Quantify User's Guide

support@rational.com
http://www.rational.com

RATIONAL
SOFTWARE

Contents

Welcome to Quantify

Using this guide ix
Using online Help ix
Conventions used in this guide x
Displaying the release notes x
Installing Quantify x
Contacting technical support xi

1 Introducing Quantify

How Quantify works 1-2
Building and running a Quantify'd program 1-3
Interpreting the program summary 1-4
Using Quantify's data analysis windows 1-5
The Function List window 1-6
Sorting the function list 1-6
Restricting functions 1-7
Finding and filtering functions 1-7
The Call Graph window 1-9
Understanding the layout of the call graph 1-10
Using the pop-up menu 1-11
Expanding and collapsing descendants 1-11
Locating functions in the call graph 1-12
Changing the focus of the call graph 1-13
Displaying additional data for functions 1-14
Changing line scale factors 1-15
Shortening function names 1-16
Saving the call graph 1-16

The Function Detail window 1-17
Understanding how time is distributed 1-18
Changing the scale and precision of data 1-19
Saving function detail data 1-19
The Annotated Source window 1-20
Changing annotations 1-21
What annotations mean 1-21
Saving performance data 1-22
Customizing Quantify’s graphical interface 1-23
Editing the .qvrc file 1-23

2 Improving Performance with Quantify

The hashtable package 2-1
Collecting baseline performance data 2-4
Uncovering an unexpected behavior 2-5
The function detail suggests long buckets 2-6
Annotated source confirms excessive calls 2-8
Saving the baseline data 2-9
Improving the performance of the testHash program 2-10
Running the improved_testHash program 2-12
Verifying the performance improvement 2-14
Other causes of slow software 2-15
Needless computation 2-15
Premature computation 2-16
Needless recomputation 2-16
Inefficient computation 2-16
Needless library or system-call requests 2-16
Excessive library or system-call requests 2-17
Expensive library or system-call requests 2-17
Environmental factors 2-17

3 How Quantify Collects Data

How Quantify records function time 3-2

How Quantify names functions 3-3
Analyzing basic blocks 3-4
How Quantify identifies basic blocks 3-5
How Quantify reports multiple basic blocks 3-9
Annotations and compiler differences 3-9
C++ templates and annotated source 3-11
How Quantify times system calls 3-13
Variations in system-call timing 3-14
How Quantify times register-window traps 3-15
Understanding recursive functions 3-16
Running on a different machine 3-20
Paging and memory cache effects 3-20

4 Customizing Data Collection

Avoiding all data recording 4-2
Timing system calls 4-2
Avoiding timing for all system calls 4-2
Controlling how system calls are timed 4-4
Avoiding timing for specific system calls 4-5
Reporting excluded system-call time 4-6
Timing shared-library operations 4-7
Recording dynamic linking 4-7
Understanding shared-library operations 4-8
Timing register-window traps 4-10
Changing the granularity of collected data 4-11
Collecting partial data for a run 4-13
Analyzing datasets containing partial data 4-14
Calling quantify _stop_recording_system_calls
and quantify_clear_data from your debugger 4-15
Annotating datasets 4-16
Saving data on signals 4-17
Collecting data in long-running programs 4-18

Collecting data for child processes 4-18
Using execve 4-19
Using fork 4-19
Using vfork 4-19
Collecting data in threaded programs 4-20
Threads and stacks 4-20
Solaris lightweight processes and threads 4-21
Analyzing data from threaded programs 4-22

5 Analyzing Data with Scripts

Exporting performance data 5-1
The export data file format 5-1
Running qv 5-2
Rerouting Quantify's output 5-3
Saving data incrementally 5-4
Automating data analysis 5-5
Automating performance regression tests 5-5
Comparing program runs with qxdiff 5-7
What qxdiff annotations mean 5-8
Managing cached object files 5-10
Deleting cached object files 5-10

A Using Quantify Options and API Functions

Using Quantify options A-2
Option syntax A-2
Using conversion characters in filenames A-3
How Quantify creates filenames A-4
Option types A-5
How Quantify processes options A-6
Using the -ignore-runtime-environment option A-8
Using Quantify API functions A-9
Calling API functions from a debugger A-9

Calling API functions from your program A-9
Linking with the Quantify stubs library A-10

B Options and API Reference

Build-time options quick reference B-1
qv options quick reference B-1
Run-time options quick reference B-2
Run-time API functions quick reference B-3
Build-time options B-4
qv options B-7
Data collection options B-8
Data collection API functions B-10
Run-time collection status API functions B-12
Threads options B-12
Child process options B-13
Options for saving data B-13
API functions for saving data B-14
Options for saving data on signals B-15
Options for automating data analysis B-15
Output options B-16
Output API functions B-16
Miscellaneous run-time options B-17

C Common Questions

Questions about instrumentation C-1
Questions about data collection C-3
Questions about data analysis C-6
Questions about performance C-6

Quantify Quick Reference

Index

Welcome to Quantify

Using this guide

This guide documents the features and capabilities of Quantify:

- Chapter 1, "Introducing Quantify," provides an overview of how to use Quantify.
- Chapter 2, "Improving Performance with Quantify," is a short tutorial that demonstrates how to use Quantify to improve the performance of a hashtable program.
- Chapter 3, "How Quantify Collects Data," explains how Quantify times a program's execution.
- Chapter 4, "Customizing Data Collection," explains how to control Quantify's data collection.
- Chapter 5, "Analyzing Data with Scripts," describes how to automate Quantify's operation by using scripts.
- Appendix A, "Using Quantify Options and API Functions," explains how to specify Quantify options and API functions.
- Appendix B, "Options and API Reference," provides a complete reference of all Quantify options and API functions.
- Appendix C, "Common Questions," provides answers to the most frequently asked questions about Quantify.

Using online Help

Quantify provides online Help through the Help menu in each data analysis window. If you select **On Context** from the Help menu, the cursor becomes a question mark (?). Click on any component of the window for specific information about that component.

Conventions used in this guide

- `<quantifyhome>` refers to the directory where Quantify is installed. To find the Quantify directory on your system, type:

  ```
  % quantify -printhomedir
  ```

- `Courier` font indicates source code, program names or output, file names, and commands that you enter.
- Angle brackets < > indicate variables.
- *Italics* introduce new terms and show emphasis.
- Sun OS4 — This icon appears next to instructions for the Sun SPARC SunOS 4 operating system.
- Solaris — This icon appears next to instructions for the Sun SPARC Solaris 2 operating system, also referred to as SunOS 5.
- HPUX — This icon appears next to instructions for the HP-UX operating system.
- IRIX — This icon appears next to instructions for the Silicon Graphics IRIX operating system.

Displaying the release notes

The Quantify `README` file is located in the `<purifyhome>` directory. It contains the latest information about this release of Quantify, including hardware and software supported, and notes about specific operating systems. To open the `README` file, select **Help > Release Notes**.

Installing Quantify

For information about licensing and installing Quantify, see the *Installation and Licensing Guide* for Rational Purify, Quantify, and PureCoverage.

Contacting technical support

If you have a technical problem and you can't find the solution in this guide, contact Rational Software Technical Support. See the back cover of this guide for addresses and phone numbers of technical support centers.

Note the sequence of events that led to the problem and any program messages you see. If possible, have the product running on your computer when you call.

For technical information about Quantify, answers to common questions, and information about other Rational Software products, visit the Rational Software World Wide Web site at `http://www.rational.com`. To contact technical support directly, visit: `http://www.rational.com/support`.

1 Introducing Quantify

Your application's run-time performance—its speed—is one of its most visible and critical characteristics. Developing high-performance software that meets the expectations of customers is not an easy task. Complex interactions between your code, third-party libraries, the operating system, hardware, networks, and other processes make identifying the causes of slow performance difficult.

Quantify is a powerful tool that identifies the portions of an application that dominate its execution time. Quantify gives you the insight to quickly eliminate performance problems so that your software runs faster. With Quantify, you can:

- Get accurate, repeatable performance data
- Control how data is collected by collecting data for a small portion of your application's execution or the entire run
- Compare *before* and *after* runs to see the impact of your changes on performance
- Easily locate and fix the problems with the highest potential for improving performance

Unlike sampling-based profilers, Quantify's reports do not include any overhead. The numbers you see represent the time your program would take *without* Quantify.

Quantify can be installed and mastered in a few hours. You incorporate it into your development process simply by adding a single word to your makefile and relinking. You don't need to recompile to use it, and you can use it on the exact code you expect to ship. Quantify works with existing makefiles, debugging tools and standard testing scripts.

Quantify instruments *all* the code in your program, including system and third-party libraries, shared libraries, and statically linked modules.

How Quantify works

Quantify counts machine cycles: Quantify uses Object Code Insertion (OCI) technology to count the instructions your program executes and to compute how many cycles they require to execute. Counting cycles means that the time Quantify records in your code is identical from run to run, assuming that the input does not change. This complete repeatability enables you to see precisely the effects of algorithm and data structure changes.

Since Quantify counts cycles, it gives you accurate data at any scale. You do *not* need to create long runs or make numerous short runs to get meaningful data as you must with sampling-based profilers: One short run and you have the data. As soon as you can run a test program, you can collect meaningful performance data and establish a baseline for future comparison.

Quantify times system calls: Quantify measures the elapsed time (wall clock) of each system call made by your program and reports how long your program waited for those calls to complete. You can immediately see the effects of improved file access or reduced network delay on your program. You can optionally choose to measure system calls by the amount of time the kernel recorded for the process, much like the `/bin/time` UNIX utility records.

Quantify distributes time accurately: Quantify distributes each function’s time to its callers so you can tell at a glance which function calls were responsible for the majority of your program’s time. Unlike `gprof`, Quantify does not make assumptions about the average cost per function. Quantify measures it directly.

Building and running a Quantify'd program

To instrument your program, add quantify to the front of the *link* command line. For example:

Add "quantify"

```
% quantify cc -g hello_world.c -o hello_world
```

Instrumenting files

```
Quantify 4.4 SunOS 4.1, Copyright 1993-1999 Rational Software Corp.
Instrumenting: hello_world.o Linking
```

Run your Quantify'd program normally:

Run the program

```
% hello_world
```

When the program starts, Quantify prints license and support information, followed by the expected output from your program.

```
**** Quantify instrumented hello_world (pid 20352 at Jan 7 08:41:27
1999)
Quantify 4.4 SunOS 4.1, Copyright 1993-1999 Rational Software Corp.
    * For contact information type: "quantify -help"
    * Quantify licensed to Quantify Evaluation User
    * Quantify instruction counting enabled.
```

Program output

```
Hello, World.
```

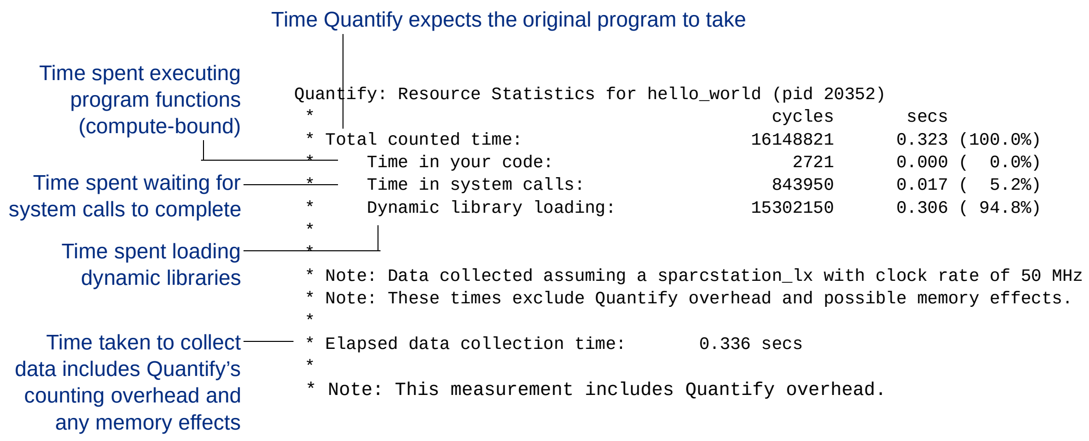
Quantify transmits the dataset to qv

```
Quantify: Sending data for 37 of 1324 functions
 from hello_world (pid 20352).........done.
```

When the program finishes execution, Quantify transmits the performance data to qv, Quantify's data analysis program.

Interpreting the program summary

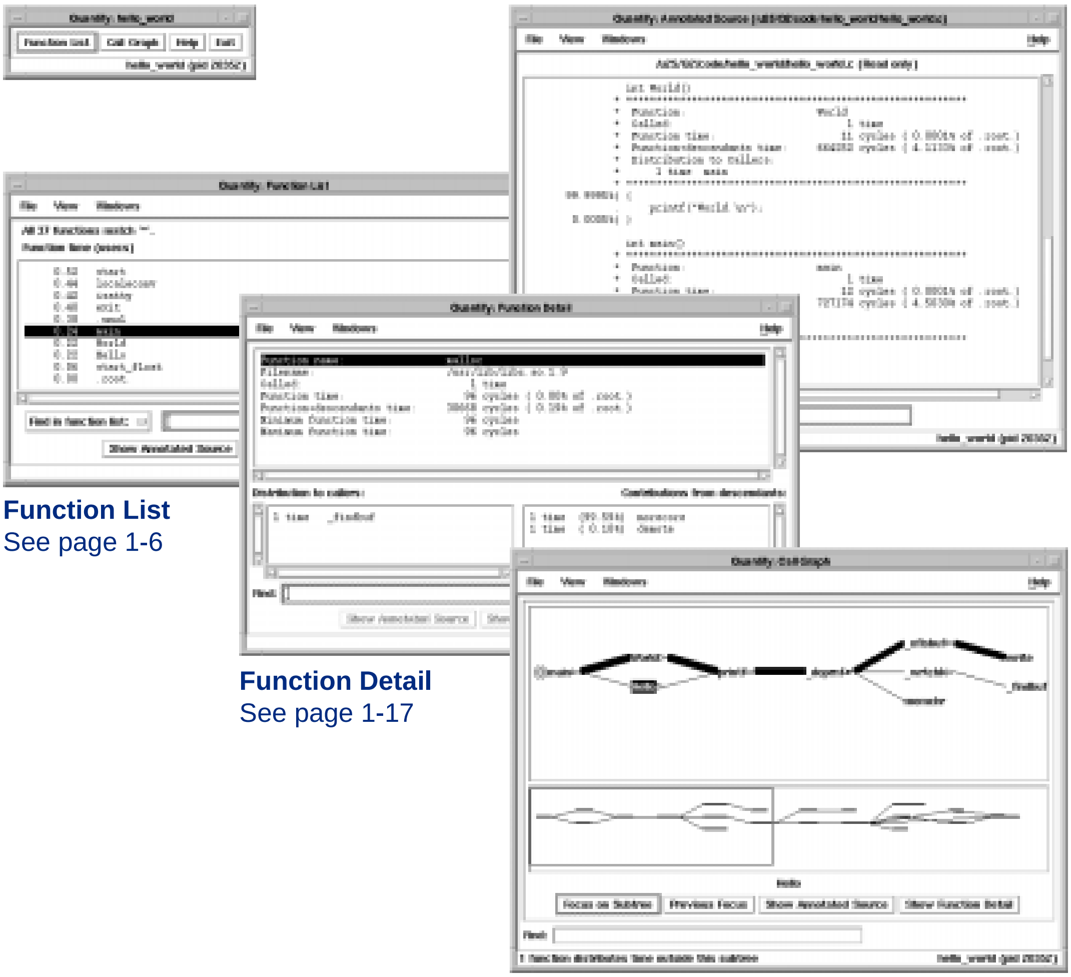
After each dataset is transmitted, Quantify prints a program summary showing at a glance how the original, non-Quantify'd program is expected to perform.


Time Quantify expects the original program to take
Time spent executing program functions (compute-bound)
Time spent waiting for system calls to complete
Time spent loading dynamic libraries
Time taken to collect data includes Quantify's counting overhead and any memory effects

```
Quantify: Resource Statistics for hello_world (pid 20352)
 *                                             cycles       secs
 * Total counted time:                       16148821      0.323 (100.0%)
 *     Time in your code:                        2721      0.000 (   0.0%)
 *     Time in system calls:                   843950      0.017 (   5.2%)
 *     Dynamic library loading:              15302150      0.306 (  94.8%)
 *
 *
 * Note: Data collected assuming a sparcstation_lx with clock rate of 50 MHz
 * Note: These times exclude Quantify overhead and possible memory effects.
 *
 * Elapsed data collection time:       0.336 secs
 *
 * Note: This measurement includes Quantify overhead.
```

Using Quantify’s data analysis windows

After transmitting the last dataset, Quantify displays the Control Panel. From here, you can display Quantify’s data analysis windows and begin analyzing your program’s performance.



Control Panel

Annotated Source
See page 1-20

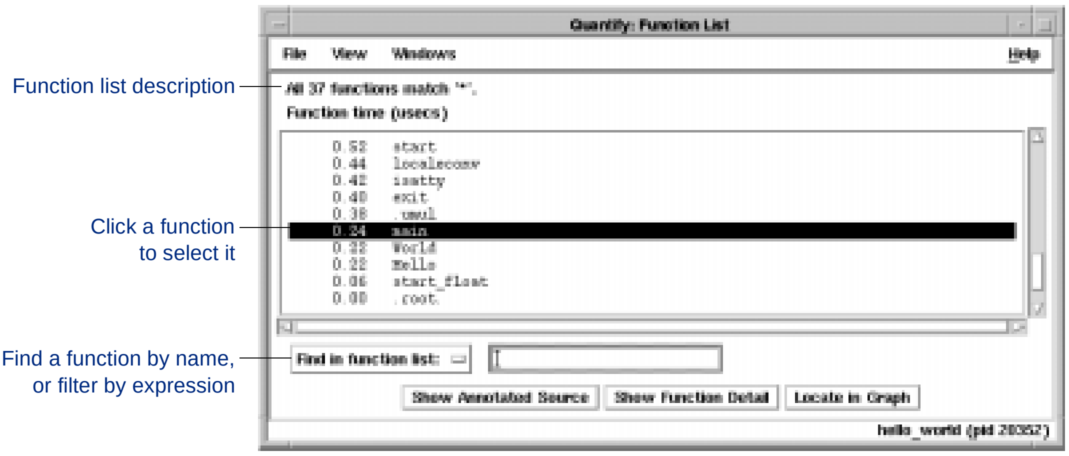
Function List
See page 1-6

Function Detail
See page 1-17

Call Graph
See page 1-9

The Function List window

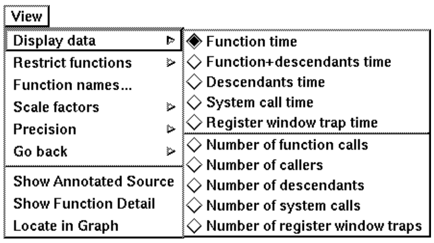
The Function List window shows the functions that your program executed. By default, it displays the top 20 most expensive functions in your program, sorted by their *function time*. This is the amount of time a function spent in your code performing computations (*compute-bound*) or waiting for system calls to complete.



```
All 37 functions match "*".
Function time (usecs)
      0.52   start
      0.44   localeconv
      0.42   isatty
      0.40   exit
      0.38   .umul
      0.24   main
      0.22   World
      0.22   Hello
      0.06   start_float
      0.00   .root
hello_world (pid 20352)
```

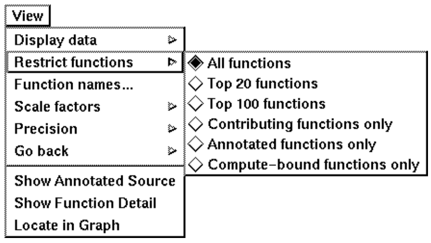
Sorting the function list

To sort the function list based on the various data Quantify collects, select **View > Display data**.



Restricting functions

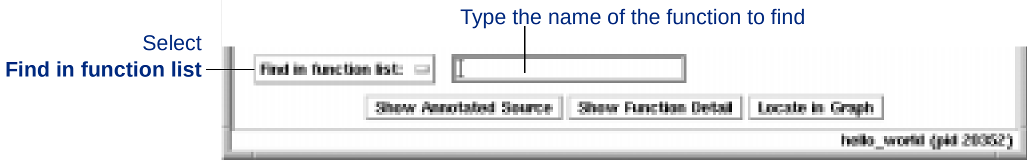
To focus attention on specific types of functions, or to speed up the preparation of the function list report in large programs, you can restrict the functions shown in the report. Select **View > Restrict functions**.



You can restrict the list to the top 20 or top 100 functions in the list, to the functions that have annotated source, to functions that are compute-bound (make no system calls), or to functions that contribute non-zero time for a recorded data type.

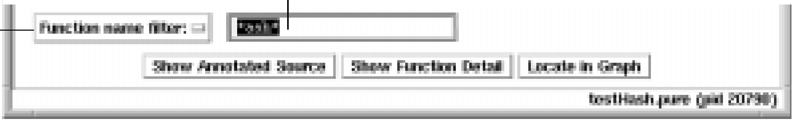
Finding and filtering functions

To search the function list for a specific function, type its name in the **Find in function list** entry field.



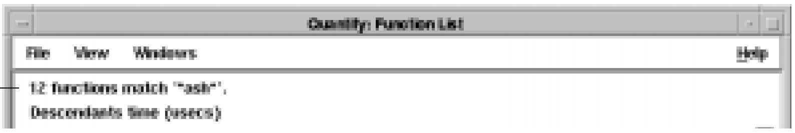
You can also filter the function list. This is useful when you have groups of related functions that contain common substrings in their names. You can search for functions with substrings such as `str` and `mem`. With C++ programs, you can find functions that are defined on different classes, such as "`List::`", and that use certain types in their argument lists such as "`(int,`".

To filter the list of functions, type a regular expression in the **Function name filter** entry field.



You can use * and ? wildcards: * represents zero or more characters; ? represents any single character. Matching is case-sensitive. The initial filter expression is *, matching all functions.

The lines at the top of the function list describe the number of functions included in the report that satisfy any restrictions and the filter expression.

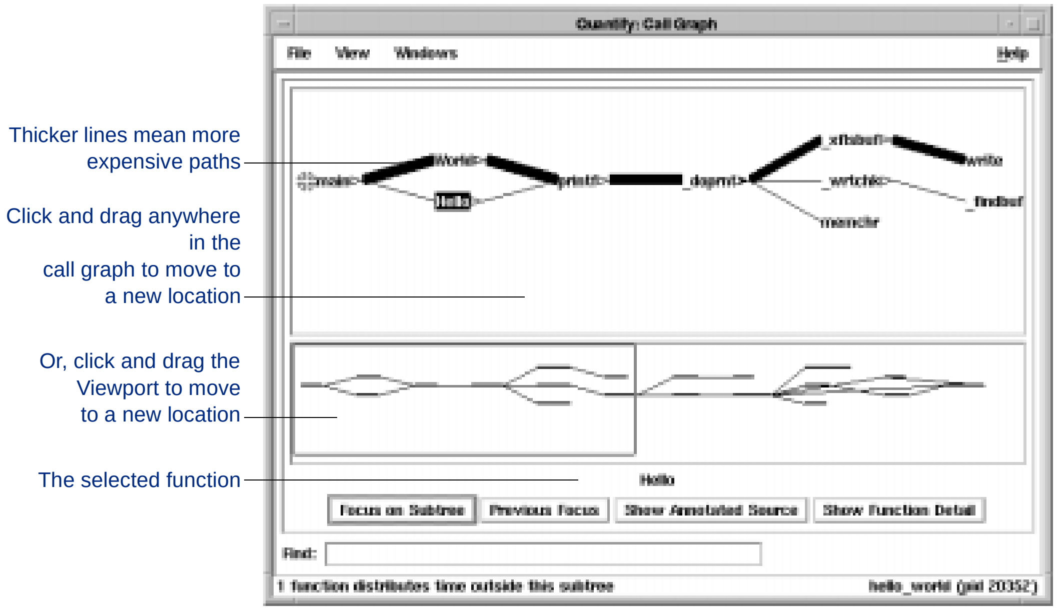

The description of the restrictions or filters that you apply
Quantify: Function List
File
View
Windows
Help

```
12 functions match "*ash*".
Descendants time (usecs)
```

The Call Graph window

The Call Graph window presents a graph of the functions called during the run. It uses lines of varying thickness to graphically depict where your program spends its time. Thicker lines correspond directly to larger amounts of time spent along a path.

The call graph helps you understand the calling structure of your program and the major call paths that contributed to the total time of the run. Using the call graph, you can quickly discover the sources of bottlenecks.



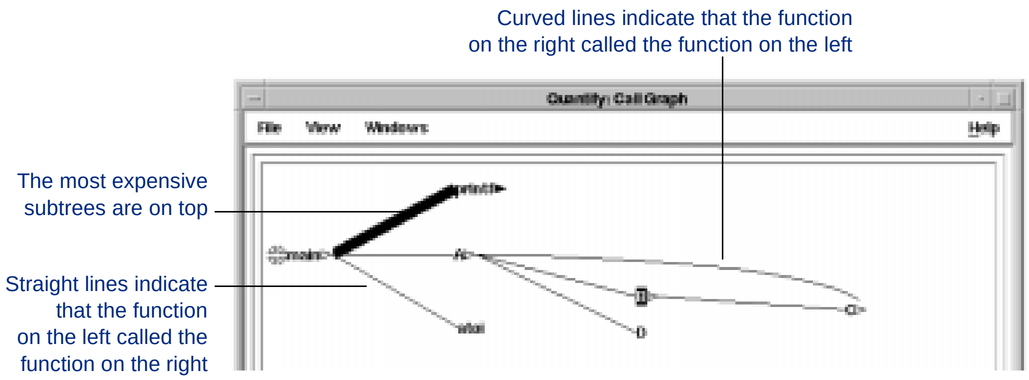
By default, Quantify expands the call paths of the top 20 functions that contribute most to the overall time of the program. The call graph begins at the .root. accumulator, which represents the total amount of time used by all the functions in the program. For more information about the .root. accumulator, see “How Quantify records function time” on page 3-2.

Understanding the layout of the call graph

In single-threaded applications, the call graph begins at the `.root.` function, with one or two branches emanating from it. One branch corresponds to the initialization sequence of the program before it reaches your program's `main` function. This branch also contains C++ static initialization and `at_exit` processing, if any.

The other branch starts at `pure_sigtramp` and is present only if your program handled any signals. Multi-threaded applications can have several additional branches emanating from the `.root.` function. See "Collecting data in threaded programs" on page 4-20.

Curved lines in the call graph often indicate the presence of recursive functions or cycles of function calls. For more information, see "Understanding recursive functions" on page 3-16.

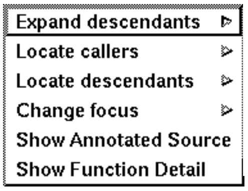

Curved lines indicate that the function on the right called the function on the left
Quantify: Call Graph
File
View
Windows
Help
The most expensive subtrees are on top
Straight lines indicate that the function on the left called the function on the right
printf
main
A
B
C
D
wait

If a function is not a leaf function, a triangle indicates how many of each function's immediate descendants are shown in the call graph.

- None of the immediate descendants are shown.
- Some of the immediate descendants are shown.
- All of the immediate descendants are shown.

Using the pop-up menu

To display the pop-up menu, right-click any function in the call graph.

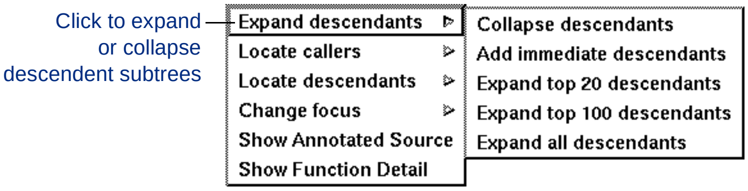


You can use the pop-up menu to:

- Expand and collapse the function's subtree
- Locate individual caller and descendant functions
- Change the focus of the call graph to the selected function
- Display the annotated source code or the function detail for the selected function

Expanding and collapsing descendants

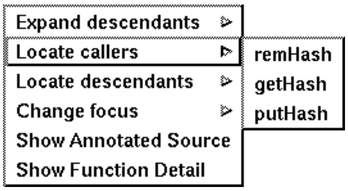
Use the pop-up menu to expand or collapse the subtrees of descendants for individual functions.



After expanding or collapsing subtrees, you can select **View > Redo layout** to remove any gaps that your changes create in the call graph.

Locating functions in the call graph

To locate a particular caller or descendant of the selected function, select **Locate callers** or **Locate descendants** from the pop-up menu.



Quantify sorts the descendants by their contribution to the function's accumulated time.

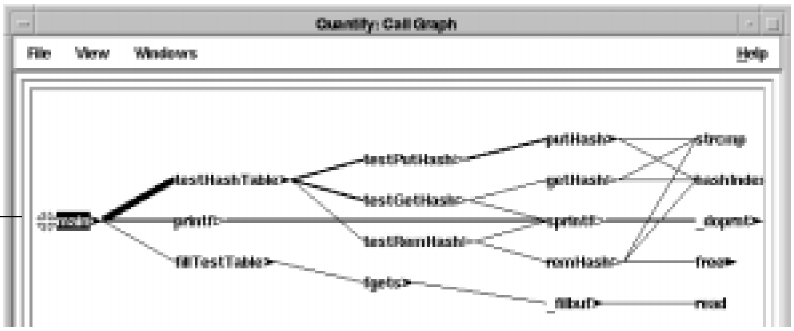
If the function you are trying to locate is not present in the call graph, Quantify displays the most expensive path from the function that is the current focus to the desired function.

