

C/C++ Productivity Tools for OS/390



Performance Analyzer

Release 1.0

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 87.

First Edition (September 1999)

This edition applies to C/C++ Productivity Tools for OS/390 Release 1.0, program number 5655-B85 and to all subsequent versions, releases, and modifications until otherwise indicated in new editions. Consult the latest edition of the applicable system bibliography for current information on these products.

Order publications through your IBM representative or through the IBM branch office serving your locality.

© Copyright International Business Machines Corporation 1999. All rights reserved.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this book.	v	Setting environment variables for Performance Analyzer	28
Who should read this book	v	Setting run-time option PROFILE for Performance Analyzer	29
Conventions used in this book	v	Creating a trace file	30
Related information	v		
How to send your comments.	vi		
 Chapter 1. Introducing the Performance Analyzer	1	 Chapter 7. Starting and exiting the Performance Analyzer	33
The Performance Analyzer for OS/390	1	Starting the Performance Analyzer	33
Performance Analyzer product files	2	Starting the Performance Analyzer from a command line.	33
 Chapter 2. Diagrams for analyzing a trace file	5	Exiting the Performance Analyzer	34
Diagrams for analyzing a trace file	5	 Chapter 8. Controlling what data is collected in the trace file	35
Call Nesting diagram	6	Collecting call frequency data	35
Dynamic Call Graph diagram	7	Tracing a specific DLL	36
Execution Density diagram	8	Tracing a Webserver application.	36
Statistics diagram	10	Specifying trace file name	37
Time Line diagram	12		
 Chapter 3. Trace file generation	13	 Chapter 9. Viewing your trace file in a diagram	39
Call frequency counting	13	Downloading the trace file from the host	39
Time stamps	14	Starting the Performance Analyzer to analyze a trace file	40
Trace events	14	Opening a trace file in a diagram	40
Function trace.	15		
Overhead time	16	 Chapter 10. Navigating the trace file view	43
Multiple process support	17	Correlating events between diagrams	43
 Chapter 4. Trace file viewing and analysis	19	Enlarging or reducing a diagram	44
Function groups	19	Seeing details by combining the zoom and correlation features	45
Pattern recognition	20	Viewing a specific time or range of time	45
Diagram filters	20		
Correlation.	22	 Chapter 11. Searching for trace data in a diagram	47
 Chapter 5. Tips for Using the Performance Analyzer to understand your program	25	Finding a specific annotation.	47
Use a combination of diagrams to understand your program.	25	Finding a specific function call or return	48
Annotate your trace file	25	Finding trace data for a specific function	49
 Chapter 6. Preparing your program for analysis	27	Finding trace data for a specific class	50
Compiling your program	27	Finding trace data for a specific executable	51
		 Chapter 12. Controlling what data is shown in the diagrams	53

Filtering events by component type	53	Tracing programs that have interlanguage	
Filtering events by function	53	calls	68
Filtering events by thread	54	Run-time option for program tracing	68
Filtering events by group	55	Run-time environment variables for	
Filtering nodes and arcs in the Dynamic		program tracing	72
Call Graph diagram.	57	Troubleshooting Performance Analyzer	
Recognizing call sequence patterns.	58	problems	74
Viewing class activity	59	Performance Analyzer error messages on	
		the host	76
Chapter 13. Analyzing Your Trace File	61	Sample Unix system service commands for	
Adding, changing, or deleting annotations	61	creating trace files	80
Determining the elapsed time between two		Sample TSO commands for creating trace	
events	61	files	82
Selecting functions to inline	62	Sample JCL for creating trace files	83
Viewing thread interactions in a		Sample trace file names from tracing a	
multithreaded program	62	multiprocess program	85
Chapter 14. Reference	65	Notices	87
Limitations when analyzing trace data	65	Trademarks and service marks	89
Limitations when creating a trace	65		
Performance Analyzer invocation			
parameters.	67		

About this book

Performance Analyzer introduces you to the Performance Analyzer and provides information about how to understand and improve the performance an OS/390 C and C++ application.

Who should read this book

Performance Analyzer is intended for application programmers who want to understand aspects of the C and C++ program on OS/390 that would otherwise be difficult to visualize and want a workstation performance analyzer to enhance their existing familiar host environment. For these users, this document introduces the Performance Analyzer and shows how to use it.

Conventions used in this book

The following conventions distinguish different text styles within this book:

plain	Window titles, folder names, icon names, and method names.
monospace	Programming examples, user input at the command line prompt or into an entry field, directory paths.
bold	Menu choices and menu names, labels for push buttons, check boxes, radio buttons, group-box controls, drop-down list boxes, combination-boxes, notebook tabs, and entry fields.
<i>italics</i>	Programming keywords and variables, and titles of documents.

Related information

For information on OS/390 C/C++ related features, news and Web sites, add this Web site to your browser's bookmark list:

<http://www.ibm.com/software/ad/c390>

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this book or any other C/C++ Productivity Tools documentation, send your comments by e-mail to torrcf@ca.ibm.com. Be sure to include the name of the book, the document number of the book, the version of C/C++ Productivity Tools, and, if applicable, the specific location of the information on which you are commenting (for example, a page number or a table number).

Chapter 1. Introducing the Performance Analyzer

The Performance Analyzer for OS/390

The Performance Analyzer helps you understand and improve the performance of your C/C++ programs. It traces the execution of a program and creates a trace file which contains data that can be examined in several diagrams at the workstation. Sometimes this tracing is also referred to as *profiling*. With the trace information, you can improve the performance of a program, examine a sequence of calls leading up to an exception, and understand the execution flow when a program runs.

The Performance Analyzer can complement other application development tools by helping you understand aspects of the program that would otherwise be difficult to visualize. For instance, with the Performance Analyzer you can:

- **Time and tune programs**
The Performance Analyzer records the time stamp of each trace event. As a result, the trace file contains a detailed record of when your program called and exited each traced function. The trace data also shows how long each function ran, which helps you identify code that you may want to tune.
- **Diagnose program abends**
The Performance Analyzer provides a complete history of events leading up to the point where a program abends.
- **Trace multithreaded programs**
After tracing a multithreaded program, you can examine the individual threads to identify their function usage.
- **Trace multiple processes**
When your POSIX programs use the fork and spawn functions to create new processes, you can still view the events in the different processes because a separate trace file is created for each process.

The Performance Analyzer performs function tracing on a program. Function tracing records information about each function call and return made during the execution of the program.

Performance Analyzer components

The Performance Analyzer has two components:

- **Host component**
Traces your host program's execution and creates a binary trace file containing the trace data that was collected. You then download this file to the workstation for analysis.

- **Workstation component**

Allows you to analyze the trace file that you have created on the host. You can take advantage of graphical and textual diagrams to assist you with the analysis of the trace data.

RELATED CONCEPTS

“Performance Analyzer product files”

“Function trace” on page 15

“Call frequency counting” on page 13

“Multiple process support” on page 17

RELATED TASKS

“Creating a trace file” on page 30

“Starting the Performance Analyzer to analyze a trace file” on page 40

“Collecting call frequency data” on page 35

“Downloading the trace file from the host” on page 39

RELATED REFERENCES

“Run-time option for program tracing” on page 68

“Run-time environment variables for program tracing” on page 72

“Limitations when creating a trace” on page 65

Performance Analyzer product files

Host Data Sets

The host component runs from a data set, CBC.SCTVMOD, which contains the following members:

- **CEEEVPRF**
Alias for the Performance Analyzer module
- **CTVMSGE**
English messages
- **CTVMSGK**
Kanji messages
- **CTVMSGT**
Message table
- **CTVPFILE**
Performance Analyzer module
- **ICTVMSGT**
Alias for the message table

OS/390 Version 2 Release 4 and subsequent releases ship the dataset. To use the Performance Analyzer product dataset you must purchase and enable the

OS/390 C/C++ Compiler with Debug feature in OS/390. The installation program adds an entry to the IFAPRDxx parmlib member with the feature name "C/C++/DEBUG" to enable this feature in OS/390. You must also apply the latest service to FMIDs, H24P111 and J24P112. See *OS/390 Planning for Installation* and *OS/390 MVS Initialization and Tuning Guide* or contact your system programmer for more information about enabling OS/390 features and applying service to this data set.

To run the Performance Analyzer on the host system, the CBC.SCTVMOD data set must be included in the OS/390 modules' search path. To do this, your system programmer can add it to the Link Pack Area, you can add it to your STEPLIB DD statement in your JCL, or you can use the export command in the OS/390 shell to add it to your STEPLIB before you run your program.

RELATED CONCEPTS

"The Performance Analyzer for OS/390" on page 1

RELATED TASKS

"Creating a trace file" on page 30

"Starting the Performance Analyzer to analyze a trace file" on page 40

Chapter 2. Diagrams for analyzing a trace file

Diagrams for analyzing a trace file

The Performance Analyzer provides several diagrams in which you can view and analyze the data contained in your trace file. Each diagram presents a different view of the trace data to give you an overall idea of how your program performs. The following list contains the names of the diagrams, the icons used to represent them, and a brief description of each; more detailed introductions to the diagrams and their uses are included in the related topics below.



Call Nesting

The Call Nesting diagram shows the trace file as a sequential series of function calls and returns. This diagram helps in diagnosing problems with critical sections, sequencing protocols, thread delays, and crashes.



Dynamic Call Graph

The Dynamic Call Graph diagram is a two-dimensional graphical representation of your program's execution. It shows the relative importance (in terms of execution time) of program components, and the call hierarchy.



Execution Density

The Execution Density diagram shows your program in terms of execution time. It shows trace data chronologically from top to bottom as thin horizontal lines of various colors in columns assigned to each traced function.



Statistics

The Statistics diagram is a textual report of cumulative information about your program's execution. It provides summary and detailed statistics on execution time and event generation for each component type: function, class, and executable.



Time Line

The Time Line diagram shows function calls and returns in chronological order along a vertical line. A function call is represented

by a short horizontal line to the right, and a function return is represented by a short horizontal line to the left. The horizontal lines are connected by vertical lines whose length is proportional to the amount of time that elapsed between the respective events.

RELATED CONCEPTS

“Call Nesting diagram”

“Dynamic Call Graph diagram” on page 7

“Execution Density diagram” on page 8

“Statistics diagram” on page 10 “Time Line diagram” on page 12

RELATED TASKS

“Opening a trace file in a diagram” on page 40

Call Nesting diagram

The Call Nesting diagram shows the trace file as a series of function calls and returns, arranged vertically. Use this diagram to diagnose problems with critical sections, sequencing protocols, program crashes, or thread delays.

Trace Data

The trace data shown by the Call Nesting diagram includes the following elements:

- The functions that were called during program execution
- The order in which the functions were called and in which they returned
- The nesting of function calls (the call stack) at any point during program execution
- The points at which control switched from one thread to another during program execution

Uses

Use the Call Nesting diagram to perform the following tasks:

- Examine the specific elements of trace data listed above.
- See the interactions among the threads.
- Get a better understanding of the program’s flow.
- View and create annotations of trace file events.

RELATED CONCEPTS

“Diagrams for analyzing a trace file” on page 5

RELATED TASKS

“Opening a trace file in a diagram” on page 40

“Filtering events by function” on page 53

“Filtering events by thread” on page 54

“Correlating events between diagrams” on page 43

“Recognizing call sequence patterns” on page 58

“Viewing thread interactions in a multithreaded program” on page 62

Dynamic Call Graph diagram

The Dynamic Call Graph diagram is a two-dimensional graphical representation of your program’s execution. It shows the relative importance in terms of execution time of the different components, and the call hierarchy.

Trace Data

The trace data shown by the Dynamic Call Graph diagram includes the following elements:

- The functions, classes, or executables (components) that ran during program execution
- The calls that were made from one component to another
- The call hierarchy
- The caller, the callee, and the number of times a call was made between each pair of components
- The relative importance of each call between components in terms of the number of calls made between components
- The relative importance of each component in terms of execution time and time on stack

You can show trace data for one type of component at a time:

- When you choose to show information on functions, the following trace data is available for each function that ran:
 - Function name
 - Compile unit
 - Object file (workstation programs) or Compile unit (host programs)
 - Executable name
 - Execution time
 - Number of calls
 - Time on stack

- When you choose to show information on classes (possible only if your trace file contains class information), the following trace data is available for each class whose code was executed:
 - Class name
 - Names of member functions and their associated statistics
 - Execution time
 - Number of calls to the member functions in the class
- When you choose to show information on executables, the following trace data is available for each executable that ran:
 - Executable name
 - Functions in the executable, and their associated statistics
 - Execution time
 - Number of calls

Uses

Use the Dynamic Call Graph diagram to perform the following tasks:

- Examine the specific types of trace data listed above.
- Get an overall view of your program and its flow.
- See the relative importance in terms of execution time of program components.
- See where time is spent in your program.
- See your program's call hierarchy.

