ProDev™ WorkShop: Performance Analyzer User's Guide

007–2581–006

CONTRIBUTORS

Written by Jenn Byrnes

Edited by Rick Thompson

Illustrated by Chrystie Danzer

Production by Glen Traefeld

COPYRIGHT

© 1996, 1999, 2001 Silicon Graphics, Inc. All rights reserved; provided portions may be copyright in third parties, as indicated elsewhere herein. No permission is granted to copy, distribute, or create derivative works from the contents of this electronic documentation in any manner, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

LIMITED RIGHTS LEGEND

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

TRADEMARKS AND ATTRIBUTIONS

IRIX and Silicon Graphics are registered trademarks and Developer Magic, ProDev, and the Silicon Graphics logo are trademarks of Silicon Graphics, Inc.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited. Vampir is a trademark of Pallas, Inc. X/Open is a trademark of X/Open Company Ltd. The X device is a trademark of the Open Grdoup.

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

New Features

This revision of the *ProDev WorkShop: Performance Analyzer User's Guide* is revised to document the Parallel Overhead View.

Record of Revision

Version	Description
001	June 1995 Original Printing.
004	June 1998 Revised to relect changes for the ProDev WorkShop 2.7 release, including the ability to present SpeedShop data within WorkShop.
005	April 1999 Supports the ProDev WorkShop 2.8 release.
006	June 2001 Supports the ProDev WorkShop 2.9 release.

Contents

About This Guide . . . . **xix**
Related Publications xix
Obtaining Publications xx
Conventions xx
Reader Comments xxi

1. Introduction to the Performance Analyzer **1**
Performance Analyzer Overview 1
The Performance Analyzer Tools 2
Sources of Performance Problems 4
CPU-Bound Processes 4
I/O-Bound Processes 4
Memory-Bound Processes 5
Bugs 5
Performance Phases in Programs 5
Interpreting Performance Analyzer Results 5
The Time Line Display 7
Resource Usage Graphs 7
Usage View (Numerical) 10
I/O View 11
MPI Stats View (Graphs) 12
MPI Stats View (Numerical) 14
The Parallel Overhead View 16
The Function List Area 16

Call Graph View 17
Butterfly View 18
Source View with Performance Annotations 20
Disassembled Code with Performance Annotations 21
Leak View, Malloc View, Malloc Error View, and Heap View 22
Memory Leakage 23
Bad Frees 23
Call Stack View 24
Working Set View 25
Cord Analyzer 26

2. Performance Analyzer Tutorial **29**
Tutorial Overview 29
Tutorial Setup 30
Changing Window Font Size 31
Analyzing the Performance Data 31
Analyzing Memory Experiments 40
Finding Memory Leaks 40
Memory Use Tutorial 42

3. Setting Up Performance Analysis Experiments **45**
Experiment Setup Overview 45
Selecting a Performance Task 46
Setting Sample Traps 47
Understanding Predefined Tasks 48
Profiling/PC Sampling 48
User Time/Callstack Sampling 49
Ideal Time/Pixie 49

Floating-Point Exception Trace 52
I/O Trace 52
Memory Leak Trace 52
R10000 and R12000 Hardware Counters 53
Custom 54
Displaying Data from the Parallel Analyzer 55

4. Performance Analyzer Reference 57

Selecting Performance Tasks 58
Specifying a Custom Task 59
Specifying Data to be Collected 60
Call Stack Profiling 60
Basic Block Count Sampling 61
PC Profile Counts 61
Specifying Tracing Data 62
`malloc` and `free` Heap Analysis 63
I/O Operations 63
Floating-Point Exceptions 63
MPI Stats Trace 63
Specifying Polling Data 64
Pollpoint Sampling 65
Call Stack Profiling 65
Specifying the Experiment Configuration 66
Specifying the Experiment Directory 67
Other Options 67
The Performance Analyzer Main Window 68
Task Field 69
Function List Display and Controls 69

- Usage Chart Area 71
- Time Line Area and Controls 72
  - The Time Line Calipers 72
  - Current Event Selection 73
  - Time Line Scale Menu 73
- Admin Menu 73
- Config Menu 75
- Views Menu 83
- Executable Menu 84
- Thread Menu 85

Usage View (Graphs) 85
- Charts in the Usage View (Graphs) Window 87
- Getting Event Information from the Usage View (Graphs) Window 88

The Process Meter Window 89

Usage View (Numerical) Window 91

The I/O View Window 93

The MPI Stats View (Graphs) Window 94

The MPI Stats View (Numerical) Window 96

The Parallel Overhead View Window 96

The Call Graph View Window 97
- Special Node Icons 98
- Annotating Nodes and Arcs 99
  - Node Annotations 99
  - Arc Annotations 99
- Filtering Nodes and Arcs 99
  - Call Graph Preferences Filtering Options 99
  - Node Menu 99
  - Selected Nodes Menu 100

Filtering Nodes through the Display Controls . . . 101
Other Manipulation of the Call Graph . . . 104
Geometric Manipulation through the Control Panel . . . 104
Using the Mouse in the Call Graph View . . . 106
Selecting Nodes from the Function List . . . 106
Butterfly View . . . 106
Analyzing Memory Problems . . . 106
Using Malloc Error View, Leak View, and Malloc View . . . 107
Analyzing the Memory Map with Heap View . . . 110
Heap View Window . . . 110
Source View `malloc` Annotations . . . 113
Saving Heap View Data as Text . . . 113
The Call Stack Window . . . 114
Analyzing Working Sets . . . 116
Working Set Analysis Overview . . . 116
Working Set View . . . 119
DSO List Area . . . 120
DSO Identification Area . . . 122
Page Display Area . . . 122
Admin Menu . . . 122
Cord Analyzer . . . 123
Working Set Display Area . . . 124
Working Set Identification Area . . . 124
Page Display Area . . . 125
Function List . . . 125
Admin Menu . . . 126
File Menu . . . 126

5. Glossary 129

Index 133

Figures

Figure 1-1 Performance Analyzer Main Window 3
Figure 1-2 Typical Performance Analyzer Time Line 7
Figure 1-3 Typical Resource Usage Graphs 9
Figure 1-4 Typical Textual Usage View 11
Figure 1-5 I/O View 12
Figure 1-6 MPI Statistical Graphs 13
Figure 1-7 MPI Statistical Text 15
Figure 1-8 An Overhead View for an OpenMP Program 16
Figure 1-9 Typical Performance Analyzer Function List Area 17
Figure 1-10 Typical Performance Analyzer Call Graph 18
Figure 1-11 **Butterfly View** 19
Figure 1-12 Detailed Performance Metrics by Source Line 21
Figure 1-13 Disassembled Code with Stalled Clock Annotations 22
Figure 1-14 Typical **Heap View** Display Area 24
Figure 1-15 Typical Call Stack 25
Figure 1-16 **Working Set View** 26
Figure 1-17 Cord Analyzer 27
Figure 2-1 Performance Analyzer Main Window—`arraysum` Experiment 33
Figure 2-2 **Usage View (Graphs)**—`arraysum` Experiment 34
Figure 2-3 Significant Call Stacks in the `arraysum` Experiment 35
Figure 2-4 Function List Portion of Performance Analyzer Window 36
Figure 2-5 Butterfly Version of the **Call Graph View** 37
Figure 2-6 Viewing a Program in the **Usage View (Numerical)** Window 38

Figure 2-7 **Source View** with Performance Metrics 39
Figure 2-8 **Performance Analyzer** Window Displaying Results of a Memory Experiment 41
Figure 3-1 **Select Task** Submenu 47
Figure 4-1 Runtime Configuration Dialog Box 66
Figure 4-2 Typical Function List Area 70
Figure 4-3 Performance Analyzer **Admin** Menu 74
Figure 4-4 **Experiment** Window 75
Figure 4-5 Performance Analyzer Data Display Options 77
Figure 4-6 Performance Analyzer Sort Options 78
Figure 4-7 Performance Analyzer **Views** Menu 84
Figure 4-8 **Usage View (Graphs)** Window 86
Figure 4-9 The **Process Meter** Window with Major Menus Displayed 90
Figure 4-10 The **Usage View (Numerical)** Window 92
Figure 4-11 The **I/O View** Window 93
Figure 4-12 Overhead View 96
Figure 4-13 **Call Graph View** with Display Controls 98
Figure 4-14 Node Menus 100
Figure 4-15 **Chain** Dialog Box 102
Figure 4-16 **Prune Chains** Dialog Box 102
Figure 4-17 **Show Important Children** Dialog Box 103
Figure 4-18 **Show Important Parents** Dialog Box 104
Figure 4-19 **Call Graph View** Controls for Geometric Manipulation 105
Figure 4-20 **Malloc Error View** Window with an **Admin** Menu 108
Figure 4-21 **Leak View** Window with an **Admin** Menu 108
Figure 4-22 **Malloc View** Window with **Admin** Menu 109
Figure 4-23 **Source View** Window with Memory Analysis Annotations 109
Figure 4-24 **Heap View** Window 111

Figure 4-25 **Heap View Save Text** Dialog Boxes 114
Figure 4-26 Performance Analyzer **Call Stack** Window 115
Figure 4-27 Working Set Analysis Process 118
Figure 4-28 **Working Set View** 120
Figure 4-29 The **Cord Analyzer** Window 125

Tables

Table 4-1 Summary of Performance Analyzer Tasks 58
Table 4-2 Basic Block Counts and PC Profile Counts Compared 62
Table 4-3 Call Stack Profiling and PC Profiling Compared 66
Table 4-4 Task Display in Usage Chart Area 71

About This Guide

This publication documents the ProDev WorkShop Performance Analyzer for release 2.9, running on IRIX systems.

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

- IRIX 6.2 or higher
- MIPSpro compilers version 7.2.1 or higher
- SpeedShop 1.4

Related Publications

The following documents contain additional information that may be helpful:

- *SpeedShop User's Guide*
- *C Language Reference Manual*
- *C++ Language System Library*
- *C++ Language System Overview*
- *C++ Language System Product Reference Manual*
- *C++ Programmer's Guide*
- *ProDev ProMP User's Guide*
- *ProDev WorkShop: Debugger User's Guide*
- *Developer Magic: Static Analyzer User's Guide*
- *Developer Magic: ProDev WorkShop Overview*
- *Fortran 77 Language Reference Manual*
- *MIPSPro 7 Fortran 90 Commands and Directives Reference Manual*
- *MIPSpro Fortran Language Reference Manual, Volume 1*
- *MIPSpro Fortran Language Reference Manual, Volume 2*

- *MIPSpro Fortran Language Reference Manual, Volume 3*

Obtaining Publications

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

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

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

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

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

Conventions

The following conventions are used throughout this document:

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

...	Ellipses indicate that a preceding element can be repeated.

Reader Comments

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

- Send us electronic mail at the following address:

  `techpubs@sgi.com`

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

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

  1–800–950–2729 (toll free from the United States and Canada)
  +1–651–683–5600

- Send a facsimile of your comments to the attention of Software Publications Group in Eagan, Minnesota, at fax number +1–651–683–5599.

We value your comments and will respond to them promptly.

Introduction to the Performance Analyzer

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

- Performance Analyzer Overview, see "Performance Analyzer Overview", page 1.
- The Performance Analyzer Tools, see "The Performance Analyzer Tools", page 2.
- Sources of Performance Problems, see "Sources of Performance Problems", page 4.
- Interpreting Performance Analyzer Results, see "Interpreting Performance Analyzer Results", page 5.

Performance Analyzer Overview

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

There are three general techniques for collecting performance data:

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

The Performance Analyzer Tools

This section describes the major windows in the Performance Analyzer toolset. The main window (see Figure 1-1, page 3) contains the following major areas:

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

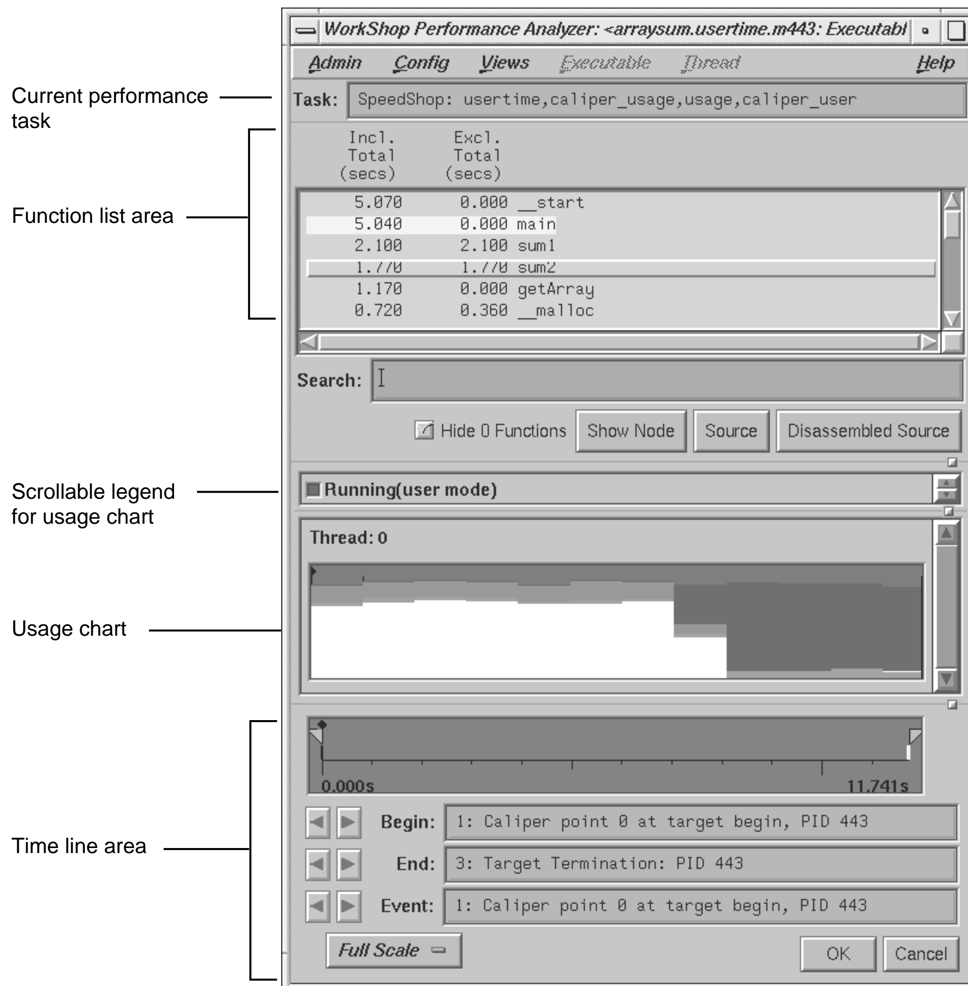

Figure 1-1 Performance Analyzer Main Window

Supplemental views bring up their own windows. For more information, see "Interpreting Performance Analyzer Results", page 5, and the subsections that follow.

Sources of Performance Problems

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

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

The following sections describe these sources of performance problems in more detail. Cache issues are mentioned in the CPU speed and availability, I/O processing, and memory size and availability descriptions.

CPU-Bound Processes

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

I/O-Bound Processes

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

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

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; cache misses result. Insufficient memory bandwidth could also be the problem.

To fix a memory-bound process, you can try to improve the memory reference patterns or, if possible, decrease the memory used by the program.

Bugs

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

Performance Phases in Programs

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

Interpreting Performance Analyzer Results

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

- The time line display shows the experiment as a set of events over time and provides caliper markers to let you specify an interval of interest. See the following section, and for more information, see "Time Line Area and Controls", page 72.
- The **Usage View (Graphs)** displays process resource usage data, such as what the program is doing at any time, the amount of data read and written, the memory

size of the program, and so on. The data is presented in the form of graphs. See "Resource Usage Graphs", page 7.

- The **Usage View (Numerical)** presents a textual display of the process and system-wide resource usage data. See "**Usage View (Numerical)**", page 10.
- The **I/O View** displays a chart of the number of bytes for each I/O transfer. See "**I/O View**", page 11.
- The **MPI Stats View (Graphs)** displays data in a graphical format for multiprocessor programs using the Message Passing Interface (MPI). See "**MPI Stats View (Graphs)**", page 12.
- The **MPI Stats View (Numerical)** displays MPI data in text format. See "**MPI Stats View (Numerical)**", page 14.
- The **Parallel Overhead View** displays the unproductive time spent in an MPI, OpenMP, or pthreads parallel program. See .
- The function list area displays the program's functions with associated performance metrics. See "The Function List Area", page 16.
- The **Call Graph View** presents the target program as nodes and arcs, along with associated metrics. See "**Call Graph View**", page 17.
- The **Butterfly View** presents a selected function along with the functions that called it and the functions that it called. See "**Butterfly View**", page 18.
- **Source View** with performance annotations, see "Source View with Performance Annotations", page 20.
- **Disassembled Source** with performance annotations, see "Disassembled Code with Performance Annotations", page 21.
- **Malloc Error View**, **Leak View**, **Malloc View**, and **Heap View**, see "Leak View, Malloc View, Malloc Error View, and Heap View", page 22.
- The **Call Stack View** shows the path through functions that led to an event. See "**Call Stack View**", page 24.
- The **Working Set View** displays a list of the DSOs in the program, with information on the efficiency of use of the text (instruction) pages. See "**Working Set View**", page 25.
- The cord analyzer lets you explore the working set behavior of an executable or DOS. See "Cord Analyzer", page 26.

The following sections describe these features in more detail.

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. Event number 4 is selected; it is labeled according to the caliper number, `third`. You can see from the graph that the phase between the selected event and event number 5 is taking more of the program's time than any of the other phases.

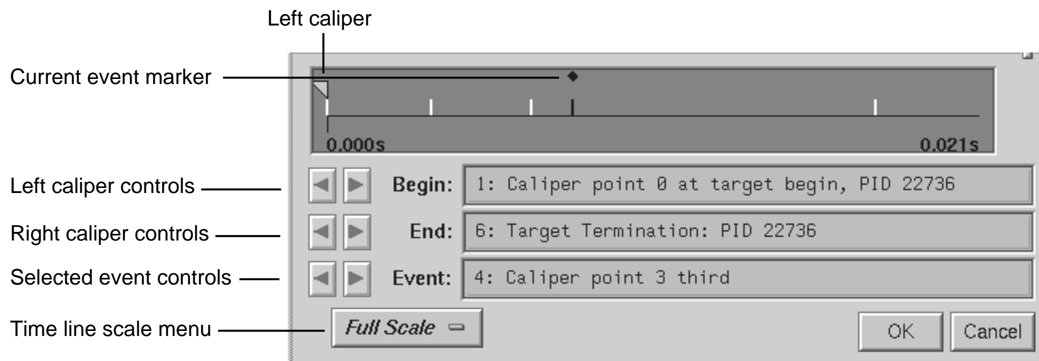

Figure 1-2 Typical Performance Analyzer Time Line

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, page 9, which shows five 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:

- The state of the program at any given time. (The states include running in user mode, running in system mode, waiting in the CPU queue, and so on.)
- Page faults.
- Context switches, or when one job is replaced in the CPU by another job.
- The size of reads and writes.
- Read and write counts.
- Poll and I/O calls. (See the `poll`(2), `ioctl`(2), and `streamio`(7) man pages for more information on what this chart measures.)
- Total system calls.
- Process signals received.
- Process size in memory.

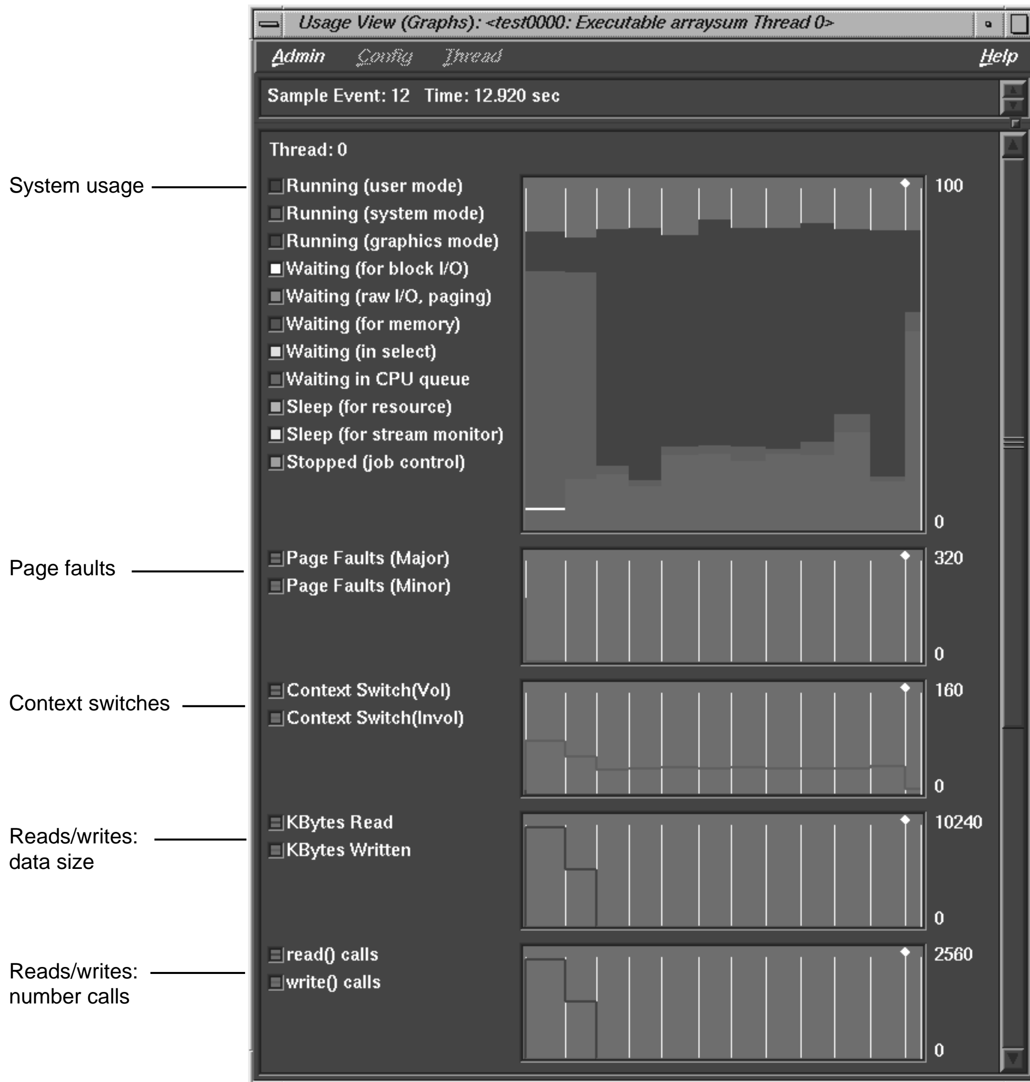

Figure 1-3 Typical Resource Usage Graphs

Resource usage data is recorded periodically: by default, every second. If you discover inconsistent behavior within a phase, you can change the interval and break the phase down into smaller phases.