If the function is not in the descendant subtree of the current focus (for example, if it is the *caller* of the current focus), Quantify attempts to find a common function whose descendant subtree contains both the current focus (and hence current selection) and the desired function. It automatically makes that function the new focus and then displays the most expensive path from the new focus to the desired function.

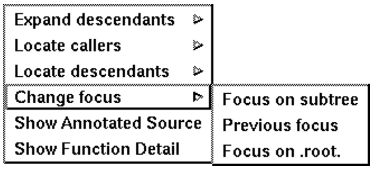
Changing the focus of the call graph

You can change the focus of the call graph to a different function.

The current focus is the function with the *cross* icon



To change the focus of the call graph to a new function, right-click the desired function and select **Change focus > Focus on subtree** from the pop-up menu.



The selected function becomes the basis for scaling and expanding operations. For example, you can select **View > Scale factors > % of focus** to see numeric data scaled for the current focus.

Functions with multiple callers

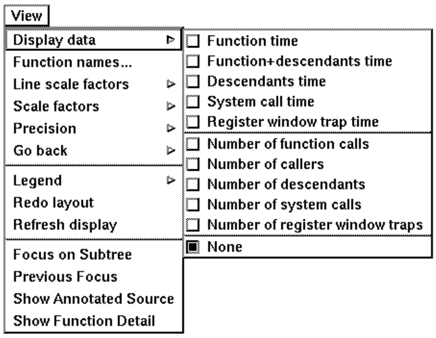
The subtree of the function you select as the new focus of the call graph might contain functions that are called from *outside* of the subtree. Consider, for example, the functions `malloc` and `free`. Both of these functions are called from many different functions in a typical C program.

If you select one of `malloc`'s callers as the new focus, its other callers are excluded from the subtree. In this case, `malloc`'s function+descendants time can *exceed* that of its individual callers. This is because `malloc` distributed the additional time to the callers that are excluded from the subtree. If you display `malloc`'s function+descendants time as a percentage of the *focus* function's function+descendants time, this percentage will *exceed* 100 percent. Quantify reports the percentage as "100+%."

Quantify warns you if the subtree of the new focus contains any functions that distribute time to functions outside of the subtree. Quantify displays a count of these functions in the status bar at the bottom of the call graph and places an asterisk (*) next to their names in the call graph.

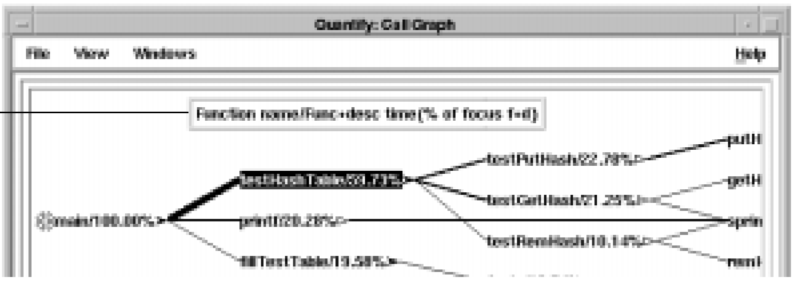
Displaying additional data for functions

To display additional data for functions in the call graph, select **View > Display data**. Select **None** to display functions without any additional data.



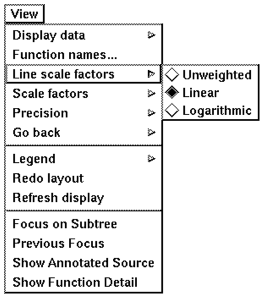
Quantify describes the data that is displayed for each function in the legend at the top of the call graph. To hide or show the legend, select **View > Legend**.

The legend describes the data displayed for the functions



Changing line scale factors

To change the scale of the lines in the call graph, select **View > Line scale factors**.



You can select from three line scale factors:

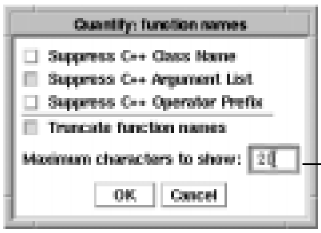
- *Unweighted* lines show only the calling relationships between functions. These lines do not carry information about how time is distributed between functions.
- *Linear* scaling shows the distributed time from a descendant function to its caller as a percentage of the total time of the current focus. This is the default line scale factor.
- *Logarithmic* scaling shows the distributed time from a descendant function to its caller as the logarithm of the percentage of the total time of the current focus. This scaling

de-emphasizes the rapid accumulation of time near the root of a subtree while emphasizing small differences between contributing functions deep in the subtree.

Shortening function names

You can shorten function names in order to tighten up the display in the call graph. Select **View > Function names**.

You can shorten function names by eliminating the C++ class name, argument list, or operator prefix. You can also specify a custom truncation length. In cases where demangling the full C++ name is ambiguous, Quantify prints ??? in the function's argument list.

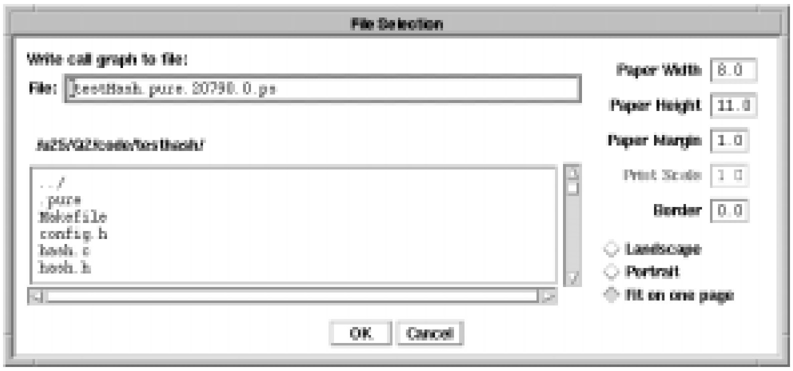


☐ Suppress C++ Class Name
▨ Suppress C++ Argument List
☐ Suppress C++ Operator Prefix
▨ Truncate function names
Maximum characters to show: 20

Specify a custom truncation length

Saving the call graph

To save a PostScript version of the current call graph, select **File > Save call graph as**.



Write call graph to file:
File: testHash.pure.20790.0.ps

/a25/Q2/code/testhash/

```
../
.pure
Makefile
config.h
hash.c
hash.h
```

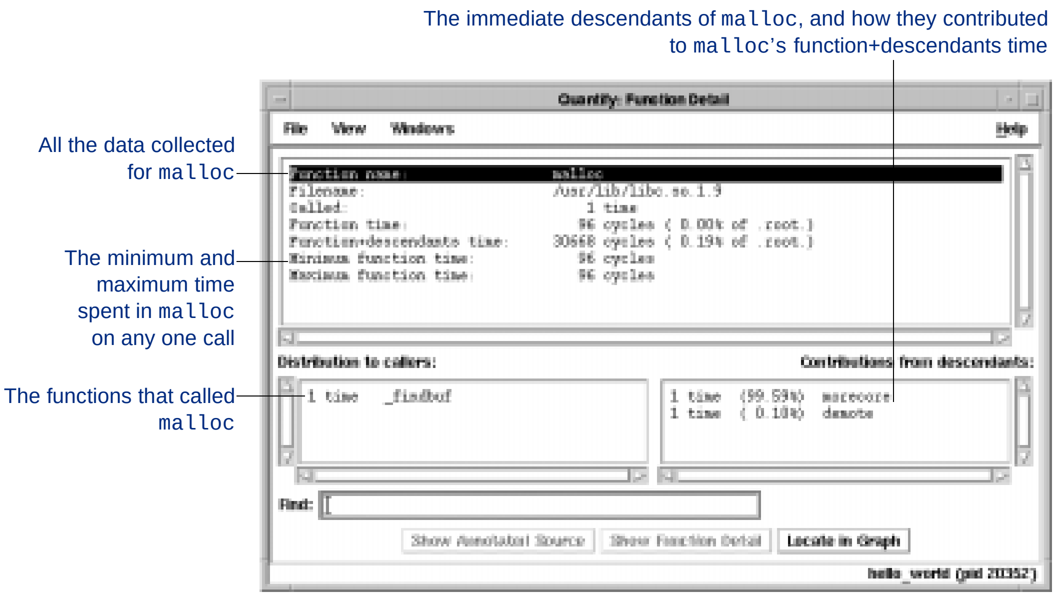
Paper Width 8.0
Paper Height 11.0
Paper Margin 1.0
Print Scale 1.0
Border 0.0

○ Landscape
○ Portrait
◆ Fit on one page

The Function Detail window

The Function Detail window presents detailed performance data for a single function showing its contribution to the overall execution of the program.

For each function, Quantify reports both the time spent in the function's own code (its *function* time) and the time spent in all the functions that it called (its *descendants* time). Quantify distributes this accumulated *function+descendants* time to the function's immediate caller.


The immediate descendants of malloc, and how they contributed to malloc's function+descendants time
All the data collected for malloc
The minimum and maximum time spent in malloc on any one call
The functions that called malloc
Quantify: Function Detail
File
View
Windows
Help
Distribution to callers:
Contributions from descendants:
Find:
Show Annotated Source
Show Function Detail
Locate in Graph
hello_world (pid 20352)

```
Function name:                 malloc
Filename:                      /usr/lib/libc.so.1.9
Called:                           1 time
Function time:                   96 cycles ( 0.00% of .root.)
Function+descendants time:    30668 cycles ( 0.19% of .root.)
Minimum function time:           96 cycles
Maximum function time:           96 cycles
```

```
1 time   _findbuf
```

```
1 time  (99.59%)  morecore
1 time  ( 0.18%)  demote
```

Double-click a caller or descendant function to display function detail for that function.

The function time and the function+descendants time are shown as a percentage of `.root`, the total accumulated time for the entire run. These percentages help you understand how this function's computation contributed to the overall time of the run. These times correspond to the thickness of the lines in the call graph.

Note: If a function calls itself recursively, the percentages displayed in the Function Detail window can *exceed* 100 percent. See “Understanding recursive functions” on page 3-16.

Times for system calls and register window traps are shown only if times were recorded for the function. See “Timing system calls” on page 4-2, and “Timing register-window traps” on page 4-10.

Understanding how time is distributed

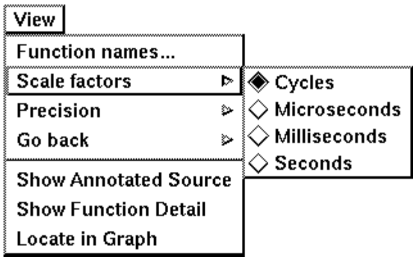
Distribution to callers lists all the functions that called the current function. For each caller, Quantify lists the number of times it called the function and the percentage of time that was spent in the current function and its descendants on behalf of that caller.

Contributions from descendants lists the immediate descendant functions called by the current function. For each descendant function, Quantify reports the number of times it was called by the current function and the percentage of time it contributed to the current function’s accumulated time.

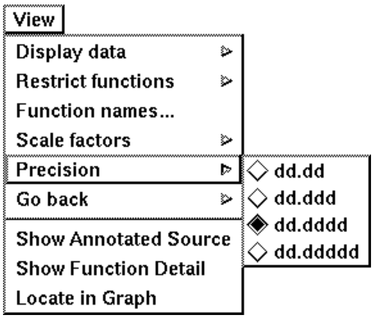
For more information, see “How Quantify records function time” on page 3-2.

Changing the scale and precision of data

Quantify can display the recorded data in cycles (the number of machine cycles) and in microseconds, milliseconds, or seconds. To change the scale of data, select **View > Scale factors**.



To change the precision of data, select **View > Precision**.



Saving function detail data

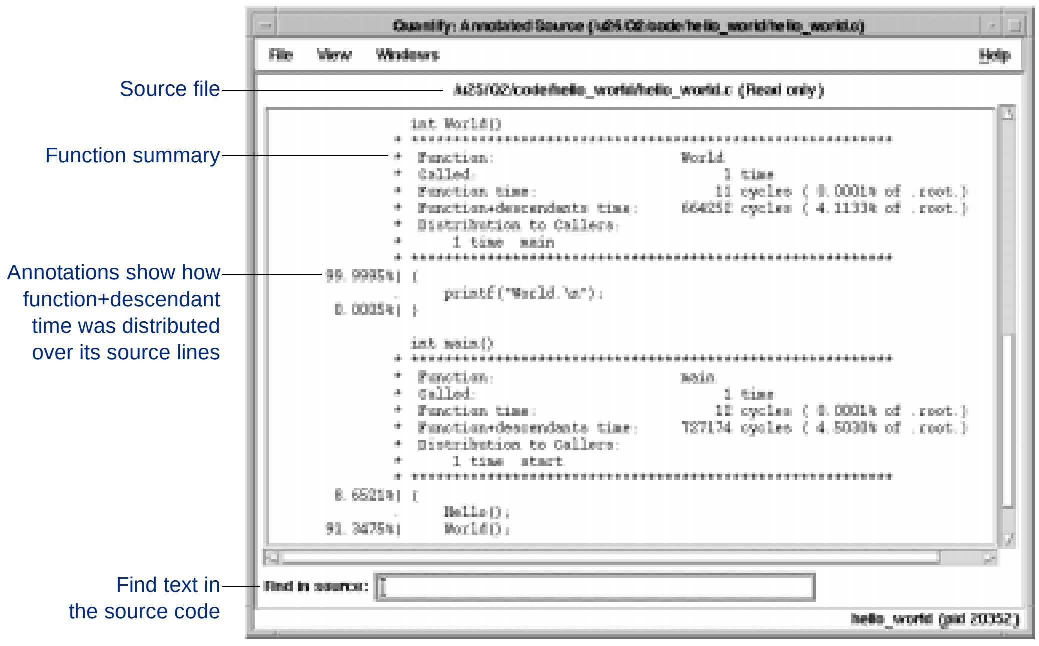
To save the current function detail display to a file, select **File > Save current function detail as**.

To append additional function detail displays to the same file, select **File > Append to current detail file**.

The Annotated Source window

The Annotated Source window presents line-by-line performance data using the function’s source code.

Note: The Annotated Source window is available only for files that you compile using the -g debugging option.


Source file
Function summary
Annotations show how function+descendant time was distributed over its source lines
Find text in the source code
Quantify: Annotated Source (/u25/G2/code/hello_world/hello_world.c)
File
View
Windows
Help
/u25/G2/code/hello_world/hello_world.c (Read only)
Find in source:
hello_world (pid 20352)

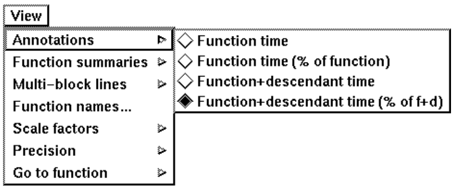
```
                 int World()
               * ************************************************************
               *  Function:                          World
               *  Called:                                 1 time
               *  Function time:                         11 cycles ( 0.0001% of .root.)
               *  Function+descendants time:     664252 cycles ( 4.1133% of .root.)
               *  Distribution to Callers:
               *      1 time  main
               * ************************************************************
       99.9995%| {
           .         printf("World.\n");
        0.0005%| }

                 int main()
               * ************************************************************
               *  Function:                          main
               *  Called:                                 1 time
               *  Function time:                         12 cycles ( 0.0001% of .root.)
               *  Function+descendants time:     727174 cycles ( 4.5030% of .root.)
               *  Distribution to Callers:
               *      1 time  start
               * ************************************************************
        8.6521%| {
           .         Hello();
       91.3475%|     World();
```

The numeric annotations in the margin reflect the time recorded for that line or basic block over all calls to the function. By default, Quantify shows the function time for each line, scaled as a percentage of the total function time accumulated by the function.

For more information about how Quantify reports data for lines and basic blocks, see “Analyzing basic blocks” on page 3-4.

Changing annotations

To change annotations, use the View menu. You can select both *function* and *function+descendants* data, either in cycles or seconds and as a percentage of the *function+descendants* time.

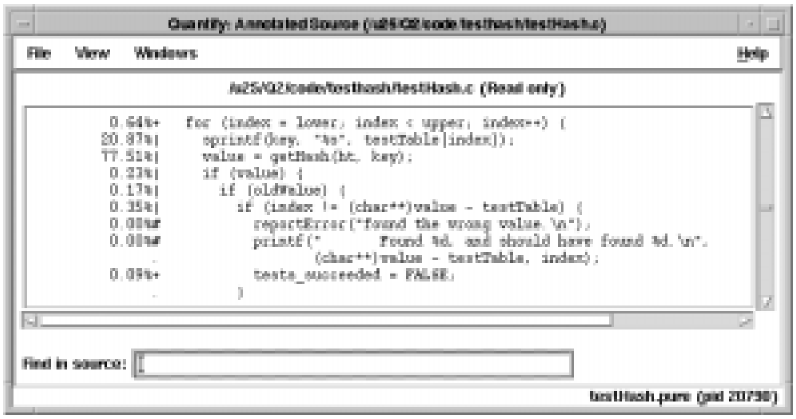


What annotations mean

Each source line in the Annotated Source window is marked with a character indicating the type of annotation.

Annotation	Meaning
*	A comment line added by Quantify. By default, Quantify inserts *function summaries*, comments for each function that reflect its detailed function data. To eliminate comments, select **View > Function summaries > Hide function summaries**.
|	The start of a single basic block or line. A basic block can span several lines.
.	The extent of basic blocks that span several lines.
+	A line containing multiple basic blocks.
#	A line containing basic blocks that were not executed. For more information about basic blocks, see “How Quantify reports multiple basic blocks” on page 3-9.

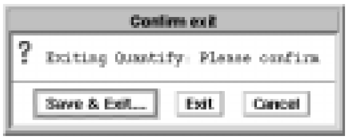
Here is an example of how annotations are used in the Annotated Source window:


Quantify: Annotated Source (/u25/Q2/code/testhash/testHash.c)
File
View
Windows
Help
/u25/Q2/code/testhash/testHash.c (Read only)
Find in source:
testHash.pure (pid 20790)

```
 0.64%+   for (index = lower; index < upper; index++) {
20.87%|     sprintf(key, "%s", testTable[index]);
77.51%|     value = getHash(ht, key);
 0.23%|     if (value) {
 0.17%|       if (oldValue) {
 0.35%|         if (index != (char**)value - testTable) {
 0.00%#           reportError("found the wrong value.\n");
 0.00%#           printf("       found %d, and should have found %d.\n",
      .                  (char**)value - testTable, index);
 0.09%+           tests_succeeded = FALSE;
      .         }
```

Saving performance data

To exit Quantify, select **File > Exit Quantify**. If you analyze a dataset interactively, Quantify does not automatically save the last dataset it receives. When you exit, you can save the dataset for future analysis.



Exiting Quantify. Please confirm.

By default, Quantify names dataset files to reflect the program name and its run-time process identifier. See “How Quantify creates filenames” on page A-4. You can analyze a saved dataset at a later time by running qv, Quantify’s data analysis program.

You can also save Quantify data in export format. This is a clear-text version of the data suitable for processing by scripts. See “Exporting performance data” on page 5-1 and “Saving data incrementally” on page 5-4.

Customizing Quantify’s graphical interface

You can modify the appearance of Quantify’s graphical interface using X resources. You can change:

- The windows that Quantify creates by default
- Fonts in titles and menus
- Foreground and background colors
- Colors of function lines in the Call Graph window

You can also specify the following default menu settings for each window:

- The data to display
- The scale factors to apply
- The precision to use
- C++ name suppressions to apply

Editing the .qvrc file

When you run Quantify interactively, in addition to any resources you have set using `.Xdefaults` (or the HP VUE resource support mechanism for HP-UX machines), Quantify loads any X resources specified in the .qvrc file in your home directory. If there is no .qvrc file in your home directory, Quantify creates one when you exit Quantify.

You can edit your `.qvrc` file to specify alternative values for Quantify’s X resources. The .qvrc file lists the values that can be changed and provides extensive comments about each value. You can remove the comments from the resource specifications and change the values. The changed values will take effect the next time you run Quantify.

If you need a fresh copy of the `.qvrc` file, delete the current version. The next time you run Quantify interactively, it will use the default settings, and then write a copy of the .qvrc file on exit.

2 Improving Performance with Quantify

This chapter provides a short tutorial using an example program called `testHash` that demonstrates how to use Quantify to improve the performance of a program. This chapter also describes some major causes of slow software.

Note: System call timing can vary due to the load on a machine. When you run the example program, the times that Quantify reports for system calls might be slightly different than the times shown in this chapter.

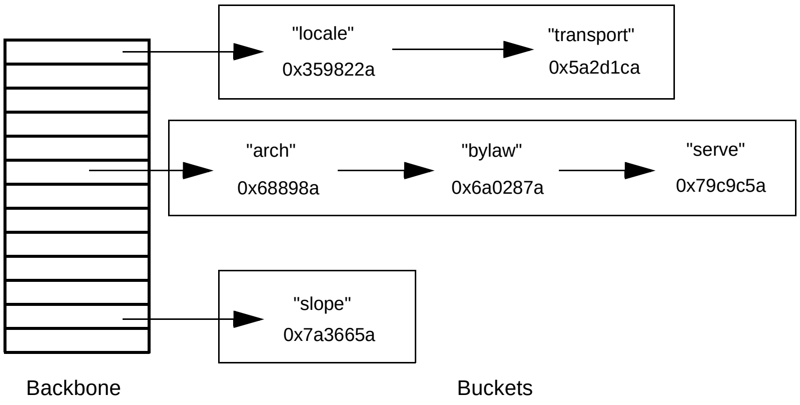
The hashtable package

Suppose you are part of a team developing a compiler and you are assigned the task of developing a symbol table that associates various programming language and user tokens with different parsing and lexical information. You implement a hashtable package and a unit test program to ensure that it works. Before you incorporate the hashtable package into the compiler, you use Quantify to find any performance bottlenecks.

Note: You can find the source code for the `testHash` program and the unit test program in `<Quantifyhome>/example_quantify`.

The unit test program reads a file of tokens and exercises the hashtable package against that test dataset. The hashtable package itself supports inserting, fetching, and deleting hashtable entries. Each hashtable is a fixed size array (called the “backbone”) containing pointers to a chain of hashtable entries.

The chain of hashtable entries from each array element is called a “bucket.”



Given a string token such as "`serve`", the hashtable package computes a 32-bit hash key, in this case `0x79c9c5a`, based on the characters in the string. The hashtable package uses the hashkey *modulo* the size of the hashtable backbone to determine what bucket to search and then scans the entries in the bucket looking for the entry with the same string.

Here is the source code for the `testHash` program.

```
/* hashEntry
 *   A hashEntry keeps the key, value pair together in a list hashEntries.
 */
typedef struct struct_hashEntry {
  char* key;                      /* The full string key for this entry */
  void* value;                   /* Pointer to user data indexed by key */
  struct struct_hashEntry* next;  /* Pointer to next bucket entry */
} hashEntry;

/* hashIndex
 *   Determines what bucket (index) to place a key in.
 *   Given a key it returns the index of the appropriate bucket.
 */
static int hashIndex(key)
     char* key;
{   char *p;
    unsigned h = 0, g;
    for (p = key; *p; p++) {
        h = (h << 4) + (*p);
        if ((g = h & 0xf0000000)) {
            h = h ^ (g >> 24);
            h = h ^ g;
        }
    }
    return h%hashtable_backbone_size;
}

/* getHash
 *   Hashes into the appropriate bucket, and then walks the
 *   chain of hash entries. It compares the keys with strcmp
 *   because the keys do not have to have the same pointer,
 *   just the same string. When a match is found the associated
 *   value is returned. NULL is returned if no match is found.
 */
void* getHash(ht, key)
     hashtable* ht;
     char* key;
{   hashEntry* entry;
    int index = hashIndex(key);
    for (entry = ht[index];(entry && strcmp(entry->key, key));
       entry = entry->next) {
    }
    if (entry)
       return (entry->value);
    else
       return (NULL);
}
```

Collecting baseline performance data

Build the testHash program using the makefile for your system. For example, on SunOS type:

```
% make -f makefile.sun testHash.pure
```

Run the `testHash` program with a test dataset.

```
% testHash.pure 500 test_words
```

Begin the analysis of the `testHash` program by reviewing Quantify's program summary:

The majority of time is spent computing

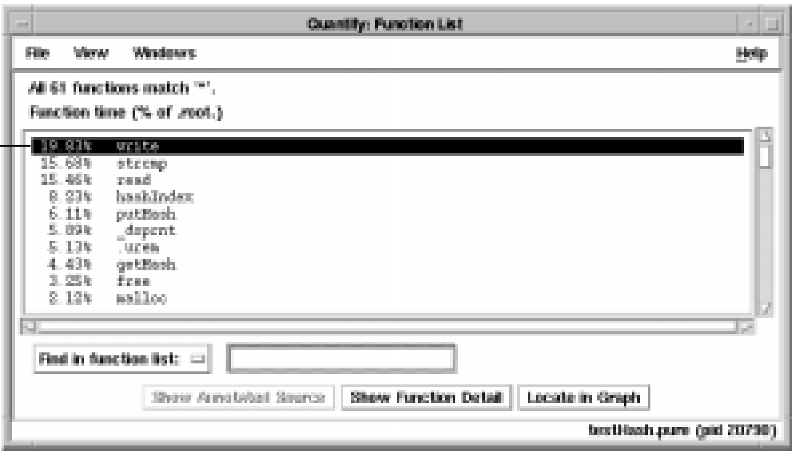
```
****  Quantify instrumented testHash.pure (pid 20790)
Quantify 4.4 SunOS 4.1, Copyright 1993-1999 Rational Software Corp.
  * For contact information type: "quantify -help"
  * Quantify licensed to Quantify Evaluation User.
  * Quantify instruction counting enabled.
Testing the first 500 entries from test_words with a hashtable of size 13.
All tests passed.

Quantify: Sending data for 61 of 187 functions
 from testHash.pure (pid 20790)..........done.

Quantify: Resource Statistics for testHash.pure (pid 20790)
 *                                             cycles       secs
 * Total counted time:                        8151282      0.163 (100.0%)
 *     Time in your code:                     5029082      0.101 ( 61.7%)
 *     Time in system calls:                  3122200      0.062 ( 38.3%)
 *
 * Note: Data collected assuming a sparcstation_lx with clock rate of 50 MHz.
 * Note: These times exclude Quantify overhead and possible memory effects.
 *
 * Elapsed data collection time:       1.962 secs
 *
 * Note: This measurement includes Quantify overhead.
 *
%
```

Testing the hashtable package involves only the compute-bound operations of inserting and removing items from a data structure. The only system calls you might expect during hashtable testing are the result of printing the test results via `printf` and requesting memory for the hashtable data structures via `malloc`. This means that you are interested in the major functions contributing to the `Time in your code` category and not in `Time in system calls` category.

Uncovering an unexpected behavior

Click **Function List** in the Control Panel to display a list of functions called in the `testhash` program.

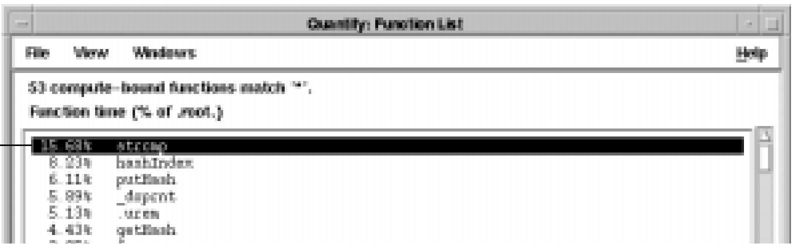
The main contributor to the hashtable test's time is `write`, the function used to print the test output from the program itself



```
Quantify: Function List
File   View   Windows                                  Help
All 61 functions match "*".
Function time (% of .root.)
 19.83%   write
 15.68%   strcmp
 15.46%   read
  8.23%   hashIndex
  6.11%   putHash
  5.89%   _doprnt
  5.13%   .urem
  4.43%   getHash
  3.25%   free
  2.12%   malloc
```

Since you are interested in the compute-bound functions of the `testHash` program, select **View > Restrict functions > Compute-bound functions only**. Quantify displays only the functions that made no operating system calls.

`strcmp` is the largest contributor to the overall execution time of the run, followed by `hashIndex`

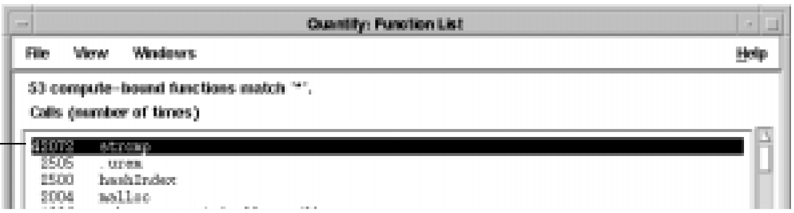


```
Quantify: Function List
File   View   Windows                                  Help
53 compute-bound functions match "*".
Function time (% of .root.)
 15.68%   strcmp
  8.23%   hashIndex
  6.11%   putHash
  5.89%   _doprnt
  5.13%   .urem
  4.43%   getHash
```

You would expect `hashIndex` to be high on the list since it is used to compute the hash keys, but `strcmp` is a surprise.