RELATED CONCEPTS

“Diagrams for analyzing a trace file” on page 5

RELATED TASKS

“Opening a trace file in a diagram” on page 40

“Filtering events by thread” on page 54

“Filtering nodes and arcs in the Dynamic Call Graph diagram” on page 57

“Enlarging or reducing a diagram” on page 44

Execution Density diagram

The Execution Density diagram shows the trace data chronologically from top to bottom, as follows:

- Each vertical column represents a function (or collapsed group of functions if you have included group information in the diagram).
- Each horizontal line represents a time slice.

- The color of each horizontal line represents the percentage of execution time spent in the given function for that time slice. (Only included threads, functions, and groups are used in calculating this percentage.)

For instance, in the default setting, functions executing more than 50% of a given time slice are represented by a red horizontal line drawn in the appropriate column at the vertical location corresponding to that time slice.

Trace Data

The trace data shown by the Execution Density diagram includes the following elements:

- The percentage of execution time spent in every traced function or group of functions for each time slice
- The total time, start time, and end time for a selected range of time

You can also show information on collections of functions, called *groups*. That is, you can define groups of functions that are meaningful to you using the **Options > Work with groups...** dialog, and can then optionally select a subset of the groups to include in the diagram using the **View > Include groups...** dialog. You can toggle the showing of group information on or off by repeatedly selecting the **View > Group filter** option.

Uses

Use the Execution Density diagram to perform the following tasks:

- Get an overview of what functions were the most active at various stages of your program's execution.
- Spot execution trends.

RELATED CONCEPTS

"Diagrams for analyzing a trace file" on page 5

"Function groups" on page 19

RELATED TASKS

"Opening a trace file in a diagram" on page 40

"Filtering events by function" on page 53

"Filtering events by thread" on page 54

"Filtering events by group" on page 55

"Correlating events between diagrams" on page 43

"Enlarging or reducing a diagram" on page 44

Statistics diagram

The Statistics diagram gives you a textual report of execution time by function (or group of functions, if you include group information in the diagram), class (if your trace file contains class information), or executable. Use this information to find hot spots in the overall program execution.

Trace Data

The trace data shown by the Statistics diagram consists of summary and detailed information.

The summary information shown by the Statistics diagram can include the following elements:

- Executable name
- Trace file description (if a description was entered when the file was created)
- Execution date
- Execution time
- Number of executables generating events
- Number of classes generating events
- Number of functions generating events
- Number of threads generating events
- Total number of events
- Total number of annotations
- Number of user events
- Maximum call nest depth
- Number of trace buffer flushes
- Total trace time excluding overhead
- Trace overhead
- Indication of whether time is task (CPU) time, or real (clock) time

Note: Data for classes is shown only if your trace file contains class information. The default class `C_Function` always exists, however.

Detailed information is shown for each component traced. Depending upon whether you are viewing information on functions, classes, or executables, the information shown by the Statistics diagram can include the following elements:

- Percent of total execution time spent in the component
- Percent of the total execution time that the component was on the call stack
- Number of times the component was called

- Cumulative execution time spent in the component
- Cumulative execution time that the component was on the call stack
- Execution time for the shortest call to the component
- Execution time for the longest call to the component
- Average execution time for a call to the component

You can also show information on collections of functions, called *groups*. That is, you can define groups of functions that are meaningful to you using the **Options > Work with groups...** dialog, and can then optionally select a subset of the groups to include in the diagram using the **View > Include groups...** dialog. If **View > Details on** is set to **Functions**, you can toggle the showing of group information on or off by repeatedly selecting the **View > Group filter** option.

Uses

Use the Statistics diagram to perform the following tasks:

- Quickly determine which components are consuming the largest amount of execution time. These components are likely to be ones for which performance tuning would prove most beneficial.
- Determine which of your functions are good candidates for inlining. If a function has a small average execution time and is called often, it is a good candidate for inlining.
- Analyze your algorithms by comparing the number of calls that were made to a particular component with the number of calls that you expected, in order to isolate possible inefficiencies.
- Determine which of several algorithms performs better by comparing the statistics recorded in a trace file generated separately for each different version of the algorithm.

RELATED CONCEPTS

“Diagrams for analyzing a trace file” on page 5

“Function groups” on page 19

RELATED TASKS

“Opening a trace file in a diagram” on page 40

“Filtering events by group” on page 55

“Correlating events between diagrams” on page 43

“Viewing class activity” on page 59

“Selecting functions to inline” on page 62

Time Line diagram

The Time Line diagram shows the sequence of nested function calls and returns, with a vertical distance between events that is proportional to the amount of time that elapsed between the respective events. It provides a direct and natural presentation of the chronological relationships of events. The Time Line diagram also shows when the flow of execution switches from one thread to another by means of a dashed horizontal line.

Trace Data

The trace data shown by the Time Line diagram includes the following elements:

- The functions that were called during program execution
- The order in which the functions were called and in which they returned
- The time at which the functions were called and at which they returned, and the time that elapsed between events
- The nesting of function calls (the call stack) at any point during program execution
- The points at which control switched from one thread to another during program execution

Uses

Use the Time Line diagram to perform the following tasks:

- Examine the specific elements of trace data listed above.
- See the interactions among various threads.
- Get a better understanding of a program's flow.
- Determine the elapsed time between two events.

RELATED CONCEPTS

“Diagrams for analyzing a trace file” on page 5

RELATED TASKS

“Opening a trace file in a diagram” on page 40

“Enlarging or reducing a diagram” on page 44

“Correlating events between diagrams” on page 43

“Determining the elapsed time between two events” on page 61

“Viewing thread interactions in a multithreaded program” on page 62

Chapter 3. Trace file generation

Call frequency counting

During function tracing, instead of collecting all tracing data, you can limit the information collected by the Performance Analyzer to the following:

- The functions that each function calls and how many times it calls them
- A count of the number of times each function is called

This data provides information about the call relationships between functions and indicates which functions are being called most frequently. You can use this information to analyze your program and to fine tune its performance.

When you specify **FUNCTION=COUNTS** as a suboption of the **PROFILE** run-time option, call frequency data is written to a trace file. After you download this file to the workstation, you can display the file graphically using the Dynamic Call Graph diagram and the Statistics diagram. No other diagrams display call frequency information.

Note: Call Frequency Counting does not provide call timing information nor does it include a chronological list of events that occur during the execution of the program (function calls, function returns, thread creation, thread switches). This reduces the size of the trace file, making it more manageable and easier to analyze.

A sample call frequency counting trace file called `CallFrequency.trc` is available in the `390ProductivityTools\samples` directory.

RELATED CONCEPTS

- “Time stamps” on page 14
- “Trace events” on page 14
- “Overhead time” on page 16

RELATED TASKS

- “Collecting call frequency data” on page 35
- “Downloading the trace file from the host” on page 39

RELATED REFERENCES

- “Limitations when creating a trace” on page 65
- “Run-time option for program tracing” on page 68
- “Run-time environment variables for program tracing” on page 72

Time stamps

During function tracing, the Performance Analyzer collects time stamp data. A time stamp is a number representing a point in time. At each trace event during the execution of a program, the Performance Analyzer records a time stamp, representing the precise time that the trace event occurred. By comparing the time stamps of two trace events, the elapsed time between the two trace events can be determined. Use the time stamp data associated with the trace events to analyze your program's performance, such as identifying performance bottlenecks.

Note: Time stamps are not collected when Call Frequency Counting is specified.

RELATED CONCEPTS

“Trace events”

“Function trace” on page 15

“Overhead time” on page 16

“Call frequency counting” on page 13

Trace events

To enable your program to be traced, compile it using the `TEST(HOOK)` and `NOGONUMBER` compile options. `NOGONUMBER` is optional, however it will significantly reduce the load module size. If the module is going to be both traced and debugged then do not specify `NOGONUMBER`. The `Test(HOOK)` option enables the compiler to generate hooks in the code at the following points:

- Function entry
- Function exit
- Before a function call
- After a function call

The Performance Analyzer uses these points in the code as trace events. A hook in the code enables the Performance Analyzer to get control and record various information about the event such as the time of the event and the thread and function associated with the event. By default, each event is recorded in chronological order.

The first event in the trace will be a call to a dummy function called `__PROGRAM__`. This event represents the program start time. The last event in the trace will be the return from the `__PROGRAM__` function and represents the program end time.

Note: When Call Frequency Counting is specified, the events are not recorded. The only processing that is done for a trace event in this case is incrementing the call counter for the function and recording call frequency data.

RELATED CONCEPTS

“Time stamps” on page 14

“Function trace”

“Overhead time” on page 16

“Call frequency counting” on page 13

Function trace

The Performance Analyzer performs function tracing, during which information is recorded about the function calls and returns made during the execution of the program.

To enable your program to be traced, compile it using the TEST(HOOK) and NOGONUMBER compile options. These options enable the compiler to generate hooks in the code at the following points:

- Function entry
- Function exit
- Before a function call
- After a function call

The Performance Analyzer uses these points in the code as trace events. A hook in the code enables the Performance Analyzer to get control and record the time of the event. These hooks call a small monitoring function that creates a time stamp on each event. The monitoring function also determines which function the trace event is for and the module that the function is in. By comparing the time stamps of the function entry and the function exit events, the Performance Analyzer determines the time taken to execute the function.

The first event in the trace will be a call to a dummy function called `__PROGRAM__`. This event represents the program start time, which may be different than the time that the first hook is encountered in the program. There is a corresponding function call return event at the end of the trace that represents the program end time. The execution time of the `__PROGRAM__` function and module represents the time spent during the execution of the program when there are no functions containing hooks on the stack. The `__PROGRAM__` execution time will be more significant when not all of the program source files are compiled with TEST(HOOK).

Use Function trace files to accomplish the following objectives:

- Identify costly or time-consuming functions
- Show logic flow (useful for constructors and destructors)
- Identify potential inline functions
- Determine which functions of a DLL are being called
- Track library calls
- Verify all built-in functions are used
- Track function calls among threads
- Track class interaction
- Track module interaction

RELATED CONCEPTS

“Time stamps” on page 14

“Trace events” on page 14

“Overhead time”

“Call frequency counting” on page 13

RELATED TASKS

“Creating a trace file” on page 30

RELATED REFERENCES

“Limitations when creating a trace” on page 65

“Run-time option for program tracing” on page 68

“Run-time environment variables for program tracing” on page 72

“Call Nesting diagram” on page 6

“Dynamic Call Graph diagram” on page 7

“Execution Density diagram” on page 8

“Statistics diagram” on page 10

“Time Line diagram” on page 12

Overhead time

Function Tracing

Whenever the Performance Analyzer logs an event for function tracing, it adds overhead to the normal execution time of your program. This increases the overall execution time of your program. The monitoring function calculates the overhead time on each event. The Performance Analyzer compensates for this overhead time by adjusting the timestamp for the event. It subtracts the overhead time from the event’s timestamp. This adjustment ensures that your program’s true execution performance is reported as accurately as possible.

When tracing is turned off, the monitoring function will not be executed, even if the code contains hooks. However, when hooks are placed in the code, the code size increases. This, along with the small amount of time to execute the hook instructions, may affect the performance of your program, even when tracing is turned off. When you have finished using the Performance Analyzer to tune your program, rebuild it without generating hooks by compiling it without the TEST(HOOK) option. This will improve the execution performance of your program.

RELATED CONCEPTS

“Trace events” on page 14

“Function trace” on page 15

“Time stamps” on page 14

Multiple process support

Use the Performance Analyzer to trace a single-process program or to trace a program that uses the fork or spawn function to create new processes. A separate trace file with a unique name is generated for each process created by the program. If the process is the result of a fork or spawn call, the name of the trace file for the process has the ID of the process (PID) appended to it.