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

Usage View (Numerical)

The usage graphs show the patterns; the textual usage views let you view the aggregate values for the interval specified by the time line calipers. Figure 1-4, page 11, shows a typical **Usage View (Numerical)** window.

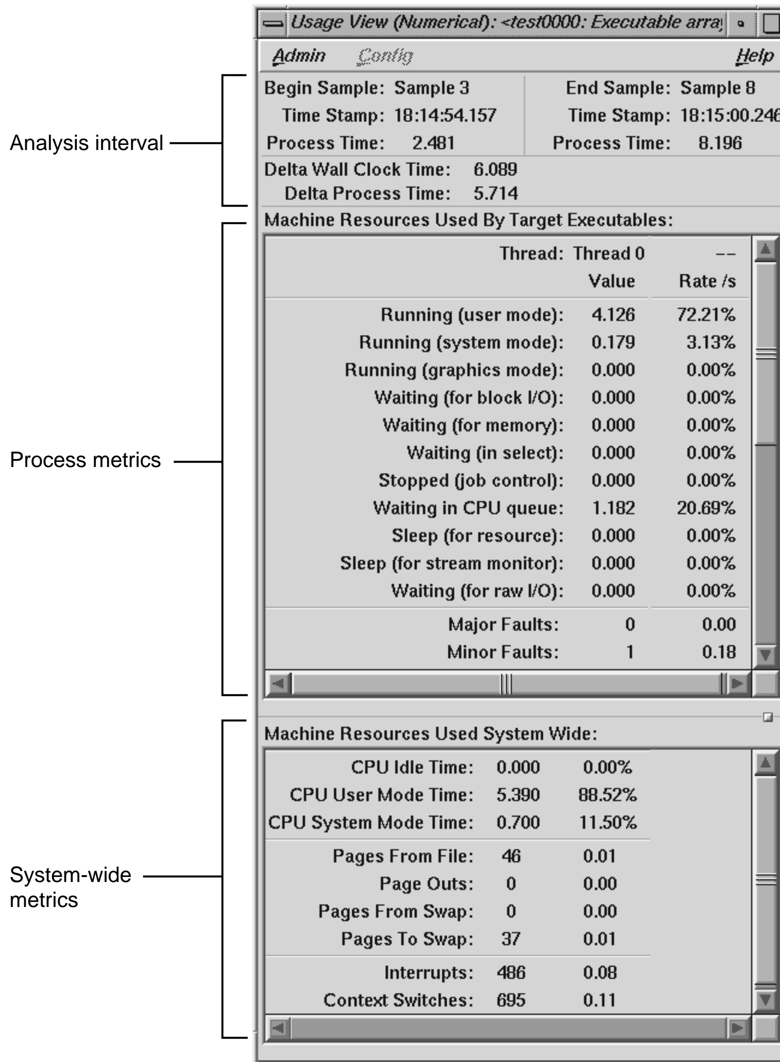

Figure 1-4 Typical Textual Usage View

I/O View

I/O View helps you determine the problems in an I/O-bound process. It produces a graph of all I/O system calls and identifies up to 10 files involved in I/O. By

selecting an event with the left mouse button, you can display the call stack corresponding to the event in the **Call Stack View**. See Figure 1-5.

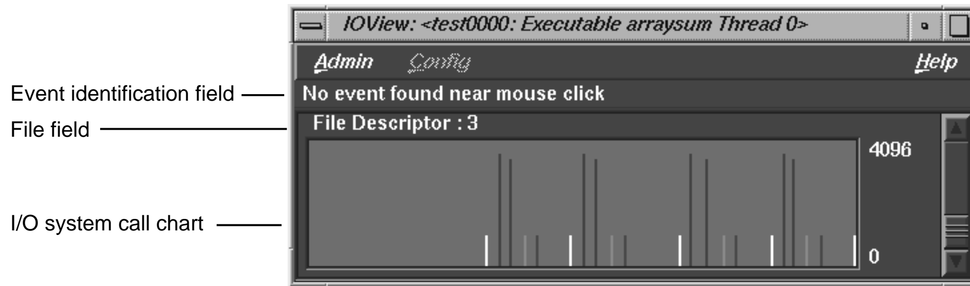

Figure 1-5 I/O View

MPI Stats View (Graphs)

If you are running a multiprocessor program that uses the Message Passing Interface (MPI), the **MPI Stats View (Graphs)** view can help you tune your program. The graphs display data from the complete program.

Both the graphs view and the numerical view (see the following section) use data collected by the MPI library and recorded by SpeedShop. Versions of the MPI library older than MPT 1.3 do not provide the data needed by these views. The MPI statistical data is recorded as part of the resource usage data, so the interval between resource usage samples is also the interval between MPI statistical samples.

The following figure shows the graphs from a large MPI program.

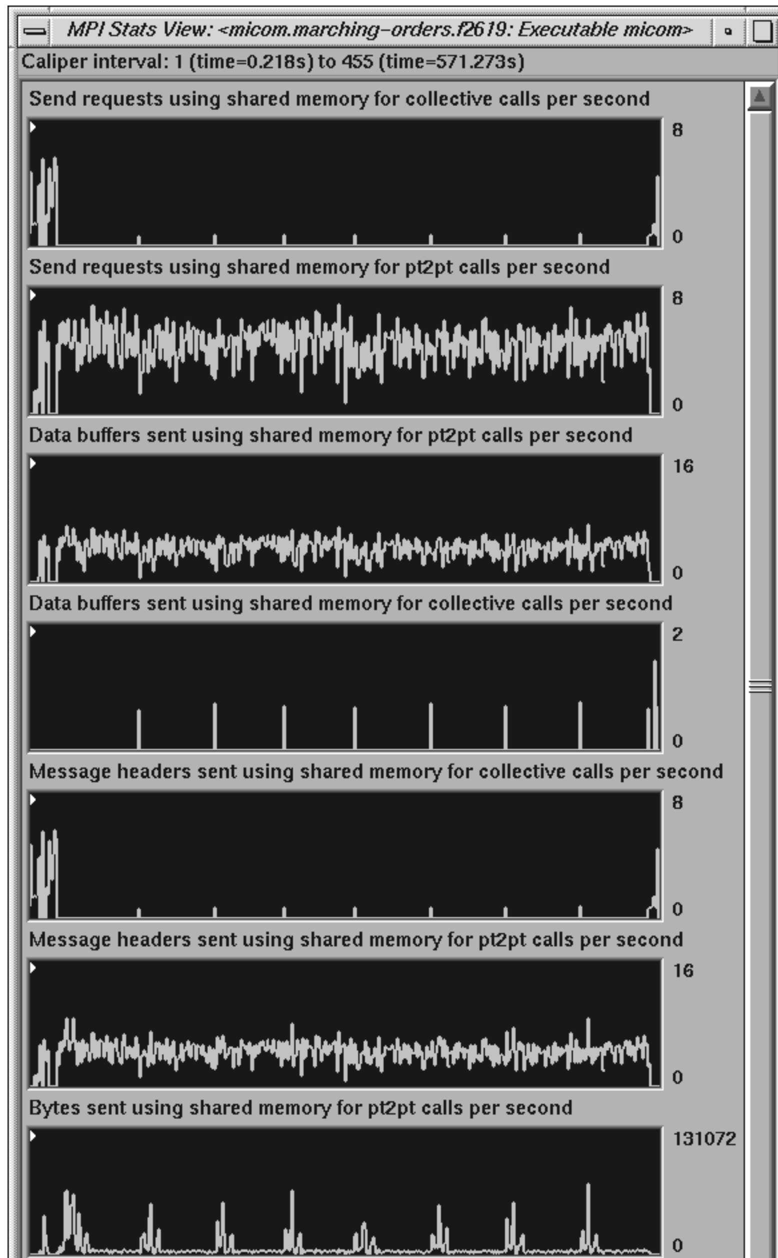

Figure 1-6 MPI Statistical Graphs

MPI Stats View (Numerical)

The **MPI Stats View (Numerical)** display gives you MPI data in text format, rather than graph format. It is a more precise measurement than the **MPI Stats View (Graphs)** display. The following figure shows the numeric version of the MPI statistics.

MPI Stats View: <micom.marching-orders.f261

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

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

Figure 1-7 MPI Statistical Text

The Parallel Overhead View

The **Parallel Overhead View** displays the overhead (or, unproductive time) spent in an MPI, OpenMP, or pthreads program. Figure 1-8, page 16 shows the overhead for an 8–processor OpenMP program. The information is drawn from all eight processors.

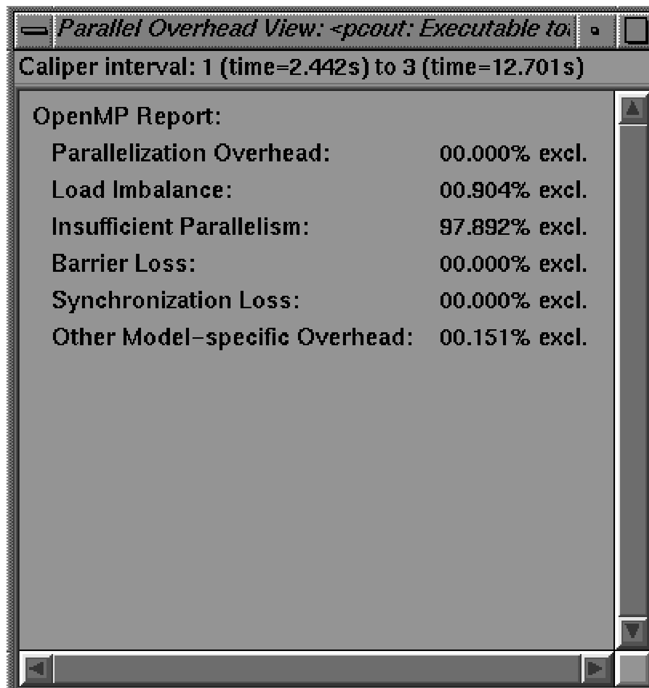

Figure 1-8 An Overhead View for an OpenMP Program

The Function List Area

The function list area 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 1-9 shows an example of the function list, ranked by inclusive CPU time.

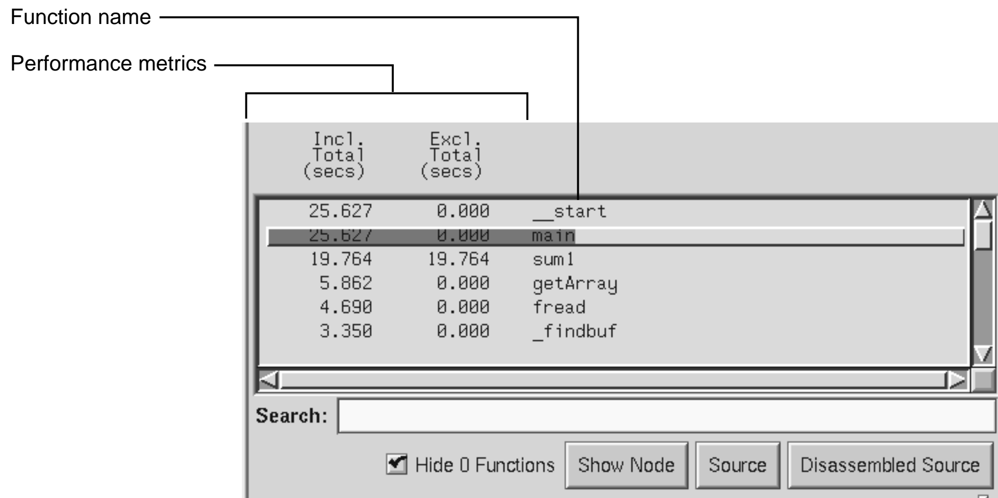

Figure 1-9 Typical Performance Analyzer Function List Area

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 the relationship between functions. The call graph displays functions as nodes and calls as arcs (displayed as lines between the nodes). The nodes are annotated with the performance metrics; the arcs come with counts by default and can include other metrics as well.

In Figure 1-10, for example, the inclusive time spent by the function `main` is 8.107 seconds. Its exclusive time was 0 seconds, meaning that the time was actually spent in called functions. The `main` function can potentially call three functions. The **Call Graph View** indicates that in the experiment, `main` called three functions: `getArray`, which consumed 1.972 seconds; `sum1`, which consumed 3.287 seconds; and `sum2`, which consumed 2.848 seconds.

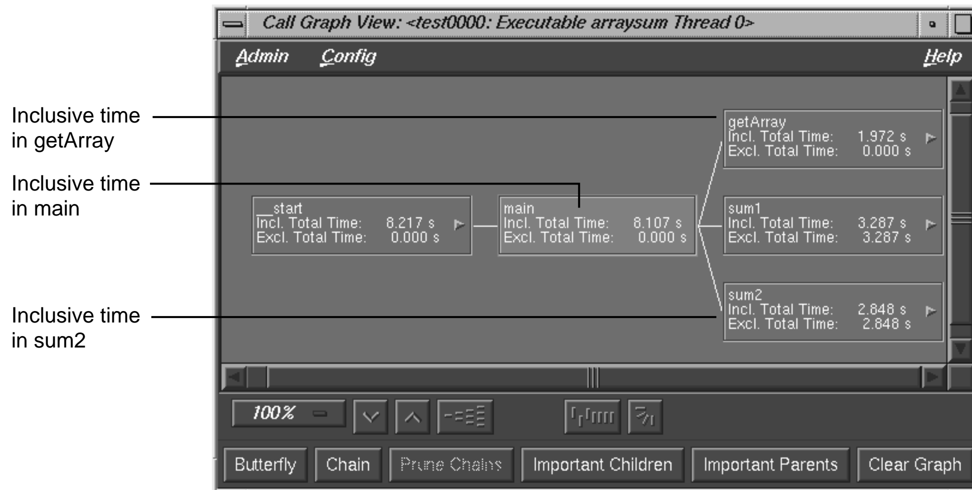

Figure 1-10 Typical Performance Analyzer Call Graph

Butterfly View

The **Butterfly View** shows a selected routine in the context of functions that called it and functions it called. For an illustration, see Figure 1-11.

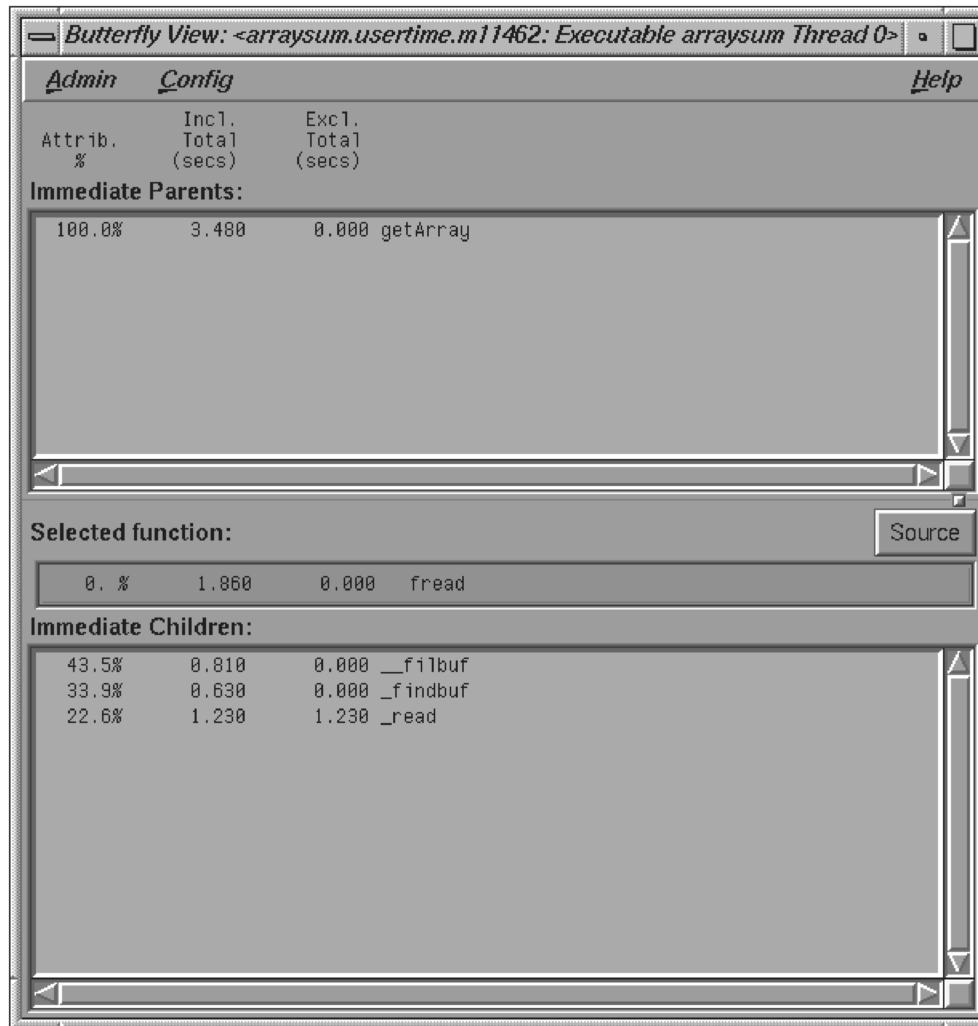

Figure 1-11 **Butterfly View**

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

The two main parts of the **Butterfly View** window identify the immediate parents and the immediate children of the selected function. In this case, the term *immediate* means they either call the selected function directly or are called by it directly.

The columns of data in the illustration show:

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

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

Source View with Performance Annotations

The Performance Analyzer lets you view performance metrics by source line in the **Source View** (see Figure 1-12, page 21) or by machine instruction in the **Disassembled Source** view. Displaying performance metrics is set in the **Preferences** dialog box, accessed from the **Display** menu in the **Source View** and **Disassembled Source** view. The Performance Analyzer sets thresholds to flag lines that consume more than 90% of a total resource. These indicators appear in the metrics column and on the scroll bar.

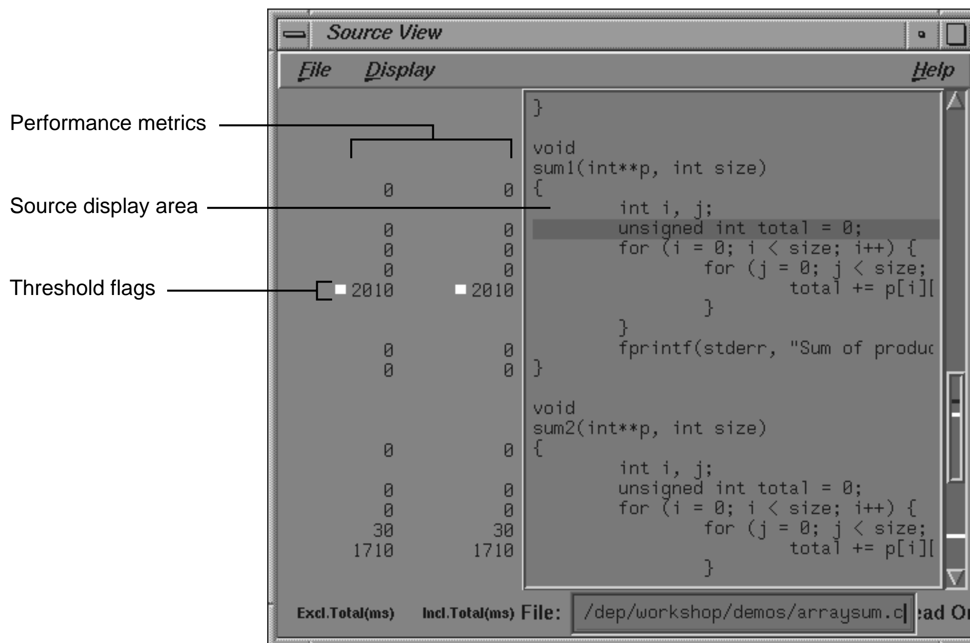

Figure 1-12 Detailed Performance Metrics by Source Line

Disassembled Code with Performance Annotations

The Performance Analyzer also lets you view performance metrics by machine instruction (see Figure 1-13, page 22). You can view any of the performance metrics that were measured in your experiment. If you ran an `Ideal Time/Pixie` experiment, you can get a special three-part annotation that provides information about stalled instructions.

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

- `B` - Branch delay

- F - Function unit delay
- O - Operand has not arrived yet

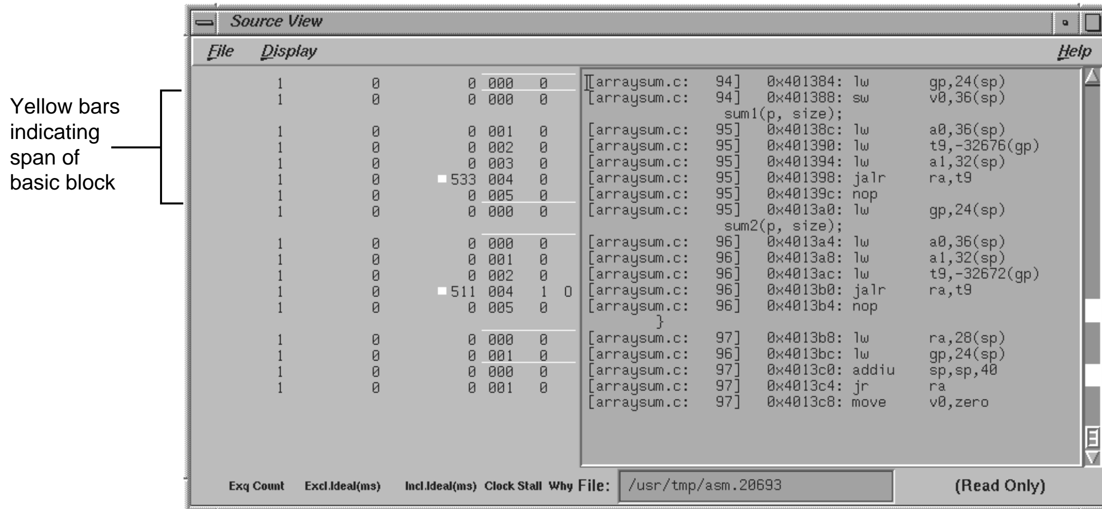

Figure 1-13 Disassembled Code with Stalled Clock Annotations

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

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

- Memory leakage, see "Memory Leakage", page 23
- Bad calls to `free`, see "Bad Frees", page 23.

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

Memory Leakage

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

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

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

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

Bad Frees

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

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

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

The **Heap View** window lets you analyze memory allocation and deallocation between selected sample events in your experiment. **Heap View** displays a memory map that indicates calls to `malloc` and `realloc`, bad deallocations, and valid deallocations during the selected period, as shown in Figure 1-14. Clicking an area in the memory map displays the address.