The `strcmp` function is considered to be efficient, so perhaps it was called a large number of times. Select **View > Display data > Number of function calls** to sort the compute-bound functions by the number of times they were called by any function.

`strcmp` remains at the head of the list with over 40,000 calls



```
53 compute-bound functions match "*".
Calls (number of times)
42072   strcmp
 2505   .urem
 2500   hashIndex
 2004   malloc
```

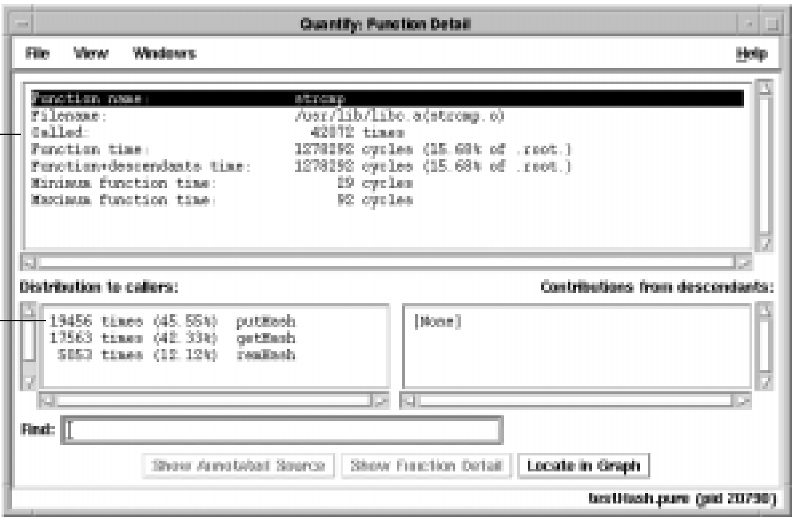
Why should `strcmp` be called so many times over such a small test dataset?

The function detail suggests long buckets

Double-click `strcmp` to open the Function Detail window.

`strcmp` never took more than 92 cycles to execute but it was called a large number of times

The calls were mostly from `putHash` and `getHash`



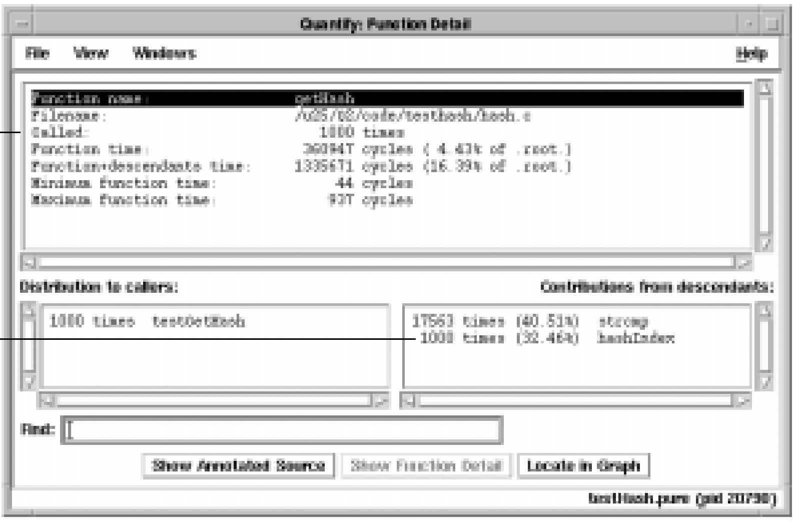
```
Function name:                 strcmp
Filename:                      /usr/lib/libc.a(strcmp.o)
Called:                          42072 times
Function time:                 1278292 cycles (15.68% of .root.)
Function+descendants time:     1278292 cycles (15.68% of .root.)
Minimum function time:              29 cycles
Maximum function time:              92 cycles
```

```
19456 times (45.55%)  putHash
17563 times (42.33%)  getHash
 5053 times (12.12%)  readHash
```

```
[None]
```

testHash.pure (pid 20730)

Double-click `getHash` to inspect its function detail.



```
Function name:                  getHash
Filename:                       /u25/02/code/testhash/hash.c
Called:                            1000 times
Function time:                   360947 cycles ( 4.43% of .root.)
Function+descendants time:      1335671 cycles (16.39% of .root.)
Minimum function time:               44 cycles
Maximum function time:              937 cycles
```

```
1000 times   testGetHash
```

```
17563 times (40.51%)   strcmp
 1000 times (32.46%)   hashIndex
```

`getHash` was called 1,000 times from the `testGetHash` driver function

`getHash` in turn called `hashIndex` 1,000 times, once per probe

The minimum and maximum time spent in the `getHash` code varies between 44 and 937 cycles. This wide variation is presumably because `getHash` had to traverse hashtable buckets of different sizes in its scanning loop.

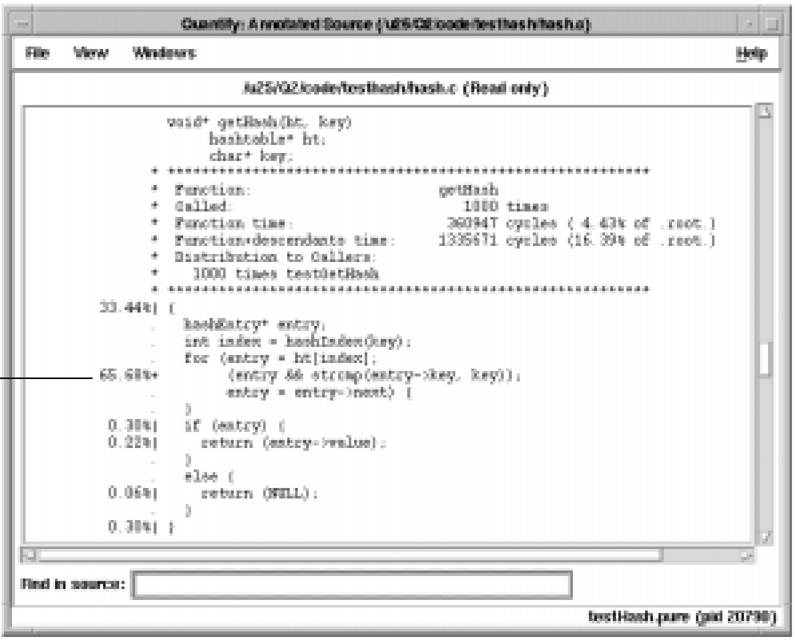
The `strcmp` function is called 10 to12 times for each call to `getHash`, making the scanning loop and the calls to `strcmp` the major contributors to `getHash`'s accumulated time.

To confirm this, you can look at the annotated source code for `getHash`.

Annotated source confirms excessive calls

Click **Show Annotated Source** in the Function Detail window to open the Annotated Source window.

The annotated source for `getHash` shows the *function+descendants* time distributed on each source line and scaled as a percentage of its overall *function+descendants* time.

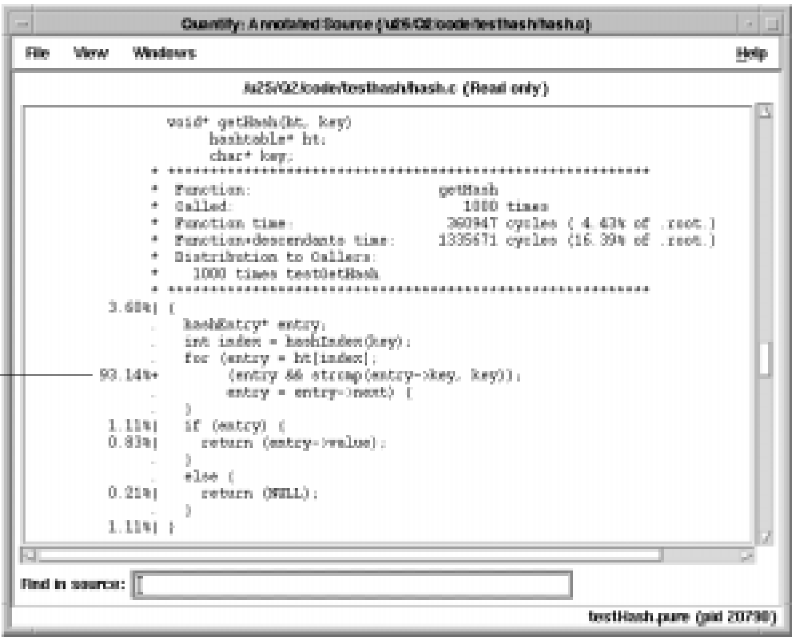

Quantify: Annotated Source (/u25/Q2/code/testhash/hash.c)
File
View
Windows
Help
/u25/Q2/code/testhash/hash.c (Read only)
Find in source:
testHash.pure (pid 20798)

```
                void* getHash(ht, key)
                     hashtable* ht;
                     char* key;
              * ************************************************************
              *  Function:                              getHash
              *  Called:                                   1000 times
              *  Function time:                          360947 cycles ( 4.43% of .root.)
              *  Function+descendants time:     1335671 cycles (16.39% of .root.)
              *  Distribution to Callers:
              *     1000 times testGetHash
              * ************************************************************
      33.44%| {
            .    hashEntry* entry;
            .    int index = hashIndex(key);
            .    for (entry = ht[index];
      65.68%*         (entry && strcmp(entry->key, key));
            .         entry = entry->next) {
            .    }
       0.38%|    if (entry) {
       0.22%|      return (entry->value);
            .    }
            .    else {
       0.06%|      return (NULL);
            .    }
       0.38%| }
```

The majority of time in `getHash` is spent in the hashtable scanning loop that calls `strcmp`

To find out how much of `getHash`'s time is spent in the loop that calls `strcmp` (exclusive of the time in the `strcmp` function itself), select **View > Annotations > Function time(% of function)**.

The Function time (% of function) view shows that over 90 percent of `strcmp`'s time was spent in the scanning loop.


Quantify: Annotated Source (/u25/Q2/code/testhash/hash.c)
File
View
Windows
Help
/u25/Q2/code/testhash/hash.c (Read only)
Find in source:
testHash.pure (pid 20798)

```
          void* getHash(ht, key)
              hashtable* ht;
              char* key;
        * ***********************************************************
        *  Function:                         getHash
        *  Called:                              1000 times
        *  Function time:                     360947 cycles ( 4.43% of .root.)
        *  Function+descendants time:        1335671 cycles (16.39% of .root.)
        *  Distribution to Callers:
        *    1000 times testGetHash
        * ***********************************************************
  3.68%| {
       .   hashEntry* entry;
       .   int index = hashIndex(key);
       .   for (entry = ht[index];
 93.14%+        (entry && strcmp(entry->key, key));
       .        entry = entry->next) {
       .   }
  1.11%|   if (entry) {
  0.83%|     return (entry->value);
       .   }
       .   else {
  0.21%|     return (NULL);
       .   }
  1.11%| }
```

Over 90 percent of `strcmp`'s time was spent in the scanning loop

Saving the baseline data

Now that you have identified the performance bottleneck, save the collected data so it can serve as a baseline against which you can measure performance changes. Select **File > Save collected data** and **File > Export data as** to save both the binary data (in case you want to rerun Quantify on this same dataset at a later time) and save the collected data in export format. Later in this chapter, you'll use the export file with the `qxdiff` script in order to verify the performance improvements you make to the program.

After saving the data, exit Quantify.

Improving the performance of the testHash program

The data Quantify reported indicates that much of the expense of the hashing operation is the number of `strcmp` comparisons that must be performed to find the requested entry. Avoiding these excessive `strcmp` comparisons would significantly improve the speed of the hash package.

There are several possible approaches you could take to improve the performance of the hashtable package. You could improve the hash key function itself in order to distribute items more uniformly in the hashtable, thereby shortening the hash buckets and thus the number of items that must be inspected to retrieve the requested item. Or, you could double the size of the hashtable array. This would distribute items into more buckets and make better use of the information in the computed hash key.

The approach you will use in this example is based on the idea that the full hash key is effectively a *compressed* version of the key string itself. The *modulo* operation, however, uses only a small fraction of the compressed information to form the index into the hashtable. The rest of the information encoded in the hash key is ignored. Since the `hashIndex` function computes the same value for identical strings, the `testHash` program could save the full hash key on the hash entry, compare the full hash keys, and then call the `strcmp` routine only if the keys were identical. This hash key comparison would be much quicker than calling `strcmp`.

The source code for the improved `testHash` program is in the file `<quantifyhome>/example_quantify/improved_hash.c`. The following changes have been made to the code in order to implement the hash key comparison.

```
/* hashEntry
 *   A hashEntry keeps the key, value pair together in a list hashEntries.
 */
typedef struct struct_hashEntry {
  int   hash_key;                /* The full hash key for this entry */
  char* key;                     /* The full string key for this entry */
  void* value;                   /* Pointer to user data indexed by key */
  struct struct_hashEntry* next;  /* Pointer to next bucket entry */
} hashEntry;

/* hashIndex
 *   Determines what bucket (index) to place a key in.
 *   Given a key it returns the index of the appropriate bucket.
 */
static int hashIndex(key, fullHashKeyp)
     char* key;
     int * fullHashKeyp;
{   char *p;
    unsigned h = 0, g;
    for (p = key; *p; p++) {
        h = (h << 4) + (*p);
        if ((g = h & 0xf0000000)) {
            h = h ^ (g >> 24);
            h = h ^ g;
        }
    }
    *fullHashKeyp = h;  /* Save full hash key in hash key parameter*/
    return h%hashtable_backbone_size;
}
/* getHash
 *   Hashes into the appropriate bucket, and then walks the
 *   chain of hash entries. It first compares the full hash
 *   keys of each entry before it compares the keys with strcmp.
 *   When a match is found the associated value is returned.
 *   NULL is returned if no match is found.
 */
void* getHash(ht, key)
     hashtable* ht;
     char* key;
{   hashEntry* entry;
     int fullHashKey;
    int index = hashIndex(key, &fullHashKey);
    for (entry = ht[index];
        (entry && ((entry->hash_key != fullHashKey) || strcmp(entry->key,key)));
         entry = entry->next) {
    }
    if (entry)
        return (entry->value);
    else
        return (NULL);
```

Running the improved_testHash program

Build and run the `improved_testHash` program.

```
% improved_testHash.pure 500 test_words
```

Compare this program summary for the `improved_testHash` program with the original program summary on page 2-4.

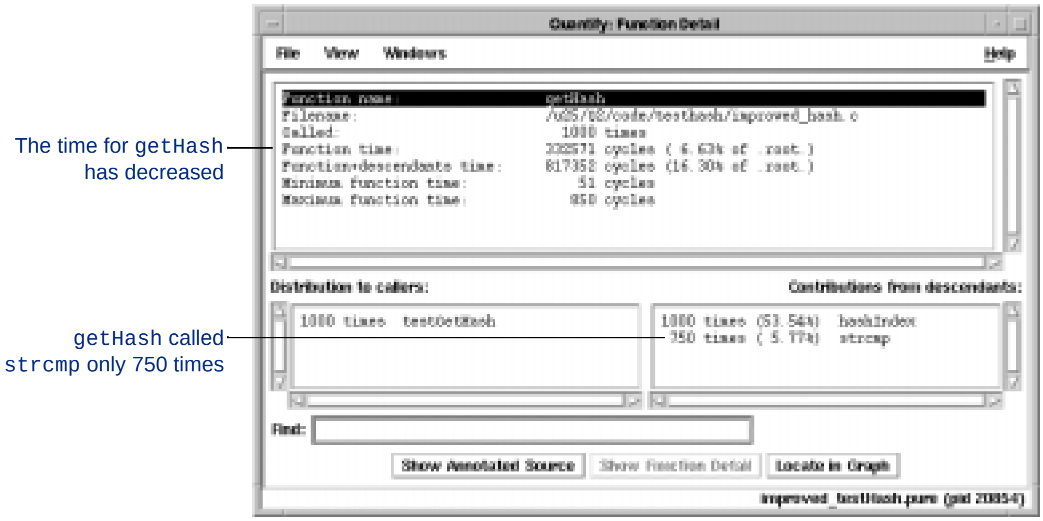
Time spent in the code has decreased from 5.0 to 3.8 million cycles

```
****  Quantify instrumented improved_testHash.pure (pid 20854)
Quantify 4.4 SunOS 4.1, Copyright 1993-1999 Rational Software Corp
  * For contact information type: "quantify -help"
  * Quantify licensed to Quantify Evaluation User.
  * Quantify instruction counting enabled.
Testing the first 500 entries from test_words with a hashtable of size 13.
All tests passed.

Quantify: Sending data for 61 of 187 functions
 from improved_testHash.pure (pid 20854)..........done.

Quantify: Resource Statistics for improved_testHash.pure (pid 20854)
 *                                            cycles       secs
 * Total counted time:                       5013898      0.100 (100.0%)
 *     Time in your code:                    3771298      0.075 ( 75.2%)
 *     Time in system calls:                 1242600      0.025 ( 24.8%)
 *
 * Note: Data collected assuming a sparcstation_lx with clock rate of 50 MHz.
 * Note: These times exclude Quantify overhead and possible memory effects.
 *
 * Elapsed data collection time:       0.906 secs
 *
 * Note: This measurement includes Quantify overhead.
 *
%
```

The counts for *Time in your code* has decreased by 1.26 million cycles—from 5.03 million cycles to 3.77 million cycles. It’s now 25 percent faster.

Display the Function Detail window for the `getHash` function.

You can see that the number of calls to `strcmp` from `getHash` has decreased dramatically, from 17,563 to 750.


The time for getHash has decreased
getHash called strcmp only 750 times
Quantify: Function Detail
File
View
Windows
Help
Distribution to callers:
Contributions from descendants:
Find:
Show Annotated Source
Show Function Detail
Locate in Graph
improved_testHash.pure (pid 20854)

```
Function name:                 getHash
Filename:                      /u25/02/code/testhash/improved_hash.c
Called:                          1000 times
Function time:                 332571 cycles ( 6.63% of .root.)
Function+descendants time:     817352 cycles (16.30% of .root.)
Minimum function time:             51 cycles
Maximum function time:            850 cycles
```

```
1000 times  testGetHash
```

```
1000 times (53.54%)  hashIndex
 750 times ( 5.77%)  strcmp
```

The time for `getHash` has decreased because, even though the routine is now comparing hash keys before calling `strcmp`, it is saving time by avoiding the cost of calling `strcmp`. Overall, the function+descendants time for `getHash` has decreased from the first to the second run.

Verifying the performance improvement

Save the export data from the improved_testHash run, then exit Quantify.

You can use Quantify's qxdiff script to compare the performance of the original testHash program with the performance of the improved_testHash run. The qxdiff script compares two export data files and reports any performance changes. Since you are interested only in the time spent in the code itself, you can use the -i option to ignore functions that make system calls.

```
% qxdiff -i testHash.pure.20790.0.qx improved_testHash.pure.20854.0.qx
```

The qxdiff report confirms a 25 percent improvement in the performance of the testHash program:

Over 40,000 calls to strcmp have been eliminated

```
Differences between:
program testHash.pure (pid 20790) and
program improved_testHash.pure (pid 20854)
                   Function name        Calls     Cycles % change
!                         strcmp       -40822   -1198640  93.77% faster
!                        putHash            0     -32912   6.61% faster
!                        getHash            0     -28376   7.86% faster
!                        remHash            0      -7856   5.91% faster
!                      hashIndex            0      10000   1.49% slower
5 differences; -1257784 cycles (-0.025 secs at 50 MHz)
25.01% faster overall (ignoring system calls).
```

The putHash, getHash, and remHash functions are faster because they now avoid unnecessary calls to strcmp. The hashIndex function is slightly slower because it is saving the full hash key into a global variable.

For more information about using the qxdiff script, see "Comparing program runs with qxdiff" on page 5-7.

Other causes of slow software

In the previous `testHash` example, you used Quantify to find one type of performance bottleneck: inefficient computation. Quantify can also help you find and resolve these other causes of slow software:

- Needless computation
- Premature computation
- Needless recomputation
- Inefficient computation
- Needless library or system-call requests
- Excessive library or system-call requests
- Expensive library or system-call requests
- Environmental factors

Needless computation

As applications evolve and algorithms are refined, or as data changes, portions of code that were needed in earlier versions can end up falling into disuse, without ever being removed. The end result is that many large programs perform computations whose results are never used. Bottlenecks are caused by time wasted on this *dead* code.

Other common useless computations are those made automatically or *by default*, even if they are not required. Applications that needlessly free data structures during a program’s shutdown, or open connections to workstations even though there isn’t a user for them, are examples of this type of bottleneck.

Quantify helps find the time that is spent in *dead* code. Once you’re convinced that the results of a computation are useless, you can remove the code.

Premature computation

Any computation that is performed *before* there is a need for its results can cause a bottleneck. For example, there may not be a reason to sort a list of numbers if the user hasn't requested that the sort be performed. Quantify can't tell you if the computation can be delayed; however, it can tell you the cost of the computation, and you can decide whether to postpone it.

Needless recomputation

Programs sometimes recompute needed values rather than caching them for later use. For example, determining the length of a constant string can result in needless computation if the computation is embedded in a loop; the length of the string is recomputed many times, each time getting the same value. Quantify can tell you where the recomputation is taking place, and you can decide to store the value after one computation.

Inefficient computation

A poor choice of algorithm or data structure layout can cause extra work for the program. The initial performance can appear acceptable, given small datasets, but then scale poorly when presented with larger or more complex datasets. This is what happened in the `testHash` program described earlier.

Quantify can tell you the cost of each computation at different scales so you can predict whether there will be a problem with still larger datasets. You can then use alternative algorithms and data structures that get the job done faster.

Needless library or system-call requests

Bottlenecks can be caused by the way your own code uses operating system or third-party library services. Making library or system-call requests when you don't need the results is the same as performing needless computations.

Quantify shows you the time spent in the operating system or third-party libraries. You can see how much a request actually costs and make an informed decision about eliminating the request or pooling similar requests for more efficient service.

Excessive library or system-call requests

It is common with operating-system requests to make more requests than necessary. Quantify helps you identify excessive requests so you can design an alternative implementation.

Expensive library or system-call requests

Some operating-system calls can vary in the amount of time they require. For example, opening and accessing files across a network can be slower when there is increased network traffic. On most UNIX file systems, opening or calling the `stat` function on a file using a fully qualified pathname requires the operating system to verify the existence of each intermediate directory. When `stat` is called using a relative pathname, the operating system starts checking from the current working directory, thereby reducing the cost of the system call. The elapsed time that Quantify reports for system calls helps you see when they slow down so you can explore less expensive implementations.

Environmental factors

External or environmental factors, such as high network delay or a high load average on the machine, can cause slow performance. Your program can also exhibit large swapping and paging effects, which Quantify cannot measure directly. These factors show up in Quantify's reports as increased system-call times.

3 How Quantify Collects Data

The time your program takes to run depends on how many instructions it executes, how many machine cycles each instruction requires, and the machine's clock rate, which is typically expressed in millions of cycles per second (MHz). Quantify analyzes your program's instructions and inserts code that counts, at run time, the actual number of machine cycles your program requires to execute. For operating system calls, Quantify times each call and converts the elapsed (wall-clock) time into the equivalent number of machine cycles. Together, these times reflect the time you can expect your original program to run.

This chapter describes how Quantify collects data. It includes:

- How Quantify records function time
- How Quantify names functions
- Analyzing basic blocks
- How Quantify times system calls
- How Quantify times register-window traps
- Understanding recursive functions
- Paging and memory cache effects

Understanding how Quantify collects data is helpful both for interpreting the reported data and for fine-tuning how the data is collected.

How Quantify records function time

Quantify starts a counter each time a function is called and counts the number of machine cycles that the function call requires, exclusive of any other function calls it makes. When the function exits and returns to its caller, Quantify records the counted time. Quantify also tracks the minimum and maximum function times recorded for each function. This is useful, for example, if the function performs initialization only on the first call, not on subsequent calls.

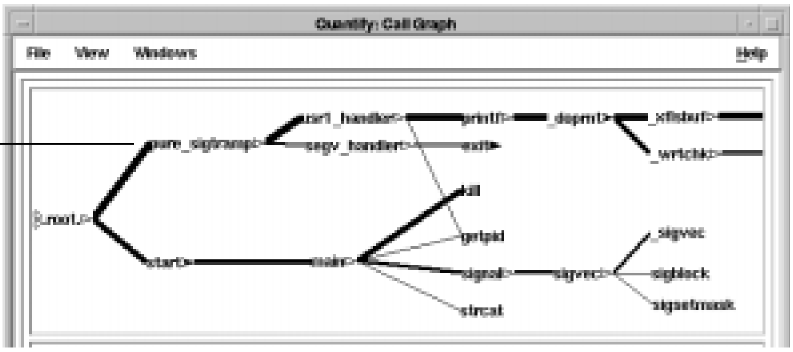
To track the callers of each function, Quantify inserts code at all function entry and exit points in your program. As the program runs, Quantify maintains a parallel stack of function calls that accumulates information about each function call. This stack is used to determine the *descendants* of each function.

When the function exits, Quantify distributes the accumulated *function+descendants* time to the function's immediate caller as part of the calling function's descendant counts. This data is shown in the Call Graph and Function Detail windows.

Quantify uses `.root.` as an accumulator for the total time consumed by all the functions in the program. Quantify treats `.root.` like a function and considers all the functions in the program to be descendents of `.root.`

Quantify reports any signal handlers as descendants of `.root` in the `pure_sigtramp` subtree.

Quantify reports signal handlers under `pure_sigtramp`



Quantify is careful not to double count a function’s time when the function calls itself directly, as in the case of a recursive function, or indirectly through other functions. If care were not taken in these cases, counts for the function would be recorded once in the *function* time accumulator and again in the function’s *descendants* count. In Quantify, *a function is never its own descendant*. A function’s descendants’ time report always reflects the time spent in all functions it called *exclusive* of calls to itself.

How Quantify names functions

Quantify names functions based on the function names in the symbol table of the object file. For C++ function names, Quantify *demangles* the name.

External function names are unique throughout the program. It is possible, however, for two or more object files of an application to contain identically named `static` functions. These names are not considered external to the object file, and there is no conflict during linking.

For ambiguous function names, Quantify appends the filename of the object file containing the function and a number indicating the function offset within the file. For example, if the static function `reset_callback` is the third function in the `graph.o` module, Quantify names it `reset_callback[graph.o/3]`.

In addition, the linker might have removed nonexternal symbols from certain object modules in order to save disk space. When Quantify finds the static function definition in the object module without a corresponding name, it names the function `unknown_static_function`. It then appends the object filename and function offset to distinguish the function from any other unknown static functions in the same or other object modules.

HPUX On HP-UX, the linker inserts stub functions in shared libraries to support distant branches within the shared library. Quantify names these functions `uwss_NNNN` (*unwind stub start*) and `uwse_NNNN` (*unwind stub end*).

Analyzing basic blocks

To determine the time spent in the function itself, Quantify analyzes the basic code blocks of each function. A basic block is a sequence of instructions that are always executed together in succession. Basic blocks typically start at the beginning of functions and other code blocks and terminate at conditional jumps to other basic blocks.

Quantify uses information about your machine's hardware to compute the expected number of machine cycles each original basic block will require to execute.

On RISC architecture machines, most instructions take a single machine cycle. Instructions such as load and store instructions can take longer and can stall, depending on the instruction stream that follows each instruction. Quantify uses this machine-specific information to estimate the number of instruction cycles each basic block will take, including the expected number of stall cycles.

Quantify inserts code that adds the expected cycle count to a basic block cycle accumulator each time the basic block is entered. These accumulators have 64-bit precision, providing accurate counts even in programs or blocks that execute for a very long time.

The counts Quantify reports reflect the time the *original* program would have taken without Quantify. The reported times are exclusive of any Quantify run-time overhead.

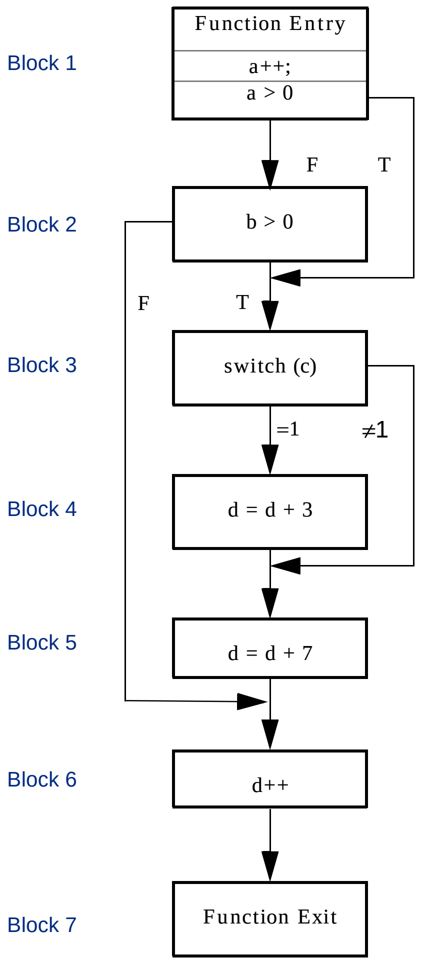
How Quantify identifies basic blocks

To understand how Quantify identifies basic blocks, consider this example program:

```
          1  int test(a, b, c, d)
          2      int a, b, c, d;
          3  {
          4       a++;
Block 1   5       if (a > 0 ||
Block 2   6           b > 0) {
          7           switch (c) {
Block 3   8               case 1:
          9                   d = d + 3;
Block 4  10               default:
         11                   d = d + 7;
Block 5  12                     }
         13       }
Block 6  14       d++;
         15       return d;
         16  }
         17
         18  int main()
Block 7  19  {
         20       test( 0, 1, 3, 8);
         21       test( 0, 1, 1, 6);     /* max */
         22       test(-1, 1, 1, 6);     /* min */
         23       return 0;
         24  }
```

The block numbers indicate the extent of the basic blocks in the function test when compiled using the -g debugging option.