The names of process and program trace files are based on the value of the `__PROF_FILE_NAME` environment variable.

RELATED CONCEPTS

“Function trace” on page 15

RELATED TASKS

“Creating a trace file” on page 30

RELATED REFERENCES

“Sample trace file names from tracing a multiprocess program” on page 85

“Run-time environment variables for program tracing” on page 72

Chapter 4. Trace file viewing and analysis

Function groups

You can create arbitrary collections of functions, called *groups*, and then analyze performance data for one or more selected groups only, instead of analyzing data for all of the functions in your program.

For example, you could create a group called *storage*, assign all storage-related functions to that group, and then analyze only the storage group to understand how much time your program is spending in storage-management activities.

You can create multiple group definitions and save them in a grouping file for later recall, and can create multiple grouping files.

You can view performance information by groups in several diagrams.

In any diagram that supports grouping, select **Collapse group** to combine all of the functions in a group into a single entry. This option is available from the **View, Selected**, or selected item pop-up menu when the diagram is showing group information and a function in one of the groups shown is selected.

You can likewise select **Expand group** to restore a collapsed group to its component entries, one per function. This option is available from the **View, Selected**, or selected item pop-up menu when the diagram is showing group information and a collapsed group is selected.

RELATED CONCEPTS

“Execution Density diagram” on page 8

“Statistics diagram” on page 10

RELATED TASKS

“Filtering events by group” on page 55

Pattern recognition

Loops in your program cause the same sequence of calls and returns to be repeated in the trace. The Performance Analyzer lets you combine like sequences in the Call Nesting diagram by using pattern recognition. The pattern recognition facility looks at a single thread and indicates patterns of calls and returns using curved arcs that show the number of repetitions of each pattern to the right.

This technique reduces the amount of screen space the diagram uses, and therefore shortens the number of pages you must scroll through to look at your trace file.

If you see a pattern repeated numerous times, for workstation programs you can group the functions in the pattern together with *pragma alloc_text* statements to limit the number of page swaps between calls, and thus improve performance. For host programs, group all of the functions in the pattern in one source file, close to each other, to limit the number of page swaps.

Pattern recognition can only be used when you include a single thread for analysis, and there are no collapsed functions in the diagram.

RELATED CONCEPTS

“Call Nesting diagram” on page 6

RELATED TASKS

“Recognizing call sequence patterns” on page 58

“Filtering events by thread” on page 54

Diagram filters

There are several techniques for filtering the trace file to reduce the amount of trace data shown in a diagram, or for isolating interesting or problematic areas in a diagram.

Some techniques have associated tasks that need to be performed in order to apply a filter; such tasks are indicated in the related topics below. Other techniques involve a straightforward manipulation of the objects shown in a diagram's display. The following list contains filtering techniques that you can use in each of the diagrams.

Call Nesting diagram

In this diagram, you can filter data in the following ways:

- Selecting specific functions or specific threads to include. The Performance Analyzer shows trace information only for those functions or threads selected.
- Selecting a time or time range to view.
- Collapsing calls so that all calls and returns subordinate to a calling function are hidden.

Dynamic Call Graph diagram

In this diagram, you can filter data in the following ways:

- Selecting specific threads to include. The Performance Analyzer shows trace information only for those threads selected.
- Removing nodes in the call path to or from a selected node.
- Hiding arcs.
- Hiding selected nodes.
- Creating a subgraph to work with. Doing so cleans up the diagram by deleting hidden nodes, repositioning remaining nodes, and centering the diagram.
- Zooming in.
- Showing data by function, class (only if your trace file contains class information), or executable by selecting from the **View > Nodes of** menu.
- Specifying that only nodes meeting selected criteria of number of calls, percent of execution time, percent of time on stack, execution time, or time on stack are shown.
- Specifying that only arcs meeting a criterion of number of calls are shown.

Execution Density diagram

In this diagram, you can filter data in the following ways:

- Selecting specific functions or threads to include. The Performance Analyzer shows trace information only for those functions or threads selected.
- Selecting specific groups of functions to include. The Performance Analyzer shows trace information only for those groups selected.
- Scaling the diagram to increase the number of pages shown.
- Zooming in.
- Selecting a time or time range to view.

Statistics diagram

In this diagram, you can filter data in the following ways:

- Selecting specific threads to include. The Performance Analyzer shows trace information only for those threads selected.

- Selecting specific groups of functions to include. The Performance Analyzer shows trace information only for those groups selected.
- Showing data by function, class (only if your trace file contains class information), or executable by selecting from the **View > Details on** menu.

Time Line diagram

In this diagram, you can filter data in the following ways:

- Scaling the diagram to increase the number of pages shown.
- Zooming in.
- Selecting a time or time range to view.

RELATED CONCEPTS

“Diagrams for analyzing a trace file” on page 5

“Function groups” on page 19

RELATED TASKS

“Filtering events by function” on page 53

“Filtering events by thread” on page 54

“Filtering events by group” on page 55

“Filtering events by component type” on page 53

Correlation

One diagram cannot show everything of interest within a trace file. Some events are easier to find in one diagram, but the information in another is more meaningful; in such cases it is helpful to find the event in one diagram and then locate that same event in one or more of the other open diagrams. Correlation allows you to do so.

The Performance Analyzer provides three diagrams whose events can be correlated between any two of them, or between all three: Call Nesting, Execution Density, and Time Line. You can correlate the diagrams based on a specific time or event, or on a range of time or events. You can also correlate (in one direction) from a function in the Statistics diagram to a call to that same function in the Call Nesting diagram; the call located is the one that used the most time of all calls to the function.

For example, use the Call Nesting diagram to identify the order and names of functions called, and then correlate to the Time Line diagram to find out how long the functions took to run. Or you can use the Execution Density diagram to see general patterns that lead up to a certain point, and then correlate to the Call Nesting diagram to see the exact order of the function calls.

Note that it is possible to correlate events between two instances of the same diagram, which could prove useful, for example, if each is scaled differently.

RELATED CONCEPTS

“Call Nesting diagram” on page 6

“Execution Density diagram” on page 8

“Statistics diagram” on page 10

“Time Line diagram” on page 12

RELATED TASKS

“Correlating events between diagrams” on page 43

Chapter 5. Tips for Using the Performance Analyzer to understand your program

Use a combination of diagrams to understand your program

The Performance Analyzer allows you to open a trace file in several diagrams, and to open multiple views of the same diagram simultaneously. Sometimes opening a trace file in two or more diagrams can help you understand a program better.

For instance, if you do not want to wade through code listings to determine how the code works, use the Dynamic Call Graph diagram, and the Call Nesting and Time Line diagrams in conjunction with each another to get a better understanding of the program's flow.

In the Call Nesting diagram you can see the order in which functions are called and return, and in the Time Line diagram you can see the timing of the calls and returns.

The Dynamic Call Graph diagram shows all of the program's threads, the relative consumption of execution time by the different functions, and the call hierarchy.

RELATED CONCEPTS

"Diagrams for analyzing a trace file" on page 5

RELATED TASKS

"Opening a trace file in a diagram" on page 40

"Correlating events between diagrams" on page 43

"Seeing details by combining the zoom and correlation features" on page 45

Annotate your trace file

An annotation is a bookmark or reminder that you can place in the trace file after it is created. In the Call Nesting diagram, select **Edit > Annotate...** to insert notes or reminders next to any function you highlight.

Workstation Programs Only: The following information is applicable to workstation programs only.

Chapter 6. Preparing your program for analysis

Compiling your program

Specify the following compile options:

- **TEST(HOOK)**
Generates the following hooks in your code:
 - Function entry and exit
 - Before function call and after function call
- **NOGONUMBER**
NOGONUMBER is optional. However, if it is specified, NOGONUMBER significantly reduces the size of the load module produced because no line number tables are generated in the code. Line number tables are required debugging, not for tracing. If the module is going to be both traced and debugged do not specify NOGONUMBER.
- **OPT(1) or OPT(2)**
Optimization will improve the performance of the application.

Note: For C programs compiled with NOOPT, specify the TEST option as TEST(HOOK,PATH,NOLINE,NOBLOCK,NOSYM).

You can build executables and DLLs from multiple compilation units (object files). You can specify the TEST(HOOK) compile option for only those compilation units that you want to trace. When your executable or DLL is running, the Performance Analyzer only captures trace information for those compilation units built with TEST(HOOK).

For example, if you use the TEST(HOOK) option when compiling both the caller function A and the called function B, the Performance Analyzer sees hooks in the following order:

1. Function A entry hook
2. Before function B call hook
3. Function B entry hook
4. Function B exit hook
5. After function B call hook
6. Function A exit hook

In the example, if function B calls a third function C in another DLL which was compiled with NOTEST, then the Performance Analyzer sees hooks in the following order:

1. Function A entry hook
2. Before function B call hook
3. Function B entry hook
4. Before function C call hook
5. After function C call hook
6. Function B exit hook
7. After function B call hook
8. Function A exit hook

Note that no hooks are available for entry and exit of function C.

Although the hooks exist, you can reduce the overhead time of tracing and, therefore, the overall program execution time by setting the environment variable `__PROF_HOOKS` to `BEFORE_AFTER` or to `ENTRY_EXIT`. This reduces the number of hooks that Performance Analyzer processes. If you set `__PROF_HOOKS` to `ALL`, the Performance Analyzer processes all hooks.

Refer to the OS/390 C/C++ User's Guide for more information on compiling programs.

RELATED CONCEPTS

"Function trace" on page 15

RELATED TASKS

"Setting environment variables for Performance Analyzer"

"Setting run-time option `PROFILE` for Performance Analyzer" on page 29

"Creating a trace file" on page 30

RELATED REFERENCES

"Run-time option for program tracing" on page 68

"Run-time environment variables for program tracing" on page 72

"Sample JCL for creating trace files" on page 83

"Sample TSO commands for creating trace files" on page 82

"Sample Unix system service commands for creating trace files" on page 80

Setting environment variables for Performance Analyzer

The following environment variables allow you to control the trace data that is created by the Performance Analyzer.

You can set these environment variables before executing your program for tracing:

- **__PROF_FILE_NAME=filename**
To specify the name of the trace file to be generated.
- **__PROF_HOOKS = ALL | ENTRY_EXIT | BEFORE_AFTER**
To identify the type of hooks that are to be processed.
- **__PROF_WEBSERVER=NO | YES**
To indicate whether tracing is to be performed in a Lotus Domino Go Webserver environment.

The following samples are available for setting the environment variables:

- “Sample JCL for creating trace files” on page 83
- “Sample Unix system service commands for creating trace files” on page 80
- “Sample TSO commands for creating trace files” on page 82

Refer to the *OS/390 C/C++ Programming Guide* and *OS/390 UNIX System Services User’s Guide* for more information on environment variables. These books are available through the OS/390 C/C++ Library page.

RELATED CONCEPTS

“Function trace” on page 15

RELATED TASKS

“Compiling your program” on page 27

“Setting run-time option PROFILE for Performance Analyzer”

“Creating a trace file” on page 30

RELATED REFERENCES

“Run-time environment variables for program tracing” on page 72

Setting run-time option PROFILE for Performance Analyzer

To enable the tracing of a program during its execution, set the Language Environment run-time option, PROFILE, using the following syntax:

```
PROFile(ON,'string')
```

You specify tracing details with the *string* suboption.

The following samples are available for setting the Language Environment run-time option, PROFILE:

- “Sample JCL for creating trace files” on page 83
- “Sample Unix system service commands for creating trace files” on page 80
- “Sample TSO commands for creating trace files” on page 82

Refer to the *Language Environment for OS/390 and VM Programming Reference* for more information on run-time options.

RELATED CONCEPTS

“Function trace” on page 15

RELATED TASKS

“Compiling your program” on page 27

“Setting environment variables for Performance Analyzer” on page 28

“Creating a trace file”

“Specifying trace file name” on page 37

RELATED REFERENCES

“Run-time option for program tracing” on page 68

“Run-time environment variables for program tracing” on page 72

Creating a trace file

When you run a program, the Performance Analyzer is started by setting the run-time option, `PROFILE(ON,'string')`. You must complete the following steps before running your program:

1. Compile your program with `TEST(HOOK)`, `NOGONUMBER`.
2. Set the Performance Analyzer environment variables `__PROF_FILE_NAME` and `__PROF_HOOKS` if values other than the defaults are desired.
3. Add the Performance Analyzer product dataset called `CBC.SCTVMOD` to the `STEPLIB` of the program if it has not been installed in the link pack area (LPA).
4. Set the run-time option, `PROFILE(ON,'string')`.