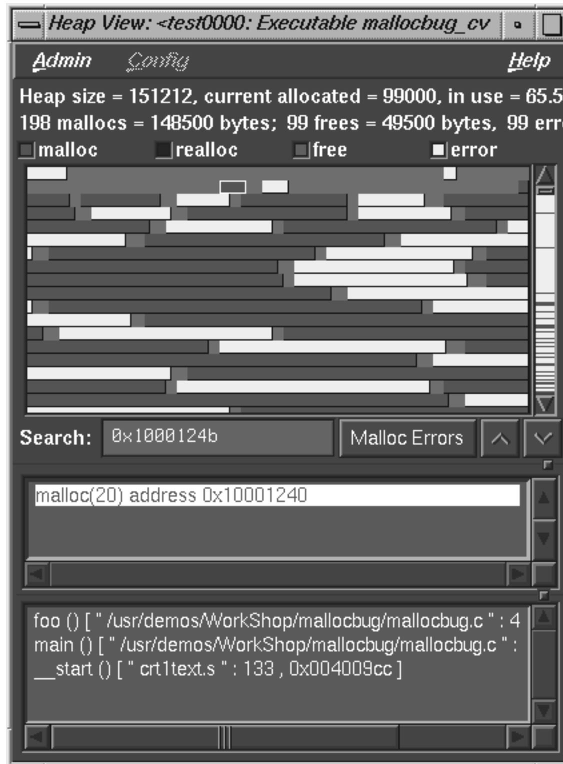

Figure 1-14 Typical **Heap View** Display Area

Call Stack View

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

back to your code. Figure 1-15 shows a typical call stack. It corresponds to sample event #3 in an experiment.

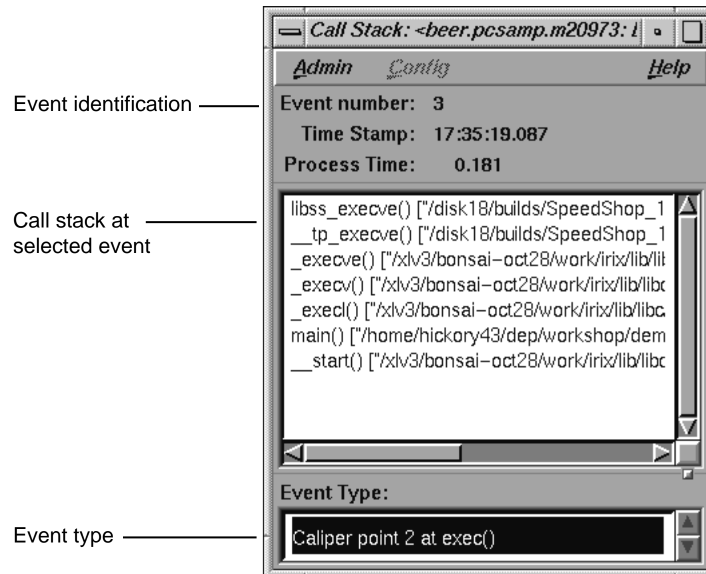

Figure 1-15 Typical Call Stack

Working Set View

Working Set View measures the coverage of the dynamic shared objects (DSOs) that make up your executable (see Figure 1-16). It indicates instructions, functions, and pages that were not used when the experiment was run. It shows the coverage results for each DSO in the DSO list area. Clicking a DSO in the list displays its pages with color coding to indicate the coverage of the page.

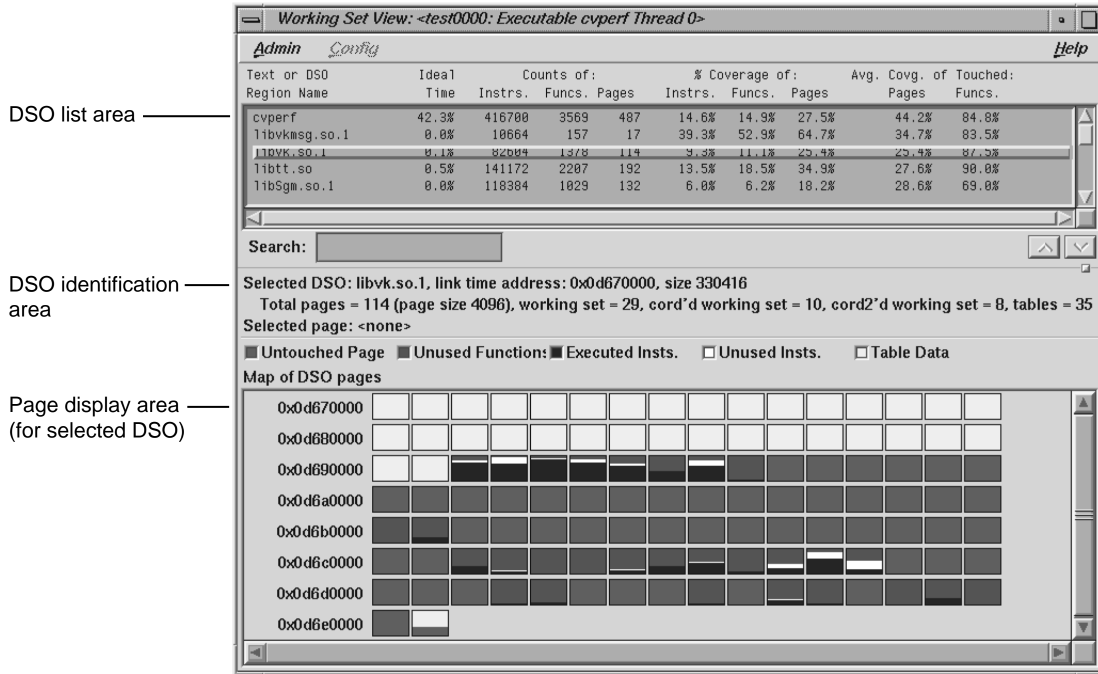

Figure 1-16 Working Set View

Cord Analyzer

The cord analyzer is not actually part of the Performance Analyzer and is invoked by typing `sscord` at the command line. The cord analyzer (see Figure 1-17) lets you explore the working set behavior of an executable or dynamic shared library (DSO). With it you can construct a feedback file for input to `cord` to generate an executable with improved working-set behavior.

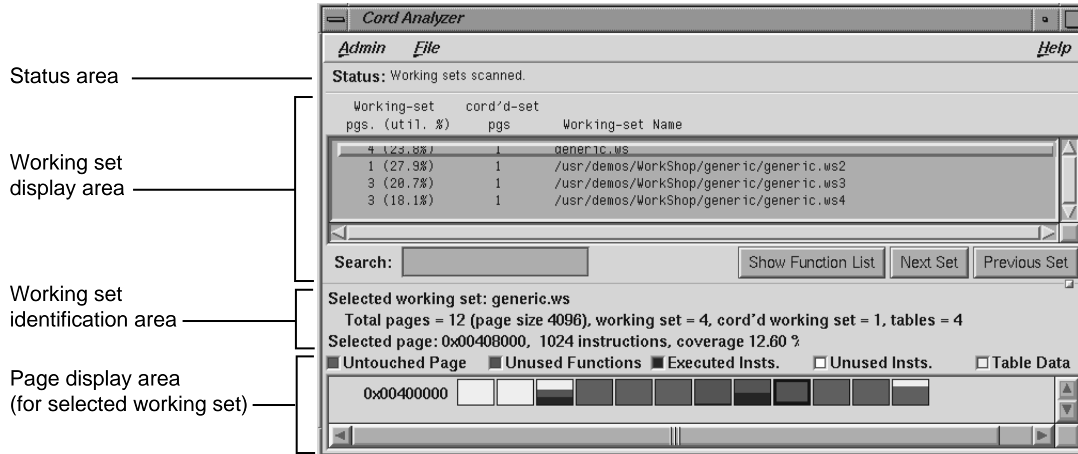

Figure 1-17 Cord Analyzer

Performance Analyzer Tutorial

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

- Tutorial Overview, see "Tutorial Overview", page 29
- Tutorial Setup, see "Tutorial Setup", page 30
- Analyzing the Performance Data, see "Analyzing the Performance Data", page 31
- Analyzing Memory through Experiments, see "Analyzing Memory Experiments", page 40.

Note: Because of inherent differences between systems and to concurrent processes that may be running on your system, your experiments will produce different results from the ones 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.

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

Tutorial Setup

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

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

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

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

   `make arraysum`

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

3. From the command line, enter the following:

   `cvd arraysum &`

   The **Debugger Main View** window is displayed. You need the Debugger to specify the data to be collected and to run the experiment. (If you want to change the font in a WorkShop window, see "Changing Window Font Size", page 31.)

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

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

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

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

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

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

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

Changing Window Font Size

If you want to change the font size on a WorkShop window, you can do so in your .Xresources or .Xdefaults file. Follow this procedure:

1. Enter the command editres(1) to get the names of the WorkShop window widgets.

2. Add lines such as the following to your .Xresources or .Xdefaults file:

```
cvmain*fontList:  6x13

cvmain*tabPanel*fontList:  fixed

cvmain*popup_optionMenu*fontList:  fixed
cvmain*canvasPopup*fontList:  6x13

cvmain*tabLabel.fontList:  6x13

cvmain*help*fontList:  6x13
cvmain*UiOverWindowLabel*fontList:  6x13
cvmp*fontList:  6x13
```

   The first changes the main window font, and the others change fonts more selectively.

3. Enter the command xrdb(1) to update the windows.

Analyzing the Performance Data

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

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

---

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

   The Performance Analyzer window now displays the information from the new experiment (see Figure 2-1, page 33).

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

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

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

   The **Usage View (Graphs)** window displays as in Figure 2-2, page 34. It shows high read activity and high system calls in the first phase, confirming our hypothesis that it is I/O-intensive.

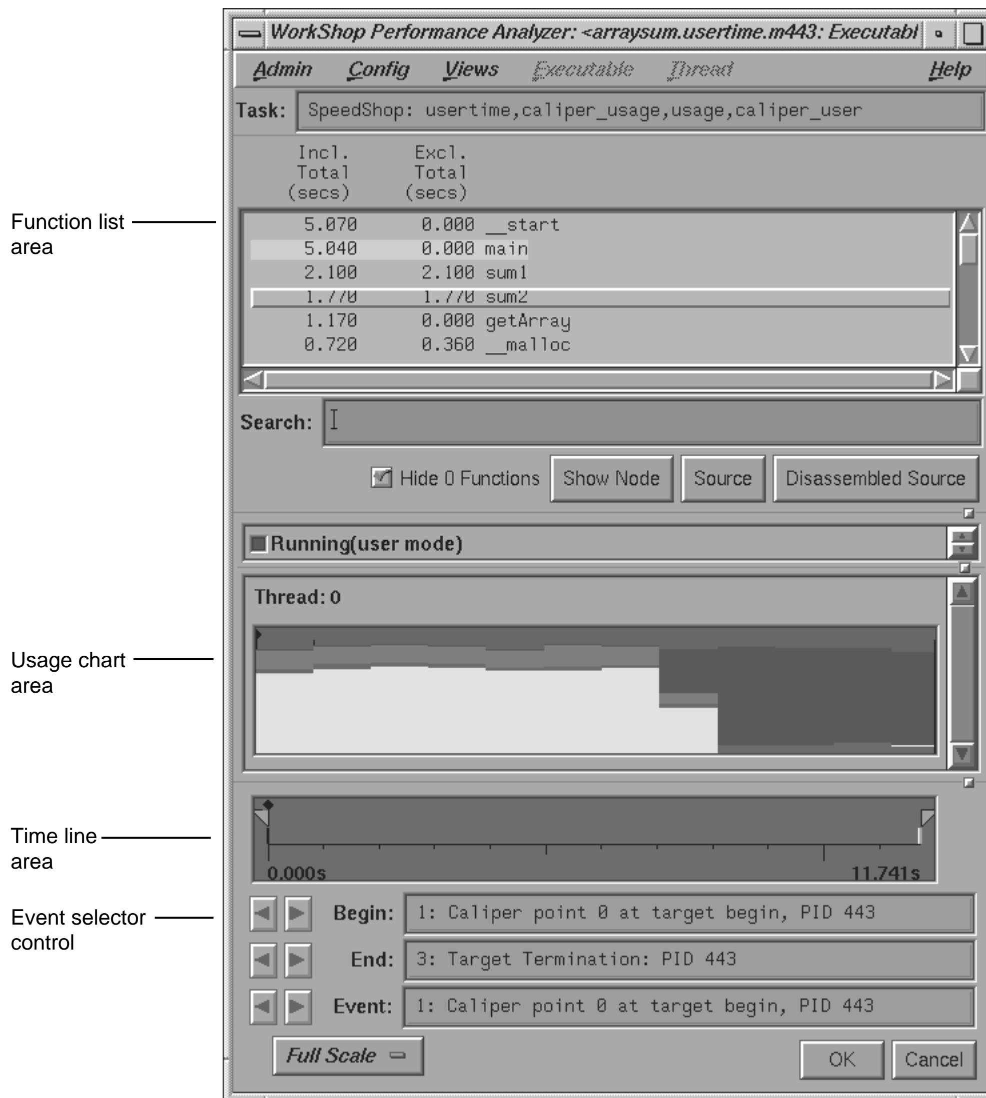

Figure 2-1 Performance Analyzer Main Window—`arraysum` Experiment

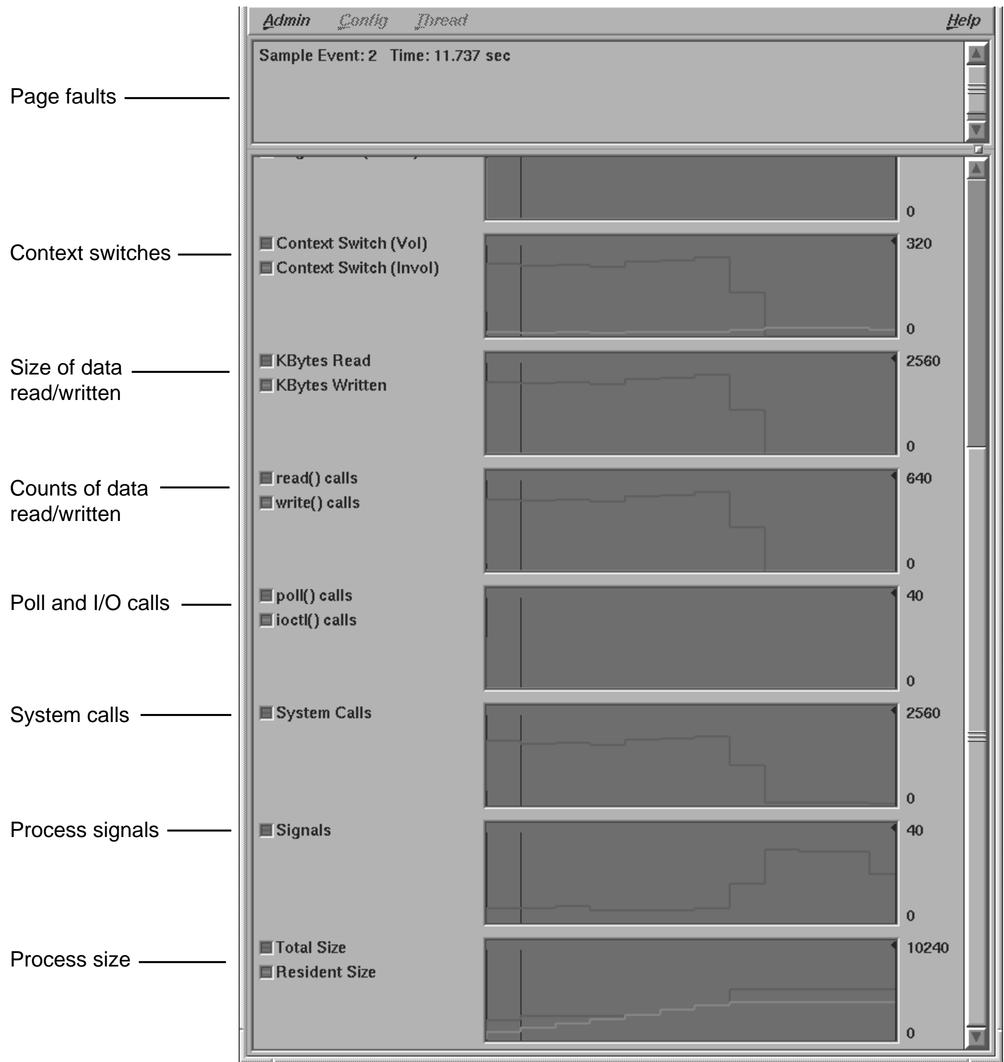

Figure 2-2 Usage View (Graphs)—`arraysum` Experiment

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

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

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

   At this point, no events have been selected so the call stack is empty. To define events, you can add calls to `ssrt_caliper_point` to record caliper points in the source file, set a sample trap from the WorkShop Debugger window, or set pollpoint calipers on the time line. (For more information on the `ssrt_caliber_point` function, see the `ssapi`(3) man page.) See Figure 2-3 for an illustration of how the **Call Stack View** responds when various caliper points are recorded.

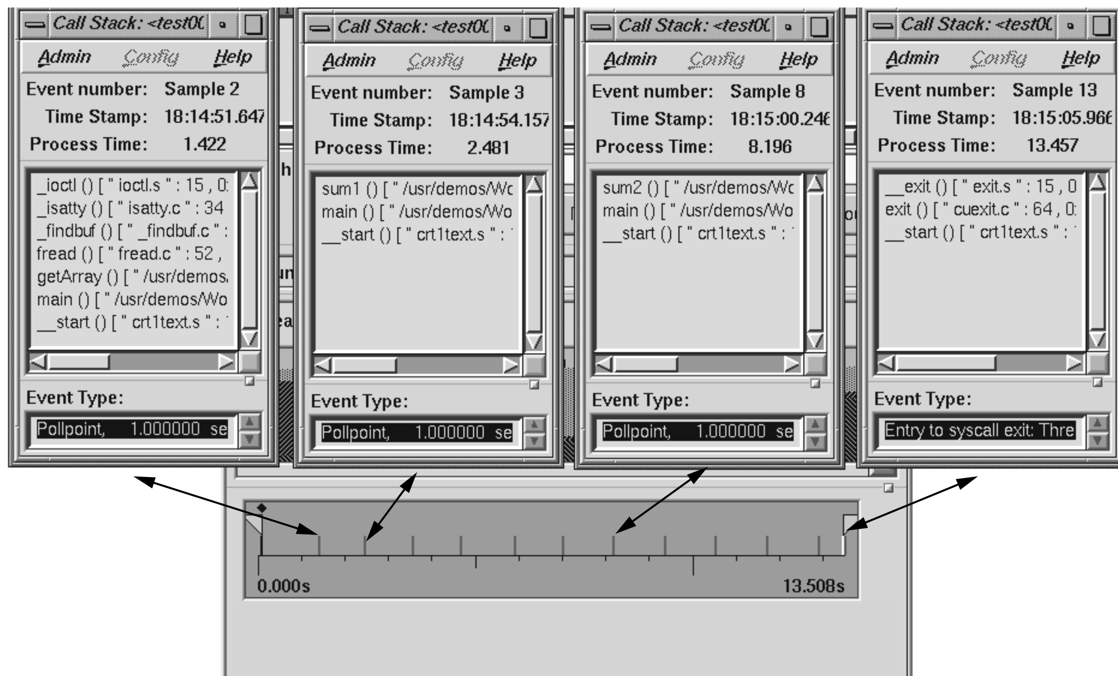

Figure 2-3 Significant Call Stacks in the `arraysum` Experiment

5. Return to the Performance Analyzer window and pull down the sash to expose the complete function list (see Figure 2-4).

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

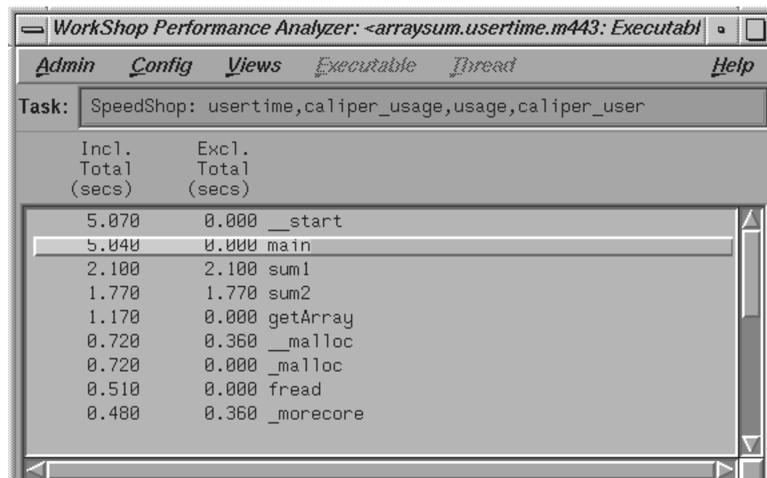

Figure 2-4 Function List Portion of Performance Analyzer Window

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

   The call graph provides an alternate means of viewing function performance data. It also shows relationships, that is, which functions call which functions. After the **Butterfly** button is clicked, the **Call Graph View** window appears, as shown in Figure 2-5, page 37. The **Butterfly** button takes the selected function (or most active function if none is selected) and displays it with the functions that call it and those that it calls.

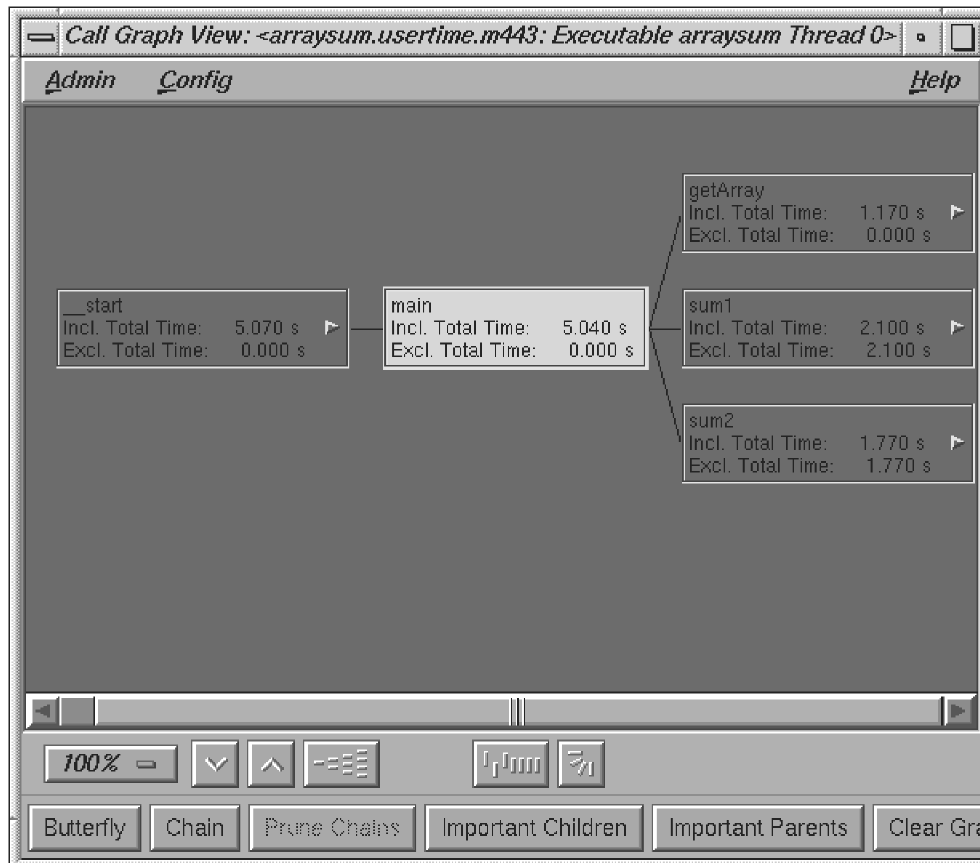

Figure 2-5 Butterfly Version of the **Call Graph View**

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

   The **Usage View (Numerical)** window appears as shown in Figure 2-6, page 38.