Here is the basic block flow structure of the test function:



The first block contains a function entry that allocates some stack space. It then increments `a` and tests whether `a>0`. If the test succeeds, the block ends in a branch to the switch statement.

Otherwise, the code enters the second block which, tests whether `b>0` and branches to the sixth block if the test fails.

Assuming that either condition is true, the third block computes the switch case to branch to based on variable `c`.

If `c==1`, the fourth block is entered, the variable `d` is incremented by 3, and the code falls through to the default case block.

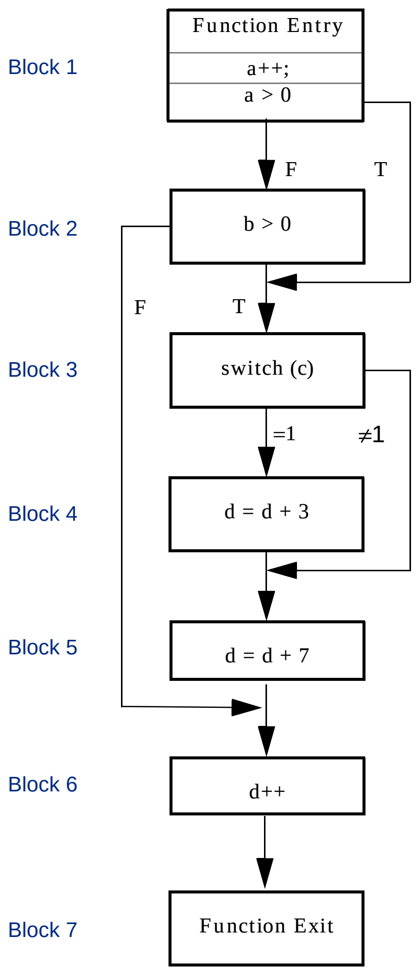
The fifth block starts in this case because the switch can branch to this case directly.

The sixth block increments the variable `d`. In nonoptimized code, this block then unconditionally branches to the exit block.

The final seventh block executes the function return sequence. It moves `d` into a return register, deallocates the stack space, and returns to the calling function.

SunOS4

Here is an example of the cycle counts for the program compiled using `cc -g` on a SPARCstation ELC:



26 cycles are required by the instructions in the first block to initialize the function, compute `a++`, test whether `a>0`, and branch to the switch statement if true.

5 cycles are required in the second block to test whether `b>0` and branch to the `d++` block if false.

3 cycles are required in the third block to compute and perform the switch branch.

6 cycles are required in the initial case in the switch.

Otherwise, **8 cycles** are required by the default case, 5 to perform the addition and another 3 to branch to the sixth block.

6 cycles, including the branch to exit, are required by the sixth block (d++).

8 cycles are required by the seventh block to perform the return function exit sequence.

To execute the first call to `test(0, 1, 3, 8)` in `main`, the program enters blocks 1, 3, 5, 6 and 7, which takes 51 cycles. The table below compares the difference in cycle counts between optimized and non-optimized code. The optimized version is noticeably faster, primarily because all the calculation occurs in registers, thereby avoiding the need to load and store values from memory.

Block	**Nonoptimized**	**Optimized**
1	26	4
3	3	2
5	8	1
6	6	1
7	8	3
Total cycles	51	11

Many optimizing compilers rearrange the execution order of machine instructions to take advantage of the RISC processor's ability to overlap operations. These instruction scheduling optimizations can have a significant impact on performance.

Since Quantify bases its analysis on the optimized instruction sequences produced by the compiler, Quantify's reports reflect any benefits of instruction scheduling performed by the compiler.

Note: Quantify's analysis does *not* reflect the additional performance improvements possible on superscalar architectures using multiple pipelines and other hardware features such as dynamic branch prediction. On such machines, Quantify's estimates are pessimistic, predicting a slower run time than what actually might be possible.

How Quantify reports multiple basic blocks

In the Annotated Source window, lines marked with a plus sign (+) indicate the start and possible continuation of multiple basic blocks over one source line. This occurs in expressions such as:

```
if ((a > 0) && (b > 0)) {c++;}
```

The two clauses of the conditional expression and the increment clause are compiled as three separate basic blocks, but all these blocks are associated with the same line number. When Quantify displays the data, the number in the margin reflects the sum of the data recorded for all the basic blocks associated with that line.

If you select **View > Multi-block lines > Show multi-block lines**, the individual times for basic blocks, as ordered in the object file, are shown on comment lines inserted immediately after the initial multiple basic block line. In some cases, the compiler might order the basic blocks differently from the order of the source code.

Annotations and compiler differences

Quantify reports counts for basic blocks in the Annotated Source window using the line number information emitted by the compiler for debugging purposes. Different compilers emit different line information in addition to different machine instruction sequences for a source file. Quantify's annotated source reports can reflect some of these internal differences.

For example, consider this code fragment:

```
if ((a > 0) && (b > 0)) {c++;}
```

When compiling without debugging information, most compilers emit three basic blocks for this code fragment, corresponding to the two test expressions and the variable increment statement. When compiling *with* debugging information, however, some compilers emit *four* basic blocks. The extra basic block corresponds to an "empty" else clause:

```
if ((a > 0) && (b > 0)) {c++;} else {}
```

At run time, if the conjunction succeeds, c will be incremented and the code will jump to the following statement. If either of the conditions fail, however, the “empty” code block will be executed, jumping to the next statement. This jump costs some machine cycles, and Quantify records those cycles in a separate basic block.

Annotations for if-then expressions

For if-then expressions written on several lines, the data from the “empty” basic block can produce annotations such as:

```
6 | if (a > 0) {
0 #     d++;  /* "then" clause was never executed */
2 +     c++;  /* executing the implicit "else" clause */
    }
```

The implied else clause can result in positive counts for the last line of the then cause.

For a description of the annotations in the Annotated Source window, see “What annotations mean” on page 1-21.

Annotations for switch expressions

Similar annotations can occur in switch statements, which are often rewritten by the compiler as if-then-else statements. Consider the following annotation fragment:

```
1     6  | switch (c) {
2        . case 2:
3     0  #     d++;
4     2  |     break;

5          case 3:
```

In this case, the compiler rewrote the switch statement as follows:

```
1        if (c == 2)
2        {
3            d++;
4            goto exit_switch; /* break */
  +      }   else { goto case3; }
5    case 3:
```

The counts on line 4 are not caused by the break expression but by the implicit else clause added by the compiler and associated with line 4. Compilers often do this because they assume that an optimizer eliminates the superfluous branches in a later pass if debugging information is not needed.

Showing multi-block lines, Quantify would display:

```
6 | switch (c) {
  .   case 2:
0 #       d++;
2 +       break;
  *       0 cycles
  *       2 cycles
      case 3:
```

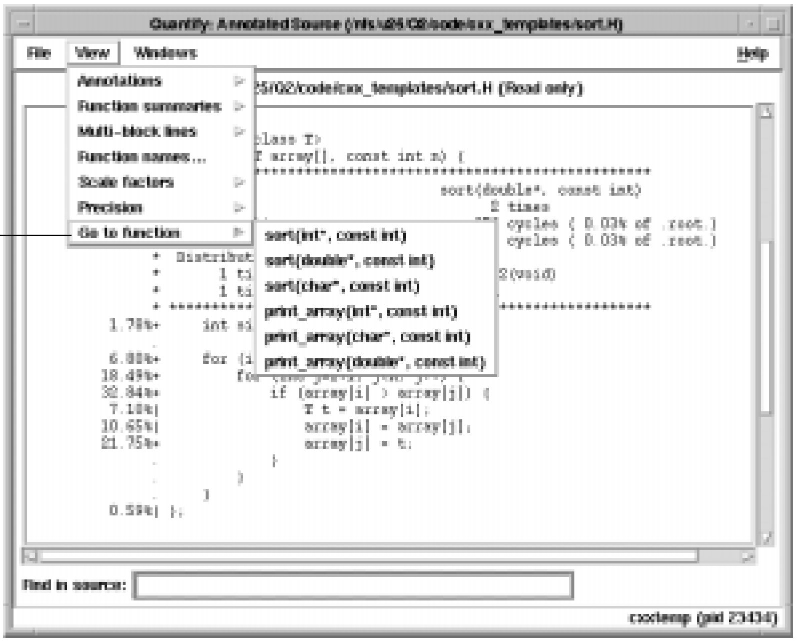
Quantify indicates that the implicit `goto exit_switch` statement, which corresponds to the original `break` statement, was never executed. However, the added implicit `else` basic block was executed. Since the sum of the multiple basic blocks under line 4 was not zero, Quantify reports the total and marks the line as being executed.

Most compilers emit many small basic blocks when compiling for debugging. The increase in the number of small basic blocks often results in a degradation in speed when Quantify is recording data in these functions, since it must record the time for each basic block separately. You can control this trade-off. See “Changing the granularity of collected data” on page 4-11.

Solaris Sun OS4 If you are recording data on register window traps, the counts for the first and last lines of the function can look quite large. Quantify assigns the register window trap times to the prevailing basic block at the time of the trap. This is typically the first or last basic block of the function.

C++ templates and annotated source

When using C++ templates, it is common to include the template declaration in a header file and define each type variant (specialization) in one or more source files. For debugging purposes, the compiler indicates that the source code for each specialization is actually found in the header file. This means that several different specializations share the same source code in the header file.



```
                 class T>
                 T array[], const int n) {
                 ******************************************************
                                           sort(double*, const int)
                                                2 times
                                                  cycles ( 0.03% of .root.)
                                                  cycles ( 0.03% of .root.)
          *  Distribut
          *       1 ti                       2(void)
          *       1 ti
          * **********                     ******************
  1.78%+       int si
        .
  6.88%+       for (i
 18.49%+           fo
 32.84%+              if (array[i] > array[j]) {
  7.18%|                  T t = array[i];
 10.65%|                  array[i] = array[j];
 21.75%+                  array[j] = t;
        .             }
        .         }
        .      }
  0.59%| };
```

Select
View > Go to function
to see the data for
other specializations

Quantify reports data for each called specialization separately, reporting each specialization as a separate function with a demangled name that indicates the specialization's data types. Quantify displays the annotated source for a function in the header file with the collected data for that function.

Note: The same display technique applies to `static` C functions defined in header files and multiple function definitions on a single line.

How Quantify times system calls

When your program requests an operating system service such as reading from or writing to a file, it executes a system call. Each system call switches the processor from user state to kernel state, permitting the operating system to process the request. When the request is complete, the operating system switches the processor back to user state and returns control to your program.

Quantify measures the time required to execute each system call. You can choose to measure this time in elapsed (wall-clock) or kernel time. See “Controlling how system calls are timed” on page 4-4. Quantify converts the measured time to machine cycles based on your processor’s clock speed, then assigns the cycles to the function containing the system call.

System call times help you to see how much time your program spends waiting for operating system requests and the variation in your program’s performance due to load fluctuations on your machine and the network. On an unloaded machine, system calls that do not involve access to other devices, such as disk drives and ethernet controllers, typically execute in nearly constant time. The time required to access remotely mounted file systems through NFS or to make requests to the X Windows server can vary depending upon the remote machines, the server process, and the network.

You can use Quantify options and API functions to customize timing for system calls. See “Timing system calls” on page 4-2.

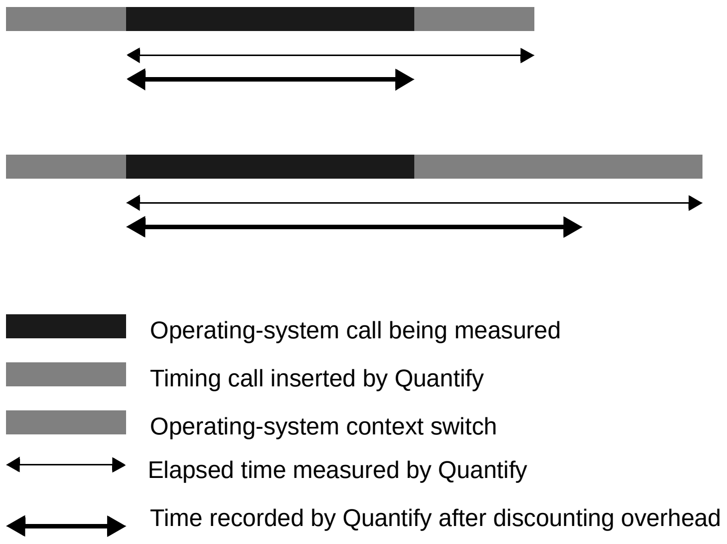
By default, Quantify does not time system calls such as `select` because they rely on unpredictable events such as a user clicking a menu item. The system calls that Quantify does not time by default are listed on page 4-5. Solaris On Solaris, Quantify also does not time a group of system calls that are associated with lightweight processes. These are listed on page 4-22.

You can have Quantify report the excluded system call times. See “Reporting excluded system-call time” on page 4-6.

Variations in system-call timing

Timing system calls can produce widely varying measurements of program performance, especially over short runs. The recorded information depends on the machine state, the contents of memory, the other processes running on the machine, the load on the network, and so on. This is to be expected and can be the very effect you want to demonstrate.

Note: Quantify uses the `gettimeofday` or `getrusage` system call to time system calls, discounting the small amount of time these calls take. Depending on the other processes that have made system requests, the kernel can switch context during Quantify's request for `gettimeofday`. If this occurs *after* the system call Quantify is timing, the measured time appears longer than it actually was.


Operating-system call being measured
Timing call inserted by Quantify
Operating-system context switch
Elapsed time measured by Quantify
Time recorded by Quantify after discounting overhead

Quantify does not adjust for this effect, since it cannot detect that it has occurred. In most programs, however, this happens infrequently.

How Quantify times register-window traps

Sun OS4 Solaris

The SPARC architecture provides a fixed set of register windows that are allocated for each function call, providing very fast access to function arguments and local variables. Allocating and releasing a new window typically takes a single instruction cycle at function entry and exit. When the set of register windows is exhausted, however, the hardware issues a trap to the operating system to save and later restore the contents of a window to memory.

Register-window trap processing by the operating system is relatively expensive, taking several hundred instruction cycles in a simple case. For programs with deeply nested function calls such as highly recursive programs, the accumulated cost of handling register-window traps can dominate the actual computation of the function itself.

By default, Quantify does *not* time register-window traps. As a consequence, the predicted times in the program summary report are optimistic, since they do *not* account for this hidden cost. Although this time is excluded, if the register-window trap times are substantial, Quantify reports this time in a separate category in the program summary. Quantify provides options and an API function that allow you to time register-window traps. See “Timing register-window traps” on page 4-10.

When you request that Quantify time register window traps, Quantify tracks each register window save-and-restore request at run time. By simulating the register-window mechanism, Quantify records the time spent by the operating system in preserving and restoring register windows required by your program. Quantify allocates the time for handling the overflow and underflow to the function that caused it.

Quantify provides this level of detail so that you can design ways to minimize the performance impact of register-window traps on your programs.

Understanding recursive functions

If a function does not call itself, the percentages listed in *Contributions from descendants* plus the function time itself, as a percentage of the function+descendants time, equals 100 percent. If the function calls itself recursively, the percentages displayed in the Function Detail window can *exceed* 100 percent. Although Quantify avoids double counting during arbitrarily complicated calling sequences, the Function Detail window can only display the contributions from all calling sequences in terms of the *immediate* descendants of those calling sequences. It cannot display the separate contributions from *each* unique calling sequence.

This display limitation is rarely a problem, since recursive calls occur infrequently. Quantify helps to identify these potentially confusing situations by reporting when a recursive call is made to a function by one of its descendants and marking these descendants with an asterisk (*). Quantify also displays a warning dialog when showing the first recursive function in the Function Detail window.

To understand why the combined percentages might exceed 100 percent, consider this function calling sequence:

$$\begin{tikzcd}
\texttt{main} \arrow[r] & \texttt{A1} \arrow[r] & \texttt{B} \arrow[r] & \texttt{C} \arrow[r] & \texttt{A2} \arrow[r] & \texttt{D} \\
 & 32 & 17 & 23 & 32 & 14
\end{tikzcd}$$

In this sequence, function `A` calls itself through a call to `B`. The calls are shown as `A1` and `A2`. The function time required by each function call is listed under each function name.

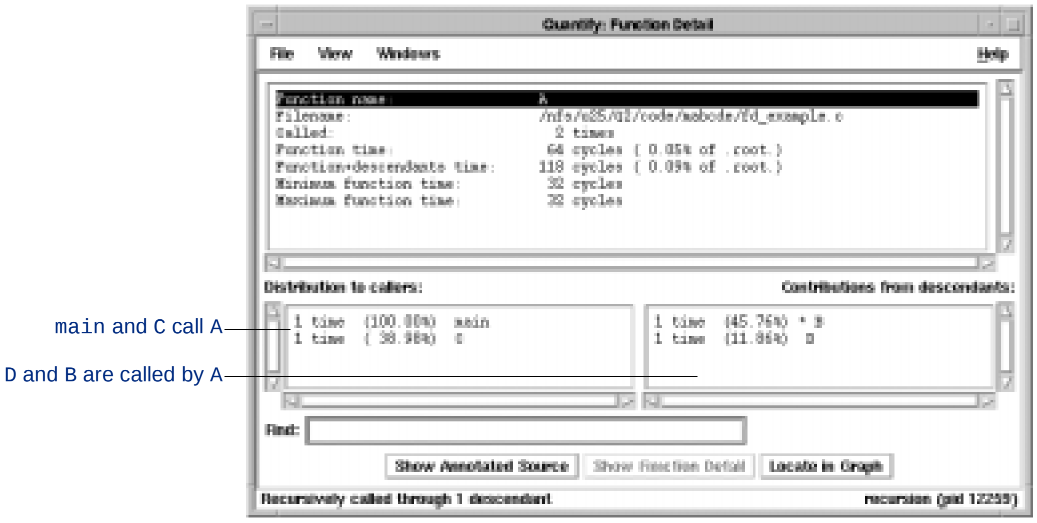
Function	**A**	**B**	**C**	**D**
Function time	64	17	23	14
Function+descendants time	118	86	69	14

Quantify adds the call times for both of these calls and reports this value as the function time of A: 32 + 32 = 64 cycles. All other functions were called once, so their function times are the same as the call times.

In Quantify, *no function is considered its own descendant*. When a function calls itself, Quantify records only the subsequent call's contribution to the function time and does not double-count that time when distributing the counts as the contributions to its callers.

The total function+descendants time for A is its function time (64 cycles) plus the call times of all other descendants (17 + 23 + 14 = 54 cycles), which in this case is 64 + 54 = 118 cycles.

The Function Detail window for A shows:



```
Function name:                A
Filename:                     /nfs/s25/q2/code/sabode/fd_example.c
Called:                         2 times
Function time:                 64 cycles ( 0.05% of .root.)
Function+descendants time:    118 cycles ( 0.09% of .root.)
Minimum function time:         32 cycles
Maximum function time:         32 cycles
```

Distribution to callers:

```
1 time  (100.00%)  main
1 time  ( 38.98%)  C
```

Contributions from descendants:

```
1 time  (45.76%) * B
1 time  (11.86%)  D
```

Recursively called through 1 descendant.   recursion (pid 12255)

Quantify does not list A as either a caller or descendant in this example. Quantify only shows the *immediate* callers and descendants of a function in the Function Detail window. If A had called itself directly, Quantify would have listed A as *both* a caller and a descendant. Notice the asterisk next to B indicating that a recursive call to A was made through the call to B.

The *Distribution to callers* shows 100 percent of the function+descendants time of A distributed to main. This makes sense since the time from everything to the right of main in the call graph comes from strict descendants of main.

The Function Detail window also shows a percentage of A's time distributed to C. This is because the second call to A distributed both D's time *and* its own time to C, since both A and D are strict descendants of C.

Turning next to the contributions from immediate descendants, computing the sum of the contribution percentages plus the function time as a percentage of the function+descendants time of A yields: 54.23% (64/118) + 45.76% (contribution from B) + 11.86% (contribution from D) = 111.85%. To understand why this happens, observe that B contributed time to A that includes B's own time plus C and D's time but, by Quantify's rule, does not include time from the second call to A.

In addition, A also received time directly from D, via the A2 call. In fact, the time contributed from D happens to be the same contribution from D that is included in the contribution from B.

Although it might appear that Quantify is double counting, in fact it is only double-*displaying* the contribution from D, not A. The first contribution is reported from the immediate call from A2 to D. The second contribution is reported when D's time is included with the time from B and C, but *not* from A2, as the contribution to A via the immediate call to B.

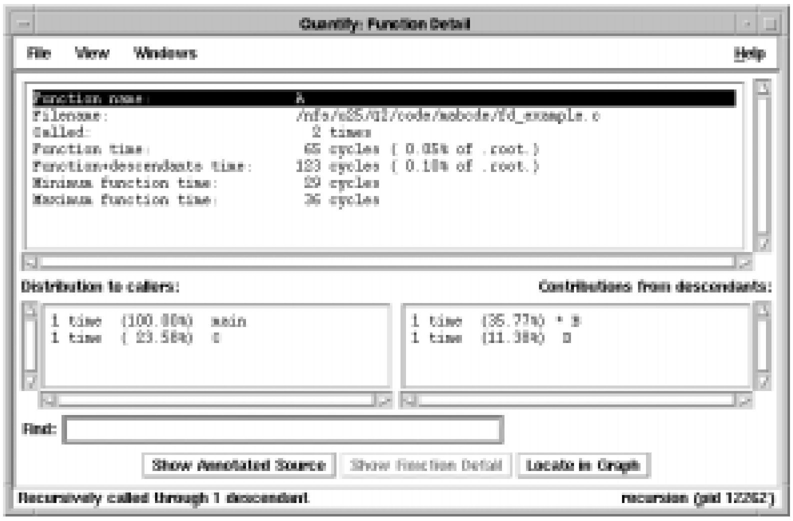
Consider the following call sequence:

$$\begin{tikzcd}
 & & & & & \texttt{D} \\
 & & & & & \texttt{14} \\
\texttt{main} \arrow[r] & \texttt{A1} \arrow[r] \arrow[uurrrr] & \texttt{B} \arrow[r] & \texttt{C} \arrow[r] & \texttt{A2} & \\
 & \texttt{36} & \texttt{17} & \texttt{27} & \texttt{29} &
\end{tikzcd}$$

In this sequence A still calls itself through a call to B. However, the first, not the second, call to A makes the call to D.

Function	A	B	C	D
Function time	65	17	27	14
Function+descendants time	123	73	56	14

The total function time for A in this case is 36 + 29 = 65 cycles and the function+descendants time for A is 65 + (17 + 27 + 14) = 123 cycles. The Function Detail window for this calling sequence is:



```
Function name:                 A
Filename:                      /nfs/u25/U2/code/mabode/fd_example.c
Called:                          2 times
Function time:                  65 cycles ( 0.05% of .root.)
Function+descendants time:     123 cycles ( 0.10% of .root.)
Minimum function time:          29 cycles
Maximum function time:          36 cycles
```

Distribution to callers:

```
1 time  (100.00%)  main
1 time  ( 23.58%)  C
```

Contributions from descendants:

```
1 time  (35.77%) * B
1 time  (11.38%)  D
```

Recursively called through 1 descendant. recursion (pid 12262)

The immediate calling relations are *identical* to the previous version. Function A still calls B and D and is called by C and main. B is still marked with an asterisk because A was called recursively through that call. In this case, however, the percentages add up to 100 percent: 52.85% (65/123) + 35.77% (contribution from B) + 11.38% (contribution from D) = 100.00%.

This is because D was not called by the recursive call to A. This critical difference in the calling sequence means that D's time is *not* displayed twice via A's call to B, as it was in the first call sequence.

Running on a different machine

Because CPUs vary in the amount of time they take to execute certain instructions, Quantify counts the number of cycles in each basic block differently for each machine.

Quantify determines the machine type and clock rate at instrumentation time. Quantify uses information from the operating system to match the machine-code values in the `.machine.<platform>` file. If a match is not found, Quantify assumes a default machine, currently `sparcstation_1` for SPARC architectures and `HP9000/715` for HPPA architectures.

Quantify records the machine type and clock rate in the Quantify'd program. It uses the stored clock rate at run time to determine the expected cycles. When you run the Quantify'd program on a different machine than the one used to instrument the program, Quantify detects the changes in the clock rate at run time and issues a warning. However, it continues to use the *original* clock rate.

You can instrument your program to collect data as if it were built and running on a different machine type. This enables you to see the changes in performance due to different clock rates. For details, see the discussion of the `-use-machine` option in page B-1.

Paging and memory cache effects

When your program uses a load instruction to request a value from memory, it makes an explicit request of the memory system. Quantify does *not* report times associated for memory system activity for the following reasons.

- If the data requested is in the processor's memory cache, the request typically takes only one or two cycles to complete. Depending on the data available in the cache, however, the memory system itself may make additional requests to retrieve or flush memory between cache memory and main memory and disk. To your program, these requests appear to make the load instructions stall, since they take longer than normal.

- The cost of these implicit operations varies with the memory system implementation. Retrieving data from main memory (DRAM) into the cache typically takes tens of cycles. Paging, that is, retrieving data from a file on disk or the swap space into the main memory, can take several thousands of cycles. If the device is remotely accessed through an NFS connection, this time depends on network delays as well as remote service contention. For a few programs, the time spent in paging and memory management dominates the computation.
- It is difficult to gather data about memory requests that trigger the additional paging requests because these additional requests occur transparently to your program. Critical memory references occur at different times, based not only on the history of the program's memory usage, but on other processes including the operating system from the time the system was booted.
- Quantify'd programs differ from the original program in that additional memory references are introduced with the new object code Quantify inserted. This skews its memory system behavior.

4 Customizing Data Collection

This chapter describes how to use Quantify options and API functions to customize data collection. It describes:

- Avoiding all data recording
- Timing system calls
- Timing shared library operations
- Timing register-window traps
- Changing the granularity of collected data
- Collecting partial data for a run
- Annotating datasets
- Saving data on signals
- Collecting data for child processes
- Collecting data in threaded programs

For more information about how Quantify collects data, see Chapter 3, “How Quantify Collects Data.”

For information about how to specify options and API functions, see Appendix A, “Using Quantify Options and API Functions.”

Note: You can change run-time options without recompiling or reinstrumenting your application. After embedding Quantify API functions in your code, you must recompile the program. You should also link with the Quantify stubs library. See “Linking with the Quantify stubs library” on page A-10.

Avoiding all data recording

You can use the -record-data option to avoid recording data during a run. This option is useful for avoiding timing data during start-up before your program gets to main. For example, you can set -record-data=no, then call the API function quantify_start_recording_data from your program at the point where you want to begin recording data.

Timing system calls

This section describes the various options that Quantify provides that let you control how system calls are timed.

Avoiding timing for all system calls

You can use the -record-system-calls option to control whether Quantify collects timing data for system calls. By default, -record-system-calls=yes so that Quantify records and distributes the elapsed (wall-clock) time of each system call. To avoid timing system calls, use -record-system-calls=no.

For example, if you run the testHash program with the default setting, Quantify times system calls. The program summary looks like this:

```
% setenv QUANTIFYOPTIONS -record-system-calls=yes
% testHash.pure 500 test_words
                  .
                  .
Quantify: Cumulative Resource Statistics for testHash.pure (pid 5150)
 *                                             cycles       secs
 * Total counted time:                        7756225      0.235 (100.0%)
 *     Time in your code:                     6159190      0.187 ( 79.4%)
 *     Time in system calls:                  1597035      0.048 ( 20.6%)
                  .
                  .                                        .
 * Note: This measurement includes Quantify overhead.
 *
```

If you run the testHash program with -record-system-calls=no, Quantify does not time system calls. The program summary looks like this:

```
% setenv QUANTIFYOPTIONS -record-system-calls=no
% testHash.pure 500 test_words
                  .
                  .
Quantify: Cumulative Resource Statistics for testHash.pure (pid 5153)
 *                                             cycles       secs
 * Total counted time:                         6159190      0.187 (100.0%)
 *     Time in your code:                      6159190      0.187 (100.0%)
 * Note: No system call times were recorded during this period.
                  .
                  .
 * Note: This measurement includes Quantify overhead.
 *
```

Time in your code remains the same, but Quantify notes that system call counts were not recorded

For more information about how Quantify records system calls, see “How Quantify times system calls” on page 3-13.

Note: When you use the qxdiff script to compare program runs, you can ignore the reporting of system call times even if Quantify has recorded system call data. This enables you to easily see just the performance improvements for compute-bound functions. For more information about the qxdiff script, see “Comparing program runs with qxdiff” on page 5-7.