The program can run under any of the following environments:

- OS/390 batch
- TSO
- OS/390 UNIX shell

Refer to the *OS/390 C/C++ User's Guide* for more information on compiling and running your applications.

RELATED CONCEPTS

“Function trace” on page 15

“Performance Analyzer product files” on page 2

RELATED TASKS

“Compiling your program” on page 27

“Setting environment variables for Performance Analyzer” on page 28

“Setting run-time option PROFILE for Performance Analyzer” on page 29

RELATED REFERENCES

“Run-time option for program tracing” on page 68

“Run-time environment variables for program tracing” on page 72

“Sample JCL for creating trace files” on page 83

“Sample Unix system service commands for creating trace files” on page 80

“Sample TSO commands for creating trace files” on page 82

Chapter 7. Starting and exiting the Performance Analyzer

Starting the Performance Analyzer

You can start the Performance Analyzer in any of the following ways:

- By double-clicking on its icon
- By entering the Performance Analyzer command on a command line

RELATED TASKS

“Starting the Performance Analyzer from a command line”

“Exiting the Performance Analyzer” on page 34

Starting the Performance Analyzer from a command line

You can start the Performance Analyzer from a command line, a command (CMD) file, or a batch (BAT) file.

To start the Performance Analyzer use the following command:

```
ianalyze [/x]
```

Where /x represents any number of Performance Analyzer invocation parameters documented in the related reference material indicated below.

You can invoke the Performance Analyzer with the Performance Analyzer - Window Manager window shown by issuing the command without invocation parameters. By issuing the command with the appropriate invocation parameters, you can invoke the Performance Analyzer so that it immediately begins to trace an executable or opens an existing trace file in one or more trace file analysis diagrams.

RELATED CONCEPTS

“Diagrams for analyzing a trace file” on page 5

RELATED TASKS

“Opening a trace file in a diagram” on page 40

RELATED REFERENCES

“Performance Analyzer invocation parameters” on page 67

Exiting the Performance Analyzer

To exit the Performance Analyzer, do the following:

1. Select **Exit the Performance Analyzer** from one of the following menus:
 - **File** menu in the Performance Analyzer - Window Manager window
 - **Trace file** menu in any of the diagrams
2. Select **Yes** if necessary.

Select **Options** > **Quick exit** in the Performance Analyzer - Window Manager window to cause the application to exit immediately without confirmation each time you select **Exit the Performance Analyzer**.

Chapter 8. Controlling what data is collected in the trace file

Collecting call frequency data

By default, the Performance Analyzer collects and records all tracing data, including timing, call count, and call relationship information for each function traced. It also records a chronological list of the events that occur during the trace. To collect only a count of the number of times each function is called and the functions each function calls, you must specify the following suboption in the **PROFILE** run-time option:

FUNCTION=COUNTS

For example:

PROFILE (ON, 'FUNCTION=COUNTS')

This capability of the Performance Analyzer is called call frequency counting.

To view the call frequency information collected with this suboption, make the trace file generated as a result of this suboption accessible on the workstation by using NFS or by downloading it. You can then display the file graphically on the workstation with the following diagrams:

Diagram	Information available	Information not available
Dynamic Call Graph	<ul style="list-style-type: none">• Call counts on arcs and information windows• Call Frequency Data• Colour-coded arcs	<ul style="list-style-type: none">• <i>Execution time</i> and <i>Time on Stack</i> values and percentages on information windows• Scaled node sizes• Colour-coded nodes
Statistics	<ul style="list-style-type: none">• Number of calls column• Partial summary section, which includes the number of executables, classes, functions, and threads plus call depth for each thread	<p>The following columns:</p> <ul style="list-style-type: none">• % of Execution• % on Stack• Execution time• Time on stack• Minimum Call• Maximum Call• Average Call

Note: Call Frequency Counting does not provide call timing information nor does it include a chronological list of events that occur during the execution of the program (function calls, function returns, thread creation, thread switches). This reduces the size of the trace file, making it more manageable and easier to analyze.

A sample call frequency counting trace file called CallFrequency.trc is available in the 390ProductivityTools\samples directory.

RELATED CONCEPTS

“Function trace” on page 15

“Call frequency counting” on page 13

RELATED TASKS

“Compiling your program” on page 27

“Creating a trace file” on page 30

RELATED REFERENCES

“Run-time option for program tracing” on page 68

Tracing a specific DLL

To trace a specific DLL, compile the files that make up the DLL with the TEST(HOOK) and NOGONUMBER options. Do not use the TEST(HOOK) option when compiling any other code that calls the DLL; otherwise, when you perform the trace, you would get trace data for the other code. Perform the trace as you would do normally by setting the PROFILE run-time option and executing the program that calls the DLL.

RELATED CONCEPTS

“Function trace” on page 15

RELATED TASKS

“Compiling your program” on page 27

“Creating a trace file” on page 30

Tracing a Webserver application

To trace a Lotus Domino Go Webserver application, do the following:

1. Compile the application code using the TEST(HOOK) and NOGONUMBER compile options. NOGONUMBER is optional and will significantly reduce the load module size. If the module is going to be both traced and debugged, do not specify NOGONUMBER.
2. Start the Lotus Domino Go Webserver with tracing turned on by doing the following:
 1. Set the Performance Analyzer environment variable `__PROF_WEBSERVER=YES`
 2. Set the Performance Analyzer environment variable `__PROF_FILE_NAME` to the desired name for the trace file.
 3. Set the Performance Analyzer environment variable `__PROF_HOOKS` if desired.
 4. Add the Performance Analyzer product dataset called CBC.SCTVMOD to the STEPLIB.
 5. Set the run-time option, PROFILE(ON,'string')
 6. Start the Lotus Domino Go Webserver in the environment where the previous steps were done.
1. Run the Webserver application.
2. Stop tracing and create a trace file by sending the SIGPROF signal to the Lotus Domino Go Webserver with the OS/390 Unix System Services kill command. For example, issue the following command from another OS/390 Unix System Services shell session:

kill -s PROF *webspid* where **webspid** is the process ID of the Lotus Domino Go Webserver.

Note: To trace another Webserver application, you must stop then restart the Lotus Domino Go Webserver with tracing turned on again.

RELATED CONCEPTS

“Function trace” on page 15

RELATED TASKS

“Compiling your program” on page 27

“Creating a trace file” on page 30

Specifying trace file name

Use the `__PROF_FILE_NAME` environment variable to name a trace file. Set this environment variable before you run your program for tracing.

If you do not set the `__PROF_FILE_NAME` environment variable, the Performance Analyzer generates a default name for the trace file. Default trace file names are explained in Run-Time Environment Variables for Program Trace.

The PID will be appended to the file name if the program is running in the OS/390 UNIX shell or running with the POSIX(ON) run-time option provided the program has more than one process; for example, *name.trc.21425534*.

RELATED CONCEPTS

“Function trace” on page 15

RELATED TASKS

“Setting environment variables for Performance Analyzer” on page 28

RELATED REFERENCES

“Run-time environment variables for program tracing” on page 72

Chapter 9. Viewing your trace file in a diagram

Downloading the trace file from the host

After creating a trace file, you can view the data contained in it by using the workstation component of Performance Analyzer. However, the trace file created on the host must be accessible on the workstation to view the data. You can use a remote file access program like Network File System (NFS) to access the binary trace file on the host, or you can physically download the file to the workstation as a binary file.

Example

The following example shows you how to use FTP to download a trace file *userid/sample.trc* to a file called *newsamp.trc* in the directory *c:\pa\traces*.

1. Create this directory on your workstation:

```
[C:\]md pa
[C:\]md pa\traces
```

2. Make that your current directory:

```
[C:\]cd pa\traces
```

3. Logon to FTP:

```
[C:\pa\traces]ftp system
Name (system):userid
Password: *****
```

This message appears:

230 *userid* is logged on. Working directory is "*userid*".

4. Transfer the file in *binary*:

```
ftp>binary
ftp>get sample.trc newsamp.trc
```

Note: You may need to change the trace file names due to differences in host and workstation file naming conventions. To rename files when transferring them from the host, specify the new file name in the **get** command. In the example, *sample.trc* is transferred and renamed to *newsamp.trc* at the workstation.

5. Logoff from FTP:

```
ftp>bye
```

RELATED CONCEPTS

"Function trace" on page 15

RELATED TASKS

“Compiling your program” on page 27

“Setting environment variables for Performance Analyzer” on page 28

“Setting run-time option PROFILE for Performance Analyzer” on page 29

“Creating a trace file” on page 30

Starting the Performance Analyzer to analyze a trace file

Once you have downloaded the trace file, you can analyze your data from your workstation. After you start the Performance Analyzer on your workstation, follow these steps:

1. In the Performance Analyzer Window Manager window, select **Analyze Trace**.
2. In the Analyze Trace window, specify the trace file name in the dialog box; or search for the trace file by clicking the **Find** button.
3. Select the appropriate diagrams to view the data.
4. Click the **OK** button.

RELATED CONCEPTS

“Function trace” on page 15

RELATED TASKS

“Creating a trace file” on page 30

“Downloading the trace file from the host” on page 39

Opening a trace file in a diagram

To open a trace file in a Performance Analyzer diagram when the Performance Analyzer is already running, use one of the following methods:

- Click the **Analyze Trace...** push button in the Performance Analyzer - Window Manager window, and then, in the Analyze Trace window, enter a trace file name and select one or more of the diagram check boxes.
- Double-click the file name or icon of a trace file in the Performance Analyzer - Window Manager window, then select one of the diagram check boxes in the Analyze Trace window and click **OK**.
- Click mouse button 2 on the file name or icon of a trace file in the Performance Analyzer - Window Manager window, then select a diagram from the trace file pop-up menu.
- From the **Trace file** menu of an open diagram, select **Open as** and then select a diagram from the cascaded menu.

- From any open diagram, click the appropriate button in the tool bar.

To immediately open a trace file in one or more Performance Analyzer diagrams when starting the Performance Analyzer from the command line, see the related procedure indicated below.

RELATED TASKS

“Starting the Performance Analyzer from a command line” on page 33

Chapter 10. Navigating the trace file view

Correlating events between diagrams

To correlate events between the Call Nesting, Execution Density, or Time Line diagrams, complete these steps:

1. Open the trace file in the diagrams between which you want to correlate events.
2. Highlight the event range of interest in one of the diagrams by taking these steps:
 - a. Click and hold mouse button 1 on the first event.
 - b. Drag the pointer to the last event.
 - c. Release the mouse button.
3. Select **Options > Correlation...** in the highlighted diagram. The Correlation window appears.
4. In the Correlation window, click the names of one or more diagrams to which you want to correlate, or click the **Select all** push button to correlate to all of the diagrams listed.
5. Click **OK**.

To correlate from a function in the Statistics diagram to the instance of the call to that same function in the Call Nesting diagram that used the most time of all calls to that function, complete these steps:

1. Open the trace file in the Statistics and Call Nesting Diagrams.
2. Highlight a single function in the Statistics diagram.
3. Select **Options > Correlation...** in the Statistics diagram. The Correlation window appears.
4. In the Correlation window, click the names of one or more diagrams to which you want to correlate, or click the **Select all** push button to correlate to all of the diagrams listed.
5. Click **OK**.

RELATED CONCEPTS

“Correlation” on page 22

RELATED TASKS

“Opening a trace file in a diagram” on page 40

“Seeing details by combining the zoom and correlation features” on page 45

Enlarging or reducing a diagram

The Dynamic Call Graph, Execution Density, and Time Line diagrams have zooming capabilities that allow you to enlarge (**Zoom in**) or reduce (**Zoom out**) the size of the diagram in order to focus on the area that is of most interest.

To enlarge the region of a diagram that is of most interest, follow these steps:

1. Highlight the area that you want to enlarge.
2. Select **View > Zoom in**.
3. Scroll until you see the area you highlighted.
4. Continue alternately selecting **Zoom in** and scrolling to the highlighted area until the diagram is enlarged to the degree you want.
5. If you zoom in too far, select **View > Zoom out** to quickly back out one step.