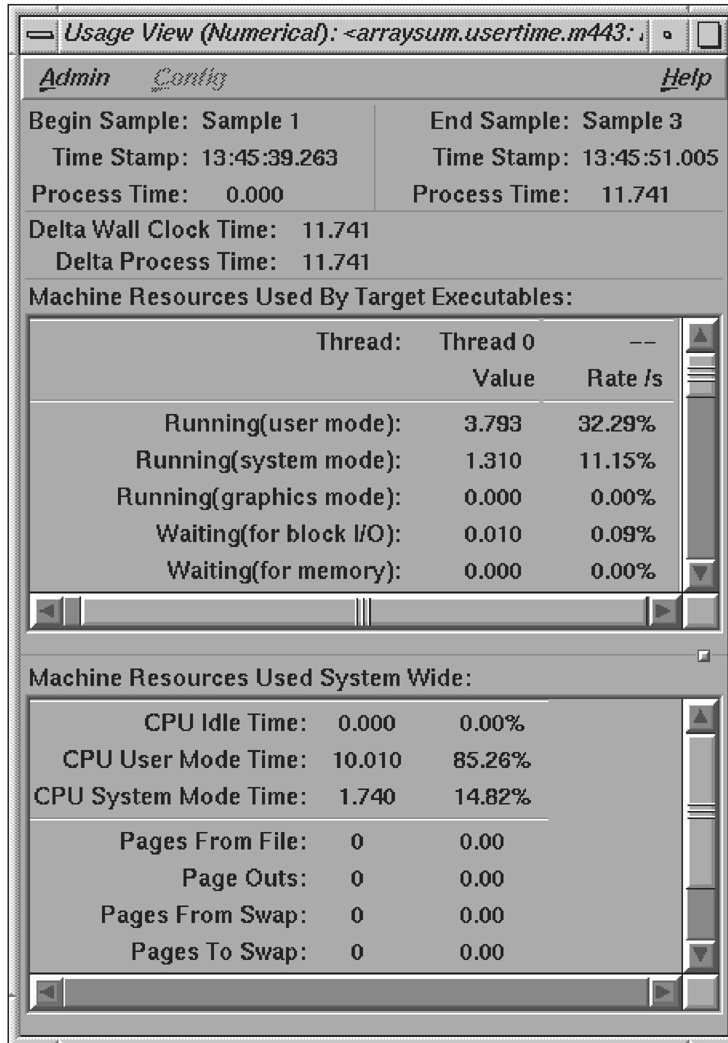

Figure 2-6 Viewing a Program in the **Usage View (Numerical)** Window

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

   The **Source View** window displays as shown in Figure 2-7, page 39, scrolled to `sum1`, the selected function. The annotation column to the left of the display area

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

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

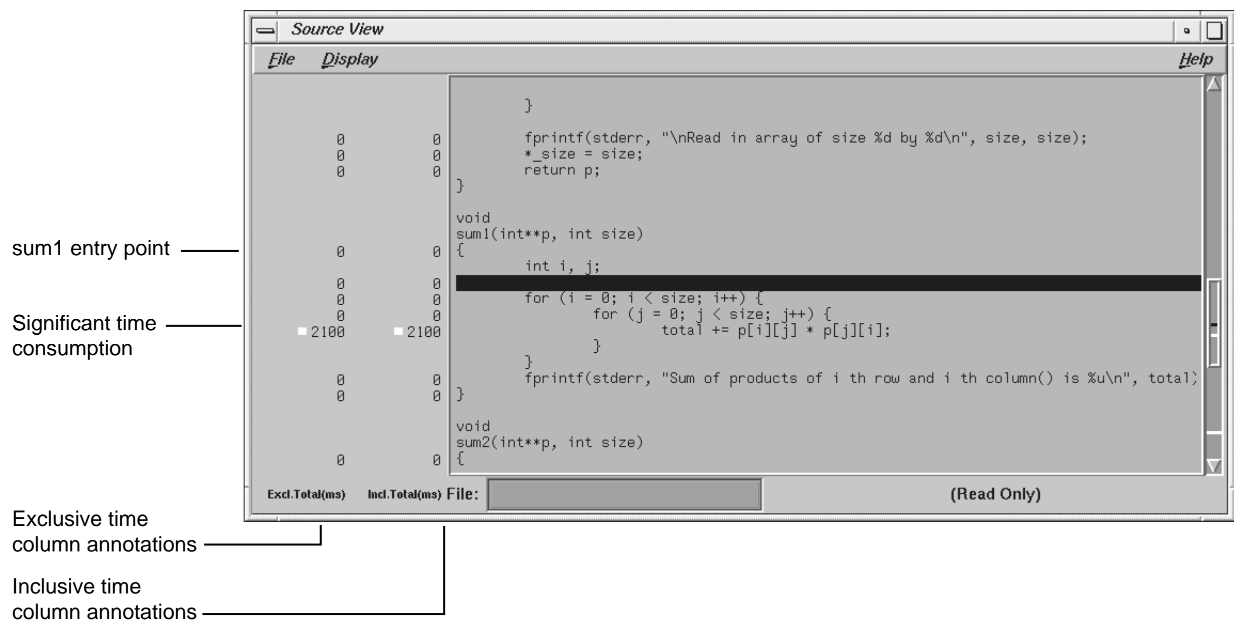

Figure 2-7 Source View with Performance Metrics

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

10. Close any windows that are still open.

This concludes the tutorial.

Analyzing Memory Experiments

Memory experiments give you information on what kinds of memory errors are happening in your program and where they are occurring.

The first tutorial finds memory leaks, situations in which memory allocations are not matched by deallocations.

The second tutorial (see "Memory Use Tutorial", page 42) analyzes memory use.

Finding Memory Leaks

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

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

   `cvd generic &`

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

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

3. Run the experiment.

   You run experiments by clicking the **Run** button.

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

   The **Performance Analyzer** window displays results appropriate to the task selected. Figure 2-8, page 41, shows the **Performance Analyzer** window after a memory experiment.

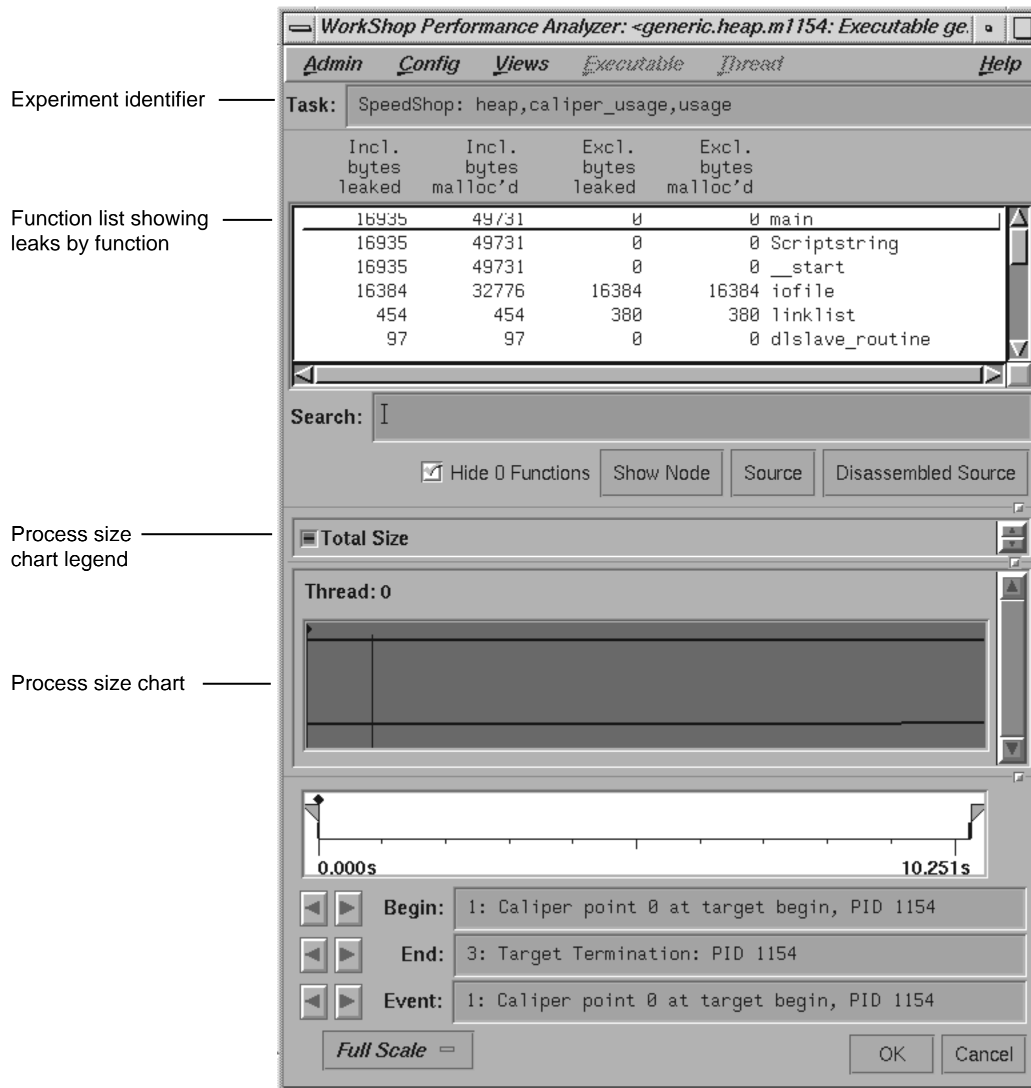

Figure 2-8 Performance Analyzer Window Displaying Results of a Memory Experiment

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

You can set other annotations in **Source View** and the function list by choosing **Preferences...** from the **Config** menu in the **Performance Analyzer** window and selecting the desired items.

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

Memory Use Tutorial

In this tutorial, you will run an experiment to analyze memory use. The program generates memory problems that you can detect using the Performance Analyzer and the following instructions:

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

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

2. Invoke the Debugger by typing:

   ```
   cvd mallocbug
   ```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

14. Close the **Source View** window.

15. Click the **Malloc Errors** button.

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

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

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, see "Experiment Setup Overview", page 45.
- Selecting a Performance Task, see "Selecting a Performance Task", page 46.
- Setting Sample Traps, see "Setting Sample Traps", page 47.
- Understanding Predefined Tasks, see "Understanding Predefined Tasks", page 48.
- Displaying Data from the Parallel Analyzer, see "Displaying Data from the Parallel Analyzer", page 55.

Experiment Setup Overview

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

Each experiment has these steps:

1. Specify the performance task.

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

   The predefined tasks ensure that only the appropriate data collection is enabled. Selecting too much data can bog down the experiment and skew the data for

collection. If you need a mix of performance data not available in the predefined tasks, you can select **Custom** from the **Select Task** submenu. It lets you enable combinations of the data collection options.

2. Specify where to capture the data.

   If you want to gather information for the complete program, this step is not needed. If you want data at specific points in the process, you need to set sample traps. See "Setting Sample Traps", page 47, for a brief description of traps or Chapter 5, “Setting Traps,” in the *Developer Magic: Debugger User’s Guide* for an in-depth discussion.

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

3. Specify the experiment configuration parameters.

   This step is not necessary if you use the defaults; if you want to make configuration changes, select **Configs** from the **Perf** menu. The dialog box lets you specify a number of configuration options, many of which depend on the experiment you plan to run. The dialog box in Figure 4-1, page 66, shows the runtime configuration choices, and the options are described in "Specifying the Experiment Configuration", page 66.

4. Run the program to collect the data.

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

5. Analyze the results.

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

Selecting a Performance Task

To set up a Performance Analyzer experiment, choose a task from the **Select Task** submenu in the **Perf** menu in the **WorkShop Debugger** window (see Figure 3-1, page 47).

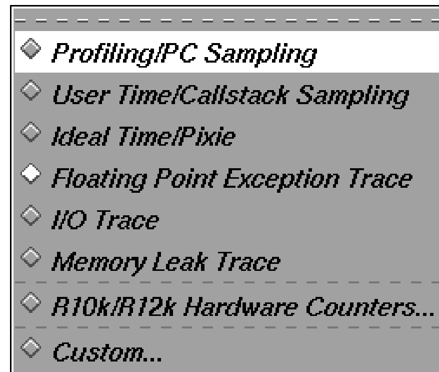

Figure 3-1 Select Task Submenu

Selecting a task enables data collection. The mode indicator in the upper right corner of the **WorkShop Debugger** window changes from **Debug Only** to **Performance**.

Setting Sample Traps

Sample traps allow you to record data when a specified condition occurs. You set traps from the **WorkShop Debugger** window: choose either the **Trap Manager** or the **Source View** from the **Views** menu. For a thorough discussion of setting traps, see Chapter 5, "Setting Traps," in the *Developer Magic: Debugger User's Guide*.

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

You can define sample traps:

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

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

Manual sample traps are triggered when you click the **Sample** button in the **WorkShop Debugger**. 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. Doing so can slow down the Performance Analyzer and skew results. To help you record data appropriate to your current objective, WorkShop provides predefined combinations of tasks, which are available in the **Select Task** submenu in the **Perf** menu (see Figure 3-1, page 47). These tasks are described in the following sections. When you select a task, the required data collection is automatically enabled.

Profiling/PC Sampling

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

- By function in the function list
- By source line in **Source View**
- By instruction in **Disassembly View**
- By machine resource usage data, captured at 1-second intervals and at sample points

This task gathers data by sampling the program counter (PC) value every 10 milleseconds (ms).

User Time/Callstack Sampling

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

- The call stack every 30 milleseconds (ms)
- Machine resource usage data at 1-second intervals and at sample points

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

- Attribute exclusive user time to the function at the bottom of each call stack (that is, the function being executed at the time of the sample).
- Attribute inclusive user time to all the functions above the one currently being executed.

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

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

Ideal Time/Pixie

Use the **Ideal Time/Pixie** task selection to tune a CPU-bound phase. The name *ideal time* is a historical name first used when processors would execute instructions in a more linear manner than the current modern processors do. Ideal time experiments would represent the best possible performance for your program.

The analysis determines the cost on a per-basic block basis; it does not deal with data dependencies between basic blocks. (A basic blocks is a set of instructions with a single entry point, a single exit point, and no branches into or out of the instructions.) It is useful when used in conjunction with the **Profiling/PC Sampling** task. Comparing the two lets you examine actual versus ideal time. The difference is the time spent as a result of the following:

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

This task records the following:

- Basic block counts
- Counts of branches taken
- Machine resource usage data at 1-second intervals and at sample points
- Function pointer traces with counts

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

- 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 because of data and functional unit *interlocks*. (Interlocks are situations caused when resources, such as data, are not available.)

The task requires instrumentation of the target executable. Counter code is inserted at the beginning of each basic block.

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

While ideal time creates a complete call graph and points out where the program spends the most time if the processor was linear in execution of instructions, it is best to run a PC sampling experiment also and compare the results.

The following is a typical case of why ideal time is not enough to learn about application performance. Because ideal only knows about instructions executed and the number of times a cycle has been executed, the loop

```
DO i=1,size
   DO j=1,size
      DO k=1,size
         u(i,j) = u(i,j) + v(i,k)*w(k,j)
      END DO
   END DO
END DO
```

would be the same for ideal as

```
    DO j=1,size
DO i=1,size
         u(i,j) = u(i,j) + v(i,k)*w(k,j)
      END DO
   END DO
END DO
```

and the same as

```
       DO k=1,size
   DO j=1,size
DO i=1,size
         u(i,j) = u(i,j) + v(i,k)*w(k,j)
      END DO
   END DO
END DO
```

But if you run it, just by using time or timex, you will see big differences.

Remember that ideal knows nothing about software pipelining, memory, page faults, caches, etc. only plain instructions executed one after another. It just counts the number of times each instruction was executed.

So, accessing `u(i,j)` or `u(j,i)` is the same for ideal, but it can make a big difference for the application runtime performance.

Floating-Point Exception Trace

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

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

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

I/O Trace

Use the **I/O Trace** task selection when your program is being slowed down by I/O calls, and you want to find the responsible code. This task records call stacks at every `read`(2), `write`(2), `readv`(2), `writev`(2), `open`(2), `close`(2), `pipe`(2), `dup`(2), and `creat`(2) system call. It also records file descriptor information and the number of bytes read or written.

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

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

Memory Leak Trace

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

- By function in the function list

- By source line in the **Source View**
- By instruction in the **Disassembly View**

R10000 and R12000 Hardware Counters

If you are running your application on a system using either the R10000 or the R12000 series CPU, you can use the **R10k/R12k Hardware Counters** task selection from the `WorkShop Debugger` window once you have focused in on the source of your problem. This task gives low-level, detailed information about hardware events. It counts the following events:

- Graduated instructions. The graduated instruction counter is incremented by the number of instructions that were graduated on the previous cycle.
- Machine cycles. The counter is incremented on each clock cycle.
- Primary instruction cache misses. This counter is incremented one cycle after an instruction fetch request is entered into the miss handling table.
- Secondary instruction cache misses. This counter is incremented after the last 16-byte block of a 64-byte primary instruction cache line is written into the instruction cache.
- Primary data cache misses. This counter is incremented on the cycle after a primary cache data refill is begun.
- Secondary data cache misses. This counter is incremented on the cycle after the second 16-byte block of a primary data cache line is written into the data cache.
- TLB (task lookaside buffer) misses. This counter is incremented on the cycle after the TLB mishandler is invoked.
- Graduated floating-point instructions. This counter is incremented by the number of floating-point instructions that graduated on the previous cycle.
- Failed store conditionals.

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

You can generate other hardware counter experiments by using the `ssrun` command. See the `ssrun`(1) man page or the *SpeedShop User's Guide* for more information.

Custom

Use the **Custom** task selection when you need to collect a combination of performance data that is not available through the predefined tasks. Selecting **Custom** brings up the same tab panel screen displayed by the **Configs...** selection (see Figure 4-1, page 66).

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

- Sampling data. This includes profiling intervals, counter size, and whether `rld`(1) will be involved in data collection.
- Tracing data. This includes `malloc` and `free` trace, I/O system call trace, and floating-point exception trace.
- Recording intervals. This includes the frequency of data recording for usage data or usage or call stack data at caliper points. You can also specify this with marching orders. (For more information on marching orders, see the `ssrun`(1) man page.)
- Call stack. This includes sampling intervals and the type of timing.
- Ideal experiments. This specifies whether or not the basic block count data is collected. It also builds a complete call graph. See "Ideal Time/Pixie", page 49 for more information.
- Hardware counter specification. This specifies the hardware event you want to count, the counter overflow value, and the profiling style (PC or call stack). Hardware counter experiments are possible only on R10000 and R12000 systems.
- Runtime. This specifies the same as those listed for the **Configs** menu selection. See "Specifying the Experiment Configuration", page 66.

Remember the basic warnings in this chapter about collecting data:

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

Displaying Data from the Parallel Analyzer

The Performance Analyzer can also display data that has been parallelized for execution on a multiprocessor system. It supports Fortran 77, Fortran 90, C, and C++ with either of the following parallelizing models:

- The automatic parallelization performed by the compilers. This is enabled by including the `-apo` option on the compiler command line. For more information on automatic parallelization, see the *MIPSpro Auto-Parallelizing Option Programmer's Guide*.
- OpenMP, a set of compiler pragmas or directives, library routines, and environment variables that help you distribute loop iterations and data among multiple processors.

ProDev ProMP is a companion product to the WorkShop suite of tools. It specifically analyzes a program that has been parallelized. It is integrated with WorkShop to let you examine a program's loops in conjunction with a performance experiment on either a single processor or multiprocessor run. (For more information, see the *ProDev ProMP User's Guide*.)

The `cvpav`(1) command reads and displays analysis files generated by the MIPSpro compilers. When you plan to view one of these files in the Performance Analyzer, use the `-e` option to `cvpav`, and specify the program executable as the argument, as follows:

```
% cvpav -e a.out
```

From the Parallel Analyzer user interface, choose the **Admin** –> **Launch Tool** –> **Performance Analyzer** menu item. Once the new window comes up, choose **Excl. Percentage** from the **Sort...** window under the **Config** menu. Doing so will list the loops in order, with the most expensive at the top, allowing you to concentrate your attention on the most compute-intensive loops.

Performance Analyzer Reference

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

- Selecting performance tasks, see "Selecting Performance Tasks", page 58.
- Specifying a custom task, see "Specifying a Custom Task", page 59.
- Specifying the experiment configuration, see "Specifying the Experiment Configuration", page 66.
- The **WorkShop Performance Analyzer** main window, see "The Performance Analyzer Main Window", page 68.
- **Usage View (Graphs)**, see "Usage View (Graphs)", page 85.
- **Process Meter**, see "The Process Meter Window", page 89.
- **Usage View (Numerical)**, see "Usage View (Numerical) Window", page 91.
- **I/O View**, see "The I/O View Window", page 93.
- **MPI Stats View (Graphs)**, see "The MPI Stats View (Graphs) Window", page 94.
- **MPI Stats View (Numerical)**, see "The MPI Stats View (Numerical) Window", page 96.
- **Parallel Overhead View**, see "The Parallel Overhead View Window", page 96.
- **Call Graph View**, see "The Call Graph View Window", page 97.
- **Butterfly View**, see "Butterfly View", page 106.
- Analyzing memory problems, see "Analyzing Memory Problems", page 106.
- **Call Stack View**, see "The Call Stack Window", page 114.
- Analyzing working sets, see "Analyzing Working Sets", page 116.

Selecting Performance Tasks

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

The tasks are summarized in Table 4-1, page 58. The Task column identifies the task as it appears in the **Select Task** menu of the WorkShop Debugger's **Perf** menu. The Clues column provides an indication of symptoms and situations appropriate for the task. The Data Collected column indicates performance data set by the task. Note that call stacks are collected automatically at sample points, poll points, and process events. The Description column describes the technique used.