Controlling how system calls are timed

The `-measure-timed-calls` option controls how system calls are timed. By default, `-measure-timed-calls=elapsed-time`. This measures the elapsed time of each call.

Specify `-measure-timed-calls=user+system` to measure the change in *user* and *system* time recorded by the kernel for the process. Unlike `elapsed-time`, `user+system` measurements do *not* reflect any time that your program and the kernel waited for other processes before the system call could be completed. The measurements Quantify reports match more closely those reported by `/bin/time` for the uninstrumented version of the program. It tells you the amount of work the kernel did to service your program *only*.

If you specify `user+system`, Quantify typically reports smaller counts for system calls. This is because the wait time is not included in these measurements. For example:

With `user+system`, system call counts are smaller

```
% setenv QUANTIFYOPTIONS -measure-timed-calls=user+system
% testHash.pure 500 test_words
                 .
                 .
Quantify: Cumulative Resource Statistics for testHash.pure (pid 5156)
 *                                            cycles       secs
 * Total counted time:                       6514930      0.197 (100.0%)
 *     Time in your code:                    6159190      0.187 ( 94.5%)
 *     Time in system calls:                  355740      0.011 (  5.5%)
 *
 * Note: Data collected assuming a sparcstation_elc with clock rate of
33 MHz.
                 .
```

You can also specify `user` or `system` individually to measure only the time the kernel spent servicing your requests.

Avoiding timing for specific system calls

You can use the -avoid-recording-system-calls option to avoid timing one or more specific operating system calls. Specify a list of system call names or numbers, separated by commas. You can find the names and numeric values in the /usr/include/sys/syscall.h header file.

The default values for -avoid-recording-system-calls are:

Sun OS4

```
SYS_exit, SYS_select, SYS_listen, SYS_sigpause
```

Solaris

```
SYS_exit, SYS_poll
```

HPUX

```
SYS_exit, SYS_select, SYS_listen, SYS_sigpause
```

In X Windows programs, Quantify routinely reports that large amounts of time were excluded from the dataset. This is because, by default, Quantify avoids recording time for system calls such as SYS_select and SYS_exit.

In the following example from a Quantify'd X Windows client, most of the unreported time is due to the X Windows server and client waiting at the select system call for the user to select an operation to perform.

```
*                                        cycles     secs
* Total counted time:                 254406924    7.709  (100.0%)
*    Time in your code:               113897874    3.451  ( 44.8%)
*    Time in system calls:            124626051    3.777  ( 49.0%)
*    Dynamic library loading:          15882999    0.481  (   6.2%)
*
* Time Quantify excluded from the dataset:
*    Time in system calls:             56195007    1.703
*          SYS_select
*      Time in register window traps:3175200     0.096
```

Quantify excluded the data for SYS_select

Note: Some UNIX client-server applications typically combine an X Windows graphical user interface and a database client interface, which communicate with the X Windows server and the database server respectively. If the database client routines also use SYS_select to wait for the database server to respond, Quantify does not report the time your program waited for the database server, since it avoids timing *all* calls to `select`.

To time the database calls, but avoid the X Windows calls, you can surround the database accesses with calls to the API functions `quantify_start_recording_system_call` and `quantify_stop_recording_system_call`. See “Collecting partial data for a run” on page 4-13.

Reporting excluded system-call time

If you avoid timing system calls completely by using `-record-system-calls=no`, or individually by using `-avoid-recording-system-calls`, Quantify does *not* include this time in the recorded data for the function and does *not* distribute the time to callers. Quantify, however, does continue to time the avoided system calls.

By default, if the excluded time exceeds 0.5 percent of the combined counted and excluded times, Quantify reports this value as *excluded* in the program summary along with the names of system calls contributing to the excluded time. This helps you to understand how much time is not accounted for by Quantify so you can include the time in the dataset if necessary.

You can use the `-report-excluded-time` option to control whether Quantify reports excluded time. Specify `-report-excluded-time=no` to eliminate the reports of excluded time. You can also specify a number between 0.0 and 100.0 as a percentage threshold, past which Quantify reports the excluded time in the program summary.

```
% setenv QUANTIFYOPTIONS -report-excluded-time=yes
% testHash.pure 500 test_words
                  .
                  .
Quantify: Cumulative Resource Statistics for testHash.pure (pid 5159)
 *                                             cycles       secs
 * Total counted time:                        7875025      0.239 (100.0%)
 *     Time in your code:                     6159190      0.187 ( 78.2%)
 *     Time in system calls:                  1715835      0.052 ( 21.8%)
 *
 * Time Quantify excluded from the dataset:
 *     Time in register window traps:            7020      0.000
```

Quantify reports the time excluded from the dataset

Timing shared-library operations

For programs linked with shared libraries, Quantify instruments the code in each shared library and records data from functions in those libraries. Quantify does *not* instrument the dynamic linker itself, the program responsible for loading the shared libraries into the running process and resolving references to entry points and data within those libraries. These dynamic-linking operations, including calls to `dlopen` under SunOS 4.1 and Solaris 2, and `shl_load` under HP-UX, can take a substantial amount of time.

Recording dynamic linking

Quantify measures the elapsed time of dynamic-linking operations.

```
% setenv QUANTIFYOPTIONS -record-dynamic-library-data=yes
% testHash.pure.dynamic 500 test_words
                      .
                      .
                      .
Quantify: Cumulative Resource Statistics for testHash.pure.dynamic (pid 5169)
 *                                             cycles       secs
 * Total counted time:                       17776759      0.539 (100.0%)
 *     Time in your code:                     6284146      0.190 ( 35.4%)
 *     Time in system calls:                  1255221      0.038 (  7.1%)
 *     Dynamic library loading:              10237392      0.310 ( 57.6%)
 *
 * Note: Data collected assuming a sparcstation_elc with clock rate of 33 MHz.
```

Elapsed time for dynamic linking operations

You can use the `-record-dynamic-library-data` option to control whether Quantify records time for dynamic linking. The default is `-record-dynamic-library-data=yes`.

To specify how Quantify measures time for dynamic linking, use the `-measure-timed-calls` option. The default is `-measure-timed-calls=elapsed-time`. This option also controls how Quantify measures times for system calls. If you change how Quantify measures time for system calls, you also change how it measures time for dynamic linking.

For more information about the `-measure-timed-calls` option, see the `-measure-timed-calls` option on page B-8.

Note: Most dynamic linkers perform an initial link of a shared library and then, as individual functions in that library are called, the final references are resolved once and a *procedure linkage table* (`PLT`) is updated by the dynamic linker so that subsequent calls to this function are processed directly. Quantify does *not* measure the time required to patch these `PLT` entries.

If the program attempts to open a dynamic library at run time that has not been instrumented, Quantify automatically instruments the library. Quantify does not measure the time required to instrument the library. Quantify records only the time required to open the instrumented library and link it into the process.

Understanding shared-library operations

Each operating system implements dynamic library initialization and support differently. As a consequence, when Quantify times dynamic library operations it often reports the times in different functions and files.

Sun OS4

On SunOS 4.1, the dynamic linker `ld.so` is loaded by a static function in the file `crt0.o` before `start` calls `main`. After `ld.so` is loaded into memory, `start` runs it and the linker loads the initial set of dynamic libraries into memory and initializes them. Quantify times both the initial load of `ld.so` and the initialization operations, and attributes the times to the `start[crt0.o/4]` function. Quantify records dynamic library loading at run time in the `dlopen` function.

Solaris

On Solaris 2, the dynamic linker `ld.so` is loaded by the standard program interpreter *before* the program is started, so Quantify cannot time this loading operation. Each dynamic library contains `init` and `fini` code that is run after the library is loaded. Quantify reports these times under the `init[<dynamic_library_name>]` and `fini[<dynamic_library_name>]` functions. Quantify records dynamic library loading at run time in the `dlopen` function.

HPUX

On HP-UX, the dynamic loader `dld.sl` is loaded by a function in the file `crt0.o` before `start` calls `main`. After `dld.sl` is loaded into memory, `start` calls it and it loads the initial set of dynamic libraries into memory and initializes them. Quantify times both the initial load of `dld.sl` and the initialization operations and attributes the times to the `_p___map_dld` function. Quantify records dynamic library loading at run time in the `shl_load` function.

Note: Unmapping dynamic libraries using `dlclose` on SunOS 4.1 and Solaris 2, and `shl_unload` on HP-UX, causes Quantify to attribute data to incorrect functions. Therefore, Quantify intercepts these functions and prevents the unmapping of dynamic libraries.

Timing register-window traps

Sun OS4 Solaris

Quantify simulates the time required to handle register window overflow and underflow conditions for SPARC processors. See "How Quantify times register-window traps" on page 3-15.

To control recording these simulated times, use the option `-record-register-window-traps`. For example:

Time reported for register-window traps

```
% setenv QUANTIFYOPTIONS -record-register-window-traps=yes
% testHash.pure.dynamic 500 test_words
                 .
                 .
                 .
Quantify: Cumulative Resource Statistics for testHash.pure.dynamic (pid
5172)
 *                                          cycles    secs
 * Total counted time:                    14418955    0.437 (100.0%)
 *     Time in your code:                  6284146    0.190 ( 43.6%)
 *     Time in system calls:               1633368    0.049 ( 11.3%)
 *     Time in register window traps:         6480    0.000 (  0.0%)
 *     Dynamic library loading:              64949610.197 ( 45.0%)
 *
 * Note: Data collected assuming a sparcstation_elc with clock rate of 33
MHz.
```

You can find the cost in machine cycles that Quantify uses for each register-window underflow and overflow trap in the `.machine.sunos4` or the `.machine.solaris2` file in `<quantifyhome>`.

By default, `-record-register-window-traps=no`. Quantify reports the excluded register-window trap time only if it is a significant fraction of the combined recorded and excluded time for the entire run.

On machines such as HPPA which do not support register windows, Quantify ignores any attempts to time these operations and issues a warning message.

Changing the granularity of collected data

Quantify can collect function time information at different levels of detail, or *granularity*. As the level of detail increases, so does the cost of collecting the data. By default, Quantify determines the level of detail based on the type of code your compiler emits.

Quantify can collect data at the following levels of granularity:

Collection granularity	Description
Function	Distinguishes counts for each function only
Basic-block	Distinguishes counts for each basic block
Line	Distinguishes counts for each line

You can use the `-collection-granularity` option to control the level of detail at which Quantify collects data. You can specify `function`, `basic-block`, or `line`.

Note: You can use `basic-block` and `line` only if debugging information is available; that is, if you compile your application using the `-g` debugging option.

- `-collection-granularity=function`

  Quantify tracks the total machine cycles over each function call but does not record any data about machine cycles for basic blocks. This level of data collection incurs the least overhead, requiring about 1 machine cycle for each basic block. This is the detail that is displayed in the Function Detail window.

  Note: When you specify `-collection-granularity=function`, the Annotated Source window is *not* available.

  If debugging information is *not* present, such as in third-party libraries and in optimized code, Quantify automatically instruments code at function granularity.

- `-collection-granularity=basic-block`

  This setting requires debugging information. Quantify tracks both the total machine cycles over each function call and the machine cycles executed in each basic block. Basic block data is used for the line annotations in the Annotated Source window.

- `-collection-granularity=line`

  This is the default if debugging information is available. `-collection-granularity=line` is identical to `basic-block` granularity, except that when a basic block extends over several lines of source code, Quantify inserts additional counters to record line-by-line counts. This improves readability of the data but at the expense of additional counting at run time.

Counting for basic blocks and lines is substantially slower than counting for function granularity, requiring approximately seven machine cycles per basic block or line counter. This is because Quantify must update both its function counter and each of the basic-block counters as it enters each basic block. Updating the basic-block counters requires updating a counter in memory.

The average cost of each basic block in normal code is 5 to 10 machine cycles. This means that, exclusive of function entry and exit overhead, Quantify's counting insertion in this case slows the function on average by a factor of two. This contrasts with function granularity which slows the same code by only about 20 percent.

To control the granularity of data collection without using the `-collection-granularity` option, you can:

- Recompile your application without using the `-g` option so Quantify automatically collects data at function granularity.
- Solaris HPUX Use `strip -l` to remove the line number information from certain files. Quantify then automatically collects data at function granularity.

To speed up data collection in an application compiled for debugging *without recompiling the code,* use the `-force-rebuild` option with the `-collection-granularity` option:

```
% quantify -force-rebuild \
     -collection-granularity=function cc ...
```

Collecting partial data for a run

You can use Quantify's API functions to fine-tune data collection. For example, to record only the cost of the last call to the `testPutHash` routine in the example `testHash` program, you can embed the `quantify_stop_recording_system_calls` and `quantify_stop_recording_data` functions in the program code:

Include "`quantify.h`"

Resets all counters

Stops recording system calls

Stops recording data

```
#include "quantify.h"
/* testHashTable
 *   Create a hashtable and do a bunch of operations on it checking that
 *   they are done correctly, and then delete the hashtable. Each operation
 *   tests whether the hashtable values should be present or not.
 */
static int testHashTable(testTable, table_size)
     testEntry *testTable;
     int table_size;
{
  int tests_succeeded = TRUE;
  /* Make the hashtable */
  hashtable* ht = testMakeHashTable();
  if (ht == NULL)
    return FALSE;
  /* Put them all in the hashtable and ensure they were not there before */
  tests_succeeded &= testPutHash(ht, testTable,  0, table_size, FALSE);
  /* Now put them in again and make sure they were already there */
  tests_succeeded &= testGetHash(ht, testTable,  0, table_size, TRUE);
  /* Lose the first half of them and make sure there was something deleted */
  tests_succeeded &= testRemHash(ht, testTable,  0,  table_size/2, TRUE);
  /* Try to remove them again and make sure there was nothing to remove */
  tests_succeeded &= testRemHash(ht, testTable,  0,  table_size/2, FALSE);
  /* Try to find the first set again and ensure no one is there */
  tests_succeeded &= testGetHash(ht, testTable,  0,  table_size/2, FALSE);
  /* But make sure the rest are still there */
  tests_succeeded &= testGetHash(ht, testTable, table_size/2, table_size, TRUE);
  /* Put the first half back in again and make sure they are all new */
  tests_succeeded &= testPutHash(ht, testTable,  0,  table_size/2, FALSE);

  if (quantify_is_running()) {
   quantify_clear_data();             /* drop any data so far */
   quantify_stop_recording_system_calls();/* don't care about calls to OS */
  }

  /* Try to put the last half in again and ensure they are already there */
  tests_succeeded &= testPutHash(ht, testTable, table_size/2, table_size, TRUE);

  if (quantify_is_running())
   quantify_stop_recording_data();  /* don't get any more data */

   tests_succeeded &= testDelHashTable(ht); /* Delete the table */

  return tests_succeeded;
}
```

Note: After embedding Quantify API functions in your code, you must recompile the program.

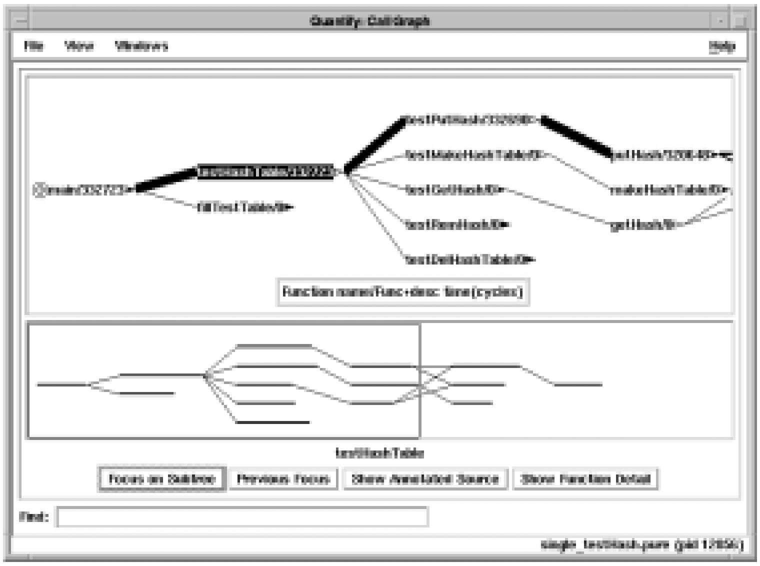
Analyzing datasets containing partial data

Look at the program summary to see the results of the `quantify_stop_recording_system_calls` and `quantify_stop_recording_data` calls. Quantify reports only the `Time in your code`, the code executed by the testPutHash call.

Not all system calls were recorded

```
Quantify: Cumulative Resource Statistics for single_testHash.pure (pid 12056)
 *                                               cycles       secs
 * Total counted time:                           332723      0.010 (100.0%)
 *     Time in your code:                        332723      0.010 (100.0%)
 * Note: Not all system call times were recorded during this period.
 *
 * Note: Data collected assuming a sparcstation_elc with clock rate of 33 MHz.
 * Note: These times exclude Quantify overhead and possible memory effects.
 *
 * Elapsed data collection time:       3.009 secs
 *
 * Note: This measurement includes Quantify overhead.
```

The call graph for this run includes the calling functions of `testPutHash`. This is because Quantify distributes time to the calling functions, even if they contributed no time themselves.



The functions that are called after the call to `testPutHash` are also recorded, even though these functions contribute no time.

Although Quantify does not record timing data after you call `quantify_stop_collecting_data`, it still tracks the function calls. This is so that if you decide to start recording data at a later point, Quantify can distribute that time correctly to the calling functions from that point on.

Calling quantify _stop_recording_system_calls and quantify_clear_data from your debugger

You can also run the `testHash` program under a debugger and place a breakpoint on the `testPutHash` line. When the program reaches the breakpoint, call the functions `quantify_stop_recording_system_calls` and `quantify_clear_data` from the debugger. Use `next` to step the program over the call to `testPutHash` and call `quantify_stop_collecting_data` or `quantify_save_data` and continue.

See “Calling API functions from your program” on page A-9.

API functions are useful when used with run-time options: The run-time options specify the *initial* recording state only; the API functions called during the execution of the program can change that state as desired.

You can call the API functions as often as you want. The overhead of using Quantify’s data collection API functions is negligible. A common Quantify practice in X Windows applications or other programs that use an event-driven architecture is to place a call to `quantify_start_recording_data` at the beginning of a callback function, and a call to `quantify_stop_recording_data` at the end of the callback function. Then run the Quantify’d application with `-record-data=no`, or place a call to `quantify_clear_data` before the program enters the main event loop. In the final dataset, Quantify reports the accumulated times recorded for exactly the callback(s) of interest.

Annotating datasets

You can add annotation strings to saved data files in order to distinguish between different data files and to record special data-collection circumstances, such as high network traffic or the use of special data files.

Use the API function `quantify_add_annotation` to add an annotation string to the next binary data file saved using `quantify_save_data`, `quantify_save_data_to_file`, or program exit. You can add as many annotations to a dataset as you want. You can also add annotations to an existing binary data file using the qv option `-add-annotation`. For example:

```
% qv -add-annotation="Run to collect dynamic library data" \
     testHash.pure.dynamic.5172.0.qv
```

Quantify automatically adds an annotation indicating the type of machine on which the data was collected. If a fatal signal is received without a user-defined signal handler, Quantify automatically saves the data to that point in the run and adds an annotation indicating the type of signal received.

To view the annotations in a data file, use the qv option `-print-annotations`. For example:

```
% qv -print-annotations testHash.pure.dynamic.5172.0.qv

****  Quantify instrumented testHash.pure.dynamic (pid 5172 at Wed Jan
13 12:54:02 1999)
Quantify 4.4 SunOS 4.1, Copyright 1993-1999 Rational Software Corp.
  * For contact information type: "quantify -help"
  * Quantify licensed to Rational Software
  * Quantify analysis enabled.
Quantify: testHash.pure.dynamic (pid 5172) run on host kenya.

Quantify: Reading data for 65 functions
 from testHash.pure.dynamic (pid 5172).........done.
Run to collect dynamic library data
%
```

The added annotation — **Run to collect dynamic library data**

Saving data on signals

Quantify installs signal handlers for many of the software signals that can be delivered to a Quantify'd process. The signal handler saves performance data to a file before proceeding with the normal signal behavior (which typically involves termination of the program). If you have already installed a signal handler for a given signal, Quantify does not save data when it receives that signal.

The initial default set of signals handled by Quantify are:

Sun OS4

```
SIGHUP, SIGINT, SIGIOT, SIGQUIT, SIGILL, SIGABRT, SIGEMT,
SIGFPE, SIGBUS, SIGSEGV, SIGSYS, SIGPIPE, SIGTERM, SIGUSR1,
SIGURSR2, SIGLOST, SIGXCPU, SIGXFSZ,
```

Solaris

```
SIGHUP, SIGINT, SIGIOT, SIGQUIT, SIGILL, SIGABRT, SIGEMT,
SIGFPE, SIGBUS, SIGSEGV, SIGSYS, SIGPIPE, SIGTERM, SIGUSR1,
SIGUSR2, SIGPOLL, SIGRTMIN, SIGRTMAX, SIGXFSZ, SIGXCPU, SIGXFSZ,
```

HPUX

```
SIGHUP, SIGINT, SIGIOT, SIGQUIT, SIGILL, SIGABRT, SIGEMT,
SIGFPE, SIGBUS, SIGSEGV, SIGSYS, SIGPIPE, SIGTERM, SIGUSR1,
SIGUSR2, SIGLOST, SIGRESERVE, SIGDIL, SIGXCPU, SIGXFSZ
```

To ignore signals in this list, use the `-ignore-signals` option. Specify a comma-separated list of signals to ignore. For example:

```
% setenv QUANTIFYOPTIONS -ignore-signals=SIGSEGV,SIGBUS
```

To handle additional signals, use the `-handle-signals` option. Specify a comma-separated list of additional signals. For example:

```
% setenv QUANTIFYOPTIONS -handle-signals=SIGALARM,SIGCHLD
```

Note: Quantify does not handle `SIGKILL`, `SIGTSTOP`, and `SIGTRAP`, since doing so interferes with normal program operation. If you specify these signals with `-handle-signals`, Quantify silently ignores them.

If you do not want Quantify to save data on any signals, specify:

```
% setenv QUANTIFYOPTIONS -save-data-on-signals=no
```

Collecting data in long-running programs

If you have a program, such as a daemon, that does not exit, you can capture Quantify data and save it to a file while the program is running, without terminating the program. You can use either of these methods:

- Write a signal handler for a given signal that calls the Quantify API function `quantify_save_data`. To clear the Quantify counters, call `quantify_clear_data`.
- Use the `-api-handler-signals` option to install signal handlers. For example, `-api-handler-signals=SIGUSR1,SIGUSR2` installs signal handlers for the `SIGUSR1` and `SIGUSR2` signals. When the program receives the first signal (`SIGUSR1`), `quantify_save_data` is called. When the program receives the second signal (`SIGUSR2`), `quantify_clear_data` is called.

  To send the signal to the process, type:

  ```
  % kill -USR1 <pid>
  ```

  This causes Quantify to write the data to the next dataset number. You can view that data with `qv` immediately after the file is written, while the daemon continues to run. Quantify automatically resets its counters to zero after saving the data, ensuring that the next incremental dataset will contain the data since the last save.

For additional information on signals, see the `signal`, `sigvec`, and `sigmask` manual pages, and the `/usr/include/signal.h` and `/usr/include/sys/signal.h` files.

Collecting data for child processes

Programs often create child processes using `fork` or `vfork`. When these child processes start, they execute from the current data state of the parent, typically to perform some computation based on data the parent provides. For example, the parent process might fork a child process to handle a request from the client to update a data record. Child processes, rather than the parent processes, are often the actual site of the performance bottlenecks in a program. To capture the performance data from these separate processes, Quantify must start a different instance of itself for each child process before the child process executes.

Using execve

Sometimes child processes immediately call execve to start running a different program altogether. In this case, the effort of starting a new copy of Quantify for the child process is wasted. Therefore, by default, Quantify *does not* start Quantify to record child process data. To record child process data, set the `-record-child-process-data` option:

```
% setenv QUANTIFYOPTIONS -record-child-process-data=yes
```

By default, Quantify saves data for each child process in a different file, since the filename prefix includes the `%p` character, which expands to the process ID of each child.

Using fork

Since the operating system makes a copy of the memory state during a `fork`, the counts for the child process include the counts from the parent process up to the point of the fork. To exclude the parent data from the child process data, you can call the `quantify_clear_data` API function after the child process begins.

Using vfork

The `vfork` system call creates a child process that shares the same memory image as its parent process. The parent process is suspended until the child exits. Since Quantify keeps its counter data in memory, as the child process executes it adds the counts to the parent's counters, thereby corrupting the parent's data. To prevent this, Quantify intercepts all `vfork` calls and converts them to `fork` calls. The `fork` system call ensures that the child is an independent process with its own memory state. Since the child is Quantify'd, its data is reported under a separate process ID, if `-record-child-process-data=yes`.

Collecting data in threaded programs

Threads are separate, independent lines of control executing within a single process. Threads share the same address space but maintain separate execution stacks. Quantify collects performance data as each thread runs and, by default, reports the composite performance data over calls to all functions from all threads.

Quantify works with most popular threads packages. For a list of supported threads packages, see the `README` file.

Threads and stacks

Quantify maintains separate accumulators for each stack and combines them to form the composite data. To request that Quantify save the per-stack data in separate qv data files, use the option:

```
% setenv QUANTIFYOPTIONS -save-thread-data=stack,composite
```

This saves both the composite and the per-stack data. If you specify `stack`, each dataset is written to a separate file. Quantify names the file by appending the value of the `%T` character to the value of the `-filename-prefix` option, followed by the .qv extension. For example, if the `-filename-prefix` option is `%v.%p.%n`, data for each stack is saved to a file named according to the expansion of `%v.%p.%n.%T.qv`. See “Using conversion characters in filenames” on page A-3 and “How Quantify creates filenames” on page A-4.

The data collection options and API functions described in this chapter affect data collection and saving for *all* threads, not just the thread that is currently executing. Thread-specific versions of the API functions are available. See the `<quantifyhome>/quantify_threads.h` file.

Typically, many threads reuse a single stack. This can happen if the thread is destroyed and the threads package recycles the stack for a later thread. Since Quantify detects stack creation and stack switches, not thread creation and destruction, the statistics it gathers reflect *all* of the threads that used a particular stack.

Quantify detects the use of a new stack by monitoring the stack pointer and comparing it against its table of known stack areas. If the stack pointer is not close to the stack of a known thread, Quantify assumes that a new stack has been created.

Use the `-thread-stack-change` option to specify how large, in bytes, a change to the stack pointer must be before Quantify recognizes a new stack. You might need to increase this value for programs that allocate large data structures on the stack using `alloca`, and decrease this value for programs that create threads with stacks very close to one another.

Quantify accesses its own internal data structures in a thread-safe way. Quantify uses mutual exclusion (`mutex`) locks to accomplish this. The exact implementation of `mutex` locks depends on the thread library being used. The `mutex` locks used by Quantify work properly with the scheduling code in your threads library.

By default, Quantify assumes that your application creates no more than 20 stacks during a run. To increase this number, use the `-max-threads` option.

Solaris lightweight processes and threads

Solaris

On Solaris 2, threads can be assigned to different lightweight processes (LWPs), which, in turn, can be assigned by the operating system to different processors if they are present.

Quantify, however, cannot easily determine when an LWP has been assigned to a different processor (since it can happen at any time) and hence cannot determine whether two LWPs (and therefore threads) might be running at the same time. Quantify therefore cannot account for true concurrency in such an application, and reports the sum of all counted times as though there is only a single processor. In this regard, Quantify’s times are pessimistic for applications running on symmetric multi-processor (SMP) machines.

Since Quantify cannot determine if several LWPs are running simultaneously, it does not time system calls that cause LWP scheduling to occur. These system calls look as though the calling thread blocked for a long time, but in fact the process was doing useful work in another thread in the same process, which Quantify is counting. If these system calls were timed, the time spent in other threads would be double-counted in the elapsed time recorded for the system call.

Quantify assumes that there are several CPUs and by default does not time system calls associated with lightweight processes. By default, `-never-record-system-calls=`:

```
SYS_sigtimedwait,SYS_lwp_sema_wait,SYS_lwp_create,
SYS_lwp_kill,SYS_lwp_mutex_unlock,SYS_lwp_mutex_lock,
SYS_lwp_cond_wait,SYS_signotifywait
```

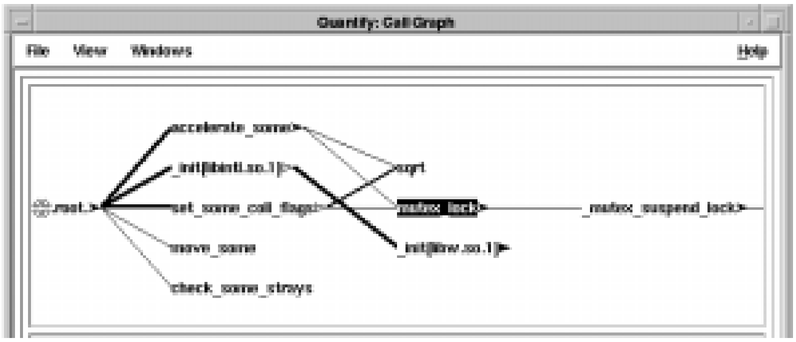
If you are running only one processor on the system, you can increase the accuracy of Quantify's times by having Quantify time the LWP system calls. You can remove specific system calls from the list specified to `-never-record-system-calls`, or specify a null string to have Quantify time all of the system calls listed above.