A quick way to zoom in on an area of interest in the Execution Density or Time Line diagram is to use the **Zoom to selected range** option in the **View** menu. It also provides an easy way to restore the diagram to full scale. To quickly restore the diagram to full scale after it has been reduced, follow these steps:

1. Select **Edit > Select all**.
2. Select **View > Zoom to selected range**.

A quick way to navigate and to zoom in or out in the Dynamic Call Graph diagram is to use the **Overview** feature. When you select **View > Overview**, a miniature version of the Dynamic Call Graph diagram appears in the Overview window, and a small gray box in the window highlights the area of the diagram that is currently in view. Use the gray box as follows:

- To change the area of the diagram that is currently in view, click and hold mouse button 1 inside the gray box, and then move the box until the desired area is in view.
- To resize the area shown in the diagram, grab and move the sides of the gray box with the mouse until the area shown has the desired size.

To restore the Dynamic Call Graph diagram to the original view and size, select **View > Re-lay graph**.

RELATED CONCEPTS

“Diagrams for analyzing a trace file” on page 5

RELATED TASKS

“Opening a trace file in a diagram” on page 40

“Seeing details by combining the zoom and correlation features”

Seeing details by combining the zoom and correlation features

A useful technique for examining specific areas of the Execution Density and Time Line diagrams is to use the zoom and correlation features together. Zooming sometimes forces the highlighted region off the page; correlation can help you quickly find it again.

To use this technique, complete these steps:

1. Open your trace file in the Execution Density or Time Line diagram.
2. Open another diagram that allows correlation (Call Nesting, Execution Density, or Time Line).
3. Highlight the area that you want to zoom in on (enlarge) in the first diagram.
4. Correlate the first diagram to the second.
5. In the first diagram, select **View > Zoom in** as many times as you want.
If you zoom in too far, select **View > Zoom out** to quickly back out one step.
6. Correlate the second diagram to the first.
The region you originally highlighted is now back in view.

RELATED CONCEPTS

“Diagrams for analyzing a trace file” on page 5

“Correlation” on page 22

RELATED TASKS

“Opening a trace file in a diagram” on page 40

“Correlating events between diagrams” on page 43

“Enlarging or reducing a diagram” on page 44

Viewing a specific time or range of time

You can view trace data for a specific time or range of time in the following diagrams:

- Call Nesting
- Execution Density
- Time Line

To view a specific time in one of these diagrams, complete these steps:

1. Select **Edit > Select Time....** The Select Time window appears.
2. Click the appropriate radio button to select the desired unit of time.
3. Use the arrows in the spin button if you want to change the time already shown.
4. Click the appropriate push button to continue.

To view a specific range of time in one of these diagrams, complete these steps:

1. Select **Edit > Select Time Range....** The Select Time Range window appears.
2. Select the start time:
 - a. Click the appropriate radio button to select the desired unit of time.
 - b. In the **Start Time** group box, use the spin button arrows to select the time at which you want the highlight to start.
3. Select the end time:
 - a. Click the appropriate radio button to select the desired unit of time.
 - b. In the **End Time** group box, use the spin button arrows to select the time at which you want the highlight to end.
4. Click the appropriate push button to continue.

Chapter 11. Searching for trace data in a diagram

Finding a specific annotation

An *annotation* is a bookmark or comment that you can place in the trace file after the trace file is created. Annotations are saved to the trace file so that you can see them at a later time. You can search for an annotation in the Call Nesting diagram.

To search for an annotation, complete these steps:

1. Select **Edit > Find**.
2. Select **Annotation...** from the cascaded menu. The Find Annotation window appears.
3. Follow the directions in the dialog window to get a list of all annotations. You can enter an asterisk (*) with a few characters of the annotation in the **Find** entry field, as follows:
 - Use an asterisk (*) to represent zero or more arbitrary characters. For example, enter:
 - * to show a list of all annotations
 - *b** to show all annotations, regardless of length, that begin with the character *b*
 - **b* to show all annotations that end with the character *b*
 - Use a question mark (?) to represent a single arbitrary character. For example, enter ?*b** to show all annotations that start with any character and have the character *b* as their second character.
4. Click the appropriate push button to continue.
5. If more than one annotation matches your search criteria, select the desired annotation in the list box, and click **OK**.

The search for the annotation begins at the currently selected annotation, user event, or function (or the first such instance in the currently selected range). The search continues until the annotation is found or the end of the diagram is reached. If the annotation is found, it is highlighted; if it is not found, a message box to that effect appears.

RELATED TASKS

“Adding, changing, or deleting annotations” on page 61

Finding a specific function call or return

You can search for a function call or return in the following diagrams:

- Call Nesting
- Execution Density
- Time Line

To search for a function call or return in one of these diagrams, complete the following steps:

1. Open the Find Function window:
 - In the Call Nesting or Time Line diagrams, select **Edit > Find > Function....**
 - In the Execution Density diagram, select **Edit > Find function....**
2. Enter the function name in the **Find** entry field. You can use wildcard characters (*) and (?) in the entry field, as follows:
 - Use an asterisk (*) to represent zero or more arbitrary characters. For example, enter:
 - * to show a list of all function names
 - *b** to show all function names, regardless of length, that begin with the character *b*
 - **b* to show all function names that end with the character *b*
 - Use a question mark (?) to represent a single arbitrary character. For example, enter *?b** to show all function names that start with any character and have the character *b* as their second character.

Wildcards are especially useful when you are searching for a fully qualified function name (for example, *myClass::function[parameter]*).
3. Be sure the **Case sensitive** box is checked if you want to enable case-sensitive searching.
4. Select the thread that you want searched.
5. Click the appropriate radio button to search for occurrences of when the function:
 - Was called
 - Returned
 - Was either called, or returned
6. Click the appropriate push button to continue.
7. If more than one function matches your search criteria, select the desired function in the list box, and click **OK**.
8. Select **Edit > Find next** to find the next occurrence of the function call or return.

The search for the function begins at the currently selected function, annotation (Call Nesting only), or user event (or the first such instance in the currently selected range). The search continues until the function is found or the end of the diagram is reached. If the function is found, it is highlighted; if it is not found, a message box to that effect appears.

Finding trace data for a specific function

You can search for trace data for a specific function in the following diagrams:

- Dynamic Call Graph
- Statistics

To search for a function in one of these diagrams, complete the following steps:

1. Make sure trace data for functions is shown in the diagram. See the Filter Events by Component Type topic for instructions.
2. Select **Options** > **Find...** The Find Function window appears.
3. Enter the function name in the **Find** entry field. You can use wildcard characters (* and ?) in the entry field, as follows:
 - Use an asterisk (*) to represent zero or more arbitrary characters. For example, enter:
 - * to show a list of all function names
 - *b** to show all function names, regardless of length, that begin with the character *b*
 - **b* to show all function names that end with the character *b*
 - Use a question mark (?) to represent a single arbitrary character. For example, enter *?b** to show all function names that start with any character and have the character *b* as their second character.

Wildcards are especially useful when you are searching for a fully qualified function name (for example, *myClass:: function[parameter]*).
4. Be sure the **Case sensitive** box is checked if you want to enable case-sensitive searching.
5. Click the appropriate push button to continue.
6. If more than one function matches your search criteria, click the desired function in the list box, and click **OK**.

The function is highlighted when found.

RELATED TASKS

“Filtering events by component type” on page 53

Finding trace data for a specific class

You can search for trace data for a specific class in the following diagrams:

- Dynamic Call Graph
- Statistics

This is possible only if your trace file contains class information.

To search for a class in one of these diagrams, complete these steps:

1. Make sure trace data for classes is shown in the diagram:
 - In the Dynamic Call Graph diagram, select **View > Nodes of > Classes**.
 - In the Statistics diagram, select **View > Details on > Classes**.
2. Select **Options > Find...** The Find Class window appears.
3. Enter the class name in the **Find** entry field. You can use wildcard characters (* and ?) in the entry field, as follows:
 - Use an asterisk (*) to represent zero or more arbitrary characters. For example, enter:
 - * to show a list of all class names.
 - *b** to show all class names, regardless of length, that begin with the character *b*.
 - **b* to show all class names that end with the character *b*.
 - Use a question mark (?) to represent a single arbitrary character. For example, enter *?b** to show all class names that start with any character and have the character *b* as their second character.
4. Be sure the **Case sensitive** box is checked if you want to enable case-sensitive searching.
5. Click the appropriate push button to continue.
6. If more than one class name matches your search criteria, click the desired class name in the list box, and click **OK**.

The class is highlighted when found.

RELATED TASKS

“Opening a trace file in a diagram” on page 40

Finding trace data for a specific executable

You can search for trace data for a specific executable in the following diagrams:

- Dynamic Call Graph
- Statistics

To search for an executable in one of these diagrams, complete these steps:

1. Make sure trace data for executables is shown in the diagram:
 - In the Dynamic Call Graph diagram, select **View > Nodes of > Executables**.
 - In the Statistics diagram, select **View > Details on > Executables**.
2. Select **Options > Find...** The Find Executable window appears.
3. Enter the executable name in the **Find** entry field. You can use wildcard characters (* and ?) in the entry field, as follows:
 - Use an asterisk (*) to represent zero or more arbitrary characters. For example, enter:
 - * to show a list of all executable names.
 - *b** to show all executable names, regardless of length, that begin with the character *b*.
 - **b* to show all executable names that end with the character *b*.
 - Use a question mark (?) to represent a single arbitrary character. For example, enter ?*b** to show all executable names that start with any character and have the character *b* as their second character.
4. Be sure the **Case sensitive** box is checked if you want to enable case-sensitive searching.
5. Click the appropriate push button to continue.
6. If more than one executable matches your search criteria, click the desired executable in the list box, and click **OK**.

The executable is highlighted when found.

RELATED TASKS

“Opening a trace file in a diagram” on page 40

Chapter 12. Controlling what data is shown in the diagrams

Filtering events by component type

You can show trace data for a specific component type in the following diagrams:

- Statistics
- Dynamic Call Graph

To show trace data for a specific component type in the Statistics or Dynamic Call Graph diagram, complete the following steps:

1. Open the trace file in one of the diagrams.
2. Select **View > Details on** in the Statistics diagram or **View > Nodes of** in the Dynamic Call Graph diagram.
3. Select the component type for which you want to show data:
 - Select **Functions** to show data on functions.
 - Select **Classes** to show data on classes (only possible if your trace file contains class information).
 - Select **Executables** to show data on executables.

A mark is shown next to the component type that is currently selected.

RELATED TASKS

“Opening a trace file in a diagram” on page 40

Filtering events by function

Filters allow you to temporarily reduce the amount of trace data shown in a diagram, or to isolate interesting or problematic areas. There are several techniques for filtering the trace data.

You can filter events by function in the following diagrams:

- Call Nesting
- Execution Density

To filter specific functions from one of these diagrams, complete these steps:

1. Select **View > Include functions....** The Include Functions window appears.
2. Scroll the list to find the function or functions you want to filter from the diagram's display.
3. Click each function you want to remove so that it is no longer highlighted.
4. Click the appropriate push button to continue.

To include specific functions in one of these diagrams, complete these steps:

1. Select **View > Include functions....** The Include Functions window appears.
2. Click the **Deselect all** push button.
3. Scroll the list to find the function or functions you want shown.
4. Select each function you want shown.
5. Click the appropriate push button to continue.

To include all functions in one of these diagrams, complete these steps:

1. Select **View > Include functions....** The Include Functions window appears.
2. Click the **Select all** push button.
3. Click the appropriate push button to continue.

In the Execution Density diagram, only those functions selected are shown. In the Call Nesting diagram, each selected function and any function in its call stack are shown.

RELATED TASKS

“Opening a trace file in a diagram” on page 40

Filtering events by thread

Filters allow you to temporarily reduce the amount of trace data shown in a diagram. There are several techniques for filtering the trace file and isolating interesting or problematic areas.

When you apply a thread filter, only the data from the selected threads is processed and displayed in the diagram. You can filter events by thread in the following diagrams:

- Call Nesting
- Execution Density
- Dynamic Call Graph
- Statistics

To filter events by thread, complete these steps:

1. Open the trace file in one of the diagrams listed above.
2. Select **View > Include threads....** The Include Threads dialog appears.
3. Select the thread or threads for which you want to see trace information in one of the following ways:
 - Select specific threads by highlighting them with the mouse.
 - Click the **Select all** push button to highlight all threads.
4. Click the **OK** push button.