Table 4-1 Summary of Performance Analyzer Tasks

Task	Clues	Data Collected	Description
Profiling/PC Sampling	CPU-bound	• PC profile counts • Fine-grained usage (1 sec.) • Call stacks	Tracks CPU time spent in functions, source code lines, and instructions. Useful for CPU-bound conditions. CPU time metrics help you separate CPU-bound from non-CPU-bound instructions.
User Time/Call-stack Sampling	Not CPU-bound	• Fine-grained usage (1 sec.) • Call stack profiling (30 ms) • Call stacks	Tracks the user time spent by function, source code line, and instruction.
Ideal Time/Pixie	CPU-bound	• Basic block counts • Fine-grained usage (1 sec.) • Call stacks	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 experiment that identifies high CPU time.

Task	Clues	Data Collected	Description
Floating Point Exception Trace	High system time in usage charts; presence of floating point operations; NaNs	• FPE exception trace • Fine-grained usage (1 sec.) • Call stacks	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.
I/O trace	Process blocking due to I/O	• I/O system call trace • Fine-grained usage (1 sec.) • Call stacks	Captures call stacks at every I/O-oriented system call. The file description and number of bytes are available in I/O View.
Memory Leak Trace	Swelling in process size	• `malloc/free` trace • Fine-grained usage (1 sec.) • Call stacks	Determines memory leaks by capturing the call stack, address, and size at all `malloc`, `realloc`, and `free` routines and displays them in a memory map. Also indicates double `free` routines.
R10k/R12k Hardware Counters...	Need more detailed information	• Wide range of hardware-level counts	On R10000 and R12000 systems only, returns low-level information by counting hardware events in special registers. An overflow value is assigned to the relevant counter. The number of overflows is returned.
Custom...		• Call stacks • 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...** from the **Select Task** submenu in the **Perf** menu in the Main View, a dialog box appears. This section provides an explanation of most of the windows involved in setting up a custom task.

The **Custom...Runtime** and **HWC Spec** (the hardware counters) windows are identical to the **Configs...Runtime** and **HWC Spec** windows. For an illustration of **Runtime**, see Figure 4-1, page 66. For information on **HWC Spec**, see "R10000 and R12000 Hardware Counters", page 53.

Specifying Data to be Collected

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

- Call stack (the **CallStack** window). See the following section.
- Basic block counts (the **Ideal** window). See "Basic Block Count Sampling", page 61.
- PC profile counts (the **PC Sampling** window). See "PC Profile Counts", page 61.

Call Stack Profiling

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

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

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

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

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

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

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

Note: The experiment may run slowly in programs with very deep call stacks and many DSOs. In such cases, increasing the sampling interval will help.

Basic Block Count Sampling

Basic block counts are translated to *ideal* CPU time (as shown in the SpeedShop `ideal` experiment) and are displayed at the function, source line, and machine line levels. The experiment uses the number of cycles for each instruction and other resources present within the type of processor being used for the experiment in calculating *ideal* CPU time. Actual time usage will be different.

See "Ideal Time/Pixie", page 49 for more information.

Memory loads and stores are assumed to take constant time, so if the program has a large number of cache misses, the actual execution time will be longer than that calculated by the ideal experiment.

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

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

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

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

PC Profile Counts

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

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

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

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

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

A comparison between basic block counts and PC profile counts is shown in Table 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	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` and `free` Heap Analysis, see "`malloc` and `free` Heap Analysis", page 63.
- I/O (`read`, `write`) Operations, see "I/O Operations", page 63.
- Floating-Point Exceptions, see "Floating-Point Exceptions", page 63.
- Message Passing Interface (MPI) Stats Trace, see "MPI Stats Trace", page 63.

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` and `free` Heap Analysis

Tracing `malloc` and `free` allows you to study your program's use of dynamic storage and to quickly detect memory leaks (`malloc` routines without corresponding `free` routines) and bad `free` routines (freeing a previously freed pointer). This data can be analyzed in the **Malloc Error View**, **Leak View**, **Malloc View**, and **Heap View** (see "Analyzing Memory Problems", page 106).

I/O Operations

I/O tracing records every I/O-related system call that is made during the experiment. It traces `read`(2), `write`(2), `readv`(2), `writev`(2), `open`(2), `close`(2), `dup`(2), `pipe`(2), and `creat`(2), along with the call stack at the time, and the number of bytes read or written. This is useful for I/O-bound processes.

Floating-Point Exceptions

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

The floating-point exceptions are as follows:

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

MPI Stats Trace

MPI tracing lets you track message-passing activity in any process of a multiprocessing job. You can view the results in the **Performance Analyzer** window

with either the **MPI Stats View (Graphs)** or **MPI Stats View (Numerical)** selections from the **Views** menu. For examples, see "**MPI Stats View (Graphs)**", page 12 and "**MPI Stats View (Numerical)**", page 14.

Unlike other performance tasks, this one cannot be initiated from the Debugger View; use the SpeedShop `ssrun`(1) command in combination with the `mpirun`(1) command. First, set the `MPI_RLD_HACK_OFF` environment variable for safety reasons and then compile the application with the MPI library:

```
setenv MPI_RLD_HACK_OFF 1
f90 -o comm comm.f -lmpi
```

Next run the `ssrun` as part of the `mpirun` command:

```
mpirun -np 4 ssrun -mpi comm
```

For this 4–processor application, five experiment files will be generated: one for each processor (the IDs begins with `f`) and one for the master process (the ID begins with `m`).

```
comm.mpi.f3221936
comm.mpi.f3224241
comm.mpi.f3225085
comm.mpi.f3227246
comm.mpi.m3226551
```

You can view any of the files with `cvperf`:

```
cvperf comm.mpi.f3225085
```

Specifying Polling Data

The following categories of polling data are available by using caliper points:

- Pollpoint Sampling, see "Pollpoint Sampling", page 65.
- Call Stack Profiling, see "Call Stack Profiling", page 65.

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

Pollpoint Sampling

Setting pollpoint sampling on the **Runtime** tab window sets caliper points that 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 use is to enable you to set boundary points for phases. In subsequent runs, you can set sample points to collect the profiling data at the phase boundaries.

Call Stack Profiling

Enabling call stack profiling in the **CallStack** tab window causes the call stack of the target executable to be sampled at the specified time interval (a minimum of 10 milliseconds) and saved. The call stack continues to be sampled when the program is not running: that is, while it is internally or externally blocked. Call stack profiling is used in the **User Time/Callstack Sampling** task to calculate total times.

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

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

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

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

Table 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, nonintrusive	Less accurate, more intrusive
Used to compute CPU time	Used to compute total time

Specifying the Experiment Configuration

To specify the experiment configuration, choose **Configs...** from the **Perf** menu. See Figure 4-1, page 66, for an illustration of the resulting window. While you can access other tabs, the only ones that are active are the **Runtime** and **General** tabs.

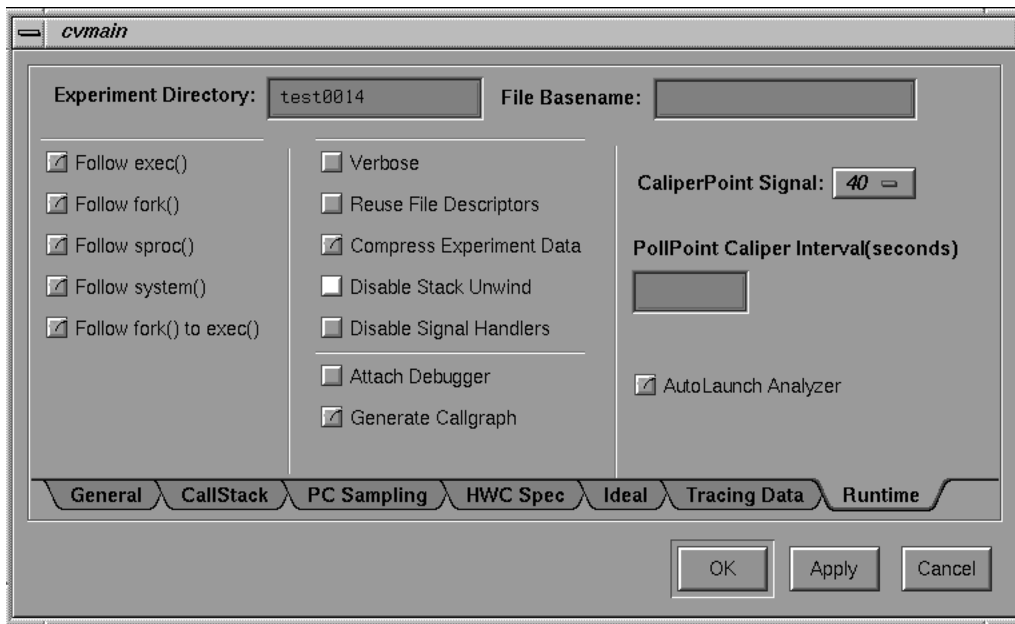

Figure 4-1 Runtime Configuration Dialog Box

Specifying the Experiment Directory

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

Other Options

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

- The **File Basename:** specifies the base name of the experiment file (if blank, it is the name of the executable).
- You can specify whether you want the Performance Analyzer to gather performance data for any processes launched by one or more of the following:
  - `exec()`
  - `fork()`
  - `sproc()`
  - `system()`
  - **Follow fork() to exec()** processes
- The center column lets you choose the following options:
  - **Verbose** output yields more explanatory information in the **Execution View**.
  - **Reuse File Descriptors** opens and closes the file descriptors for the output files every time performance data is to be written. If the target program is using `chdir()`, the `_SPEEDSHOP_REUSE_FILE_DESCRIPTORS` environment variable is set to the value selected by this configuration option.
  - **Compress Experiment Data** saves disk space.
  - **Disable Stack Unwind** suppresses the stack unwind as is done in the SpeedShop `usertime`, `totaltime`, and other call stack-based experiments.

- **Disable Signal Handlers** disables the normal setting of signal handlers for all fatal and exit signals.
- **Attach Debugger** lets you debug the running program.
- **Generate Callgraph** displays which functions called, and were called by, other functions.

- **CaliperPoint Signal** sets the value of the signal sent by the sample button to cause the process to write out a caliper point. The default value is 40.
- **PollPoint Caliper Interval (seconds)** specifies the interval at which pollpoint caliper points are taken.
- **AutoLaunch Analyzer** launches the Performance Analyzer automatically when the experiment finishes.

The Performance Analyzer Main Window

The Performance Analyzer main window is used for analysis after the performance data has been captured. It contains a time line area indicating when events took place over the span of the experiment, a list of functions with their performance data, and a resource usage chart. The following sections cover these topics:

- Task field, see "Task Field", page 69.
- Function list display and controls, see "Function List Display and Controls", page 69.
- Usage chart area, see "Usage Chart Area", page 71.
- Time line area and controls, see "Time Line Area and Controls", page 72.
- Admin menu, see "Admin Menu", page 73.
- Config menu, see "Config Menu", page 75.
- Views menu, see "Views Menu", page 83.
- Executable menu, see "Executable Menu", page 84.
- Thread menu, see "Thread Menu", page 85.

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

`cvperf` [`-exp`] *directory*

`cvperf` *speedshop_exp_files*

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

The arguments to these commands are as follows:

directory	A directory containing data from old WorkShop performance experiments.
speedshop_exp_files	One or more SpeedShop experiment files generated either by the `ssrun`(1) command or by using the **Select Task ...** submenu of the **Perf** menu on the WorkShop Debugger window.
pixie.counts_files	An output file from `pixie`(1) measuring code execution frequency. The `ideal` task generates a *pixie.counts_file*.

Task Field

The **Task** field identifies the task for the current experiment and is read-only. See "Selecting Performance Tasks", page 58, for a summary of the performance tasks. For an in-depth explanation of each task, refer to "Understanding Predefined Tasks", page 48.

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 4-2.

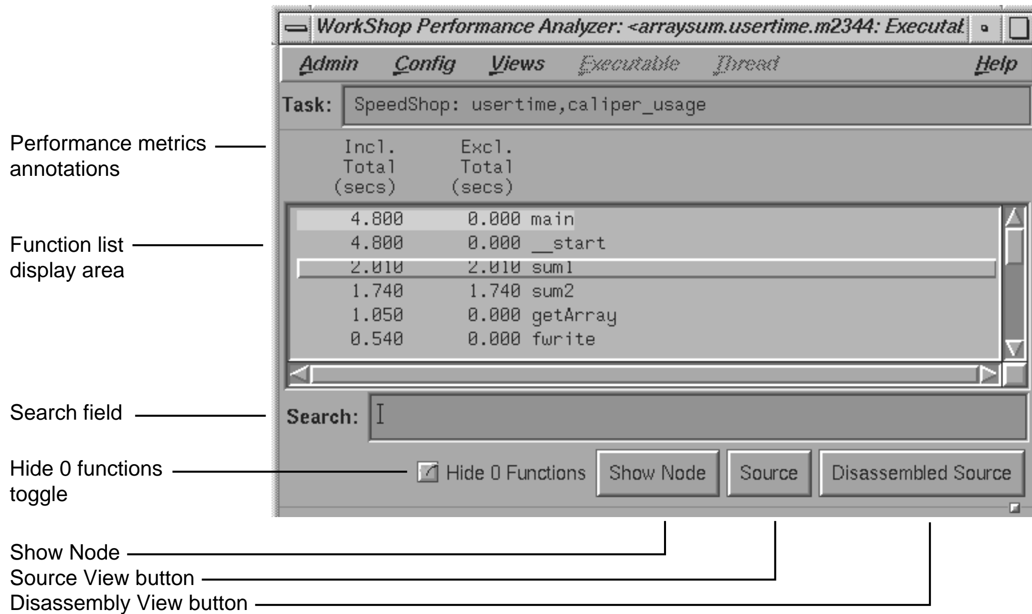

Figure 4-2 Typical Function List Area

The main features of the function list are:

Function list display area
: Shows all functions in the program annotated with their associated performance data. The column headings identify the metrics.

  You select the performance data to display from the **Preferences...** selection in the **Config** menu. The order of ranking is set by the **Sort...** selection in the **Config** menu. The default order of sorting (depending on availability) is:

  1. Inclusive time
  2. Exclusive time

	3. Counts
Search field	Lets you look for a function in the list and in any active views.
Hide 0 Functions toggle button	Lets you filter functions with 0 time from the list.
Show Node button	Displays the specified node in the **Call Graph View**.
Source button	Displays the **Source View** window corresponding to the selected function. The **Source View** window displays performance metrics in the annotation column. **Source View** can also be displayed by double-clicking a function in the function list or a node or arc (lines between nodes) in the call graph.
Disassembled Source button	Displays the **Disassembly View** window corresponding to the selected function. The **Disassembly View** is annotated with the performance metrics.

Usage Chart Area

The usage chart area in the Performance Analyzer main window displays the stripchart most relevant to the current task. The upper subwindow displays the legend for the stripchart, and the lower subwindow displays the stripchart itself. This gives you some useful information without having to open the **Usage View (Graphs)** window. Table 4-4, 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
User Time/Callstack Sampling	User versus system time
Profiling/PC Sampling	User versus system time
Ideal Time/Pixie	User versus system time
Floating Point Exception Trace	Floating-point exception event chart
I/O Trace	`read()`, `write()` system calls
Memory Leak Trace	Process Size stripchart

Task	Data in Usage Chart Area
R10000 or R12000 Hardware Counters	Depends on experiment
Custom task	User versus system time, unless one of the tracing tasks from this list has been selected

You can expand either subwindow to show more information by dragging the boxes at the right of the subwindow.

Time Line Area and Controls

The time line shows when each sample event in the experiment occurred. Figure 1-2, page 7, shows the time line portion of the Performance Analyzer window with typical results.

The Time Line Calipers

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

Specifying an interval is done as follows:

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

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

2. Set the right caliper to the sample event at the end of the interval. This is similar to setting the left caliper.

3. Confirm the change by clicking the **OK** button in the control area.

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

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

Current Event Selection

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

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

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

Time Line Scale Menu

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

Admin Menu

The **Admin** menu and its options are shown in Figure 4-3. The **Admin** menu has selections common to the other WorkShop tools. The following selections are different in the Performance Analyzer:

Experiment... Lets you change the experiment directory and displays a dialog box (see Figure 4-4, page 75).

Save As Text... Records a text file with preference information selected in the view and displays a dialog box. You can use the default file name or replace it with another name in the **Selection** dialog box that displays. You can specify the

number of lines to be saved. The data can be saved as a new file or appended to an existing one.

Figure 4-3 Performance Analyzer **Admin** Menu

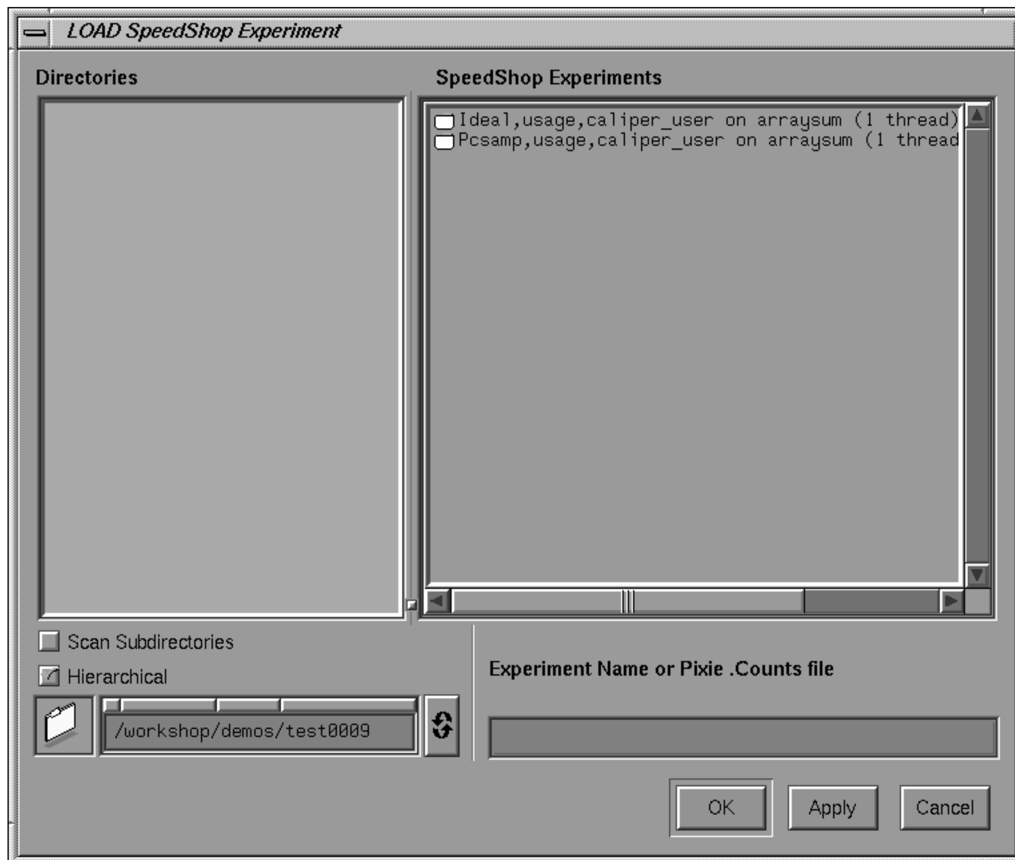

Figure 4-4 Experiment Window

Config Menu

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

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

Preferences... Brings up the **Data Display Options** window, which lets you select which metrics display and whether they

appear as absolute times and counts or percentages. Remember, you can only select the types of metrics that were collected in the experiment. You can also specify how C++ file names (if appropriate) are to display:

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

For an illustration of the **Data Display Options** window, see Figure 4-5, page 77.

Sort...

Brings up the **Sort Options** window, which lets you establish the order in which the functions appear; this helps you find questionable functions. The default order of sorting (depending on availability) is:

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

For an illustration, see Figure 4-6, page 78.

The selections for the **Display Data Options** window and the **Sort Options** window are similar. The difference between the inclusive (`Incl.`) and exclusive (`Excl.`) metrics is that inclusive data includes data from other functions called by the function, and exclusive data comes only from the function.

Figure 4-5 Performance Analyzer Data Display Options

Figure 4-6 Performance Analyzer Sort Options

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

Incl. Percentage, **Excl. Percentage**

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

Incl. Total Time, **Excl. Total Time**

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

Incl. CPU Time, **Excl. CPU Time**

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

Incl. Ideal Time, **Excl. Ideal Time**

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

Incl. HWC Data, **Excl. HWC Data**

Number of events measured.

Incl. Cycles, **Excl. Cycles**

Number of machine cycles.

Incl. Instr'ns, **Excl. Instr'ns**

Number of instructions.

Incl. FP operations, **Excl. FP operations**

Number of floating-point operations.

Incl. Load counts, **Excl. Load counts**

Number of `load` operations.

Incl. Store counts, **Excl. Store counts**

Number of `store` operations.

Incl. System calls, **Excl. System calls**

Number of system calls.

Incl. Bytes Read, **Excl. Bytes Read**

Number of bytes in a `read` operation.

Incl. Bytes Written, **Excl. Bytes Written**

Number of bytes in a `write` operation.

Incl. FP Exceptions, **Excl. FP Exceptions**

Number of floating-point exceptions.

Incl. Page faults, **Excl. Page faults**

Number of page faults.

Incl. bytes leaked, **Excl. bytes leaked**

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

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

Number of bytes allocated in `malloc` operations.

Incl. bytes MPI/Sent, **Excl. bytes MPI/Sent**

Number of bytes of data sent by an MPI routine.

Incl. bytes MPI/Recv, **Excl. bytes MPI/Recv**

Number of bytes of data received by an MPI routine.

Incl. MPI Send-Ops, **Excl. MPI Send-Ops**

Number of times an MPI send routine was executed.

Incl. MPI Recv-Ops, Excl. MPI Recv-Ops

Number of times an MPI receive routine was executed.

Incl. MPI Barriers, **Excl. MPI Barriers**

Number of times an `MPI_Barrier` routine was executed.

Address

Address of the function.

Instr'n Coverage

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

Calls

Number of times a function is called.

Pixstats/Cycles-per instr'n

Shows how efficient the code is written to avoid stalls or to take advantage of super scalar operation. A cycles per-instruction count of 1.0 means that an instruction is executed every cycle. A count greater than 1.0 means some instructions took more than one cycle. A count less that 1.0 means that sometimes more than one instruction was executed at a given cycle. The R10000 and R12000 processors can potentially execute up to 4 instructions on every cycle.

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

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

`Display Data As:`
Times/Counts
Percentages

Lets you choose whether you want to display your performance metrics as times and counts (for instance, the time a function required to execute) or as percentages (the percentage of the program's time a function used). The default is **Times/Counts**.

Hide 0 Functions in Function List and **Hide 0 Functions in Graph**

Lets you filter functions with 0 counts from the list or graph.

