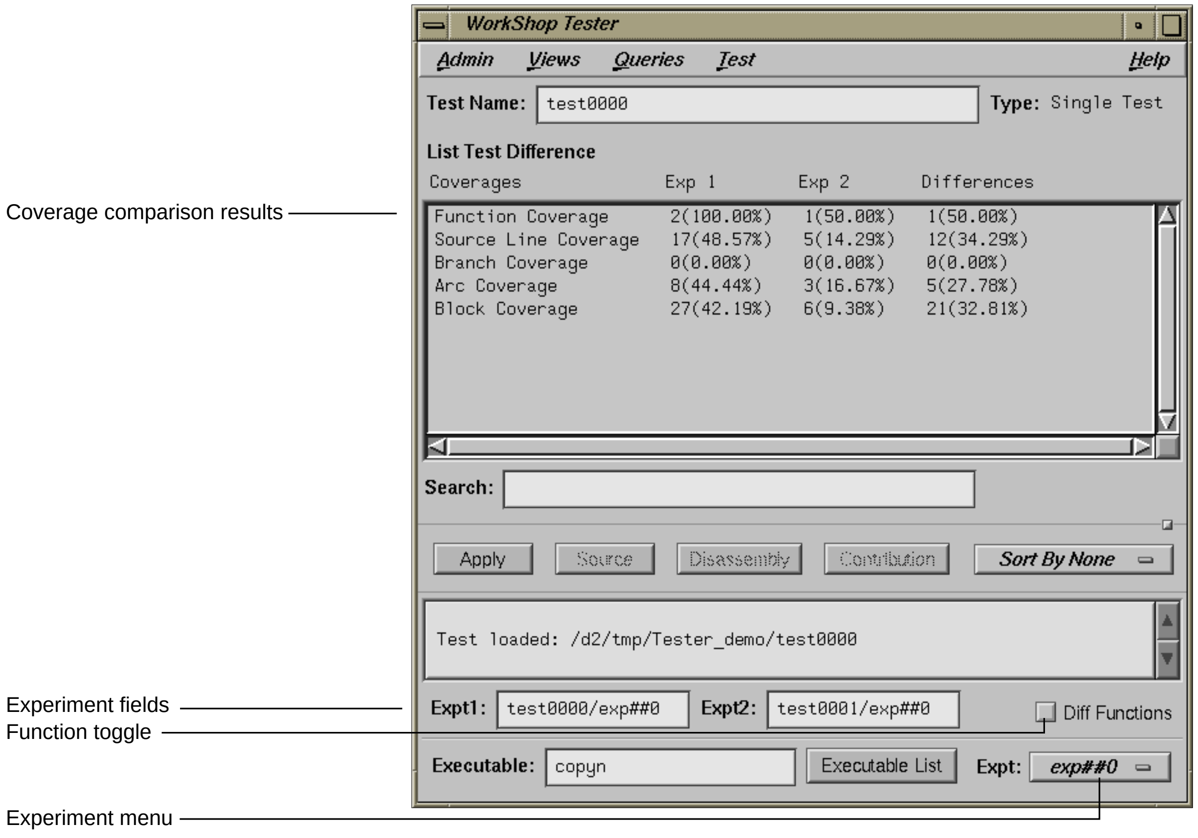
Query Test : test0000

☐ Recursive List

Figure 9-26 “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 9-27.



Test Name: test0000 Type: Single Test

List Test Difference

```
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           27(42.19%)   6(9.38%)     21(32.81%)
```