This list might change for different release levels of the Solaris 2 operating system. Consult the installation notes and `README` for the updated list.

Analyzing data from threaded programs

The call graph looks different for multi-threaded applications.

With multi-threaded applications, there are additional functions emanating from `.root.` corresponding to the unique starting points for the different threads started during the dataset. Some thread packages place their own routines on those stacks, which then call your functions. Other packages arrange for your functions to be called directly. You are also likely to see functions emanating from `.root.` reflecting the activity on any scheduler and signal handling threads the thread library sets up.



Solaris

Since threaded programs must lock accesses to shared data, you see support functions calling various different locking primitives supplied by the threads package. On Solaris 2, even under a single-threaded application, the linked libraries are *MT-safe* (multiple-thread safe). This means they call locking primitives as well, even though they are stubbed out in a single-thread application. These locking calls increase the complexity in the call graph. You can use the call graph pop-up menu to collapse the subtrees under these locking primitives.

5 Analyzing Data with Scripts

This chapter discusses how to use scripts to automate the analysis of Quantify's performance data.

Exporting performance data

You can save Quantify data in an export format that is suitable for processing by shell scripts and other programs.

There are several ways to create an export data file:

- You can use the `-write-export-file` option at run time.
- You can select **File > Save export data** in any Quantify data analysis window.
- If you have saved the binary data in a file, you can use the `-write-export-file` option on the `qv` command line to write the export file. For example, the command:

  ```
  % qv -write-export-file a.out.83475.0.qv
  ```

  writes an export data file named `a.out.83475.0.qx` in the current directory.

Quantify creates export filenames by expanding any conversion characters in the `-filename-prefix` option, then adding the `.qx` extension. See "Using conversion characters in filenames" on page A-3 and "How Quantify creates filenames" on page A-4.

Note: The export file does not provide data for basic blocks or lines. All C++ function names are *demangled*.

The export data file format

Each line of an export data file contains a keyword followed by tab-separated data fields appropriate for that type of line. The keyword *keyword* describes the format of each keyword. All data for a keyword appears on a single line. The primary keywords are:

- *Comment*: Each line contains a comment.
- *Keyword*: Each line describes a keyword and its data fields.
- *Program*: This line contains the name of the program, the process ID, the type of machine the data was collected on, the clock rate for that machine, and the version of Quantify.
- *Function*: Each line contains one function's detailed data.
- *Caller*: Each line contains one caller of a function and the distribution of time from the function to the caller.

Running qv

You can view a saved binary dataset at any time by running the qv program.

```
% qv a.out.513.0.qv
```

Alternatively, you can use the command:

```
% quantify -view a.out.513.0.qv
```

Note: You do not need to specify or retain the original executable to view your data. Each dataset file contains all function name and source file information required to display the performance data. In addition, dataset files are highly compressible. This is especially important for large datasets that need to be retained for comparison purposes.

The qv program supports a number of command-line options. See "qv options" on page B-7.

If you specify `-write-export-file`, `-write-summary-file`, `-add-annotation`, or `-print-annotations`, qv does *not* start the X Windows display. To start the X Windows display, include `-windows=yes` on the command line. For example:

```
% qv -write-summary-file -windows=yes a.out.513.0.qv
```

To connect to an X Window display, set the environment variable `DISPLAY` to the display name where windows should be created. The named display must have proper access permissions.

`qv` also supports standard X Windows options such as `-display`, `-background`, and `-language`. You can specify values for these options by using either the standard X Windows format in which option name and value are specified as separate tokens, or by using the Quantify options format:

```
% qv -windows=yes -display=ginza:0 \
     -language japanese a.out.513.0.qv
```

Rerouting Quantify's output

You can save Quantify reports and messages to a log file. This allows you to run a Quantify'd program and then inspect the output from Quantify at your convenience.

You can use the `-logfile` option to reroute Quantify output, separating it from the program's normal output:

```
% setenv QUANTIFYOPTIONS '-logfile=/tmp/pureout \
-write-summary-file=none'


% a.out

Hello, World

%
```

Notice that there is no Quantify banner. Quantify writes the banner to the log file. The option `-write-summary-file=none` prevents the printing of the program summary report, which will *not* be added to the log file. You can print the program summary report later, using the command:

```
% qv -write-summary-file=- a.out.<pid>.0.qv
```

To view the program output and Quantify output at the same time, open a separate terminal window and view the log file in this window while your program is running. Use the command:

```
% tail -f /tmp/pureout
```

You can add informative messages to the log file by calling one of several Quantify API functions from your program.

- The function `pure_printf` adds a message to all forms of Quantify output.
- The function `pure_logfile_printf` is a variation, adding a message if the output goes to a log file.

These functions take a format string and a variable number of arguments just like `printf`.

Note: The full `%` escape syntax of `printf` is *not* supported. You must limit escapes to simple `%d`, `%u`, `%n`, `%s`, `%c`, `%e`, `%f`, or `%g`. No field width or precision specifiers are allowed, and `%e, %f, %g` are equivalent to `%10.2f`.

To interleave your program's output with Quantify messages, use the option `-copy-fd-output-to-logfile`. Specify a list of file-descriptor numbers. For example, specifying `1,2` causes Quantify to *copy all* output written on file descriptors `1` and `2`, interleaving it with Quantify output.

Saving data incrementally

You can use the API functions `quantify_save_data` and `quantify_save_data_to_file` to save the collected performance data on demand. After saving the data, these functions automatically call `quantify_clear_data` to clear the accumulators so that saved files do not contain overlapping data.

Use the function `quantify_save_data_to_file` to save data in binary form to the file you specify. You can use conversion characters in filenames. See "Using conversion characters in filenames" on page A-3.

Use the function quantify_save_data to save data in binary form to a file whose name is based on the value of the -filename-prefix option and the .qv extension. The default value for -filename-prefix is %v.%p.%n. For example, for a program named new with a process ID of 513, Quantify would name the file new.513.0.qv. See “How Quantify creates filenames” on page A-4.

By default, Quantify saves each dataset and prints an incremental summary of the timing data recorded for each dataset. This behavior is controlled by the default behavior of the -write-summary-file option.

If -windows=yes, Quantify runs with the graphical interface and displays the *last* dataset it receives. In addition to the incremental summary on this last dataset, Quantify also prints a summary of all the timing data recorded over all the datasets it received. If no data was recorded in a dataset, it is not transmitted or saved.

Automating data analysis

You can use the -run-at-save and -run-at-exit options to automate repetitive performance analysis and reporting tasks. Both options take a quoted string containing sh(1) shell commands. Quantify passes the string to sh whenever data is saved with the API functions quantify_save_data or quantify_save_data_to_file and when the program exits, respectively. The shell command has access to all the environment variables available at the start of the Quantify’d program.

You can use conversion characters in the string. Quantify expands the conversion characters before the string is passed to the shell. See “Using conversion characters in filenames” on page A-3.

Automating performance regression tests

If you have an automated overnight build process, you can use the -run-at-exit option to detect any regression in your program’s performance. Add a step to the build procedure to Quantify the program and then run it on a standard test data set. The qxchange script uses `qxdiff` to compare a saved export data file with a given export file for an executable. It then copies the given export file to become the saved export data file for subsequent comparisons.

```
#!/bin/sh
# Runs qxdiff to compare a recent run with a previously saved export file
# Then replaces the saved export data with the recent data.

# Usage: qxchange <executable_name> <export_file> [qxdiff options]
# Typically qxchange %v %x -i

executable="$1"
shift
current_export_file="$1"
shift
previous_export_file="${executable}.last_qx"

if [ -f "$previous_export_file" ]; then
   qxdiff "$@" $previous_export_file $current_export_file
else
   echo "No previous data.  Comparison will be made on subsequent runs."
fi
cp $current_export_file $previous_export_file
```

To run this script during the nightly build, set the environment variable `QUANTIFYOPTIONS` to:

```
'-windows=no -run-at-exit="qxchange %v %x -i"'
```

The `-windows=no` option specifies that Quantify should be run without the graphical interface during the automated run. The `-run-at-exit` option specifies that the `qxchange` script should run whenever the program saves a dataset and that comparisons should ignore system call times.

By default, the output from `qxchange` is written to `stdout`. To email the comparison to yourself, you can use:

```
'-windows=no -run-at-exit="qxchange %v %x -i | mail $USER"'
```

Inspecting incremental datasets during a run

If your program calls the API function `quantify_save_data` on a regular basis in order to capture incremental performance data, you can use the `-run-at-save` option to view each dataset as it is saved. To view each dataset as it is saved, use:

```
%  setenv QUANTIFYOPTIONS -run-at-save="qv %b &"
```

Comparing program runs with qxdiff

The `qxdiff` script compares two export data files from runs of a Quantify'd program and reports any changes in performance. To use the `qxdiff` script:

1 Save baseline performance data to an export file. Select **File > Export Data As** in any data analysis window.

2 Change the program and run Quantify on it again.

3 Select **File > Export Data As** to export the performance data for the new run.

4 Use the `qxdiff` script to compare the two export data files. For example:

```
% qxdiff -i testHash.pure.20790.0.qx improved_testHash.pure.20854.0.qx
```

You can use the `-i` option to ignore functions that make calls to system calls.

Below is the output from this example.

`qxdiff` lists the functions that have changed

And summarizes the differences for the entire run

```
Differences between:
program testHash.pure (pid 20790) and
program improved_testHash.pure (pid 20854)
                  Function name         Calls      Cycles   % change
!                       strcmp       -40822    -1198640  93.77% faster
!                      putHash            0      -32912   6.61% faster
!                      getHash            0      -28376   7.86% faster
!                      remHash            0       -7856   5.91% faster
!                    hashIndex            0       10000   1.49% slower
5 differences; -1257784 cycles (-0.025 secs at 50 MHz)
25.01% faster overall (ignoring system calls).
```

What qxdiff annotations mean

Each line in the `qxdiff` report lists a function whose time has changed. The annotations on each line mean the following:

Symbol	Meaning
-	Called in the baseline run only
!	Called in both runs; timing changed
+	Called in the changed run only

Unless you change the calling structure of the program, `qxdiff` reports only the functions whose performance has changed (lines marked “!”). You can use the `-l` option to print a description of the columns and annotations in the `qxdiff` report.

The `qxdiff` report summarizes the total number of differences, the change in the total function time count, and the overall percentage change between the baseline and the changed run. The `qxdiff` script prints the report to `stdout`.

By default, `qxdiff` reports changes in operating system call times. These changes can be caused by a different number of calls or changed network traffic loads. You can use the `-i` option to tell `qxdiff` to ignore functions that make system calls in order to focus on the changes in compute-bound functions only.

The example on the following page shows the effect of the `-i` option.

For example, you can use `qxdiff -i` to see the effect of a different compiler optimization level on the `improved_testHash` program.

Compile an optimized version of the improved_testHash program

```
% cc  -c -O4 improved_hash.c -o improved_hash.o
% make improved_testHash.pure

quantify cc  -Bstatic -g  -o improved_testHash.pure testHash.o improved_hash.o
Quantify 4.4 SunOS 4.1, Copyright 1993-1999 Rational Software Corp.
Instrumenting: improved_hash.o Linking
```

Run the optimized program

```
% setenv QUANTIFYOPTIONS '-windows=no -write-export-file= '
% improved_testHash.pure 500 test_words

****  Quantify instrumented improved_testHash.pure (pid 9952 at Wed Jan 13
15:04:39 1999)
Quantify 4.4 SunOS 4.1, Copyright 1993-1999 Rational Software Corp.
  * For contact information type: "quantify -help"
  * Quantify licensed to Rational Software
  * Quantify instruction counting enabled.
Testing the first 500 entries from test_words with a hashtable of size 13.
All tests passed.

Quantify: Sending data for 62 of 186 functions
 from improved_testHash.pure (pid 9952)..........done.
To view your saved Quantify data, type:
 qv improved_testHash.pure.9952.0.qv

Quantify: Cumulative Resource Statistics for improved_testHash.pure (pid 9952)
 *                                            cycles       secs
 * Total counted time:                       5422968      0.164 (100.0%)
 *     Time in your code:                    3552297      0.108 ( 65.5%)
 *     Time in system calls:                 1870671      0.057 ( 34.5%)
 *
 * Note: Data collected assuming a sparcstation_elc with clock rate of 33 MHz.
 * Note: These times exclude Quantify overhead and possible memory effects.
 *
 * Elapsed data collection time:       0.747 secs
 *
 * Note: This measurement includes Quantify overhead.
```

Use `qxdiff` to compare the original run with the optimized run

```
% qxdiff -i improved_testHash.pure.9928.0.qx improved_testHash.pure.9952.0.qx

Differences between:
program improved_testHash.pure (pid 9928) collected on sparcstation_elc (33
MHz)
program improved_testHash.pure (pid 9952) collected on sparcstation_elc (33
MHz)
                   Function name Calls     Cycles % change
!                      hashIndex     0    -614212  64.72% faster
!                        putHash     0    -347142  51.81% faster
!                        getHash     0    -214193  45.94% faster
!                        remHash     0     -92545  51.69% faster
!                   delHashTable     0     -10322  55.70% faster
!                  makeHashTable     0        -13  41.94% faster
6 differences; -1278427 cycles (-0.039 secs at 33 MHz)
26.46% faster overall (ignoring system calls).
```

The optimized run is 26% faster

Managing cached object files

To improve build-time performance, Quantify caches its instrumented versions of all the libraries and object files that are used by the program. When you rebuild a program, Quantify updates only the new or modified files; otherwise it uses the cached versions.

You can identify an instrumented cache file by its name. It includes _pure_ and a Quantify version number. It can also include information about the size of the original file, or the name and version number of the operating system.

Quantify writes Quantify'd files to the original file's directory if that is writable, or to the global cache directory.

You can control how instrumented libraries and files are cached by:

- Specifying the global cache directory
- Directing Quantify to save all cache files in the global cache directory
- Restricting Quantify from caching files in certain directories

See "Build-time options" on page B-4.

Deleting cached object files

Since Quantify rebuilds cached files as needed, you can remove them at any time in order to conserve disk space.

Using the pure_remove_old_files script

Note: The `pure_remove_old_files` script also removes Purify and PureCoverage cache files.

To remove cache files, use the `pure_remove_old_files` script located in the `<quantifyhome>` directory. For example, to remove all cache files that are 14 days or older:

```
% pure_remove_old_files / 14
```

The first argument (/) specifies the path, the second argument (14) specifies the number of days. This command removes files 14 days or older recursively from the root directory /.

Using a cron job

To automate the removal of cache files, create a cron job that periodically removes the files. For example, to remove files that have not been accessed in two weeks, type:

```
% crontab -e
```

Add this entry to the crontab file:

```
15 2 * * * <quantifyhome>/pure_remove_old_files / 14
```

This runs pure_remove_old_files each day at 2:15 A.M., and removes all cached files starting at the root directory / that have not been read within the last 14 days.

To remove all of the cache files in the current directory and subdirectories, use:

```
% pure_remove_old_files . 0
```

This is useful in clean targets of makefiles.

A Using Quantify Options and API Functions

This appendix describes how to use Quantify options and Application Programming Interface (API) functions. It includes:

- Quantify option syntax
- Quantify option types
- Quantify option processing
- Using the `-ignore-runtime-environment` option
- Calling Quantify API functions from a debugger
- Calling Quantify API functions from your program
- Linking with the Quantify stubs library

For a complete list of Quantify options and API functions, see Appendix B, "Options and API Reference."

Using Quantify options

Option syntax

A Quantify option consists of a word or phrase that begins with a hyphen. For example:

```
-record-system-calls=no
```

- The leading hyphen is required.
- No space is allowed on either side of the equal sign (=).
- Quantify ignores case, hyphens, and underscores in the option name. For example, the option `-record-system-calls` is equivalent to `-record_system_calls` and `-RecordSystemCalls`.
- For options that take a list of directories, you can specify the directory names separated by colons (:). For example:

  ```
  -forbidden-directories=/usr/home/program:/usr/home/program1
  ```

- Specify a list of signals separated by commas (,). For example:

  ```
  -handle-signals=SIGUSR1,SIGUSR2,SIGILL
  ```

- You can use wildcards. For example, in filenames: `program*` matches `program4`, `/dira/dirb/program.o`, and `/dira/dirb/program1.o`.

Using conversion characters in filenames

You can use conversion characters when you specify filenames for options such as `-filename-prefix`, `-log-file`, `-run-at-save`, and `-run-at-exit`. Quantify supports these conversion characters:

Character	Converts to
%V	Full pathname of the program with "/" replaced by "_"
%v	Program name
%p	Process ID (pid)
%T	Thread identifier
%t	Current time (hh:mm:ss)
%d	Current date (yymmdd)
%n	Sequence number, starting at 0. This value is incremented as each dataset is saved.
%b	Name of a `/tmp` file containing binary data
%s	Name of a `/tmp` file containing program summary
%x	Name of a `/tmp` file containing export data

Quantify expands conversion characters at run time to form uniquely named datasets.

If the filename is unqualified (does not contain "/"), Quantify writes it to the directory where the program resides. Qualified filenames can be absolute, or relative to the current working directory. For example, if you specify the option:

```
-log-file=./%v.qlog
```

Quantify writes the log file to the current working directory. If the program is called `test`, the log file is called `./test.qlog`.

How Quantify creates filenames

By default, Quantify forms filenames by expanding any conversion characters in the value of the `-filename-prefix` option and appending an extension. By default, `-filename-prefix=%v.%p.%n`. Quantify saves data to these files:

Filename	Data saved
`%v.%p.%n.qv`	The binary dataset file
`%v.%p.%n.qx`	The export data file
`%v.%p.%n.qs`	The program summary file
`%v.%p.%n.ql`	The current Function List
`%v.%p.%n.qfd`	The current Function Detail data
`%v.%p.%n.ps`	The Call Graph, in PostScript form
`<source>.%p`	The annotated source file
`$HOME/.qvrc`	The X resource file for customizing Quantify

Note: The value of `%n` is incremented each time the dataset is saved.

Option types

Quantify uses three types of options: boolean, string, and integer.

- Boolean options take the values yes or no, or `true` or `false`. If you do not specify an explicit value, the value is yes. For example, the option settings `-record-data` and `-record-data=yes` are identical.
- String options can be a string of any kind. String options are used for programs, directories, file names, lists of file descriptor numbers, system call and signal numbers, shell commands, floating point numbers, and directory paths.

  If you do not specify an explicit value for a string option, the value is cleared. For example, the option

  ```
  -write-summary-file=./quantifyout
  ```

  routes Quantify reports to the file `quantifyout` in the current directory. The option `-write-summary-file=`, without a value, clears any default specification of a summary file and writes the reports to the standard output.
- Integer options can be set to any whole number. For example, the option `-max-threads=60` instructs Quantify to expect a maximum of 60 threads to be created during a run. Integer values cannot be cleared.

How Quantify processes options

You can specify Quantify options in environment variables and on the link line. Quantify processes options in the following order (highest precedence first):

1. Options specified in the `QUANTIFYOPTIONS` or `PUREOPTIONS` environment variables
2. Options specified on the link line

Specifying options in environment variables

You can specify any Quantify option in the `QUANTIFYOPTIONS` and `PUREOPTIONS` environment variables. Values in `PUREOPTIONS` apply to Quantify, PureCoverage, and Purify software products. The values specified in `QUANTIFYOPTIONS` take precedence over `PUREOPTIONS`.

Quantify applies *build-time* options specified in environment variables when a Quantify'd application is built. Any build-time options on the link line override environment variables.

Quantify applies *run-time* options specified in environment variables when you run the Quantify'd program. The environment values in force when you run the program override any defaults specified on the link line.

If an option is specified more than once in an environment variable, Quantify applies the *first* value it sees. To add an overriding value for the `-log-file` option without changing other options specified, use a command like:

```
csh     % setenv QUANTIFYOPTIONS "-log-file=new $QUANTIFYOPTIONS"
sh, ksh $ QUANTIFYOPTIONS="-log-file=new $QUANTIFYOPTIONS"; \
      export QUANTIFYOPTIONS
```

Setting site-wide options

You can use the `PUREOPTIONS` environment variable to set options that apply to Quantify, PureCoverage, and Purify software products.

For example, if your site has a central shared file that is sourced by all users' `.cshrc` or `.profile` files, you can set `-cache-dir=alternate/dir` in the `PUREOPTIONS` environment variable to apply to all users.

Specifying options on the link line

You can specify any Quantify option on the link line. For example:

```
quantify -cache-dir=$HOME/qcache -always-use-cache-dir $CC ...
```

Quantify applies build-time options to the Quantify build command being run. Quantify builds run-time options into the executable so that they become the default values for the Quantify'd executable. This is a convenient way to build a program with nonstandard default values for run-time options.

For example, the link line:

```
% quantify -collection-granularity=function \
       -record-system-calls=no cc ...
```

instructs Quantify to instrument the program at build time to collect data only at the function level, and at run time to avoid collecting system call data.

Using the -ignore-runtime-environment option

You can use the `-ignore-runtime-environment` option when you build your executable to make sure that the run-time options you specify remain in effect whenever the executable is run.

The `-ignore-runtime-environment` builds into an executable all the run-time options specified on the link line along with any run-time options specified in the `QUANTIFYOPTIONS` and `PUREOPTIONS` environment variables.

When the Quantify'd program is run, Quantify ignores the current option values in the environment variables and uses the built-in values.

Use the `-ignore-runtime-environment` option when:

- You want someone else to run your program without their run-time environment modifying your run-time option specifications.
- Your program is started automatically by another program, and you cannot set the environment variable for that program.
- You have several Quantify'd programs running at one time, and you cannot specify options for each program.

Note: Use the `-ignore-runtime-environment` option at build time only. Quantify ignores this option if you specify it at run time.

Using Quantify API functions

You can call Quantify API functions from a debugger or from your program. Unless otherwise specified, Quantify functions return 1 (`True`), indicating success.

Calling API functions from a debugger

You can use Quantify's API functions by setting breakpoints in your debugger and then calling the appropriate function when the breakpoint is reached:

Solaris SunOS4

```
(gdb) call quantify_clear_data()
```

Solaris SunOS4

```
(dbx) print quantify_clear_data()
```

HPUX

```
x(xdb) p quantify_clear_data()
```

Note: You can get help about using the API functions at run time by calling the API function `quantify_help()`. This function prints a list of the Quantify API functions, and supports cut and paste of the function names into the debugger.

Calling API functions from your program

To call Quantify functions from ANSI C and C++ programs, include the file `quantify.h`:

```
# include <quantify.h>
```

This header file is located in the same directory as Quantify. You might need to add the compiler option `-I<quantifyhome>` in your makefile to locate it.

Note: After embedding API functions in your code, you must recompile your program.

Linking with the Quantify stubs library

If you call Quantify functions in your program, you should link with the Quantify API stubs library. This is a small library that stubs out all the Quantify API functions when you are *not* using Quantify. When you *are* using Quantify, the stubs are ignored.

Add the library `<quantifyhome>/libquantify_stubs.a` to your link line.

Here is an example of a makefile:

```
# Build Hello World
# Use: make -f Makefile.hello_world a.out.pure

# Quantify-related flags
QDIR                  = ‘quantify -print-home-dir‘
QFLAGS                = -ignore-runtime-options
QUANTIFY              = quantify $(QFLAGS)
QSTUBS                = $(QDIR)/quantify_stubs.a

# General flags
CC                    = cc
CFLAGS                = -g -I$(QDIR)

# Targets
all:       a.out a.out.pure

a.out:     hello_world.c
           $(CC) $(CFLAGS) -o $@ $? $(QSTUBS)

a.out.pure: hello_world.c
           $(QUANTIFY) $(CC) $(CFLAGS) -o $@ $? $(QSTUBS)
```

B Options and API Reference

This appendix presents a complete list of Quantify's options and API functions. For a description of how to specify Quantify options and API functions, see Appendix A, "Using Quantify Options and API Functions."

Build-time options quick reference

Build-time options	Default	Page
`-always-use-cache-dir`	no	B-4
`-cache-dir`	`<quantifyhome>/cache`	B-4
`-collection-granularity`	`line`	B-4
`-collector`	not set	B-4
`-forbidden-directories`	system-dependent	B-5
`-force-rebuild`	no	B-5
`-g++`	no	B-5
`-ignore-runtime-environment`	no	B-5
`-linker`	system-dependent	B-5
`-use-machine`	system-dependent	B-6
`-print-home-dir`	not set	B-6
`-version`	not set	B-6

qv options quick reference

qv options	Default	Page
`-add-annotation`	not set	B-7
`-print-annotations`	no	B-7
`-windows`	`yes`	B-7
`-write-export-file`	`none`	B-7
`-write-summary-file`	`/dev/tty`	B-7

Run-time options quick reference

Quantify options	Default	Page
`-api-handler-signals`	not set	B-15
`-append-logfile`	`no`	B-16
`-auto-mount-prefix`	`/tmp_mnt`	B-17
`-avoid-recording-system-calls`	system-dependent	B-8
`-copy-fd-output-to-logfile`	not set	B-16
`-fds`	`26`	B-17
`-filename-prefix`	`%v.%p.%n`	B-13
`-handle-signals`	not set	B-15
`-ignore-signals`	not set	B-15
`-logfile`	not set	B-16
`-max-threads`	`20`	B-12
`-measure-timed-calls`	`elapsed-time`	B-8
`-never-record-system-calls`	system-dependent	B-8
`-output-limit`	`1000000`	B-16
`-program-name`	`argv[0]`	B-17
`-record-child-process-data`	`no`	B-13
`-record-data`	`yes`	B-9
`-record-dynamic-library-data`	`yes`	B-9
`-record-register-window-traps`	`no`	B-9
`-record-system-calls`	`yes`	B-9
`-report-excluded-time`	`0.5`	B-9
`-run-at-exit`	not set	B-15
`-run-at-save`	not set	B-15
`-save-data-on-signals`	`yes`	B-15
`-save-thread-data`	`composite`	B-12
`-thread-stack-change`	`0x1000`	B-12
`-threads`	`no`	B-12
`-user-path`	not set	B-17
`-windows`	`yes`	B-7
`-write-export-file`	`none`	B-7
`-write-summary-file`	`/dev/tty`	B-7

Run-time API functions quick reference

Unless otherwise indicated, all Quantify API functions return 1 (true), indicating success.

Quantify API functions	Page
pure_printf (char *format, ...)	B-16
pure_logfile_printf (char *format, ...)	B-16
quantify_help (void)	B-12
quantify_is_running (void)	B-12
quantify_print_recording_state (void)	B-12
quantify_save_data (void) quantify_save_data_to_file (char *filename)	B-14
quantify_add_annotation (char *annotation)	B-14
quantify_clear_data (void)	B-14
quantify_disable_recording_data (void)	B-11
quantify_start_recording_data (void) quantify_stop_recording_data (void) quantify_is_recording_data (void)	B-10
quantify_start_recording_dynamic_library_data (void) quantify_stop_recording_dynamic_library_data (void) quantify_is_recording_dynamic_library_data (void)	B-11
quantify_start_recording_register_window_traps (void) quantify_stop_recording_register_window_traps (void) quantify_is_recording_register_window_traps (void)	B-11
quantify_start_recording_system_call (char *system_call_string) quantify_stop_recording_system_call (char *system_call_string) quantify_is_recording_system_call (char *system_call_string)	B-10
quantify_start_recording_system_calls (void) quantify_stop_recording_system_calls (void) quantify_is_recording_system_calls (void)	B-10
-save-data-on-signals -handle-signals -ignore-signals	B-15

Build-time options

Build-time options	Default
`-always-use-cache-dir`	**`no`**
Specifies whether all Quantify'd libraries and object files are written to the global cache directory, even if they reside in writable directories.	
`-cache-dir`	**`<quantifyhome>/cache`**
Specifies the location of the global directory where Quantify caches instrumented versions of object files and libraries. See "Deleting cached object files" on page 5-10.	
`-collection-granularity`	**`line`**
Specifies the level of collection granularity for files containing debugging information. You can specify `function`, `basic-block`, or `line`. See "Changing the granularity of collected data" on page 4-11.	
`-collector`	not set

Specifies the name of the collect program to be used to sequence and collect static constructors in C++ code. You must set this option to the name of the collect program used by the g++ compiler.
 	
To find the name of the collect program used by the g++ compiler, type:
 	
`% g++ -v myprogram.c`
 	
For example, if the collect program is:
 	
`/usr/local/lib/gcc-lib/ld`
 	
use the command:
 	
`% quantify -g++=yes \`
`-collector=/usr/local/lib/gcc-lib/ld g++ myprogram.c`
 	
Note: g++ on Solaris 2 does not use a collector for C++ programs. Quantify on Solaris ignores this option.	