RELATED TASKS

“Opening a trace file in a diagram” on page 40

Filtering events by group

Filters allow you to temporarily reduce the amount of trace data shown in a diagram, or to isolate interesting or problematic areas. There are several techniques for filtering the trace data.

You can filter events by function group in the following diagrams:

- Execution Density
- Statistics

To define one or more groups of functions that can be analyzed as a whole in one of the diagrams that supports groups, complete the following steps:

1. Select **Options > Work with groups....**
2. On the Grouping File page of the Work with Groups window, define a grouping file by entering a file name and optional description in the appropriate fields.
3. On the Group Names page of the Work with Groups window, enter a group name and optional description in the appropriate fields. Click **Add**. Repeat this step for as many groups of functions as you want to define in the grouping file.
4. On the Group Definition page of the Work with Groups window, add functions to a group by completing the following steps:
 - a. Select the group you want to add functions to from the **Group Name** drop-down combination box.
 - b. Enter a search string in the **Filter Mask** entry field that matches the function names you want to add to the group. Use ***** as a wildcard to match zero or more characters, or **?** as a wildcard to match any single character.

- c. Select the appropriate check boxes to indicate whether you want to select functions that currently belong to no group, to exactly one group, or to more than one group.
- d. Be sure the **Case sensitive** box is checked if you want to enable case-sensitive searching.
- e. Click **Filter** to see the function names that match your search criteria.
- f. In the **Available Functions** container, highlight the functions you want to add to the group, and click **Add**.

Repeat this step for each group of functions you want to define in the grouping file.

Note that if you add a function to more than one group, and are showing group information in a diagram, the function appears once for each group it belongs to that is included in the diagram.

5. When you are finished defining groups and their functions, select the **Save grouping file on OK** check box in the Work with Groups window, and click **OK**. (If you just click **OK** without having saved your changes to a grouping file, the defined groups remain in effect for the current trace file in the current Performance Analyzer session only.)

To indicate which collection of function groups you want to analyze, complete these steps:

1. Select **Options > Work with groups....**
2. On the Grouping File page of the Work with Groups window, click **Find....**
3. Select a grouping file from the files found, and click **OK**.
4. Click **Open**.

To include specific groups in one of the diagrams that supports groups, complete these steps:

1. Select **View > Include groups....** The Include Groups window appears.
2. Click the **Deselect all** push button.
3. Scroll the list to find the groups you want.
4. Select each group you want to include.
5. Click the appropriate push button to continue.
6. To view groups in a diagram, you must be viewing functions in that diagram:
 - In the Statistics diagram, select **View > Details on > Functions** if you are not currently viewing functions.

To view group information in a diagram after having removed it from the diagram, select **View > Group filter**. A mark appears next to the option to indicate that it is in effect.

To remove group information from a diagram that is showing group information, select **View > Group filter**. The mark next to the option is removed to indicate that it is not in effect.

RELATED CONCEPTS

“Function groups” on page 19

“Execution Density diagram” on page 8

“Statistics diagram” on page 10

Filtering nodes and arcs in the Dynamic Call Graph diagram

To define a specific cross section of nodes that you want shown in the Dynamic Call Graph diagram, complete these steps:

1. Select **View > Filters > Nodes....** The Nodes Filter window appears.
2. Select the check boxes for the desired filter criteria, and fill in the corresponding values by which you want to filter the nodes.
3. Click the **And** radio button to show the nodes that meet the values for all the selected criteria. Alternatively, click the **Or** radio button to show the nodes that meet the values of at least one of the selected criteria.
4. Select one or more compile units in which you want to search for nodes that meet the filter criteria.
5. Click the **OK** push button to apply the filters and close the Nodes Filter window. (The **Apply** push button applies the filters but leaves the Nodes Filter window open.)
The nodes that meet the filter criteria are shown and the nodes that do not meet the criteria are hidden.

To define a specific cross section of arcs that you want shown in the Dynamic Call Graph diagram, complete these steps:

1. Select **View > Filters > Arcs....** The Arcs Filter window appears.
2. Select the check box for the **Number of Calls** criterion, and fill in the corresponding values by which you want to filter the arcs.
3. Select one or more compile units in which you want to search for arcs that meet the filter criterion.
4. Click the **OK** push button to apply the filter and close the Arcs Filter window. (The **Apply** push button applies the filter but leaves the Arcs

Filter window open.)

The arcs that meet the filter criterion are shown and the arcs that do not meet the criterion are hidden.

At any time, you can restore nodes or arcs that were hidden as a result of the filtering by bringing up the appropriate filter window again, clearing the check boxes, and clicking **OK**. Alternatively, you can select the **Restore graph** or **Restore subgraph** option, as appropriate, from the **View** menu.

RELATED CONCEPTS

“Dynamic Call Graph diagram” on page 7

RELATED TASKS

“Opening a trace file in a diagram” on page 40

Recognizing call sequence patterns

The Performance Analyzer lets you combine like sequences of calls and returns in the Call Nesting diagram, by using a pattern recognition facility.

To recognize patterns in the Call Nesting Diagram, complete these steps:

1. If any calls in the diagram are collapsed (indicated by a plus (+) sign next to the function name), return to the diagram and expand them by selecting **View > Expand all**.
2. Select **View > Include threads...**
3. Select a single thread for which you want to see patterns by highlighting it with the mouse.
4. Select the **Use pattern recognition** check box.
5. Click **OK**.

The Call Nesting diagram indicates patterns with curved lines that show the number of repetitions on the right.

RELATED CONCEPTS

“Pattern recognition” on page 20

“Call Nesting diagram” on page 6

RELATED TASKS

“Filtering events by thread” on page 54

Viewing class activity

If you are analyzing a trace file that contains class information, you can view class activity in a Dynamic Call Graph or Statistics diagram. For example, you might want to view class activity to do one of the following:

- Discern patterns more easily. Viewing class details in a Dynamic Call Graph diagram condenses the data shown because activity in all functions defined for a class is combined, which simplifies observing patterns.
- Identify the function that consumes the most time. Viewing class details in a Statistics diagram shows you how long a particular process (group of functions) takes to run, which helps you reduce the number of functions that you must look at when trying to identify the one consuming the most time.

To view a Dynamic Call Graph or Statistics diagram by class, do the following:

1. Create a trace file.
2. Open the trace file in a Dynamic Call Graph or Statistics diagram.
3. In the Dynamic Call Graph diagram, select **View > Nodes of > Classes**. Each node then represents the data for every member function contained in that class.

In the Statistics diagram, select **View > Details on > Classes**. The Summary and Details panes then provide statistics about the classes used in your executable.

RELATED TASKS

“Opening a trace file in a diagram” on page 40

Chapter 13. Analyzing Your Trace File

Adding, changing, or deleting annotations

An annotation is a bookmark or comment that you can place in the trace file after it is created. You can add, change, or delete annotations in the Call Nesting diagram. The Performance Analyzer saves the annotations to the trace file so you can see them later.

To add or change an annotation, complete these steps:

1. Select the function in which you want to add or change the comment.
2. Select **Edit > Annotate....**
3. Type the comment in the window. The comment is limited to 64 characters.
4. Click the appropriate push button to continue.

To delete an annotation, complete these steps:

1. Select the comment you want to remove.
2. Select **Edit > Annotate....**
3. Click **Remove**.

RELATED TASKS

“Finding a specific annotation” on page 47

Determining the elapsed time between two events

To determine the elapsed time between two events, complete these steps:

1. Create a trace file.
2. Open the trace file in the Time Line diagram.
3. Highlight the area between the two events in the Time Line diagram. To highlight an area:
 - a. Click and hold mouse button 1 on the first event.
 - b. While holding mouse button 1, drag the pointer to the second event.
 - c. Release the mouse button.
4. Check the status area of the Time Line diagram for the elapsed time between events. Elapsed time is shown at the beginning of the line labeled **Selected region**.

RELATED CONCEPTS

“Time Line diagram” on page 12

RELATED TASKS

“Opening a trace file in a diagram” on page 40

Selecting functions to inline

Your trace file can help you determine which functions to inline. To do this, complete these steps:

1. Use the *OPT* compiler option.
2. Create a trace file and view it in the Statistics diagram. If an inlined function appears in the Statistics diagram, the compiler chose not to inline it.
3. Look for functions in the Statistics diagram that were called frequently and had small average execution times. These functions could be good candidates for inlining.

Although inlining functions improves the performance of your application, it also increases the size of your executable.

RELATED TASKS

“Opening a trace file in a diagram” on page 40

Viewing thread interactions in a multithreaded program

To view thread interactions in a multithreaded program, complete these steps:

1. Create a trace file.
2. Open the trace file in a Call Nesting or Time Line diagram.
3. Scroll through the diagram using the vertical scroll bar. Look for horizontal dashed lines, which indicate that your program has switched from one thread to another.

Doing so allows you to see the flow of execution across threads, and could be helpful in identifying timing problems.

RELATED CONCEPTS

“Correlation” on page 22

“Call Nesting diagram” on page 6

“Time Line diagram” on page 12

RELATED TASKS

“Opening a trace file in a diagram” on page 40

Chapter 14. Reference

Limitations when analyzing trace data

Note the following limitations when using the Performance Analyzer:

- The Performance Analyzer runs only on Windows 95 or Windows NT. It does not run on Windows 3.1x or Windows for Workgroups, even with the Win32s extension.
- The pattern recognition feature in the Call Nesting diagram has the following limitations:
 - The Performance Analyzer only searches for patterns in the first 32,768 events in the selected thread.
 - The maximum number of patterns for which the Performance Analyzer searches is 8191.

If the Performance Analyzer reaches either of these limits, it stops searching for patterns.

- The Dynamic Call Graph diagram cannot analyze trace files that have more than 7000 functions.
- If you minimize the Overview window in the Dynamic Call Graph diagram, it does not appear in the list of open applications.

Workstation Programs Only: The following Performance Analyzer limitations apply to workstation programs only.

Limitations when creating a trace

When creating a trace with the Performance Analyzer on OS/390, the following limitations apply:

- There is no support for the Performance Analyzer under a CICS environment.
- The Performance Analyzer can trace C/C++ applications that have multiple threads. However, it cannot accurately determine when the processor switches active threads. It can determine only the following points because they are the points where the Performance Analyzer gets control:
 - When a thread is created
 - When a function is called or returns within a thread
 - When a thread terminates

Consequently, thread switches are recorded only at function call and return points.

- Because an OS/390 system can have multiple processors, threads can execute concurrently. For a particular function call in one thread, the Performance Analyzer calculates the function's execution time independent of the execution time calculations for functions running in other threads. On an OS/390 system with a single processor, this may result in the reporting of function execution time that is larger than the actual execution time. Another consequence of this method of function execution time calculation for a multithread application is that the data in the Statistics diagram may show a total execution time for all functions exceeding the total execution time for the program.
- Because of the overhead time required for tracing, the program takes longer to execute when it is tracing. This overhead time is factored out in the Performance Analyzer data.
- When tracing multithreaded applications with TASK time, the source code for the parent thread must be compiled with the TEST(HOOK) option in order for the child threads to be traced.
- The time period between the time a C++ exception is thrown and the time the next method is called will be charged to the method that throws the exception. A consequence of this is that time spent in a method that catches the exception but does not call any other methods or functions will not be charged to that method but to the method that threw the exception.
- When you compile your program with the TEST(HOOK) option, tracing hooks are inserted into your program. This increases your program's size, and because time is required to execute the hook instructions, the program runs slower even when tracing is turned off. For this reason, it is recommended that you rebuild your program without the TEST(HOOK) option after you have finished using the Performance Analyzer to trace and tune your program.

RELATED CONCEPTS

"Function trace" on page 15

RELATED TASKS

"Creating a trace file" on page 30

"Compiling your program" on page 27

RELATED REFERENCES

"Run-time option for program tracing" on page 68

Performance Analyzer invocation parameters

When you start the Performance Analyzer from a command line, a command (CMD) file, or a batch (BAT) file, you can control which of the following initial actions the Performance Analyzer takes:

- Show the Performance Analyzer - Window Manager window
- Open an existing trace file (in one or more diagrams)

This is accomplished by specifying the appropriate invocation parameters, as described below.