Incl. Percentage Show inclusive percentages on the **Call Graph View** window.

Incl. Total Time Show inclusive total time on the **Call Graph View** window.

Incl. CPU Time Show inclusive CPU time on the **Call Graph View** window.

Incl. Ideal Time Show inclusive ideal time on the **Call Graph View** window.

Incl. HWC Data Show inclusive hardware counter data on the **Call Graph View** window.

Incl. System calls Show inclusive system calls on the **Call Graph View** window.

Incl. Bytes Read Show inclusive bytes read on the **Call Graph View** window.

Incl. Bytes Written Show inclusive bytes written on the **Call Graph View** window.

Incl. FP Exceptions Show inclusive floating–point exceptions on the **Call Graph View** window.

Incl. Page faults Show inclusive page faults on the **Call Graph View** window.

Incl. bytes leaked Show inclusive bytes leaked as a result of `malloc` operations not followed by matching `free` operations on the **Call Graph View** window.

Incl. bytes malloc'd Show inclusive bytes allocated with a `malloc` operation on the **Call Graph View** window.

Calls Show the number of calls to that function on the **Call Graph View** window.

The following option is available on the **Sort Options** window only:

Alphabetic	Sort alphabetically by function name.

Views Menu

The **Views** menu in the Performance Analyzer (see Figure 4-7, page 84) provides the following selections for viewing the performance data from an experiment. Each view displays the data for the time interval bracketed by the calipers in the time line.

Usage View (Graphs)	Displays resource usage charts and event charts. See "Usage View (Graphs)", page 85.
Usage View (Numerical)	Displays the aggregate values of resources used. See "Usage View (Numerical) Window", page 91.
I/O View	Displays I/O events. See "The I/O View Window", page 93.
MPI Stats View (Graphs)	Displays MPI information in the form of graphs. See "The MPI Stats View (Graphs) Window", page 94.
MPI Stats View (Numerical)	Displays MPI information in the form of text. See "The MPI Stats View (Numerical) Window", page 96.
Call Graph View	Displays a call graph that shows functions and calls and their associated performance metrics. See "The Call Graph View Window", page 97.
Butterfly View	Displays the callers and callees of the function. See "Butterfly View", page 106.
Leak View	Displays individual leaks and their associated call stacks. See "Using Malloc Error View, Leak View, and Malloc View", page 107.
Malloc View	Displays individual `malloc` routines and their associated call stacks. See "Using Malloc Error View, Leak View, and Malloc View", page 107.
Malloc Error View	Displays errors involving memory leaks and bad calls to `free`, indicating error locations and the total number of errors. See "Using Malloc Error View, Leak View, and Malloc View", page 107.
Heap View	Displays a map of heap memory showing `malloc`, `realloc`, `free`, and bad `free` operations. See

	"Analyzing the Memory Map with Heap View", page 110.
Call Stack View	Displays the call stack for the selected event and the corresponding event type. See "The Call Stack Window", page 114.
Working Set View	Measures the coverage of the DSOs that make up the executable, noting which were not used. See "**Working Set View**", page 25.

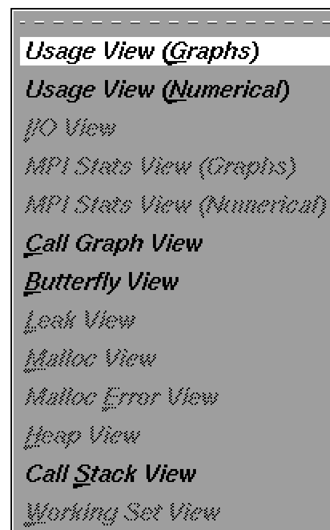

Figure 4-7 Performance Analyzer **Views** Menu

Executable Menu

If you enabled **Track Exec'd Processes** for the current experiment, the **Executable** menu will be enabled and will contain selections for any `execed` processes. (The **Track Exec'd Processes** selection is in the **Performance** panel of the **Executable** menu.) These selections let you see the performance results for the other executables.

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

Thread Menu

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

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

Usage View (Graphs)

The **Usage View (Graphs)** window displays resource usage and event charts containing the performance data from the experiment. These charts show resource usage over time and indicate where sample events took place. Sample events are shown as vertical lines. Figure 4-8, page 86, shows the **Usage View (Graphs)** window.

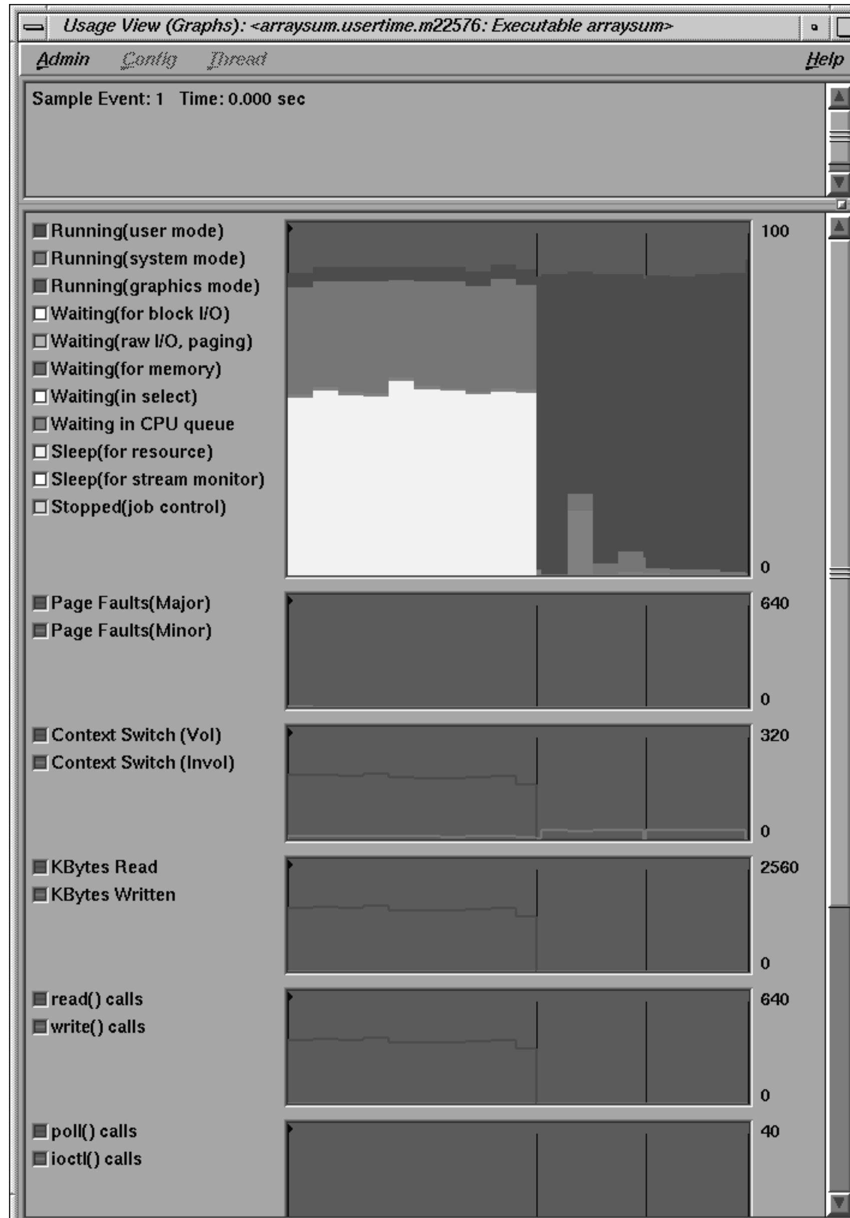

Figure 4-8 **Usage View (Graphs)** Window

Charts in the Usage View (Graphs) Window

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

User versus system time

Shows CPU use. Whenever the system clock ticks, the process occupying the CPU is charged for the entire ten millisecond interval. The time is charged either as user or system time, depending on whether the process is executing in user mode or system mode. The graph provides these annotations to show how time is spent during an experiment's process: **Running (user mode)**, **Running (system mode)**, **Running (graphics mode)**, **Waiting (for block I/O)**, **Waiting (raw I/O, paging)**, **Waiting (for memory)**, **Waiting (in select)**, **Waiting in CPU queue**, **Sleep (for resource)**, **Sleep (for stream monitor)**, and **Stopped (job control)**.

Page Faults

Shows the number of page faults that occur within a process. *Major faults* are those that require a physical read operation to satisfy; *minor faults* are those where the necessary page is already in memory but not mapped into the process address space.

Each major fault in a process takes approximately 10 to 50 milliseconds. A high page-fault rate is an indication of a memory-bound situation.

Context Switch

Shows the number of voluntary and involuntary context switches in the life of the process.

Voluntary context switches are attributable to an operation caused by the process itself, such as a disk access or waiting for user input. These occur when the process can no longer use the CPU. A high number of voluntary context switches indicates that the process is spending a lot of time waiting for a resource other than the CPU.

Involuntary context switches happen when the system scheduler gives the CPU to another process, even if the target process is able to use it. A high number of

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

Getting Event Information from the Usage View (Graphs) Window

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

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

- The point becomes selected, as indicated by the diamond marker above it. The marker appears in the time line, resource usage chart, and **Usage View (Graphs)** charts if the window is open.
- The current event line at the top of the window identifies the event and displays its time.
- The call stack that corresponds to this sample point is displayed in the **Call Stack** window (see "The Call Stack Window", page 114).

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

The Process Meter Window

The process meter lets you observe resource usage for a running process without conducting an experiment. To call the process meter, select **Process Meter** from the **Views** menu in the **WorkShop Debugger** window.

A **Process Meter** window with data and its menus displayed appears in Figure 4-9, page 90. The **Process Meter** window uses the same **Admin** menu as the WorkShop Debugger tools.

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

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

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

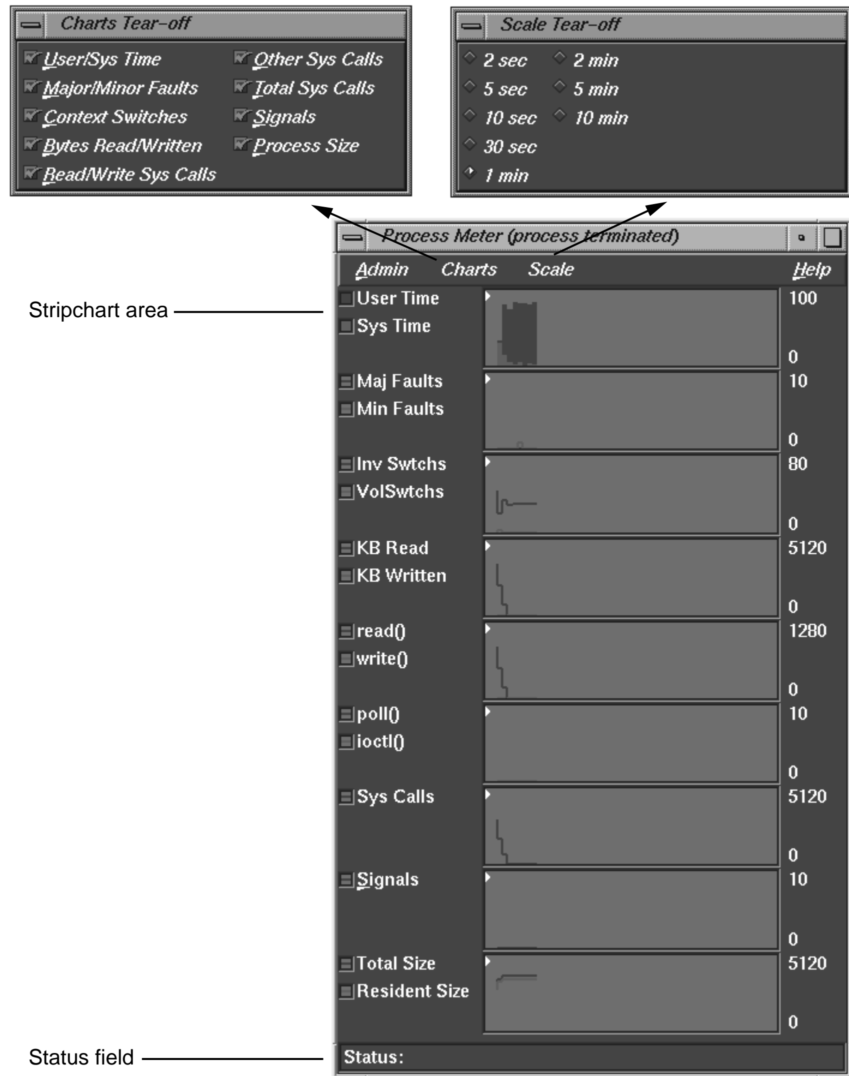

Figure 4-9 The **Process Meter** Window with Major Menus Displayed

Usage View (Numerical) Window

The **Usage View (Numerical)** window (see Figure 4-10, page 92) shows detailed, process-specific resource usage information in a textual format for a specified interval. The interval is 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.

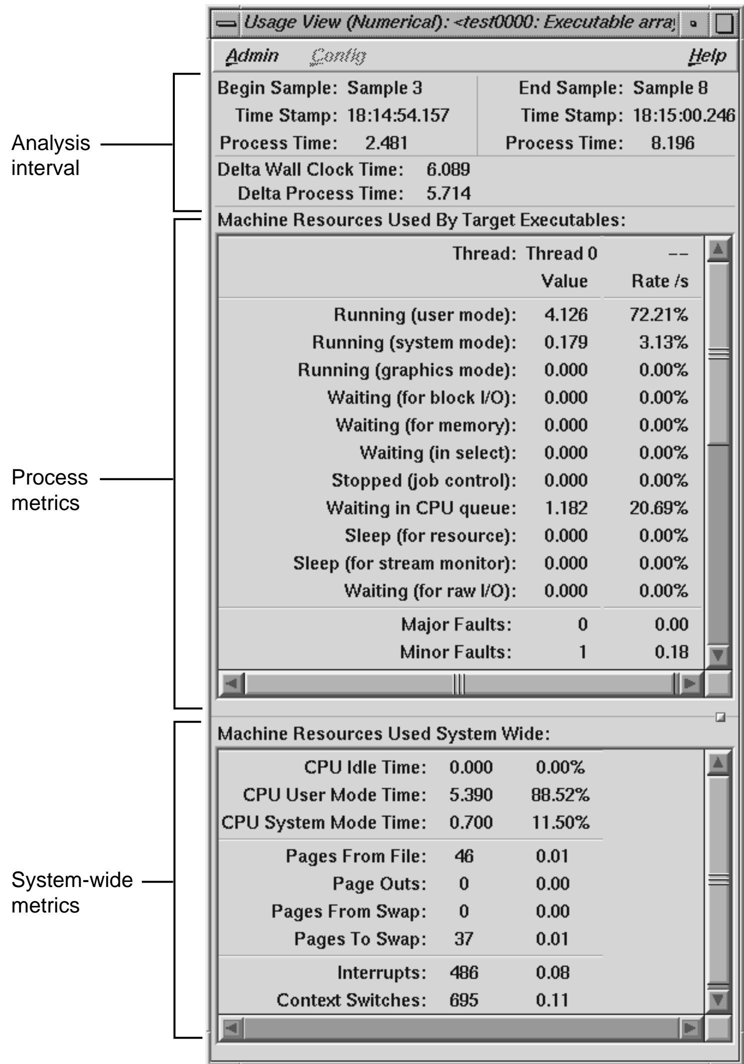

Figure 4-10 The **Usage View (Numerical)** Window

The I/O View Window

The **I/O View** window helps you determine the problems in an I/O-bound process. It produces graphs of all I/O system calls for up to 10 files involved in I/O. Clicking an I/O event with the left mouse button displays information about it in the event identification field at the top of the **I/O View** window. See Figure 4-11.

For a list of the system calls traced, see "I/O Trace", page 52.

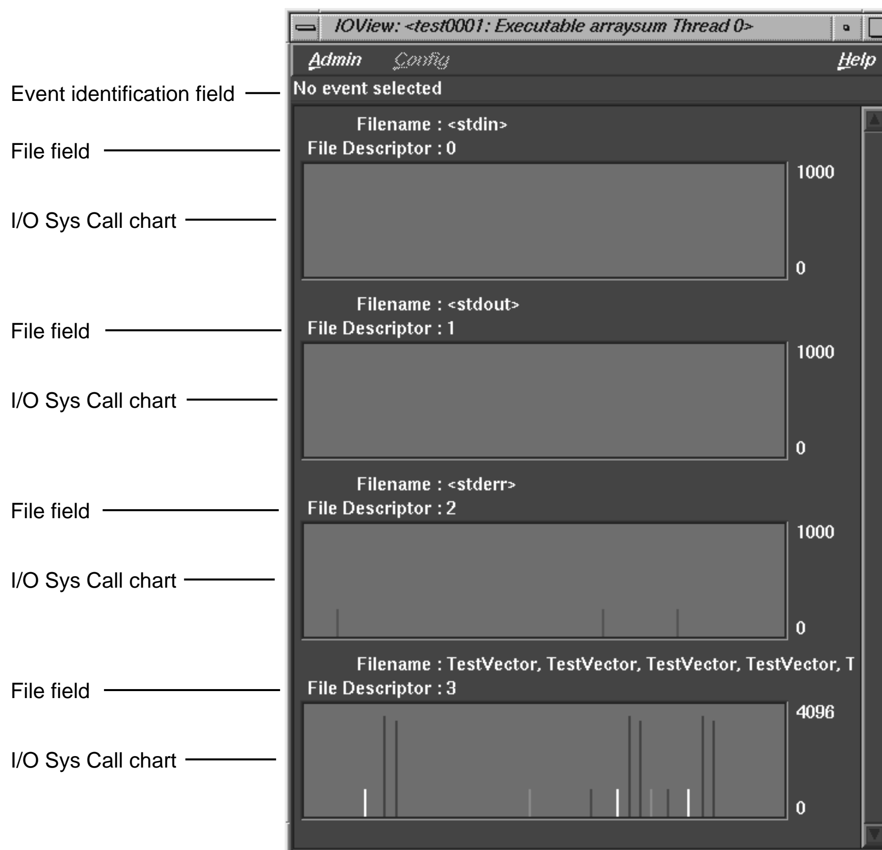

Figure 4-11 The **I/O View** Window

The MPI Stats View (Graphs) Window

The **MPI Stats View (Graphs)** window displays information on as many as 32 aspects of an MPI program in graph format. For an illustration of the window, see Figure 1-6, page 13.

If a graph contains nothing but zeros, it is not displayed.

In the following list of information that may be displayed in the graphs, *shared memory* refers to memory in a multiprocessor system that can be accessed by any processor. The High Performance Parallel Interface (*HIPPI*) is a network link, often used to connect computers; it is slower than shared memory transfers but faster than TCP/IP transfers. *TCP/IP* is a networking protocol that moves data between two systems on the Internet.

Collective calls are those that move a message from one processor to multiple processors or from multiple processors to one processor. `MPI_Bcast`(3) is a collective call. A *point-to-point call*, such as `MPI_Send`(3) or `MPI_Ssend`(3), moves a message from one processor to one processor.

Note: The MPI tracing experiment does not track down communicators, and it does not trace all collective operations. This may also affect the translation of some events using `ssfilter`(1).

The following information can be displayed in the **MPI Stats View (Graphs)** window.

- Retries in allocating MPI headers per procedure for collective calls
- Retries in allocating MPI headers per host for collective calls.
- Retries in allocating MPI headers per procedure for point–to–point calls
- Retries in allocating MPI headers per host for point–to–point calls
- Retries in allocating MPI buffers per procedure for collective calls
- Retries in allocating MPI buffers per host for collective calls
- Retries in allocating MPI buffers per procedure for point–to–point calls
- Retries in allocating MPI buffers per host for point–to–point calls
- The number of send requests using shared memory for collective calls
- The number of send requests using shared memory for point–to–point calls

- The number of send requests using a HIPPI bypass for collective calls
- The number of send requests using a HIPPI bypass for point–to–point calls
- The number of send requests using TCP/IP for collective calls
- The number of send requests using TCP/IP for point–to–point calls
- The number of data buffers sent using shared memory for point–to-point calls
- The number of data buffers sent using shared memory for collective calls
- The number of data buffers sent using a HIPPI bypass for point–to-point calls
- The number of data buffers sent using a HIPPI bypass for collective calls
- The number of data buffers sent using TCP/IP for point–to-point calls
- The number of data buffers sent using TCP/IP for collective calls
- The number of message headers sent using shared memory for point–to-point calls
- The number of message headers sent using shared memory for collective calls
- The number of message headers sent using a HIPPI bypass for point–to-point calls
- The number of message headers sent using a HIPPI bypass for collective calls
- The number of message headers sent using TCP/IP for point–to-point calls
- The number of message headers sent using TCP/IP for collective calls
- The total number of bytes sent using shared memory for point–to-point calls
- The total number of bytes sent using shared memory for collective calls
- The total number of bytes sent using a HIPPI bypass for point–to-point calls
- The total number of bytes sent using a HIPPI bypass for collective calls
- The total number of bytes sent using TCP/IP for point–to-point calls
- The total number of bytes sent using TCP/IP for collective calls

The MPI Stats View (Numerical) Window

The **MPI Stats View (Numerical)** window displays the same information as the **MPI Stats View (Graphs)** window (see the preceding section), but it presents it in text form. For an illustration, see Figure 1-7, page 15.

Unlike the **MPI Stats View (Graphs)** window, this window includes all of the data, whether or not it is zero.

The Parallel Overhead View Window

The **Parallel Overhead View** window shows the overhead incurred by a parallel program. MPI, OpenMP, and pthread parallel programming models are supported. The following figure illustrates the overhead for the `total.f` Fortran program, located in the `/usr/demos/WorkShop/mp` directory.

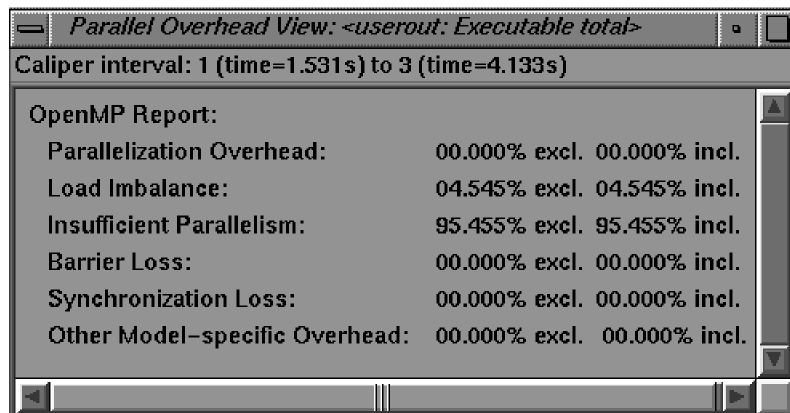

Figure 4-12 Overhead View

The meaning for each of the data items is as follows for this OpenMP demo. Other programming models generate slightly different data.