(Build-time options continued on next page)

Build-time options	Default
`-forbidden-directories`	system-dependent
Specifies a colon-separated list of directories into which Quantify cannot write files, even if the directories listed are writable. All the subdirectories of forbidden directories are also forbidden. The default values are: Sun OS4 `/lib:/opt:/usr/lib:/usr/5lib:/usr/ucb/lib:/usr/lang:/usr/local` Solaris `/lib:/opt:/usr/lib:/usr/4lib:/usr/ucblib:/usr/lang:/usr/local` HPUX `/lib:/usr/lib:/usr/local:/opt`	
`-force-rebuild`	**`no`**
Specifies whether Quantify creates a new instrumented version of every file required by the application. You can use this option when changing the granularity level with the `-collection-granularity` option.	
`-g++`	**`no`**
Specifies that the g++ compiler should be used. This option additionally tells the linker to invoke the appropriate collect program (specified using the `-collector` option).	
`-ignore-runtime-environment`	**`no`**
Specifies whether the run-time Quantify environment overrides the option values used in building the program. This option is useful if you are building a Quantify'd program for someone else to run, and you want to make sure that the options you specify are in effect at run time. See "Using the -ignore-runtime-environment option" on page A-8.	
`-linker`	system-dependent
Specifies the name of the linker that Quantify should invoke to produce the executable. Use this option only if you need to bypass the default linker. The default linkers are: Sun OS4 HPUX `/bin/ld` Solaris `/usr/ccs/bin/ld`	

(Build-time options continued on next page)

Build-time options	Default
`-use-machine`	system-dependent

Specifies the build-time analysis of instruction times according to a particular machine. The default value for this option is one of the machines defined in the `.machine.<platform>` file.

This option is particularly useful if you build your application on a fast compile server but want to test the performance on a more modest machine. You can instrument the application during the build and test it later without having to re-Quantify the application. See “How Quantify times register-window traps” on page 3-15.

The entries in a `.machine.<platform>` file describe:

- Processors and the cost of different instructions
- Floating-point accelerators used to perform floating point operations
- Machines that are combinations of specific processors and fpas running at different clock rates

The `.machine.<platform>` file supplied with Quantify defines machines currently available. However, new machines are being released and the data in the `.machine.<platform>` file might change. In addition, you can define anticipated or imaginary machines to see what performance difference a change of hardware might provide.

Contact Rational Software Technical Support for additional information on the .machine.`<platform>` file and any new machine entries.

`-print-home-dir`	not set

Prints the name of the directory where Quantify is installed, then exits. You can use this option to build the compiler command when including the `quantify.h` file from the installation directory:

```
$CC -c $CFLAGS -I‘quantify -print-home-dir‘ myprogram.c
```

`-version`	not set

Prints Quantify’s version number string to `stdout` and then exits. You can use this option to identify which version of Quantify is in use while running a test suite by incorporating these lines in your test harness scripts:

```
#!/bin/sh
...
echo "Run monitored by : ‘quantify -version‘"
...
```

qv options

For instructions on how to use , see “Running qv” on page 5-2.

qv options	Default
`-add-annotation` Specifies a string to add to the binary file. The annotation string in a file can be printed at a later time with the `-print-annotations` option.	not set
`-print-annotations` Specifies whether to write annotations to `stdout`.	**no**
`-windows` Specifies whether `qv` runs with the graphical display. When it is set to `yes`, `qv` displays the Control Panel. When set to `no`, `qv` saves the collected data to a file. See “Automating data analysis” on page 5-5.	**yes**
`-write-export-file` Specifies the name of the file used to write the export data file for a dataset. By default, the value `none` specifies that no export file is written. If you specify no value using `-write-export-file`, Quantify creates the export filename based on the value of the `-file-prefix` option. Quantify appends the `.qx` extension to the option value. See “Exporting performance data” on page 5-1.	**none**
`-write-summary-file` Specifies the name of the file used to write the program summary for a dataset. By default, Quantify writes the program summary to the current output. If you specify the value `none`, Quantify does not write a program summary. If you specify `-write-summary-file` without a value, Quantify creates the summary filename based on the value of the `-file-prefix` option. Quantify appends the `.qs` extension to the option value. See “Exporting performance data” on page 5-1.	**/dev/tty**

Data collection options

Data collection options	Default
`-avoid-recording-system-calls`	system-dependent
Specifies that Quantify not time specified system calls. The `syscall_list` is a comma-delimited list of system call names or numbers. See “Avoiding timing for specific system calls” on page 4-5. Specify a list of system-call names or numbers, separated by commas. The names and numeric values in the `/usr/include/sys/syscall.h` header file. The default values for `-avoid-recording-system-calls` are: Sun OS4 `SYS_exit, SYS_select, SYS_listen, SYS_sigpause` Solaris `SYS_exit, SYS_poll` HPUX `SYS_exit, SYS_select, SYS_listen, SYS_sigpause`	
`-measure-timed-calls`	**`elapsed-time`**
Specifies how Quantify measures the time required for system calls and dynamic library operations. You can specify `elapsed-time`, `user+system`, `user`, or `system`. If `-measure-timed-calls=elapsed-time`, Quantify records the elapsed (wall-clock) time the operation took using the `gettimeofday` system call. If the value is `user`, `system` or `user+system`, Quantify records the time the kernel recorded for that operation using the `getrusage` system call. See “Controlling how system calls are timed” on page 4-4.	
Solaris **`-never-record-system-calls`**	system-dependent
Specifies that Quantify not time certain system calls. The `syscall_list` is a comma-delimited list of system call names or numbers. Unlike `-avoid-recording-system-calls,` this option ensures that time is never collected on these system calls and that time is never added to the excluded time Quantify tracks and reports. Use this option on Solaris to avoid timing system calls that cause LWP switches to occur in symmetric multi-processor (SMP) installations. Setting this option avoids double-counting the system call time that elapsed while one thread was blocked but another thread was able to execute. See “Collecting data in threaded programs” on page 4-20. The default on Solaris is: `SYS_sigtimedwait`, `SYS_lwp_sema_wait`, `SYS_lwp_create`, `SYS_lwp_kill`, `SYS_lwp_mutex_lock`, `SYS_lwp_mutex_unlock`, `SYS_lwp_cond_wait`, `SYS_signotifywait`	

Data collection options	Default
`-record-data`	**`yes`**
Specifies whether Quantify records any data over the entire program, including data for system calls and register-window traps. This option establishes the initial recording state only. See “Avoiding all data recording” on page 4-2.	
`-record-system-calls`	**`yes`**
Specifies whether Quantify records the time spent by a function making calls to the operating system. This option establishes the initial recording state only. See “Timing system calls” on page 4-2.	
Sun OS4 Solaris **`-record-register-window-traps`**	**`no`**
Specifies whether Quantify records the time spent by a function in saving and restoring register windows. This option establishes the initial recording state only. See “Timing register-window traps” on page 4-10.	
`-record-dynamic-library-data`	**`yes`**
Specifies whether Quantify records the time spent performing dynamic library operations. This option establishes the initial recording state only. See “Timing shared-library operations” on page 4-7.	
`-report-excluded-time`	**`0.5`**
Specifies whether to report any time that has been excluded from a dataset. The excluded time is reported if, as a percentage of the combined total counted time and the excluded time, it exceeds the specified value. `no` sets the threshold at `0`, so Quantify never reports excluded times. `yes` sets the threshold at `100`, so Quantify always reports excluded times. See “Reporting excluded system-call time” on page 4-6.	

Data collection API functions

Data collection functions

`int quantify_start_recording_data (void)`
`int quantify_stop_recording_data (void)`
`int quantify_is_recording_data (void)`

Starts and stops the recording of all count data, including code, system call, and register-window trap data.

`quantify_is_recording_data` returns TRUE if Quantify is recording data, FALSE otherwise. See also, "Collecting partial data for a run" on page 4-13.

`int quantify_start_recording_system_calls (void)`
`int quantify_stop_recording_system_calls (void)`
`int quantify_is_recording_system_calls (void)`

Starts and stops the recording of system-call timing data.

`quantify_is_recording_system_calls` returns TRUE if Quantify is recording system calls, FALSE otherwise.

`int quantify_start_recording_system_call (char * syscall_string)`
`int quantify_stop_recording_system_call (char * syscall_string)`
`int quantify_is_recording_system_call (char * syscall_string)`

Starts and stops the recording of system-call timing data for specific system calls. The first two functions take a string containing one or more system-call names or numbers. They return TRUE if the argument list is well-formed, FALSE otherwise.

`quantify_is_recording_system_call` takes a *single* system-call name or number and returns TRUE if Quantify is recording that system call, FALSE otherwise. The system-call names and numbers are located in the system header file `/usr/include/sys/syscall.h`.

Note: Overall system-call timing must be enabled for these functions to have an effect. To enable system-call timing, use the `-record-system-calls` option or the `quantify_start_recording_system_calls` API function.

(Data collection functions continued on next page)

Data collection functions

`int quantify_start_recording_dynamic_library_data (void)`
`int quantify_stop_recording_dynamic_library_data (void)`
`int quantify_is_recording_dynamic_library_data (void)`

Starts and stops the recording of dynamic library timing data, for example, during `dlopen` operations under SunOS 4.1 and Solaris 2, or `shl_load` operations under `HP-UX` machines. The function `quantify_is_recording_dynamic_library_data` returns `TRUE` if Quantify is recording dynamic library calls, `FALSE` otherwise. See “Timing shared-library operations” on page 4-7.

Sun OS4 Solaris

`int quantify_start_recording_register_window_traps (void)`
`int quantify_stop_recording_register_window_traps (void)`
`int quantify_is_recording_register_window_traps (void)`

Starts and stops the recording of register-window trap data. The function `quantify_is_recording_register_window_traps` returns `TRUE` if Quantify is recording register-window traps, `FALSE` otherwise. On non-SPARC platforms, these functions always return `FALSE`. See “Timing register-window traps” on page 4-10.

`int quantify_disable_recording_data (void)`

Disables collection of *all* data by Quantify. This function always returns `TRUE`.

Once this function is called, you cannot re-enable data collection for this process. No data is recorded and no data is saved.

Run-time collection status API functions

Run-time collection status functions
`int quantify_is_running (void)`
Returns `TRUE` if the executable is Quantify'd, `FALSE` otherwise. Use this function to enclose special-purpose application code to execute only in a Quantify'd environment.
`int quantify_help (void)`
Prints a message describing most of Quantify's API functions. Always returns `TRUE`.
`int quantify_print_recording_state (void)`
Prints the current recording state of the process. Always returns `TRUE`.

Threads options

Threads options	Default
`-max-threads`	**`20`**
Specifies the maximum number of threads run at any time.	
`-save-thread-data`	**`composite`**
Specifies whether to save the per-stack performance data as separate datasets. The value is a comma-delimited string containing one or more keywords: `stack` saves each per-stack dataset. `composite` saves the composite dataset. By default, Quantify saves only the `composite` dataset.	
`-threads`	**`no`**
Specifies thread support. This option is automatically set to `yes` when you link with a supported threads package.	
`-thread-stack-change`	**`0x1000`**
Specifies the minimum size, in bytes, of a change to the stack pointer that signals a thread context switch. See "Collecting data in threaded programs" on page 4-20.	

Child process options

Child process options	Default
`-record-child-process-data`	**no**
Specifies whether Quantify records the data for any child processes created by `fork` and `vfork`. See “Collecting data for child processes” on page 4-18.	

Options for saving data

Options for saving data	Default
`-filename-prefix`	**`%v.%p.%n`**
Specifies the prefix name for report filenames. See “How Quantify creates filenames” on page A-4.	
`-write-export-file`	**none**
Specifies the name of the file used to write the export data file for a dataset. By default, the value `none` specifies that no export file is written. If you specify no value using `-write-export-file`, Quantify creates the export filename based on the value of the `-file-prefix` option. Quantify appends the `.qx` extension to the option value. See “Exporting performance data” on page 5-1.	
`-write-summary-file`	**`/dev/tty`**
Specifies the name of the file used to write the program summary for a dataset. By default, Quantify writes the program summary to the current output. If you specify the value `none`, Quantify does not write a program summary. If you specify `-write-summary-file` without a value, Quantify creates the summary filename based on the value of the `-file-prefix` option. Quantify appends the `.qs` extension to the option value. See “Exporting performance data” on page 5-1.	

API functions for saving data

Functions for saving data

`int quantify_clear_data (void)`

Clears all the performance data recorded to this point. You can use this function, for example, to ignore the performance data collected about the start-up phase of your program. This function always returns `TRUE`.

`int quantify_save_data (void)`
`int quantify_save_data_to_file (char * filename)`

Saves all the data recorded since program start or the last call to `quantify_clear_data` into a data file. You can view the data file using `qv`. After saving the data, these functions call `quantify_clear_data` in order to ensure non-overlapping datasets. Both functions return `TRUE` if successful, `FALSE` otherwise. See "Saving data incrementally" on page 5-4.

The function `quantify_save_data` creates a filename based on the value of the `-filename-prefix` option. The function `quantify_save_data_to_file` takes a string that overrides the value of the `-filename-prefix` option. You can use conversion characters in filenames. The default `-filename-prefix` is `%v.%p.%n`, which writes data files based on the application basename, the current process ID, and the current dataset number. Quantify *always* adds the extension `.qv` to each dataset file it writes.

`%n` is incremented each time the dataset is saved.

See "Using conversion characters in filenames" on page A-3, and "How Quantify creates filenames" on page A-4.

`int quantify_add_annotation (char * annotation)`

Adds the given string to the *next* saved dataset. You can view the annotation strings later, using the `qv` option `-print-annotations`. Use this function to record any special data-collection circumstances with a particular `qv` file. This function returns the length of the string, or `FALSE` if passed a `NULL` pointer.

Options for saving data on signals

Options for saving data on signals	Default
`-save-data-on-signals`	**`yes`**
Specifies whether Quantify installs default signal handlers for any fatal signals your program does not handle itself. This option controls whether Quantify automatically saves data on fatal signals. See “Saving data on signals” on page 4-17.	
`-handle-signals` **`-ignore-signals`**	not set not set
Specifies whether Quantify saves the collected performance data whenever the program receives a signal that would be fatal. If you want Quantify to save data on any nonfatal signals, specify the signals with the `-handle-signals` option. If you do not want Quantify to save data on some fatal signals, specify them with the `-ignore-signals` option. See “Saving data on signals” on page 4-17.	
`-api-handler-signals`	not set
Installs signal handlers. For example: `-api-handler-signals= SIGUSR1,SIGUSR2` The first signal specified is associated with `quantify_save_data`; the second signal is associated with `quantify_clear_data`. See “Collecting data in long-running programs” on page 4-18.	

Options for automating data analysis

Options for automating data analysis	Default
`-run-at-exit`	not set
Specifies a shell script to run at program exit. See “Automating data analysis” on page 5-5.	
`-run-at-save`	not set
Specifies a shell script to run each time the program saves counts. See “Automating data analysis” on page 5-5.	
`-windows`	**`yes`**
Specifies whether qv runs with the graphical display. When it is set to `yes`, `qv` displays the Control Panel. When set to no, qv saves the collected data to a file. See “Automating data analysis” on page 5-5.	

Output options

Output options	Default
`-logfile`	not set
Specifies the name of the log file to which Quantify's output is sent.	
`-append-logfile`	**`no`**
Specifies that Quantify output be appended to the current log file rather than replacing it.	
`-output-limit`	**`1000000`**
Use with the `-logfile` option to restrict the size of the log file and conserve disk space. Specify the maximum size, in bytes, of the log file. All output is truncated beyond this size.	
`-copy-fd-output-to-logfile`	not set
Specifies that output from file descriptors be appended to the log file. Specify a comma-delimited list of file descriptors. Output written to these file descriptors is copied into the log file and helps you reconstruct the actions of the user. For example, to copy output written to `stdout` and `stderr` into the log file, interspersed with Quantify output, use: `% quantify -copy-fd-output-to-logfile=1,2 cc foo.c`	

Output API functions

Output functions
`int pure_logfile_printf (char *format, ...)`
Prints output from the program to the log file if set.
`int pure_printf (char *format, ...)`
Prints output from the program to the log file. If not set, output goes to `stderr`.

Miscellaneous run-time options

Special run-time options	Default
`-auto-mount-prefix`	**`/tmp_mnt`**
Specifies the directory prefix used by the file system auto-mounter, usually `/tmp_mnt`, to mount remote file systems in NFS environments. Use this option to strip the prefix, if present. **Note:** If your auto-mounter alters the prefix, instead of adding a prefix, use the syntax: `-auto-mount-prefix=/tmp_mnt/home:/homes` to specify that the real filename is constructed from the apparent by replacing `/tmp_mnt/home with /homes`. If this option is not set correctly, Quantify might be unable to access files on auto-mounted file systems. The auto-mounter might not recognize their names.	
`-fds`	**26**
Specifies the default set of file descriptors used by Quantify in case they clash with the ones used by your program. To use file descriptors `57` and `58` instead of the default `26` and `27`, use: `% setenv QUANTIFYOPTIONS -fds=57`	
`-program-name`	**`argv[0]`**
Specifies the full pathname of the Quantify'd program if `argv[0]` contains an undesirable or incorrect value. This can occur if you change `argv[0]` in your program. For example, many programs that are invoked by a script rename their `argv[0]` to the name of the script.	
`-user-path`	not set
Specifies a colon-delimited list of directories that Quantify should search to find the program at run time. Normally, the program can be found by looking at `argv[0]` and your `$PATH`, but it is possible to execute a program not on your path. If your program is forked by another and Quantify cannot find it to read the debug symbols, set this option to the directory where your program is stored. This option takes precedence over `$PATH`. Use this option if Quantify cannot find source files when attempting to display Annotated Source windows.	

C Common Questions

This appendix contains answers to the most frequently asked questions about using Quantify.

Questions about instrumentation

Does Quantify work with Purify, and PureCoverage?

You cannot use Quantify on the same files that you instrument with Purify or with PureCoverage. The instruction sequences Quantify inserts are different from those inserted by Purify and PureCoverage.

What are the `.pure` and `*_pure_q512_200.o` files created in my working directory? Is it safe to delete them?

The `*_pure_q512_200.o` files are the Quantify'd versions of your object files for your particular machine. They are kept so that if you change one object file, the rest of the objects do not have to be re-Quantify'd. The `.pure` file is a part of the file locking mechanism Quantify uses when creating instrumented files. If you delete any of these files, Quantify will rebuild them when necessary. See "Managing cached object files" on page 5-10.

Does Quantify work with shared libraries?

Yes, including the use of dlopen on SunOS 4.1 and Solaris 2, and shl_load on HP-UX. If you dynamically open a library that Quantify has not seen before, or if you delete a cached Quantify'd shared library, Quantify builds it, if required, at run time. See "Timing shared-library operations" on page 4-7.

Why do I get the following failure?

```
Memory allocation failed during a request for 12345678 bytes.
```

If you get this message, you need to add at least 13 megabytes of swap space to your configuration. You can increase the swap space on your machine or instrument your application on a machine that has more swap space.

The first time you link with Quantify, you need more swap space than on subsequent links, because Quantify instruments and caches versions of all the libraries your application uses.

To determine the current swap size and increase it, use the appropriate commands for your operating system. See the manual pages for each command's options and arguments.

Sun OS4 To determine the current swap size, use the `pstat -s` command. To increase your swap space, use the `mkfile` and `swapon` commands.

Solaris To determine the current swap size, use the `swap -s` command. To increase your swap space, use the `mkfile` and `swap -a` commands.

HPUX To determine the current swap size, use the `swapinfo` command. To increase your swap space or increase the data-segment size limit, use the `sam` (System Administration Manager) application. You must be root to use these commands.

Sun OS4

Why do I get an ld out-of-date warning?

```
ld: /usr/local/quantify/cache/lib/libc_pure_q512_200.sa.1.7:
 warning: table of contents for archive is out of date;
rerun ranlib(1)
```

Your workstation's clock is out of sync with your file server. To synchronize them, use the command:

```
root% rdate file-server
```

You must be root to do this.

Questions about data collection

Can I collect data over just part of my program?

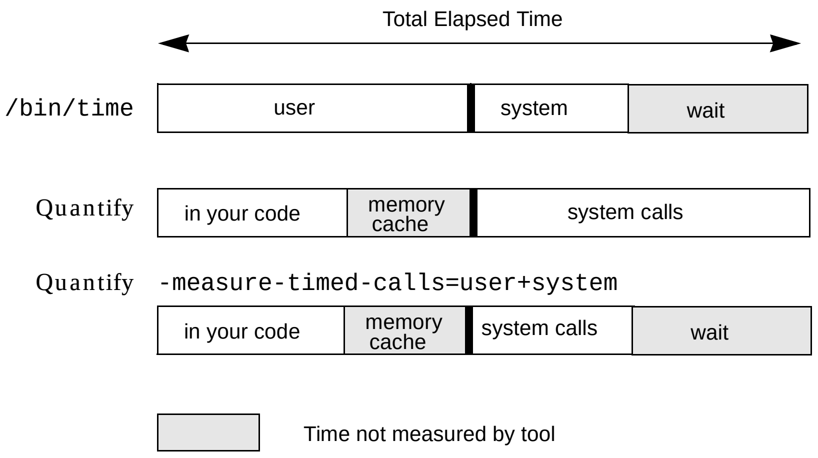
Yes, by using Quantify's API functions. See "Collecting partial data for a run" on page 4-13.

Why are the times reported by /bin/time different from Quantify's predictions?

Quantify and /bin/time measure time in your code and time in system calls differently.

The system utility /bin/time reports times measured *by the kernel* for your process. When the kernel switches to and from user state for your process, it adds the elapsed (wall-clock) time to the user time measurement for your process. In kernel state, when the kernel switches to and from another process, it adds the elapsed time it spent on your process to the sys time measurement for your process. When the kernel waits for a device, this elapsed time is not added to your sys time measurement. These measurements include time spent handling cache misses and memory paging but exclude time the kernel (and your process) waited for response from a device or another process.

In contrast, Quantify counts instructions using a technique that excludes cache miss and paging effects. By default, Quantify also measures elapsed time for operating-system calls which, from your process's point of view, includes any time required for the kernel to wait on devices or for other processes to run.



Quantify underestimates the time `/bin/time` reports in user code, since it does not measure memory effects, and predicts longer system times than `/bin/time` because its measurements include wait times that `/bin/time` excludes, but that represent a true cost to your program.

You can tell Quantify to measure system-call time using the same technique as `/bin/time` by using the option `-measure-timed-calls`. See "Timing system calls" on page 4-2.

Why do I get two different times for the same program when nothing has changed in the program?

Because Quantify measures the elapsed time required by the operating system to perform an operation such as a call to `write`, the measured time varies from one run to another. In particular, the measured time depends on the status of other processes as well (your process might be swapped out) and the cost to perform the operation on a particular device. Since Quantify counts instructions, the `Time` in your code will be identical between runs.

Why does Quantify sometimes report excessive system-call times compared with `/bin/time`?

Quantify measures system-call times by calling either the `gettimeofday` or the `getrusage` functions. If the operating system performs a context switch during one of these additional function calls, the measured time of the operating-system call will appear longer than it actually was. Quantify cannot adjust for this effect because it cannot detect that it has occurred.

This typically happens because your program is making a very large number of very fast system calls in a short period of time, for example, calling the function `gettimeofday` in a loop. This effect is negligible in normal programs. You should consider avoiding recording system-call times during these periods. You can also avoid recording the specific system calls with high-call counts that occur during this period.

See “How Quantify times system calls” on page 3-13, and “Variations in system-call timing” on page 3-14.

Does Quantify save data even if my program encounters a segmentation fault?

Yes. Quantify installs a default handler that saves your data in the event of a segmentation violation. Quantify installs its handler before the code begins execution. If you install your own handler, Quantify calls it rather than the default handler. If you want to save data in that case, your handler should call the API functions `quantify_save_data` or `quantify_save_data_to_file`.

How are the options `-run-at-save` and `-run-at-exit` different? What do they mean?

The `-run-at-save` and `-run-at-exit` options provide a mechanism for you to invoke arbitrary commands or scripts at different times during your program execution. The option `-run-at-exit` allows you to invoke your command *once* at program exit time. The option `-run-at-save` is invoked whenever Quantify saves data.

Questions about data analysis

What are the unknown_static_function function names that appear in Quantify's output?

When Quantify finds a static-function definition without a corresponding name in an object module, it names the function unknown_static_function. It then appends the object filename and function offset to distinguish the function from any other unknown static functions in the same or other object modules. See "How Quantify names functions" on page 3-3.

HPUX

What are the uwss_NNNN and uwse_NNNN function names that appear in Quantify's output?

In shared libraries, the linker creates stub functions for various purposes but does not give these functions names. Quantify, however, requires names, so it gives these functions names such as uwss_NNNN. These names do not appear in gprof.

Why are the line numbers listed by Quantify in the Annotated Source window occasionally off by a line or two?

Different compilers build their debugging information in different ways. Some C++ compilers tend to put incorrect line numbers in the debugging information in the code. Quantify uses whatever debugging information is provided to indicate line numbers. You can sometimes see similar behavior in debuggers.

Questions about performance

What are the disk requirements for using Quantify?

Quantify makes instrumented copies of your object files before passing them to the linker. As a time-saving measure, Quantify caches these instrumented versions of your libraries and object files. See "Managing cached object files" on page 5-10.

The amount of disk space consumed by the cache files depends on the size of the object files and libraries your program requires. Quantify typically increases the size of these files by 50 percent.

If these cached files take too much disk space, you can remove them at any time. They will be rebuilt as needed.

What are the swap requirements for Quantify?

At build-time, Quantify requires swap space approximately equal to two to five times the size of your largest uncached object file or library.

At run time, your Quantify'd program requires approximately 1.5 times the swap space required by your non-Quantify'd program. This is due to the increased size of the instrumented code and the presence of dynamically allocated data accumulators and other data structures.

What is the run-time performance impact of Quantify?

Your Quantify'd program typically runs two to five times slower than your non-Quantify'd program. How much slower depends primarily on the number of basic blocks executed by your program, and on the collection granularity of the code. The fastest Quantify'd code is optimized code or code that is instrumented with function granularity. Code instrumented at basic-block or line granularity is significantly slower. See "Changing the granularity of collected data" on page 4-11.

Run-time speed also depends on how large your program's virtual memory space is relative to your machine's physical memory. If your program is using a large amount of virtual memory relative to the amount of real memory, the increase in the memory requirements can cause your program to *thrash*, that is, to spend nearly all its time paging to disk instead of running.

If your Quantify'd program is more than five times slower, your program is probably thrashing. Use the `vmstat` command to see how much time your program is spending in `kernel` vs. user mode. User mode time should typically be greater than 80 percent.

If user mode drops to less than 50 percent, run your program on a machine with more real memory, try to increase the locality of your program’s memory references to reduce paging, or increase the real memory of your machine.

Quantify Quick Reference

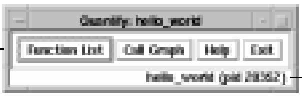
Using Quantify

The CONTROL PANEL appears by default when running a Quantify'd program.
You can also invoke `qv` on a saved `.qv` file:

```
%  qv a.out.23.0.qv
```

Control Panel: Open data analysis windows

Click to open data analysis windows



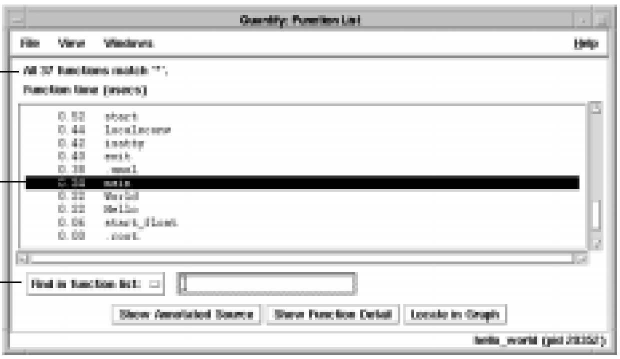
Program and process ID

Function List window: Sort functions to find bottlenecks

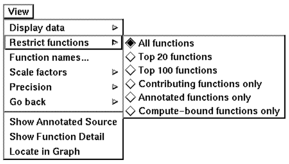
Description of current function list

Click a function to select it

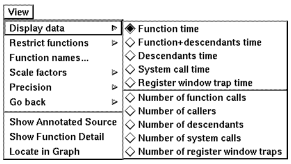
Find a function, or filter by expression



Restrict functions



Sort by different data



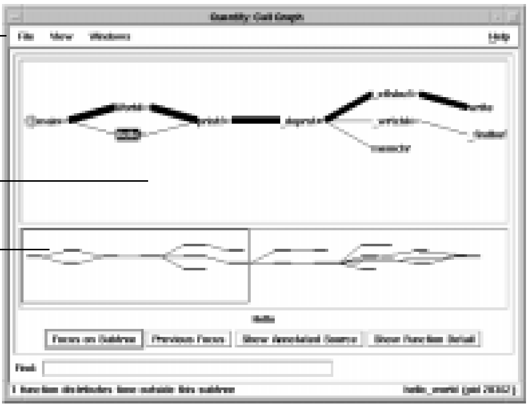
Call Graph window: Understand your application's calling structure

By default, Quantify expands the top 20 descendents of `.root`.