Show the Performance Analyzer - Window Manager window

To start the Performance Analyzer and show the Performance Analyzer - Window Manager window, use the following command:

```
ianalyze
```

Open an Existing Trace File

To start the Performance Analyzer and open an existing trace file in one or more trace file analysis diagrams, use the following command:

```
ianalyze /x myprog.trc
```

Where:

/x Represents one or more of the following options. If you have already created a trace file, these options cause the trace file to be shown in their respective diagrams. You can quickly open the diagrams by entering one or more of these options in your startup command.

<i>/cn</i>	Shows the trace file in the Call Nesting diagram.
<i>/ed</i>	Shows the trace file in the Execution Density diagram.
<i>/cg</i>	Shows the trace file in the Dynamic Call Graph diagram.
<i>/ss</i>	Shows the trace file in the Statistics diagram.
<i>/tl</i>	Shows the trace file in the Time Line diagram.

myprog

Represents a trace file name.

For example, if you want to show the *myprog.trc* trace file in both the Call Nesting and Execution Density diagrams, enter the command:

```
ianalyze /cn /ed myprog.trc
```

RELATED CONCEPTS

“Diagrams for analyzing a trace file” on page 5

RELATED TASKS

“Opening a trace file in a diagram” on page 40

Tracing programs that have interlanguage calls

The Performance Analyzer does not trace calls to functions written in languages other than C and C++.

If your C or C++ application makes an interlanguage call, and the `__PROF_HOOKS` environment variable was set to `ALL` or `BEFORE_AFTER`, the called function is displayed with name `Unknown_Function_xxxxxxx` in the Performance Analyzer trace diagrams, where `xxxxxxx` is the hex offset within the module of the function entry point. . If `__PROF_HOOKS` was set to `ENTRY_EXIT`, the interlanguage function call does not appear in the function trace diagrams.

If the non-C/C++ function in turn calls a C or C++ function, the called C or C++ function appears in the function trace if it was compiled with `TEST(HOOK)` and `__PROF_HOOKS` was set to `ALL` or `ENTRY_EXIT`.

In the class views of the Performance Analyzer diagrams, C function calls are included in a class called `C_Function` and calls to routines of programming languages other than C++ are included in a class called `Unknown_Language`.

RELATED CONCEPTS

“Function trace” on page 15

RELATED TASKS

“Creating a trace file” on page 30

“Compiling your program” on page 27

RELATED REFERENCES

“Run-time option for program tracing”

“Run-time environment variables for program tracing” on page 72

Run-time option for program tracing

To enable the tracing of a program during its execution, you must set the Language Environment run-time option, `PROFILE`, which has the following syntax:

```
PROFILE(ON|OFF,'string')
```

or

PROF (ON|OFF,'string')

The PROFILE run-time option has two suboptions:

- ON|OFF switch to turn on or off the trace for program execution
- 'string' to specify the type of tracing to be performed

Suboption ON|OFF

Specify ON to activate the Performance Analyzer tracing. Specify OFF if you do *not* want the Performance Analyzer to take any trace. You *must* specify the ON|OFF suboption.

Suboption String

The suboption *string* consists of a list of parameters enclosed in either single or double quotation marks. The parameters are separated by commas (,), and can occur in any order in the *string*. They can also be in any case or mixed case because all values are converted to uppercase before being processed. The parameters are provided in the following table:

Parameters	Abbreviation	Default	Description
FUNCTION=ALL or FUNCTION=COUNTS	F=A or F=C	F=A	<p>FUNCTION=ALL counts how many times each function is called, records what functions are called by each function and how many times they are called, provides total execution and stack times for each function, and records event data (every function call/return, thread creation and time of that event).</p> <p>FUNCTION=COUNTS collect a count of the number of times each function is called and what functions are called by each function and how many times they were called..</p> <p>Note: The TASK REAL suboption is ignored if FUNCTION=COUNTS is specified.</p>

Parameters	Abbreviation	Default	Description
TASK REAL	T R	T	<p>Specifies the type of time used during tracing. Specify TASK for CPU time or REAL for elapsed time.</p> <p>Note: This suboption is ignored if FUNCTION=COUNTS is specified.</p>

Note:

- Invalid parameters result in a warning message.
- Defaults are used for unspecified parameters.
- If conflicting parameters in the suboption *string* are specified, the last one is used. For example, if the suboption *string* is "REAL, FUNCTION=COUNTS, TASK, FUNCTION=ALL", the Performance Analyzer will use TASK and FUNCTION=ALL.

Multiple Specifications of PROFILE Options

Because run-time options can be specified a number of ways, multiple PROFILE settings may exist when the program executes. The *whole* suboption string is used as a single value. If the suboption string (including a null string) is specified in the run-time option, the whole suboption string is used. If not, the suboption string from #pragma runopts is used. If no suboption string is specified in #pragma runopts, the suboption string in the installation defaults is used. If the installation defaults do not have a PROFILE suboption string, then the default parameters (which are FUNCTION=ALL, and TASK) are used. The IBM supplied installation default for the PROFILE run-time option is PROFILE(OFF,"")

Note: When a suboption *string* is used, the Performance Analyzer fills in missing parameters in the string with defaults. For example if neither TASK nor REAL was specified in the suboption string, the Performance Analyzer uses TASK as the type of time when tracing.

Examples

1. **Installation Default:** PROFILE(OFF,'FUNCTION=COUNTS,REAL')
Program Code: #pragma runopts(prof(on))
Specified Run-time PROFILE Option: None
PROFILE Option Used for Tracing:
 PROFILE(ON,'FUNCTION=COUNTS,REAL')
Suboption String Used for Tracing:FUNCTION=COUNTS,REAL
2. **Installation Default:** PROFILE(ON,'FUNCTION=COUNTS,REAL')
Program Code: #pragma runopts(prof(off,'function'))

- Specified Run-time PROFILE Option:** prof(on)
PROFILE Option Used for Tracing: PROFILE(ON,'FUNCTION')
Suboptions String Used for Tracing: FUNCTION=ALL,TASK
3. **Installation Default:** PROFILE(OFF,'FUNCTION=COUNTS')
Program Code: No #pragma runopts was specified
Specified Run-time PROFILE Option: prof(on)
PROFILE Option Used for Tracing:
PROFILE(ON,'FUNCTION=COUNTS')
Suboption String Used for Tracing: FUNCTION=COUNTS,TASK
 4. **Installation Default:** PROFILE(OFF,"")
Program Code: #pragma runopts(prof(on,'f=c,real'))
Specified Run-time PROFILE Option: prof(on,'task')
PROFILE Option Used for Tracing: PROFILE(ON,'TASK')
Suboption String Used for Tracing: FUNCTION=ALL,TASK
 5. **Installation Default:** PROFILE(ON,'FUNCTION=COUNTS')
Program Code: #pragma runopts(prof(off,'function=counts,real'))
Specified Run-time PROFILE Option: prof(on,"")
PROFILE Option Used for Tracing: PROFILE(ON,"")
Suboption String Used for Tracing: FUNCTION=ALL,TASK

Refer to the Language Environment for OS/390 and VM Programming Reference for more information on run-time options.

RELATED CONCEPTS

- "Function trace" on page 15
- "Call frequency counting" on page 13

RELATED TASKS

- "Setting run-time option PROFILE for Performance Analyzer" on page 29
- "Specifying trace file name" on page 37
- "Creating a trace file" on page 30
- "Collecting call frequency data" on page 35

RELATED REFERENCES

- "Run-time environment variables for program tracing" on page 72
- "Sample JCL for creating trace files" on page 83
- "Sample TSO commands for creating trace files" on page 82
- "Sample Unix system service commands for creating trace files" on page 80

Run-time environment variables for program tracing

Before tracing a program during its execution, the following environment variable can be set:

`__PROF_FILE_NAME=filename`

Default:

The *filename* is *name.trc*, where *name* is the name of the executable or DLL which has the first main function is used as the *name*. If no main function is encountered, the name of the first executable or DLL encountered is used as the *name*.

When tracing an OS/390 UNIX System Services (formerly known as OpenEdition) application in the OS/390 UNIX shell, the file is written to the current directory.

When tracing an OS/390 batch or TSO application, the trace data is written to a sequential data set. A high level qualifier may be added to the file name, depending on the configuration of your system.

Description:

Use this environment variable to specify the name of the output trace file(s) created by the Performance Analyzer.

- **HFS output file**

When you are tracing an OS/390 UNIX application, you can specify a file name for the output trace file. If you want to write the file to a directory other than the current directory, specify a path with *filename*, for example,

`__PROF_FILE_NAME=/u/smith/trace.trc`.

- **Output to sequential data set**

If you want to force the output file to an OS/390 sequential data set, you can prefix *filename* with double slashes (`//`), for example `__PROF_FILE_NAME=//smith.trace`. The *filename* is prefixed with the *userid* as the high-level qualifier. You can specify a fully qualified *filename* by adding single quotation marks (`'`) around the filename, for example `__PROF_FILE_NAME=/'smith.trace'`.

`__PROF_WEBSERVER=NO | YES`

Default: NO

Description:

To trace the application running in a Lotus Domino Go Webserver environment, this variable must be set to YES. Otherwise, set this variable to NO.

Examples of Generated Trace Files

__PROF_FILE_NAME=	Generated Trace File For Program Running Under	
	OS/390 Batch/TSO	OS/390 UNIX
trace.ftrc	Sequential data set <i>USERID.TRACE.TRC</i>	HFS file ./trace.trc
trace	Sequential data set <i>USERID.TRACE</i>	HFS file ./trace
'test.trace.trc'	Sequential data set TEST.TRACE.TRC	HFS file ./'test.trace.trc'
/u/smith/test/trace.trc	HFS file /u/smith/test/trace.trc	HFS file /u/smith/test/trace.trc
//test.trace.trc	Sequential data set <i>USERID.TEST.TRACE.TRC</i>	Sequential data set <i>USERID.TEST.TRACE.TRC</i>
//'first.test.trace.trc'	Sequential data set FIRST.TEST.TRACE.TRC	Sequential data set FIRST.TEST.TRACE.TRC
./trace.trc	HFS file ./trace.trc	HFS file ./trace.trc

__PROF_HOOKS = ALL | ENTRY_EXIT | BEFORE_AFTER

Default: **__PROF_HOOKS = ALL**

Description

Use this environment variable to control which hooks are processed for function calls.

- **ENTRY_EXIT**

Trace data is only collected at entry and exit points of a function. Exercise caution when specifying **ENTRY_EXIT** because you may lose some trace information. For example, if you specified **ENTRY_EXIT**, and function B in another file was called, unless the function B's file was built with **TEST(HOOK)**, there would be no trace record indicating that function B was ever executed. This would be conspicuous if a whole DLL is built without the **TEST(HOOK)** compile option because none of the calls to the DLL would be recorded.

- **BEFORE_AFTER**

Trace data is collected before and after a function call.

- **ALL**

Events are processed before and after a function call and also at the entry and exit points of a function. If all files of a program are compiled with **TEST(HOOK)**, then tracing the program will be faster when you specify **BEFORE_AFTER** instead of **ALL** because fewer hooks are processed. However, the main function does not appear in the trace data because nothing calls it.

Refer to the OS/390 C/C++ Programming Guide and the OS/390 UNIX System Services User's Guide for more information on environment variables.

RELATED CONCEPTS

“Function trace” on page 15

“Call frequency counting” on page 13

RELATED TASKS

“Setting environment variables for Performance Analyzer” on page 28

“Collecting call frequency data” on page 35

“Creating a trace file” on page 30

RELATED REFERENCES

“Sample JCL for creating trace files” on page 83

“Sample TSO commands for creating trace files” on page 82

“Sample Unix system service commands for creating trace files” on page 80

Troubleshooting Performance Analyzer problems

No Trace File Created

By default, the Performance Analyzer creates the trace file as `xxxxxx.trc`, where `xxxxxx` is the name of the module in which the first main function is found. If the Performance Analyzer cannot find the main function, it creates a file named `module_name.trc`, where `module_name` is the name of the first executable or DLL encountered.