Search:

```
Test loaded: /d2/tmp/Tester_demo/test0000
```

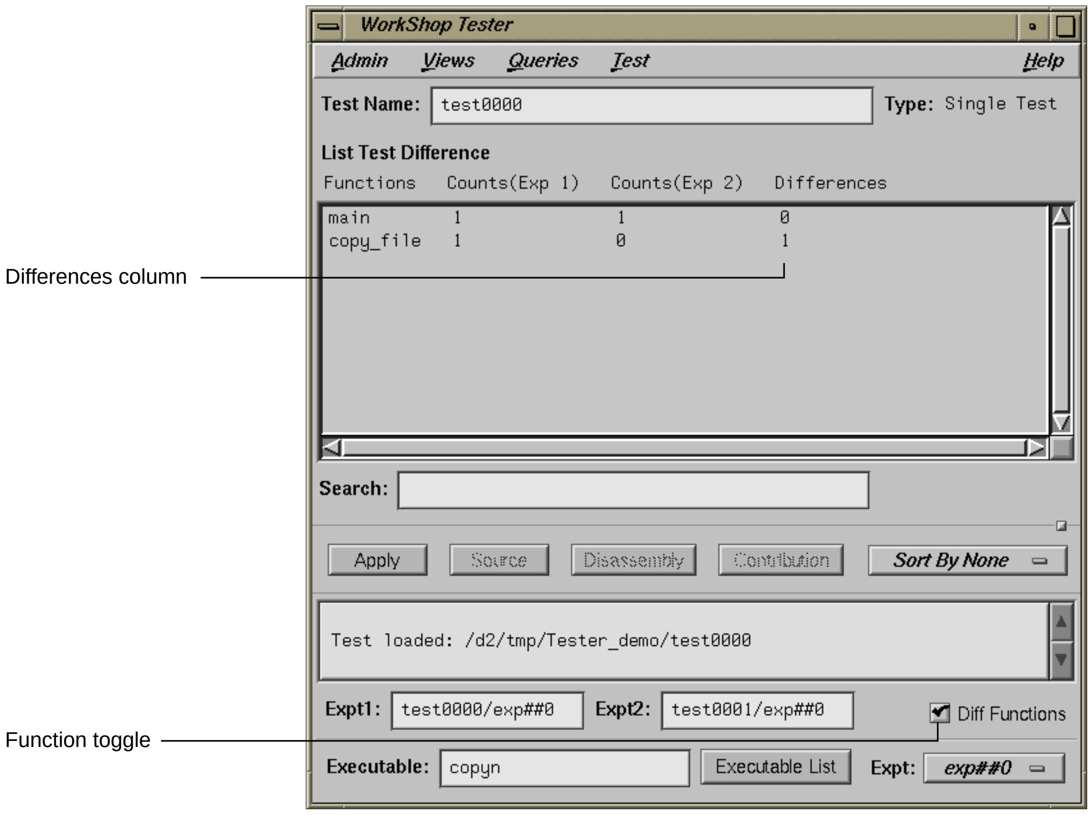
Expt1: test0000/exp##0 Expt2: test0001/exp##0

Executable: copyn Expt: exp##0

Figure 9-27 “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 9-28 shows a typical function difference example.



Test Name: test0000 Type: Single Test

List Test Difference

```
Functions   Counts(Exp 1)   Counts(Exp 2)   Differences
main        1               1               0
copy_file   1               0               1
```

Search:

```
Test loaded: /d2/tmp/Tester_demo/test0000
```

Expt1: test0000/exp##0 Expt2: test0001/exp##0 ✓ Diff Functions

Executable: copyn Expt: exp##0

Figure 9-28 “Compare Test” Example — Function Differences

Admin Menu Operations

The Admin menu is shown in Figure 9-29.

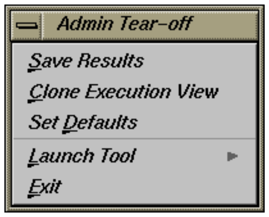


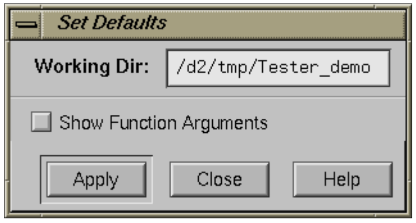
Figure 9-29 Admin Menu

The Admin menu provides these selections:

"Save Results"
: brings up the standard CASEVision 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 9-30.



Working Dir: /d2/tmp/Tester_demo

Figure 9-30 "Set Defaults" Dialog Box

"Launch Tool"

The Launch Tool submenu contains commands for launching other WorkShop applications (see Figure 9-31).

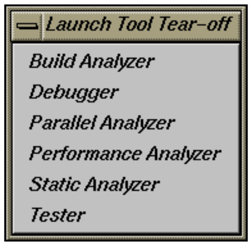


Figure 9-31 "Launch Tool" Submenu

If any of these tools are not installed on your system, the corresponding menu item will be grayed out.

"Exit" closes all Tester windows.

Index

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 007-2581-002.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  – On the Internet: techpubs@sgi.com
  – For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- 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