`Parallelization Overhead`

The percentage of the total overhead time spent making the code parallel. In the example, this time is negligible.

`Load Imbalance`

The percentage of the overhead time caused by *load imbalance*. Load imbalance means the parallel work is not evenly distributed among the processors, causing some processors to wait while the others finish their tasks.

`Insufficient Parallelism`

The percentage of the overhead time spent in regions of the code that are not parallel.

`Barrier Loss`

The percentage of overhead time consumed by the barrier mechanism. This is not the time spent waiting at a barrier.

`Synchronization Loss`

The percentage of the overhead time consumed by synchronization mechanisms other than barriers.

`Other Model-specific Overhead`

The percentage of the overhead time due to other operations of the parallel programming model, in this case OpenMP.

Overhead data is collected automatically when you create an experiment file. To see the total picture, aggregate the experiment files from each processor into a single file, as follows:

```
% ssaggregate -e total.usertime* -o userout
```

Then view the single output file, `userout`, through `cvperf`.

The Call Graph View Window

The **Call Graph View** window displays the functions as nodes, annotated with performance metrics, and their calls as connecting arcs (see Figure 4-13, page 98). Bring up the **Call Graph View** window by selecting **Call Graph View** from the **Views** menu.

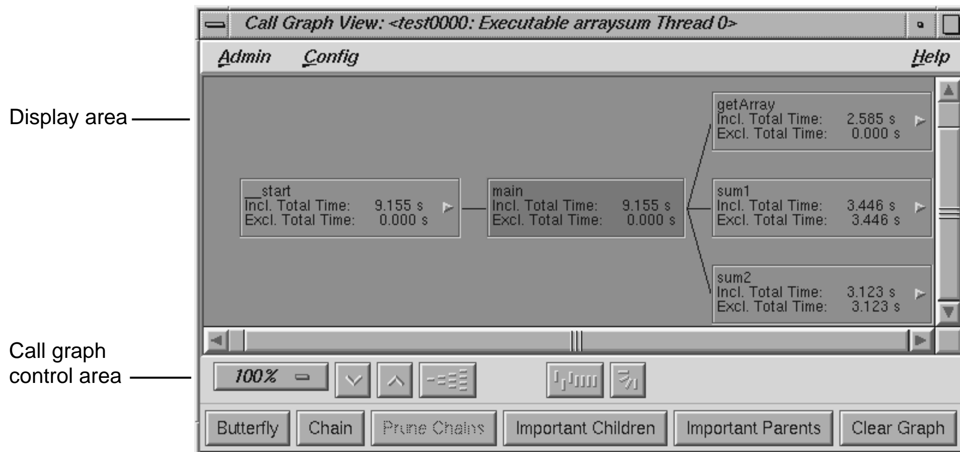

Figure 4-13 Call Graph View with Display Controls

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

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 described in the following sections.

Node Annotations

To specify the performance metrics that display inside a node, use the **Preferences** dialog box in the **Config** menu from the Performance Analyzer main view. (For an illustration of the **Data Display Options** window, see Figure 4-5, page 77.)

Arc Annotations

Arc annotations are specified by selecting **Preferences...** from the **Config** menu in the **Call Graph View** window. (For an illustration of the **Data Display Options** window, see Figure 4-5, page 77.) You can display the counts on the arcs (the lines between the functions). You can also display the percentage of calls to a function broken down by incoming arc. For an explanation of the performance metric items, see "Config Menu", page 75.

Filtering Nodes and Arcs

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

Call Graph Preferences Filtering Options

The **Preferences** selection in the **Call Graph View Config** menu also lets you hide functions and arcs that have 0 calls. See Figure 4-5, page 77.

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 4-14.

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

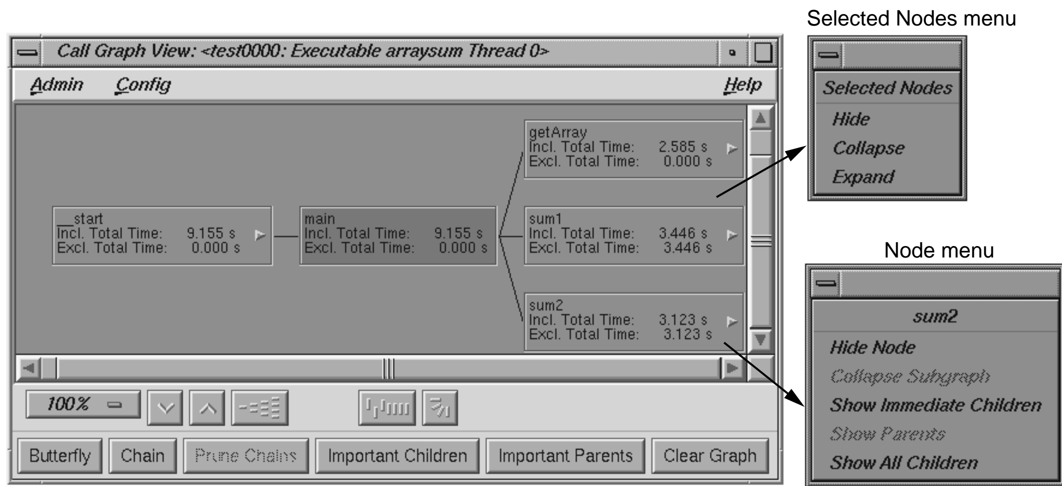

Figure 4-14 Node Menus

The **Node** menu selections are as follows:

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

Selected Nodes Menu

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

Hide	Removes the selected nodes from the call graph display

Collapse Removes the nodes called by the selected nodes (and descendant nodes) from the call graph display

Expand Displays all the functions (descendants) called by the selected nodes

Filtering Nodes through the Display Controls

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

You can perform these display operations:

Butterfly Presents the call graph from the perspective of a single node (the target node), showing only those nodes that call it or are called by it. Functions that call it are displayed to the left and functions it calls are on the right. Selecting any node and clicking **Butterfly** redraws the display with the selected node in the center. The selected node is displayed and highlighted in the function list.

Chain Lets you display all paths between a given source node and target node. The **Chain** dialog box is shown in Figure 4-15, page 102. You designate the source function by selecting it or entering it in the **Source Node** field and clicking the **Make Source** button. Similarly, the target function is selected or entered and then established by clicking the **Make Target** button. If you want to filter out paths that go through nodes and arcs with zero counts, click the toggle. After these selections are made, click **OK.**

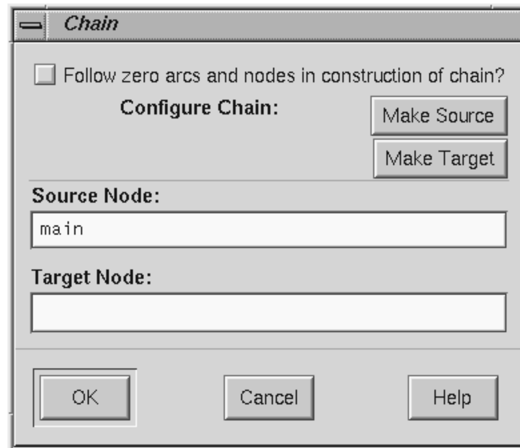

Figure 4-15 Chain Dialog Box

Prune Chains Displays a dialog box that provides two selections for filtering paths from the call graph (see Figure 4-16).

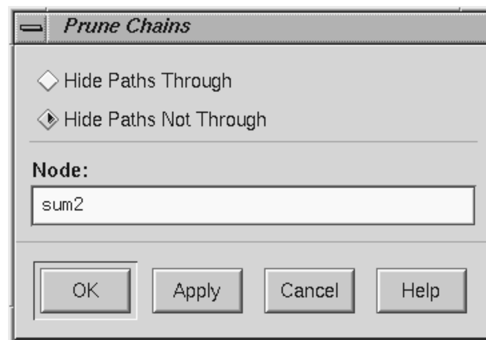

Figure 4-16 Prune Chains Dialog Box

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

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

Important Children Lets you focus on a function and its descendants and set thresholds to filter the descendants. You can filter the descendants either by percentage of the caller's time or by percentage of the total time. The **Threshold key** field identifies the type of performance time data used as the threshold. See Figure 4-17.

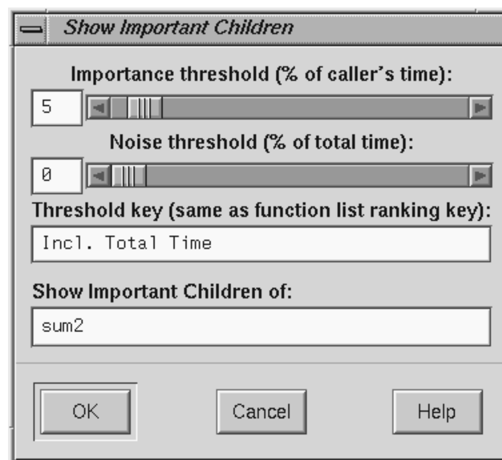

Figure 4-17 Show Important Children Dialog Box

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 4-18.

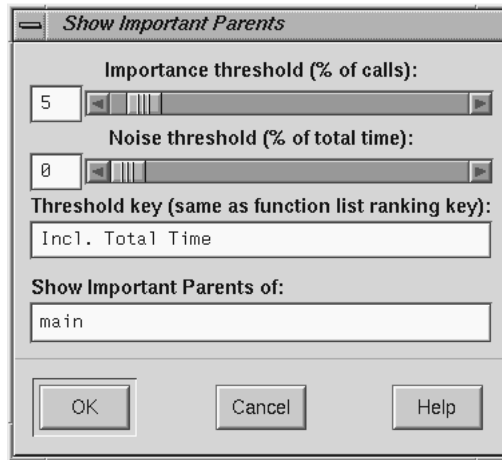

Figure 4-18 Show Important Parents Dialog Box

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

Other Manipulation of the Call Graph

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

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 4-19, page 105).

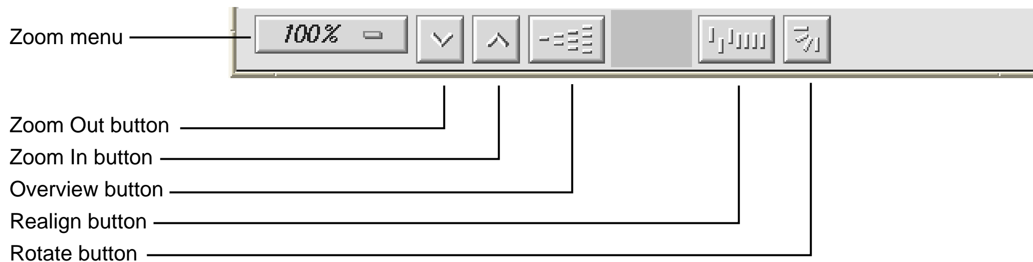

Figure 4-19 Call Graph View Controls for Geometric Manipulation

These controls are:

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

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

Using the Mouse in the Call Graph View

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

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

Selecting Nodes from the Function List

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

Butterfly View

The **Butterfly View** shows a selected function, the functions that called it (the **Immediate Parents**), and the functions it calls (the **Immediate Children**). For an illustration, see Figure 1-11, page 19.

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

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

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

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.

A memory leak occurs when memory that is allocated in the program in not freed later. As a result, the size of the program grows unnecessarily.

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 4-20, page 108), **Leak View** (see Figure 4-21, page 108), or **Malloc View** (see Figure 4-22, page 109). **Malloc View** is the most general, showing all memory allocation operations. **Malloc Error View** shows only those memory operations that caused problems, identifying the cause of the problem and how many times it occurred. **Leak View** displays each memory leak that occurs in your executable, its size, the number of times the leak occurred at that location during the experiment, and the corresponding call stack (when you select the leak).

Each of these views has three major areas:

- Identification area—This indicates which operation has been selected from the list. **Malloc View** identifies `malloc` routines, indicating the number of `malloc` locations and the size of all `malloc` operations in bytes. **Malloc Error View** identifies leaks and bad `free` routines, indicating the number of error locations and how many errors occurred in total. **Leak View** identifies leaks, indicating the number of leak locations and the total number of bytes leaked.
- List area—This is a list of the appropriate types of memory operations according to the type of view. Clicking an item in the list identifies it at the top of the window and displays its call stack at the bottom of the list. The list displays in order of size.
- Call stack area— This displays the contents of the call stack when the selected memory operation occurred. Figure 4-23, page 109, shows a typical **Source View** window with leak annotations. (You can change the annotations by using the **Preferences...** selection in the **Performance Analyzer Config** menu). Colored boxes draw attention to high counts.

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

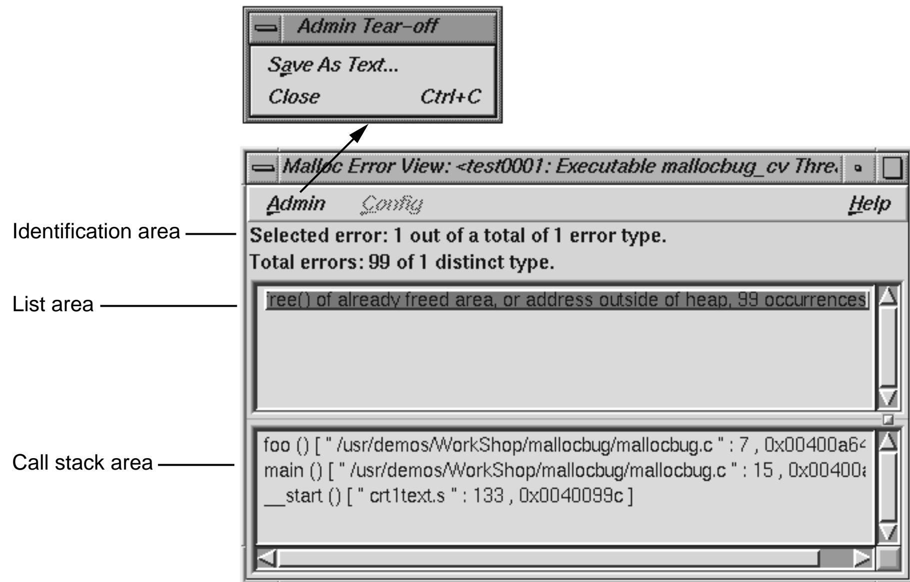

Figure 4-20 **Malloc Error View** Window with an **Admin** Menu

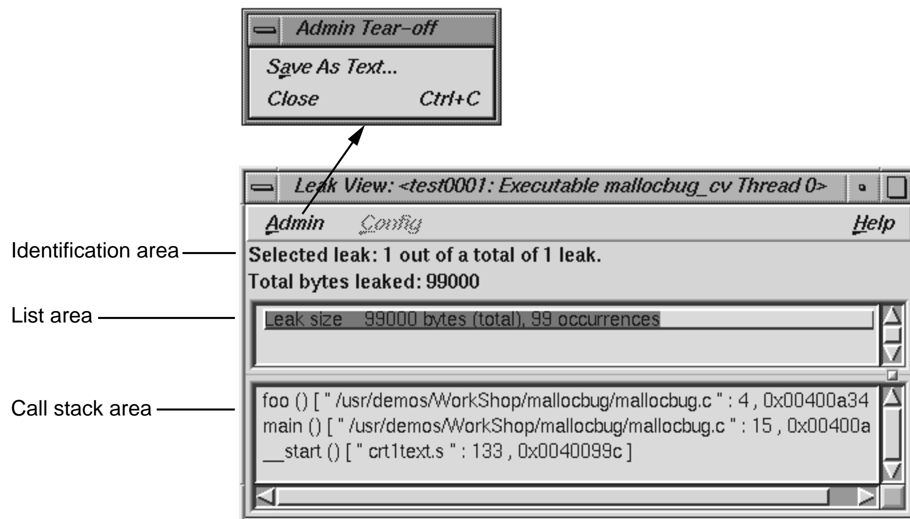

Figure 4-21 **Leak View** Window with an **Admin** Menu

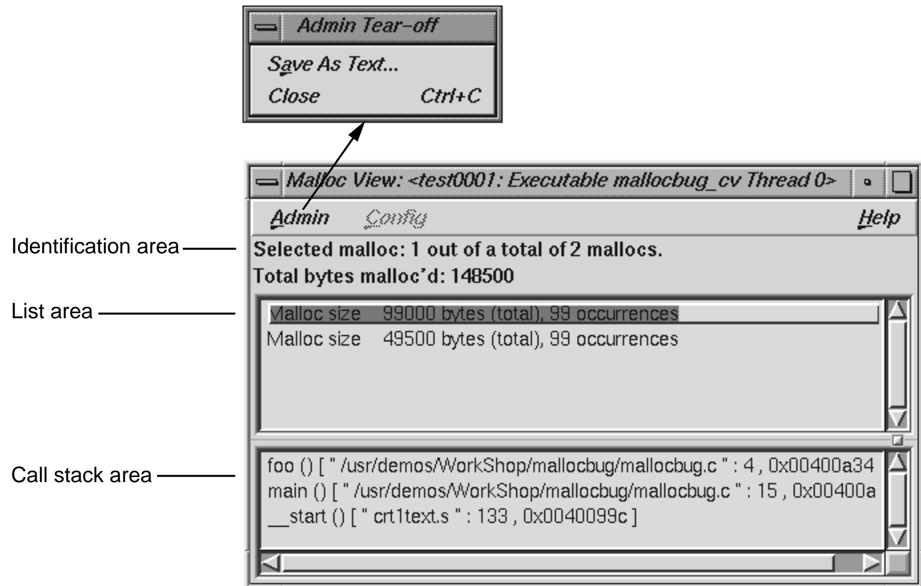

Figure 4-22 Malloc View Window with **Admin** Menu

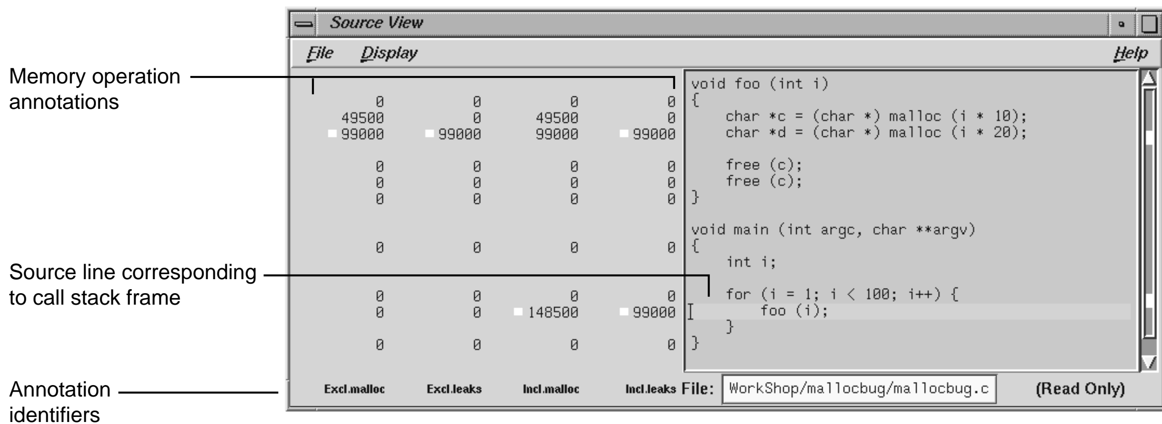

Figure 4-23 Source View Window with Memory Analysis Annotations

Analyzing the Memory Map with Heap View

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

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

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

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

Heap View Window

A typical **Heap View** window with its parts labeled appears in Figure 4-24, page 111.

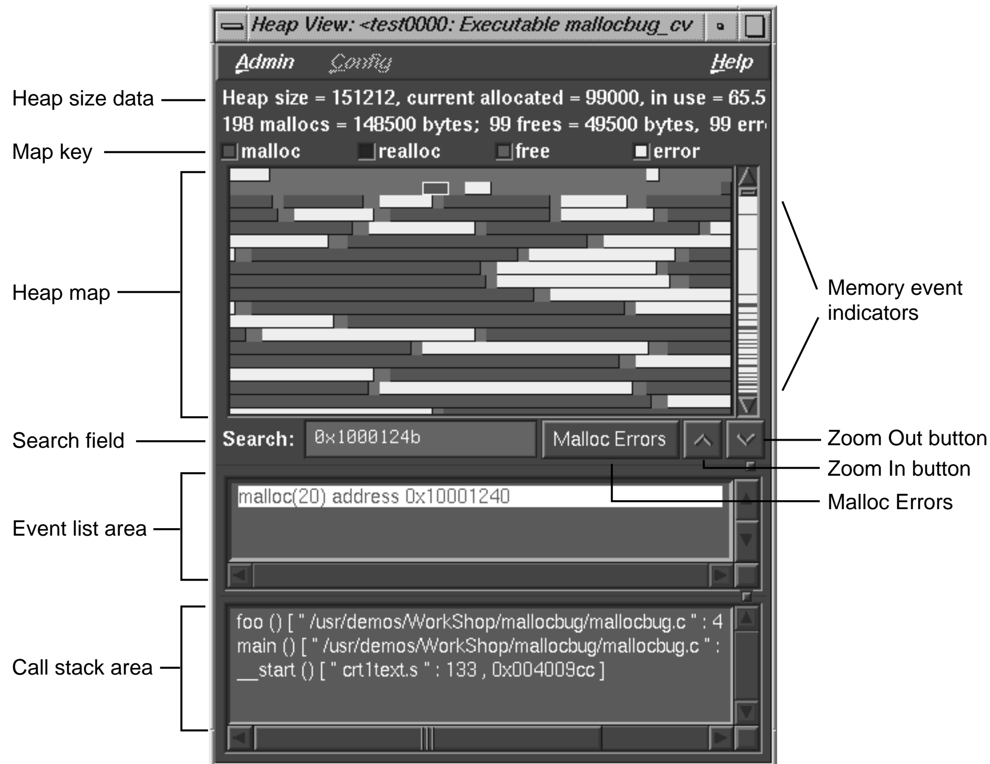

Figure 4-24 Heap View Window

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

Map key — Appears at the top of the heap map area to identify blocks by color. The actual colors depend on your color scheme.

Heap map — Shows heap memory as a continuous, wrapping, horizontal rectangle. The memory addresses begin at the upper left corner and progress from left to right, row by row. The rectangle is broken up into color-coded segments according to memory use status. Clicking a highlighted area in the heap map identifies the type of problem, the memory address where it

occurred, its size in the event list area, and the associated call stack in the call stack display area.

Note in Figure 4-24, page 111, that there are only a few problems in the memory at the lower addresses and many more at the higher addresses.