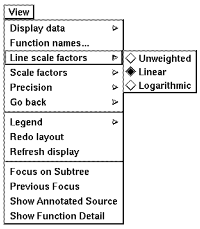
Save call graph as PostScript file

Click and drag anywhere in the call graph to move to a new location

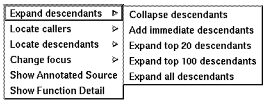
Click and drag Viewport to move to a new location



Change how data is displayed

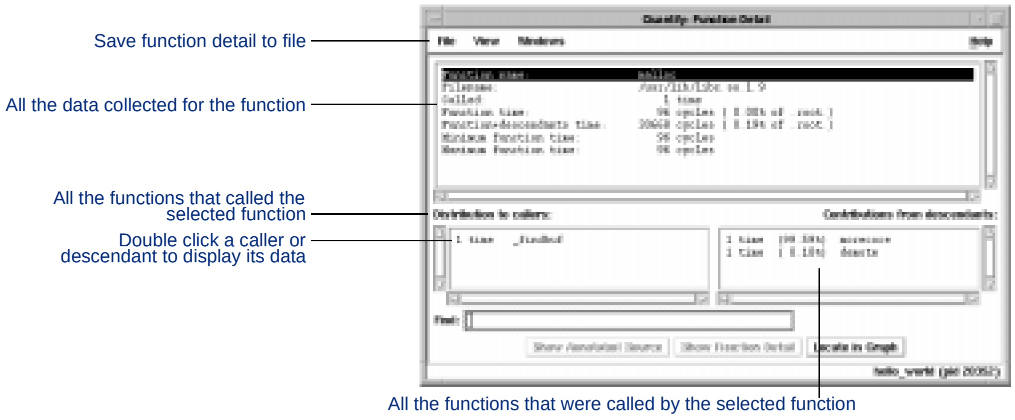


Right-click any function to display the pop-up menu



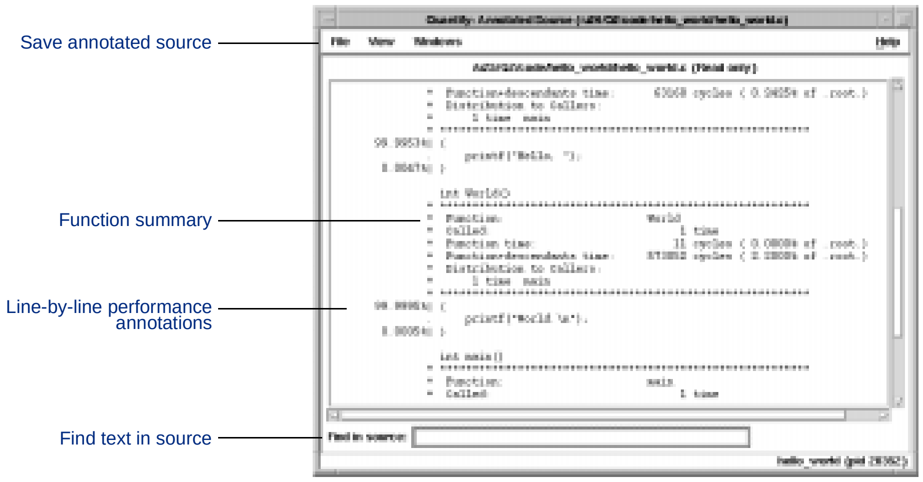
Quantify Quick Reference

Function Detail window: Examine how a function's calling time is distributed


Save function detail to file
All the data collected for the function
All the functions that called the selected function
Double click a caller or descendant to display its data
All the functions that were called by the selected function
Quantify: Function Detail
File
View
Windows
Help
Distribution to callers:
Contributions from descendants:
Find:
Show Annotated Source
Show Function Detail
Locate in Graph

Annotated Source window: View line-by-line performance data

The Annotated Source window is available if you compile your program using the `-g` debugging option.



Conversion characters for filenames

Use these conversion characters when specifying filenames for options.

Character	Converts to	Character	Converts to
%d	Current date (yymmdd)	%t	Current time (hh:mm:ss)
%p	Process id (pid)	%v	Program name
%n	Sequence number, starting at 0. Increments each time the dataset is saved.	%V	Full pathname of the program with "/" replaced by "_"
%T	Thread identifier		

Quantify Quick Reference

API functions

Include `<quantifyhome>/quantify.h` in your code and link with `<quantifyhome>/quantify_stubs.a`

Commonly used functions	Description
`quantify_help (void)`	Prints description of Quantify API functions
`quantify_is_running  (void)`	Returns `true` if the executable is Quantify'd
`quantify_print_recording_state (void)`	Prints the recording state of the process
`quantify_save_data (void)`	Saves data from the start of the program or since last call to `quantify_clear_data`
`quantify_save_data_to_file (char * filename)`	Saves data to a file you specify
`quantify_add_annotation (char * annotation)`	Adds the specified string to the next saved dataset
`quantify_clear_data (void)`	Clears the performance data recorded to this point
`quantify_<action>†_recording_data (void)`	Starts and stops recording of all data
`quantify_<action>†_recording_dynamic_library_data (void)`	Starts and stops recording dynamic library data
`quantify_<action>†_recording_register_window_traps (void)`	Starts and stops recording register-window trap data
`quantify_<action>†_recording_system_call (char *system_call_string)`	Starts and stops recording specific system-call data
`quantify_<action>†_recording_system_calls (void)`	Starts and stops recording of all system-call data

† `<action>` is one of: `start`, `stop`, `is`. For example: `quantify_stop_recording_system_call`

Build-time options

Specify build-time options on the link line to build Quantify'd programs:

```
 % quantify -cache-dir=$HOME/cache -always-use-cache-dir cc ...
```

Commonly used build-time options	Default
`-always-use-cache-dir` Specifies whether Quantify'd files are written to the global cache directory	`no`
`-cache-dir` Specifies the global cache directory	`<quantifyhome>/cache`
`-collection-granularity` Specifies the level of collection granularity	`line`
`-collector` Specifies the collect program to handle static constructors in C++ code	not set
`-ignore-runtime-environment` Prevents the run-time Quantify environment from overriding option values used in building the program	`no`
`-linker` Specifies an alternative linker to use instead of the system linker	system-dependent
`-use-machine` Specifies the build-time analysis of instruction times according to a particular machine	system-dependent

Quantify Quick Reference

qv run-time options

To run qv, specify the option and the saved .qv file: `% qv -write-summary-file a.out.23.qv`

qv options	Default
`-add-annotation` Specifies a string to add to the binary file	not set
`-print-annotations` Writes the annotations to `stdout`	`no`
`-windows` Controls whether Quantify runs with the graphical interface	`yes`
`-write-export-file` Writes the recorded data in the dataset to a file in export format	not set
`-write-summary-file` Writes the program summary for the dataset to a file	not set

Run-time options

Specify run-time options using the environment variable `QUANTIFYOPTIONS`:

```
 % setenv QUANTIFYOPTIONS "-windows=no"; a.out
```

Commonly used run-time options	Default
`-avoid-recording-system-calls` Avoids recording specified system calls	system-dependent
`-measure-timed-calls` Specifies measurement for timing system calls	`elapsed-time`
`-record-child-process-data` Records data for child processes created by `fork` and `vfork`	`no`
`-record-system-calls` Records system calls	`yes`
`-report-excluded-time` Reports time that was excluded from the dataset	`0.5`
`-run-at-exit` Specifies a shell script to run when the program exits	not set
`-run-at-save` Specifies a shell script to run each time the program saves counts	not set
`-save-data-on-signals` Saves data on fatal signals	`yes`
`-save-thread-data` Saves composite or per-stack thread data	`composite`
`-write-export-file` Writes the dataset to an export file as ASCII text	none
`-write-summary-file` Writes the program summary for the dataset to a file	`/dev/tty`
`-windows` Specifies whether Quantify runs with the graphical interface	`yes`

Index

Symbols

. in annotated source 1-21
# in annotated source 1-21
$HOME/.qvrc A-4
%b 5-6, A-3
%d A-3
%n A-3, A-4
%p A-3
%T A-3
%t A-3
%V A-3
%v 5-6, A-3
%x 5-6, A-3
*
  in annotated source 1-21
  in call graph 1-14
  in filenames A-2
+ in annotated source 1-21, 3-9
??? in function name 1-16
_p___map_dld function 4-9
| in annotated source 1-21

Numerics

100+% function+descendants time 1-14
64-bit precision 3-4

A

-add-annotation 4-16, B-7
-always-use-cache-dir B-4
Annotated Source window
  and -g option 1-20
  C++ templates 3-11
  changing annotations 1-21
  compiler differences 3-9
  function summaries 1-21
  multiple basic blocks 3-9
  single basic block 1-21
Annotated Source window symbols 1-21
annotations
  adding to datasets 4-16
  and -user-path B-17
  Annotated Source window symbols 1-21
  for if-then expressions 3-10
  for switch expressions 3-10
  printing 4-16
  removing from call graph 1-14
API functions
  annotating data B-14
  calling from a debugger A-9
  calling from a program A-9
  clearing collected data B-14
  printing recording state B-12
  recording data B-10
  recording dynamic-library data B-11
  recording partial data 4-13
  recording register-window traps B-11
  recording system calls B-10
  run-time B-3
  stubs library A-10
  using with options 4-15
API functions (by name)
  pure_logfile_printf B-16
  pure_printf B-16
  quantify_add_annotation 4-16, B-14
  quantify_clear_data 4-15, 4-19, 5-4, B-14
  quantify_disable_recording_data B-11
  quantify_help B-12
  quantify_is_recording_data B-10
  quantify_is_recording_dynamic_library_data B-11
  quantify_is_recording_register_window_traps B-11
  quantify_is_recording_system_calls B-10

quantify_is_running B-12
quantify_print_recording_state B-12
quantify_save_data 4-15, 4-16, 5-6
quantify_save_data_to_file 4-16, 5-4, B-14
quantify_start_recording_data 4-15
quantify_start_recording_dynamic_library_data B-11
quantify_start_recording_register_window_traps B-11
quantify_start_recording_system_calls B-10
quantify_stop_recording_data B-10
quantify_stop_recording_dynamic_library_data B-11
quantify_stop_recording_register_window_traps B-11
quantify_stop_recording_system_calls 4-15
-api-handler-signals 4-18, B-15
appearance, customizing Quantify's 1-23
appending function detail to file 1-19
-append-logfile B-16
architecture
HPPA 4-10
RISC 3-4, 3-8
SPARC 3-15
superscalar 3-8
at_exit 1-10
automating data analysis 5-5
-auto-mount-prefix B-17
-avoid-recording-system-calls 4-5, 4-6, B-8

B

background color, changing 1-23
baseline data 2-9
basic blocks
accumulators 3-4
collection granularity 4-11
definition 3-4
identification of 3-4
multiple 1-21
binary dataset
conversion character (%b) A-3
file directive (%b) 5-6
/bin/time 1-2, 4-4, C-3, C-5
blocks, basic 3-4
bottlenecks
environmental factors 2-17
in library or system-call requests 2-16, 2-17
premature computation 2-16
types of 2-15
branch
instruction 3-4
prediction 3-8
bucket 2-2
building a Quantify'd program 1-3
build-time options B-1, B-4

C

C++ applications
annotated source for templates 3-11
argument lists 1-7
classes 1-7, 1-16
finding functions in 1-7
function name demangling 1-16, 5-1
function names 3-3
static initialization 1-10
cache
directory B-4
memory effects 3-20, C-3
memory miss C-3
options B-4
removing files 5-10
-cache-dir B-4
Call Graph window
changing focus 1-13
controlling complexity 1-11
displaying data for functions 1-14
displaying function data 1-14
expanding and collapsing subtrees 1-11
finding functions 1-12
layout 1-10
multi-threaded applications 1-10
redoing layout 1-11
saving 1-16
saving PostScript version 1-16
single-threaded applications 1-10
callers, distribution 1-18
calling functions from 4-15

calls, *see* system calls
changing annotations 1-21
characters
  conversion A-3
  See also Annotated Source window symbols
child processes
  collecting data about 4-18
  options B-13
  using execve 4-19
  using fork 4-19
  using vfork 4-19
client-server applications 4-5
clock rate 3-1
code, *see* Annotated Source window
collapsing subtrees 1-11
collection API functions B-12
collection granularity
  controlling 4-12
  default 4-11
  levels 4-11
  option B-4
-collection-granularity 4-11, 4-12, B-4
-collector B-4
colors, changing 1-23
comparing program runs 5-7
compilers
  annotation differences 3-9, C-6
  -g switch 3-5
composite data 4-20
computation, needless 2-15
compute-bound functions 1-4, 1-6, 1-7, 2-5
context switch 3-14
contributions from descendants 1-18
conversion characters A-3
-copy-fd-output-to-logfile 5-4, B-16
counts
  clearing B-14
  expected cycle 3-4
  in optimized code 3-8
  register-window traps 3-15
  relation to original program 3-4
  system calls 3-13
cron job, for removing files 5-11
crt0.o 4-8
current focus 1-12
curved lines in call graph 1-10
customizing Quantify's interface 1-23
cycles
  and register windows 3-15
  scale factor 1-19

D

daemons, collecting data from 4-18
data
  automating analysis 5-5
  baseline 2-9
  comparing export files 5-7
  composite 4-20
  exporting 1-22, 5-1
  recording 4-2
  saving incrementally 5-4
  saving on exit 1-22
  stack 4-20
  start/stop recording B-10
  transmission to qv 1-3
data collection
  API functions B-8
  granularity 4-11
  multi-threaded programs 4-22
  on threaded programs 4-20
  options B-8
  partial 4-13, C-3
  speeding up 4-12
  starting 4-2
data recording, starting 4-2
data-analysis questions C-6
database applications 4-5
datasets
  adding annotations to 4-16
  annotating 4-16
  containing partial data 4-14
dead code 2-15
debugger 4-15
debugging information C-6
deeply nested function calls 3-15
default machine 3-20
demangling function names 1-16
descendants
  called from outside subtrees 1-13
  contributions from 1-18
  distribution of time 1-18
  expanding and collapsing in call graph 1-11
detail of collected data 4-11

directives
  current date A-3
  current time A-3
  export data A-3
  full pathname A-3
  process ID A-3
  program name A-3
  program summary A-3
  sequence number A-3
  thread identifier A-3
directories
  cache B-4
  search list B-17
  searched for sources 1-9
disk
  requirements C-6
  swap space C-2
distribution of time
  descendants time 3-2
  to callers 1-18
dlclose 4-9
dld.sl 4-9
dlopen 4-7, C-2
dynamic branch prediction 3-8
dynamic libraries
  timing 1-4, 4-8, B-11
  unmapping 4-9
dynamic linker, ld.so 4-8, 4-9
dynamic linking
  timing 4-8

E

elapsed time 3-13, 4-4, C-3
empty code blocks 3-10
environment variables
  PUREOPTIONS A-6
  QUANTIFYOPTIONS A-6
  specifying options in A-6
excluded time 4-6
execve 4-19
exiting and saving data 1-22
expanding subtrees 1-11
expansion characters for filenames A-3
exporting data 1-22
  default filename 5-1, 5-7
  file format 5-1
expression, filter 1-8
expressions
  if-then 3-10
  switch 3-10

F

-fds B-17
file descriptors B-17
-filename-prefix 4-20, 5-5, A-4, B-13
filenames
  conversion characters A-3
  default A-4
  unqualified A-3
files
  appending function detail data to 1-19
  cache C-7
  created by Quantify C-1
  export data 5-2
  libquantify_stubs.a A-10
  log 5-4
  .machine 4-10, B-6
  program summary A-3
  .pure C-1
  quantify_threads.h 4-20
  quantify.h A-9
  .qvrc 1-23
  removing old 5-10
  signal.h 4-18
  syscall.h 4-5, B-8
  /tmp A-3
  .Xdefaults 1-23
filtering function names 1-8
finding functions in function list 1-7
fini code 4-9
focus
  changing in call graph 1-13
  current 1-12
fonts, changing 1-23
-forbidden-directories B-5
-force-rebuild 4-12, B-5
foreground color changing 1-23
fork 4-18, 4-19
free 1-13
Function Detail window 1-17
  appending data to file 1-19
  saving data 1-19
Function List window
  filtering function names 1-7
  finding functions 1-7
  finding top contributors 1-6

header 1-8
restricting functions 1-7
function names
demangling 1-16, 5-1
filtering 1-7, 1-8
making unique C-6
truncating 1-16
unique 3-3
function time
minimum and maximum 1-17, 3-2
sorting 1-6
function(s)
changing focus in call graph 1-13
collection granularity 4-11, 4-12
compute-bound 1-7
displaying data in call graph 1-14
filtering function list 1-8
finding in call graph 1-12
finding in function list 1-7
finding top contributors 1-6
multiple callers of 1-13
recursive 3-3, 3-16
recursive in call graph 1-10
restricting display of 1-7
sorting 1-6
static 3-3
summaries 1-21
See also API functions
function+descendants
100+% time 1-14
time 1-17, 1-21

G

-g compiler switch 3-5
-g++ B-5
getrusage system cal 3-14
gettimeofday system call 3-14
global options A-6
granularity of collected data 4-11

H

handlers, signal 4-18
-handle-signals 4-17, B-15
hashtable example 2-1
header files 4-18
help
in debugger A-9
technical xi
using online Help ix
high load average 2-17
HPPA 4-10
HP-UX 3-3, 4-9

I

-i option for qxdiff script 2-14
if-then expressions,
annotations 3-10
-ignore-runtime-environment A-8,
B-5
-ignore-signals 4-17, B-15
incomplete subtrees 1-13
incremental datasets 5-6
incrementing %n A-3, A-4
inefficient computation 2-16
init code 4-9
inspecting incremental datasets 5-6
installing Quantify x
instruction
branch 3-4
load and store 3-4
scheduling 3-8
stalls 3-4
instrumentation
analysis 3-4
first time C-1
granularity 4-11
interface, customizing 1-23

K

keywords in export files 5-2
kill 4-18

L

ld.so 4-8, 4-9
leaf function 1-10
library
dynamic, time loading 1-4
shared C-2
stubs A-10
lightweight processes (LWPs) 4-21
line scale factors 1-15
changing 1-15
line(s)
changing scale factors 1-15
collection granularity 4-12
curved in call graph 1-10
linear 1-15

logarithmic scaling 1-15
removing number information 4-12
unweighted 1-15
linear scaling 1-15
link, disk requirements C-2, C-6
-linker B-5
linker options B-5
load and store instructions 3-4
load fluctuations 3-13
log files 5-4
logarithmic scaling, lines 1-15
-logfile 5-3, B-16
long-running programs 4-18

M

.machine file 3-20, 4-10, B-6
machine(s)
counting cycles 1-2
default 3-20
instructions 3-4
running of different 3-20
SMP 4-21
makefile example A-10
malloc 1-13, C-2
maximum function time 1-17
-max-threads 4-21, B-12
measured time 3-13
-measure-timed-calls 4-4, 4-8, B-8, C-4
measuring timed calls 4-4
Mega-Hertz (MHz) 3-1
memory
cache effects 3-20, C-3
fetching 3-21
paging effects 3-20, C-3
memory access options B-12, B-13
menu, pop-up 1-11
minimum function time 1-17
miss, cache memory C-3
MT-safe (multiple-thread safe) 4-23
multiple basic blocks 1-21
multiple callers of functions 1-13
multi-threaded programs 1-10, 4-22
mutual exclusion (mutex) locks 4-21

N

n, incrementing %n A-3, A-4
names
shortening function 1-16
network delay 2-17
-never-record-system-calls 4-22, B-8
NFS 3-13
non-exiting programs 4-17
numbers for lines, removing 4-12
numeric annotations 1-21

O

Object Code Insertion (OCI) 1-2
object files
caching and removing 5-10
Quantify'd version C-1
operating system
affect on time 3-1
optimized code 3-8
options
build-time B-4
caching B-4
child process B-13
in environment variables A-6
in makefiles A-10
invoking shell scripts 5-5
link line A-7
linker B-5
memory access B-12, B-13
processing A-6
qv B-1, B-7
recording register-window traps 3-15
running scripts at save 5-5
run-time B-2
site-wide A-6
syntax A-2
threads 4-20, B-12
timing system calls 4-2, 4-4
types A-5
using with API functions 4-15
windows 5-2, 5-5, B-7
options (by name)
-add-annotation B-7
-always-use-cache-dir B-4
-api-handler-signals 4-18, B-15
-append-logfile B-16
-auto-mount-prefix B-17
-avoid-recording-system-calls 4-5, 4-6, B-8
-cache-dir B-4

-collection-granularity 4-11, 4-12, B-4
-collector B-4
-copy-fd-output-to-logfile B-16
-fds B-17
-filename-prefix 4-20, 5-5, A-4, B-13
-forbidden-directories B-5
-force-rebuild 4-12, B-5
-g++ B-5
-handle-signals 4-17, B-15
-ignore-runtime-environment B-5
-ignore-signals 4-17, B-15
-linker B-5
-logfile 5-3, B-16
-max-threads 4-21, B-12
-measure-timed-calls 4-4, 4-8, B-8
-never-record-system-calls 4-22, B-8
-output-limit B-16
-print-annotations 4-16, B-7
-print-home-dir B-6
-program-name B-17
-record-child-process-data 4-19, B-13
-record-data 4-2, 4-15, B-9
-record-dynamic-library-data 4-8, B-9
-record-register-window-traps 4-10, B-9
-record-system-calls 4-2, 4-6, B-9
-report-excluded-time 4-6, B-9
-run-at-exit B-15, C-5
-run-at-save B-15, C-5
-save-data-on-signals 4-17, B-15
-save-thread-data 4-20, B-12
-threads B-12
-thread-stack-change 4-21, B-12
-use-machine 3-20, B-6
-user-path B-17
-version B-6
-windows 5-5, B-7, B-15
-write-export-file 5-1, B-7, B-13
-write-summary-file 5-3, 5-5, B-7, B-13
output, re-routing to log file 5-3
-output-limit B-16
overflow register windows 3-15
overhead 1-4, 3-4, C-7
dynamic library 4-7
Quantify 1-1

P

paging effects 2-17, 3-20, 3-21, C-3
parallel stack of function calls 3-2, 4-20
partial data collection 4-13
performance
data transmission 1-3
impact C-6
load fluctuations 3-13
pessimistic timing 3-8
PLT, procedure linkage table 4-8
pop-up menu 1-11
PostScript, saving call graph 1-16
precision, 64-bit 3-4
-print-annotations 4-16, B-7
printf escape syntax 5-4
-print-home-dir B-6
printing PostScript call graph 1-16
procedure linkage table (PLT) 4-8
process
identifier B-14
swapping C-4
processes, child 4-18
program summary
excluded time 4-6
interpreting 1-4
program(s)
comparing runs 5-7
long-running 4-18
output 1-3
-program-name B-17
.ps file extension A-4
.pure file C-1
pure_logfile_printf 5-4, B-16
pure_printf 5-4, B-16
pure_sigtramp 1-10, 3-2
PureCoverage C-1
PUREOPTIONS A-6
Purify C-1

Q

.qfd file extension A-4
.ql file extension A-4
.qs file extension A-4
Quantify
accuracy 3-20
disk space requirements C-6, C-7
how it works 1-2, 3-2, 3-3
overhead 1-1, 1-4, 3-4, 3-14
variations in timing 3-14

quantify
  _add_annotation 4-16, B-14
  _clear_data 4-15, 4-19, B-14
  _disable_recording_data B-11
  _help B-12
  _is_recording_data B-10
  _is_recording_dynamic_library_data B-11
  _is_recording_register_window_traps B-11
  _is_recording_system_calls B-10
  _is_running B-12
  _print_recording_state B-12
  _save_data 4-15, 4-16, 5-4, 5-6, B-14
  _save_data_to_file 4-16, 5-4, B-14
  _start_recording_data 4-2, 4-15, B-10
  _start_recording_dynamic_library_data B-11
  _start_recording_system_call 4-6
  _start_recording_system_calls B-10
  _stop_recording_data 4-15, B-10
  _stop_recording_dynamic_library_data B-11
  _stop_recording_register_window_traps B-11
  _stop_recording_system_call 4-6
  _stop_recording_system_calls 4-13, 4-15, B-10
quantify_threads.h 4-20
Quantify'd programs
  building C-1
  inherent differences in 3-21
  run-time memory usage C-7
  run-time performance differences C-7
QUANTIFYOPTIONS A-6
questions
  about data analysis C-6
  about data collection C-3
  about instrumentation C-1
  about performance C-6
  general C-1
qv 1-3, 5-2
  options B-1
  running 5-2
.qv file extension A-4
.qx file extension 5-1, A-4
qxchange script 5-5
qxdiff script 2-9, 4-3, 5-7
  example of using 2-14
  -i option 2-14

R

Rational Software World Wide Web xi
-record-child-process-data 4-19, B-13
-record-data 4-2, 4-15, B-9
-record-dynamic-library-data 4-8, B-9
recording data 4-2
recording dynamic library data
  API functions B-11
-record-register-window-traps 4-10, B-9
-record-system-calls 4-2, 4-6, B-9
recursive functions 1-10, 3-3, 3-16
  and register window traps 3-15
redoing layout of call graph 1-11
register windows
  API functions B-10
  options 4-10, B-9
  recording B-11
  trap data 3-11
regular expression 1-8
release notes, locating x
removing
  annotations in call graph 1-14
  old files 5-10
-report-excluded-time 4-6, B-9
reports
  program summary 1-4
  re-routing to log file 5-3
  sending by e-mail 5-6
RISC architecture 3-4, 3-8
.root.
  accumulator 1-9, 1-10
  as accumulator 3-2
  as focus 1-9
  as function 3-2
  in call graph 1-10
-run-at-exit 5-5, B-15, C-5
-run-at-save 5-5, B-15, C-5
runs, comparing program 5-7
run-time
  API functions B-3
  options B-2
  overhead 3-4
  performance C-7

S

-save-data-on-signals 4-17, B-15
-save-thread-data 4-20, B-12
saving
  call graph 1-16
  data B-13
  data on exit 1-22
  incremental data 5-4
scale factors 1-19
  >% of focus 1-13
  line 1-15
scripts
  qxchange 5-5
  qxdiff 2-9, 4-3, 5-7
  remove_old_files 5-10
select() 3-13
sending data to qv 1-3
sequence number %n A-3
shadow stack 3-2, 4-20
shared libraries C-2
  operations 4-8
  timing 4-7
shell scripts 5-5
shl_load 4-7, C-2
shl_load function 4-9
shortening function names 1-16
signal handlers 3-2
  default set 4-17
  installing 4-18
  See also signals
signal.h 4-18
signals
  ignoring 4-17
  not handled 4-17
  reported in call graph 1-10
  saving data on 4-17
  See also signal handlers
SIGUSR1, SIGUSR2 4-18, B-15
single-threaded applications 1-10
site-wide options A-6
slow software, causes of 2-15
Solaris 2 3-15, 4-9, 4-21
source, *see* Annotated Source window
SPARC architecture 3-15
stacks, and threads 4-20
starting recording data 4-2
static functions 3-3, 3-12
stored clock rate 3-20
strings, annotation 4-16
strip -l 4-12
stubs library A-10
subtrees
  changing focus 1-13
  expanding and collapsing 1-11
  incomplete 1-13
summary for functions 1-21
SunOS 4.1 4-8
superscalar architectures 3-8
support, technical xi
swap space 3-20, C-2, C-7
swapping effects 2-17
switch expressions, annotations 3-10
symbols, *see*
  Annotated Source window symbols
  conversion characters
symmetric multi-processors (SMP) 4-21
syntax, options A-2
SYS_exit 4-5
SYS_select 4-5
syscall.h 4-5, B-8, B-10
system calls
  and database clients 4-5
  avoiding timing 4-5
  ignoring with -i option 2-14
  start/stop recording B-10
  SYS_exit 4-5
  SYS_select 4-5
  timing 1-2, 3-13, 4-2
  timing in X Windows programs 4-5

T

technical support xi
templates, C++ 3-11
thrashing C-7
-threads B-12
threads
  collecting data in 4-20
  identifier (%T) A-3
  lightweight processes, and 4-21
  options 4-20, B-12
  reported in call graph 1-10, 4-22
  single-threaded applications 1-10
  stacks, and 4-20
-thread-stack-change 4-21, B-12

time(s)
  compute-bound 1-4
  current time directive A-3
  elapsed (wall-clock) 3-1, 3-13
  exceeding 100% 1-13, 1-14
  excluding overhead 3-4
  function+descendants 1-17
  in code 1-4, 2-4
  in system calls 2-4
  kernel 3-13, C-7
  loading dynamic libraries 1-4
  memory swapping 3-21
  network delays 3-13
  optimistic 3-15
  pessimistic 3-8, 4-21
  predicted 3-15
  system calls 3-13
  to collect the data 1-4
  unexpected amounts 1-13
timing
  recursive functions 3-3
  register-window traps 4-10
  system calls 4-2
  variations in 3-20
triangle icons 1-10
truncating function names 1-16

U

unique function names 3-3, C-6
unknown_static_function 3-3, C-6
unqualified filename A-3
unweighted lines 1-15
-use-machine 3-20, B-6
user+system measurements 4-4
-user-path B-17
uwse_NNNN C-6
uwss_NNNN C-6

V

variations in timing 3-20
verifying improvements 5-7
-version B-6
vfork 4-18, 4-19
viewport 1-9

W

wall-clock time, *see* elapsed time
wildcards
  in filenames A-2
  in filter expressions 1-8
-windows 5-5, B-7, B-15
windows, data analysis 1-5
World Wide Web site, Rational Software xi
-write-export-file 5-1, B-7, B-13
-write-summary-file 5-3, 5-5, B-7, B-13

X

X resources, customizing Quantify 1-23
X Windows 3-13, 5-2
  applications 4-5
  avoiding call timing 4-6
  options 5-3
.Xdefaults 1-23