Memory event indicators	The events appear color-coded in the scroll bar. Clicking an indicator with the middle button scrolls the display to the selected problem.
Search field	Provides two functions: • If you enter a memory address in the field, the corresponding position will be highlighted in the heap map. If there was a problem at that location, it will be identified in the event list area. If there is no problem, the event list area displays the address at the beginning of the memory block and its size. • If you hold down the left mouse button and position the cursor in the heap map, the corresponding address will display in the **Search** field.
Event list area	Displays the events occurring in the selected block. If only one event was received at the given address, its address is shown by default. If more than one event is shown, double-clicking an event will display its corresponding call stack.
Call stack area	Displays the call stack corresponding to the event highlighted in the event list area.
Malloc Errors button	Causes `malloc` errors and their addresses to display in the event list area. You can then enter the address of the `malloc` error in the **Search** field and press the `Enter` key to see the error's `malloc` information and its associated call stack.
Zoom in button	An upward-pointing arrow, it redisplays the heap area at twice the current size of the display. If you reach the limit, an error message displays.

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

Source View `malloc` Annotations

Like **Malloc View**, if you double-click a line in the call stack area of the **Heap View** window, the **Source View** window displays the portion of code containing the corresponding line. The line is highlighted and indicated by a caret (^), with the number of bytes used by `malloc` in the annotation column. See Figure 4-23, page 109.

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 4-25, page 114). This lets you select the file name in which to save the **Heap View** data. The default file name is *experiment-filename*`.out`. When you click **OK**, the data for the current caliper setting and the list of unmatched `free` routines, if any, are appended to the specified file.

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

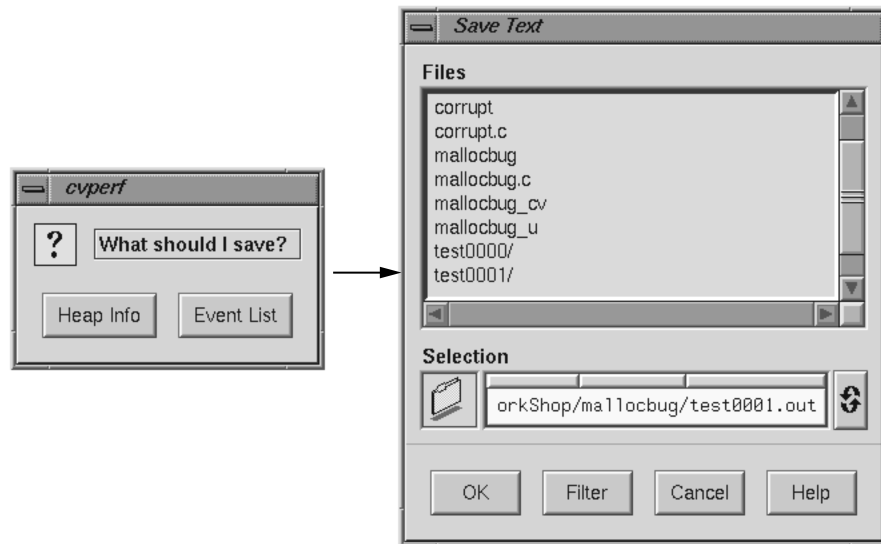

Figure 4-25 Heap View Save Text Dialog Boxes

The Call Stack Window

The **Call Stack** window, which is accessed from the Performance Analyzer **Views** menu, lets you get call stack information for a sample event selected from one of the Performance Analyzer views. See Figure 4-26, page 115.

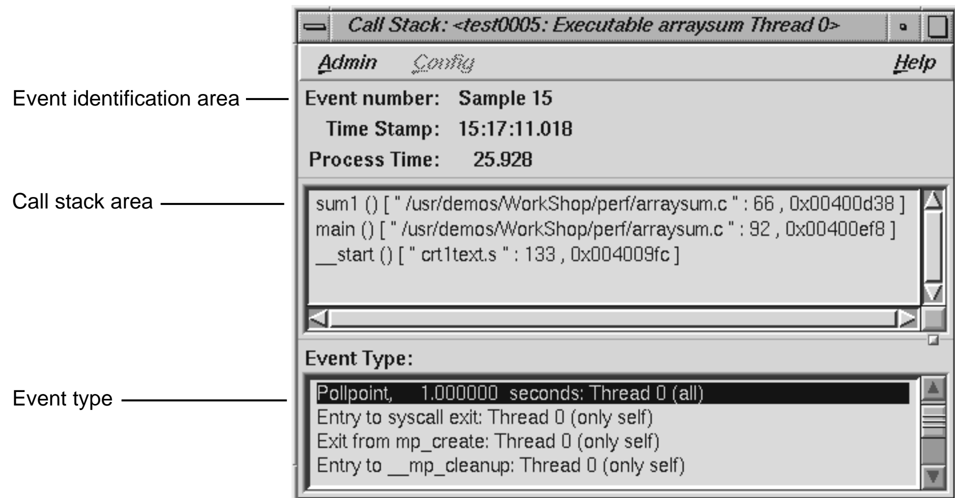

Figure 4-26 Performance Analyzer **Call Stack** Window

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

Event identification area

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

Call stack area

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

Event type area

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

Analyzing Working Sets

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

The term *working set* refers to those executable pages, functions, and instructions that are actually brought into memory during a phase or operation of the executable. If more pages are required than can fit in memory at the same time, *page thrashing* (that is, swapping in and out of pages) may result, slowing down your program. Strategic selection of which pages functions appear on can dramatically improve performance in such cases.

You do this by creating a file containing a list of functions, their sizes, and addresses called a *cord mapping file*. The functions should be ordered so as to optimize page swapping efficiency. This file is then fed into the `cord` utility, which rearranges the functions according to the order suggested in the `cord` mapping file. See the `cord`(1) man page for more information.

Working set analysis is appropriate for:

- Programs that run 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, `sscord`(1), is separate from the Performance Analyzer and is invoked by typing **`sscord`** at the command line. It displays a list of the working sets that make up a cord mapping file, shows their utilization efficiency, and, most importantly, computes an optimized ordering to reduce working sets.

Figure 4-27, page 118, presents an overview of the process of conducting working set analysis.

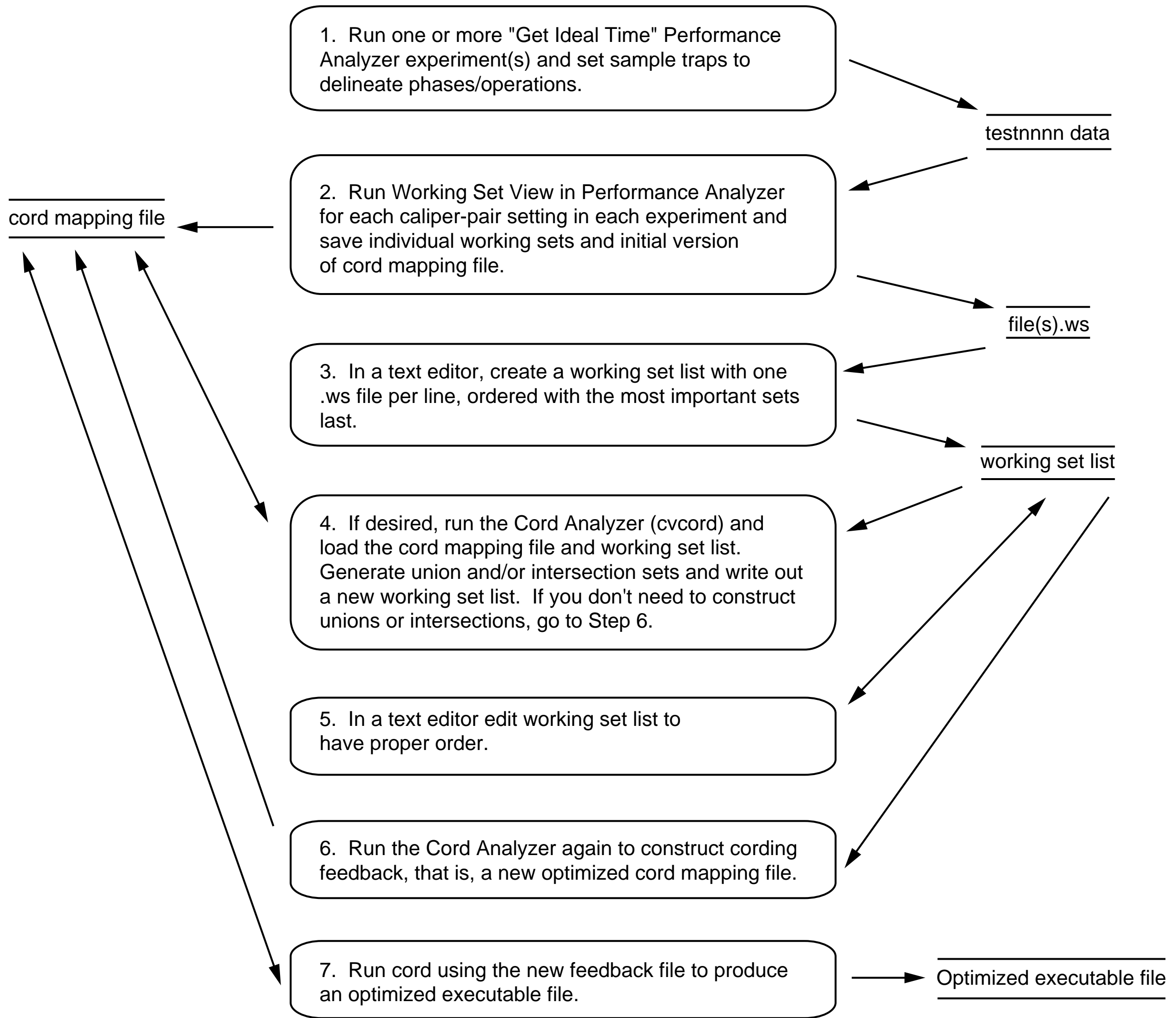

Figure 4-27 Working Set Analysis Process

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

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

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

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

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

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

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

Working Set View

The **Working Set View** measures the coverage of the dynamic shared objects (DSOs) that make up your executable (see Figure 4-28, page 120). It indicates instructions, functions, and pages that were not used when the experiment was run. It shows the coverage results for each DSO in the DSO list area. Clicking a DSO in the list displays its pages with color-coding to indicate the coverage of the page.

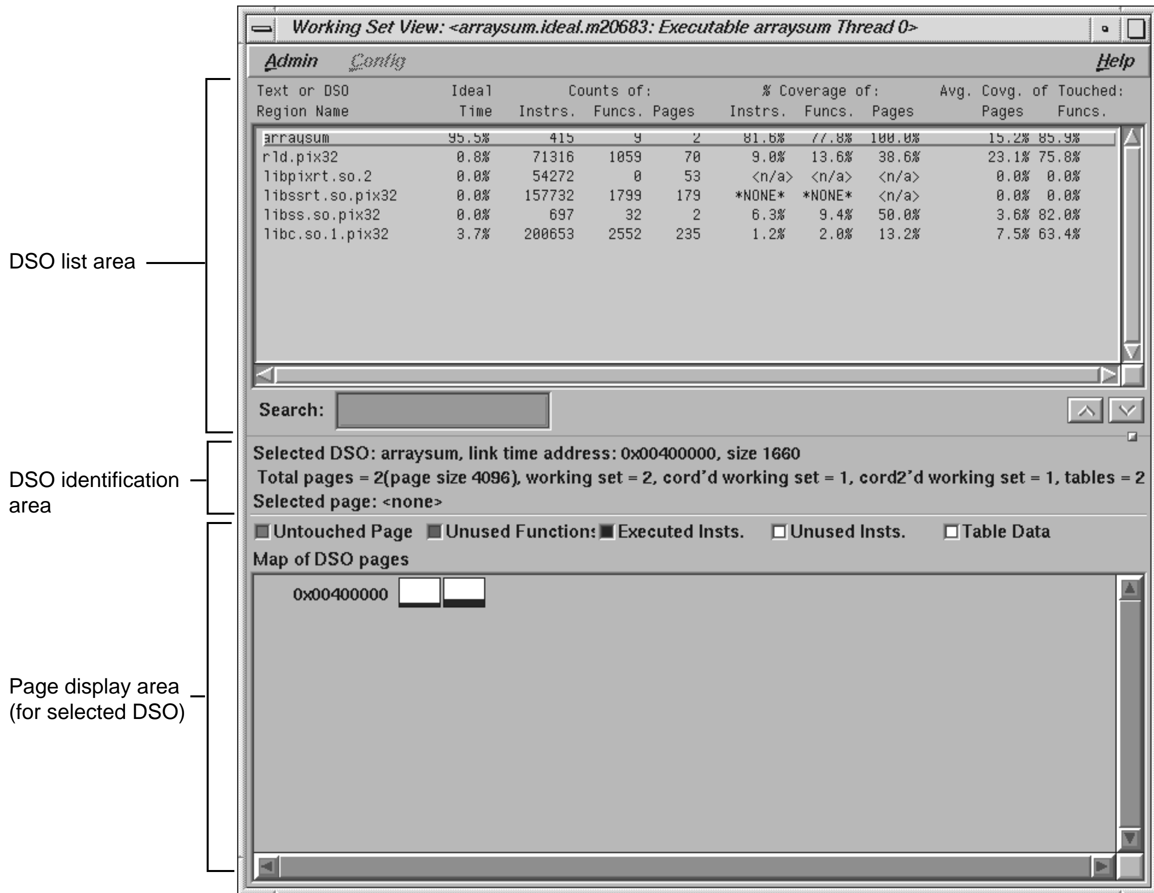

Figure 4-28 **Working Set View**

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

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

Counts of: Instrs.

Lists the number of instructions contained in the DSO.

Counts of: Funcs.

Lists the number of functions contained in the DSO.

Counts of: Pages

Lists the number of pages occupied by the DSO.

% Coverage of: Instrs.

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

% Coverage of: Funcs.

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

% Coverage of: Pages

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

Avg. Covg. of Touched: Pages

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

Avg. Covg. of Touched: Funcs

Lists the average percentage use of instructions within used functions.

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

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

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

Admin Menu

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

Save Working Set

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

Save Cord Map File

Saves all of the functions in the DSOs in a cord mapping file for input to the Cord Analyzer. This file corresponds to the feedback file discussed on the `cord`(1) man page.

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 is discussed in this part of the manual because it works in conjunction with the **Working Set View**. The cord analyzer lets you explore the working set behavior of an executable or shared library (DSO). With it you can construct a feedback file for input to the `cord`(1) utility to generate an executable with improved working set behavior. Invoke the cord analyzer at the command line using the following syntax:

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

The `sscord` command accepts the following arguments:

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

`-v|-V`	Verbose output. If specified, mismatches between working sets and the executable or DSO are noted.
executable	Specifies a single executable file name as input.

The **Cord Analyzer** window is shown in Figure 4-29, page 125, with its major areas and menus labeled.

Working Set Display Area

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

Working-set pgs. (util. %)

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

cord'd set pgs

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

Working-set Name

Identifies the path for the working set.

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

Working Set Identification Area

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

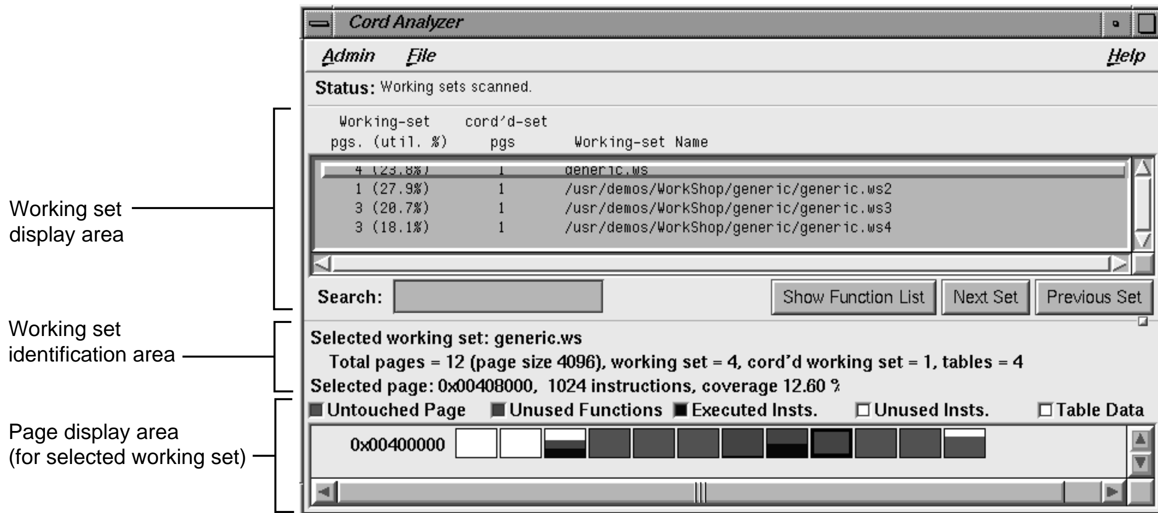

Figure 4-29 The **Cord Analyzer** Window

Page Display Area

The page display area at the bottom of the window shows the starting address for the DSO and its pages, and their use in terms of untouched pages, unused functions, executed instructions, unused instructions, and table data related to `rld`(1). It includes a color legend at the top to indicate how pages are used.

Function List

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

Use — Count of the working sets containing the function.

Address — Starting address for the function.

Insts. — Number of instructions in the function.

Function (File) — Name of the function and the file in which it occurs.

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

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

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

Admin Menu

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

Save Working Set List

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

File Menu

The **File** menu contains the following selections:

Delete All Working Sets

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

Delete Selected Working Set

Removes the selected working set from the working set list.

Add Working Set

Includes a new working set in the working set list.

Add Working Set List from File

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

Construct Gray-code Cording Feedback

Generates an ordering to minimize the working sets, placing the highest priority set first. It compacts each set and orders it to minimize the transitions between each set and the one that follows.

Gray code is believed to be superior to weighted ordering, but you might want to experiment with them both.

Construct Weighted Cording Feedback

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

Construct Union of Selected Sets

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

Construct Intersection of Selected Sets

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

Read Feedback File

Loads a new cord mapping file into the Cord Analyzer.

Glossary

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

bead
A record in an experiment file.

blocking
Waiting in the kernel for a resource to become available.

caliper points
Markers in the time domain that can be used to delimit a performance analysis. For instance, you may want to analyze only the CPU-bound part of your code.

call stack
A software stack of functions and routines used by the running program. The functions and routines are listed in the reverse order, from top to bottom, in which they were called. If function `a` is immediately below function `b` in the stack, then `a` was called by `b`. The function at the bottom of the stack is the one currently executing.

collective calls
Move a message from one processor to multiple processors or from multiple processors to one processor.

context switch
When the system scheduler stops a job from executing and replaces it with another job.

cord mapping file
A file containing a list of functions, their sizes, and their addresses.

CPU time
Process virtual time (see the glossary entry) plus time spent when the system is running on behalf of the process, performing such tasks as executing a system call. This is the time returned in `pcsamp` and `usertime` experiments.

disassembly
Assembly language version of the program.

exclusive time
The time spent only in the function itself, not including any functions it might call.

heartbeat resource data
Resource usage data (such as CPU time, wait time, I/O transfers, and so on) recorded at regular intervals. The `cvperf` usage view graphs are drawn using this data.

HIPPI
The High Performance Parallel Interface is a network link, often used to connect computers. It is slower than shared memory transfers but faster than TCP/IP transfers.

inclusive time
The total time spent in a function and all the functions it calls.

instrumenting
A method of collecting data by inserting code into the executable program to count events, such as the number of times a section of the program executes.

interlock
A feature of the CPU that causes a stall when resources are not available.

load imbalance
When the work in a parallel program is not evenly distributed among the processors, causing some processors to wait while the others finish their tasks.

memory leak
Making `malloc` calls without the corresponding calls to `free`. The result is, the amount of heap memory used continues to increase as the process runs.

memory page
The smallest unit of memory handled by the operating system. It is usually either 4 or 16 Kbytes.

page fault
A problem resulting in the possible loss of data. A high page fault rate is an indication of a memory-bound situation.

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

phase
A part of a program that concentrates on a single activity. Examples are the input phase, the computation phase, and the output phase.

point-to-point call
Moves a message from one processor to another single processor.

pollpoint
A regular time interval at which performance data is captured.

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

the time when the operating system is in control, such as executing a system call for the program.

profiling
A method of collecting data by periodically examining and recording the program's program counter (PC), call stack, and hardware counters that measure resource consumption.

profiling time
This is the same as *CPU time*.

real time
The same as *wall–clock time*.

sample event
A point in the program at which the PC or some resource is sampled.

system time
The time during a program's execution during which the system has control. It could be performing I/O or executing a system call.

TCP/IP
A networking protocol that moves data between two systems on the Internet.

thrashing
Accessing data from different parts of memory, causing frequent loads of pages of memory into cache. Using random access on an array might be an example.

threshold
An upper limit. For example, in the Source View, any line of code that exceeds a threshold of resource usage is flagged in the display.

total time
The same as *wall-clock time*.

user time
The same as *CPU time*.

virtual address
A location in memory as it appears in a program. For example, `a[10]` is the virtual address of element `10` of the array `a`. Internally, the virtual address is translated into the computer's physical address.

virtual time
The same as *process virtual time*.

wall-clock time
The total time a program takes to execute, including the time it takes waiting for a CPU. This is real time, not computer time.

working set
Executable pages, functions, and instructions that are actually brought into memory during a phase or operation of the executable.

Index

B

bad frees, 23
Butterfly button, 101

C

calipers, 72
call graph, 104
call stack data collection, 60
Call stack window, 114
chain operation, 101
Charts menu, 89
Context switch stripchart, 87
cord analyzer, 26, 123
CPU time, 49
custom task, 54

D

Disassembled source button, 71

E

environment variable
    _SPEEDSHOP_OUTPUT_FILENAME, 31
experiments
    Performance Analyzer, 45

F

floating point exception trace, 52
function list, 69, 106

H

Heap view, 110
Heap view tutorial, 42
Hide 0 functions toggle, 71

I

I/O trace, 52
ideal time task, 50

L

leak experiments, 40

M

Make source, 101
Make target, 101
malloc/free tracing, 63
memory leak experiments, 40
memory leakage, 23
memory leaks, 52
memory problems, 22

O

Overview button, 105

P

Page faults stripchart, 87

PC Sampling, 48
performance analysis theory, , 4, 1
Performance Analyzer
  experiments, 45
  tasks, 48
  tutorial, 29
Performance panel, 46
poll and I/O calls stripchart, 88
poll system calls, 88
pollpoint sampling, 65
Process Meter, 89
Process size stripchart, 88

R

Read/Write
  data size stripchart, 88
  system calls stripchart, 88
Realign button, 105
resource usage data, 8
Rotate button, 105

S

sample traps, 47
Scale menu, 89
Search field, 71
select system calls, 88
Show node button, 71
Source button, 71
Source view with leak annotations, , 109

T

target directory, 67
Trace I/O, 52
tracing data, 62

U

unmatched frees, 23
user vs system time stripchart, 87

W

working set analysis, 116
Working Set View, 25, 119

Z

Zoom in, 105
Zoom menu, 105
Zoom out, 105