If you specified the environment variable `__PROF_FILE_NAME`, then the trace file will be created with the name you specified. If you cannot find the file, check the following:

- The PROFILE run-time option must be turned on.
- If your application is running with run-time option POSIX(ON), your trace file may be stored as an HFS file.
- If your application creates other processes with the fork or spawn functions, the names of the trace files created for the different processes of the application will have the process ID's appended to the name that was specified with `__PROF_FILE_NAME`.
- Your program must be compiled with the TEST(HOOK) option.
- Check for Performance Analyzer error messages in stdout or stderr.
- If an OS/390 UNIX kill command was used to stop the *process*, no trace files will be generated.
- If the operator used a cancel command to stop the *process*, no trace files will be generated.

ianalyze Not Found - Workstation Error

Ensure that the workstation part of the Performance Analyzer was installed properly. For more details on installing it on the workstation, see *Installing and Getting Started with OS/390 C/C++ Productivity Tools for Windows NT*.

Error Reading the Trace File - Workstation Error

If you are using the **ianalyze** command to display a graph, the following message may appear:

28104E: Error reading trace file,"trace file name"

it means that the trace file cannot be displayed. The cause of the error may be one or more of the situations described in the following table:

Situation	Response
The trace file was downloaded from the host machine as a text file instead of a binary file.	Download the file again as a binary file.
The version of the Performance Analyzer's host component is different from the version of the workstation component.	Ensure that both the host and workstation components of Performance Analyzer are at the most recent service level.
The workstation is out of storage.	Close other programs that are running and use the ianalyze command again.

29104E: No events were logged in trace file error - Workstation Error

You cannot open the Call Nesting, Time Line, or Execution Density diagrams with a trace file generated by specifying the call frequency counting sub-option **FUNCTION=COUNTS**. Only the Dynamic Call Graph diagram and Statistics diagram show call frequency information.

34002E: Error number 100 occurred - Workstation Error

This error message is issued when a diagram cannot be displayed because the data in the trace file includes a thread call depth of greater than 512. The Call Nesting, Execution Density, and Time Line diagrams do not support thread call depths greater than 512.

Loading Help Error - Workstation Error

When the Performance Analyzer is invoked at the workstation to analyze a trace file, the following error message may appear: "Failed to load Help Manager. IPTPW10.HLP must be in your **HELP** and **DPATH** environment variables" The cause of the error is usually a user **HELP** variable setting that

does not include %HELP%. The installation of OS/390 C/C++ Productivity Tools sets a system HELP environment variable to include the path containing the required file. If you have specified a user HELP variable, add %HELP% at the end of your path specifications. Note: The DPATH environment variable does not need to be set.

RELATED CONCEPTS

“Function trace” on page 15

“Call frequency counting” on page 13

RELATED TASKS

“Creating a trace file” on page 30

“Collecting call frequency data” on page 35

RELATED REFERENCES

“Performance Analyzer error messages on the host”

“Run-time option for program tracing” on page 68

“Run-time environment variables for program tracing” on page 72

“Limitations when creating a trace” on page 65

Performance Analyzer error messages on the host

These are error messages that the Performance Analyzer generates during tracing:

CTV0001

Cannot allocate memory.

Explanation:

The Performance Analyzer is unable to acquire some heap storage and cannot continue.

Programmer Response:

Run the program with a larger storage region.

CTV0002

Invalid value specified for environment variable%1. The default value %2 is used.

Explanation:

%1 is the environment variable name and %2 is the default environment variable value. An invalid value was specified for the indicated environment variable. The default value is used instead. Tracing continues.

Programmer Response:

Ensure that the environment variable is set to the desired value and try again.

CTV0004

Cannot create data space.

Explanation:

Data space creation failed. Data spaces might not be supported by the system. Data space support is required to run the Performance Analyzer. Tracing is discontinued.

Programmer Response:

Contact your systems administrator to determine whether or not data spaces are supported on your system.

CTV0005

Cannot release storage.

Explanation:

A failure occurred while trying to release data space or address space storage. If this condition continues, a storage shortage may occur.

Programmer Response:

Try tracing again. If this message continues to reappear, notify your systems administrator of this problem.

CTV0007

Tracing cannot continue due to a Performance Analyzer internal error.

Explanation:

The Performance Analyzer has encountered inconsistent data and cannot continue processing the data. If a trace file is generated, it is incomplete.

Programmer Response:

Try tracing again. If this message continues to reappear, contact your IBM representative to report this problem.

CTV0009

Cannot open iconv table.

Explanation:

A failure occurred trying to open the iconv conversion table of the Language Environment Run-Time Library. Tracing is stopped and no trace file is created.

Programmer Response:

Try tracing again. If this message continues to reappear, contact your systems administrator to ensure that the Language Environment run time is installed properly and available to your program.

CTV0010

Cannot open the trace file %1.

Explanation:

%1 is the trace file name. The specified trace file cannot be opened for writing. Tracing is stopped and no trace file is created.

Programmer Response:

Ensure that space is available in the file system or volume and that you have the correct permissions to create the file.

CTV0011

Cannot write to the trace file.

Explanation:

A failure occurred writing to the Performance Analyzer trace file. Tracing is stopped and the trace file is incomplete.

Programmer Response:

Ensure that the file system or volume is operational and not full, then try tracing again. If this message continues to reappear, contact your IBM representative to report the problem.

CTV0012

Cannot convert string to ASCII.

Explanation:

A failure occurred doing code page conversion using the iconv functions of the Language Environment Run-Time Library. Tracing is stopped and the Performance Analyzer trace is incomplete.

Programmer Response:

Try tracing again. If this message continues to reappear, contact your systems administrator to ensure that the Language Environment run time is installed properly and available to your program.

CTV0013

Cannot read from data space.

Explanation:

Reading from a data space failed. Tracing continues but the Performance Analyzer trace is incomplete. A serious system problem may exist.

Programmer Response:

If this message continues to be reappear, a serious system problem may exist. Contact your systems administrator to ensure that data space support is operational. Try tracing again. If the problem persists, contact your IBM representative to report the problem.

CTV0015

%1 is an invalid Performance Analyzer option and is ignored.

Explanation:

%1 is the invalid option that was specified. An invalid Performance Analyzer option was specified. The option is ignored and tracing continues.

Programmer Response:

Specify valid Performance Analyzer options in the "PROFILE" run-time option and try again.

CTV0016

The %1 feature of OS/390 is not enabled. Contact your system programmer.

Explanation:

%1 is the name of the OS/390 feature. This feature of OS/390 is not enabled at your installation. This feature is required in order to use the Performance Analyzer. Your system programmer can contact IBM OS/390 service to have this element enabled.

Programmer Response:

Contact your system programmer to have this feature enabled.

CTV0017

No tracing data is available because no hooks were encountered.

Explanation:

No tracing hooks were found in the program code. The program cannot be traced.

Programmer Response:

Compile the program code with the TEST(HOOK) option and try tracing again.

CTV0018

Cannot write to data space.

Explanation:

Writing to a data space failed. Tracing continues but is incomplete. A serious system problem may exist.

Programmer Response:

If this message continues to reappear, a serious system problem may exist. Contact your systems administrator to ensure that data space support is operational. Try tracing again. If the problem persists, contact your IBM representative to report the problem.

CTV0019

Data space is full and it cannot be extended.

Explanation:

Extending a data space failed. Tracing continues but is incomplete. A serious system problem may exist.

Programmer Response:

If this message continues to reappear, a serious system problem may exist. Contact your systems administrator to ensure that data space support is operational. Try tracing again. If the problem persists, contact your IBM representative to report the problem.

RELATED CONCEPTS

“Function trace” on page 15

RELATED TASKS

“Creating a trace file” on page 30

RELATED REFERENCES

“Troubleshooting Performance Analyzer problems” on page 74

Sample Unix system service commands for creating trace files

Here are some examples of OS/390 UNIX shell commands used to compile and execute a program and turn on the Performance Analyzer for function tracing.

Compile and Bind C Programs for Tracing

Use the **c89** command to compile and bind test.c:

```
c89 -o ./test -0 -Wc,"TEST(HOOK),NOGONUMBER" test.c
```

Compile and Bind C++ Programs for Tracing

Use the **c++** command to compile and bind test.cxx:

```
c++ -o ./test -0 -Wc,"TEST(HOOK),NOGONUMBER" test.cxx
```

Set Run-Time PROFILE Option and Environment Variables

Use the export command for setting run-time options and environment variables, for example:

```
export _CEE_RUNOPTS="PROFILE(ON,'FUNCTION=ALL,REAL')"  
export __PROF_FILE_NAME=./test.trc
```

Set the STEPLIB environment variable so that Language Environment can find the Performance Analyzer module (only required if it is not included in the Link Pack Area):

```
export STEPLIB=IBC.SCTVMOD:$STEPLIB
```

Execute Your Program and Start Performance Analyzer

Start your program as follows, and the Performance Analyzer tracing will begin at the same time:

```
test
```

Turn off tracing after running your program by setting the _CEE_RUNOPTS environment variable as follows:

```
export _CEE_RUNOPTS="PROFILE(OFF)"
```

Note: If tracing is not turned off in this way, any other program that is executed in the current shell will be traced, including some OS/390 UNIX shell commands.

Refer to the *OS/390 UNIX System Services Command Reference* for more information on the use of the OS/390 UNIX commands.

RELATED CONCEPTS

“Function trace” on page 15

RELATED TASKS

“Compiling your program” on page 27

“Setting environment variables for Performance Analyzer” on page 28

“Setting run-time option PROFILE for Performance Analyzer” on page 29

RELATED REFERENCES

“Sample JCL for creating trace files” on page 83

“Sample TSO commands for creating trace files” on page 82

“Sample trace file names from tracing a multiprocess program” on page 85

Sample TSO commands for creating trace files

Here are some examples of TSO commands used to compile and execute your program and turn on the Performance Analyzer for function tracing.

Compile and Bind C Programs for Tracing

Use the following to compile and bind your C program:

```
cc testprof.c(testcpgm) (test(hook) nogonumber search('cee.sceeh.+') obj(testpgm.obj(test  
cxxbind obj(testpgm.obj(testcpgm)) load(testpgm.load(testcpgm))
```

Compile and Bind C++ Programs for Tracing

Use the following to compile and bind your C++ program:

```
cxx testprof.cpp(testcpp) (test(hook) nogonumber lsearch(testprof.hpp)  
se('cbc.sclbh.+','cee.sceeh.+') obj(testpgm.obj(testcpp))  
cxxbind obj(testpgm.obj(testcpp)) load(testpgm.load(testcpp))
```

Set Run-Time PROFILE Option and Environment Variables and Run Your Program

You use the following to set run-time options and environment variables and run your program. In this case, the program being executed is testpgm.load(testprof).

```
tsolib act dsname('CBC.SCTVMOD')  
call testpgm(testprof) 'PROFILE(ON,"F=A,R"),ENVAR(__PROF_FILE_NAME=testprof.trace)/'
```

Refer to the *OS/390 C/C++ User's Guide* for more information on compiling and running your applications.

RELATED CONCEPTS

“Function trace” on page 15

RELATED TASKS

“Compiling your program” on page 27

“Setting environment variables for Performance Analyzer” on page 28

“Setting run-time option PROFILE for Performance Analyzer” on page 29

RELATED REFERENCES

“Sample JCL for creating trace files” on page 83

“Sample Unix system service commands for creating trace files” on page 80

“Sample trace file names from tracing a multiprocess program” on page 85

Sample JCL for creating trace files

You can customize this sample JCL to execute your program and turn on the Performance Analyzer for function tracing. The JCL compiles, binds, and traces the sample program CBC3GDC1 that comes with the OS/390 C/C++ Compiler.

```
//PROFFUNC JOB 1,'PP5647-A01',MSGLEVEL=(1,1),MSGCLASS=A
// SET #CPP=CBC
// SET #PA=CBC
// SET #LE=CEE
//PROC JCLLIB ORDER=(&#CPP..SCBCPRC,
// &#LE..SCEEPROC)
//-----
/* PROFFUNC - OS/390 C/C++ Performance Analyzer Sample JCL For
/*      Function Level Trace
/*
/* COPYRIGHT:
/* LICENSED MATERIALS - PROPERTY OF IBM.
/*
/* 5647-A01
/* (C) COPYRIGHT IBM CORP. 1997,1999 ALL RIGHTS RESERVED
/* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,
/* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA
/* ADP SCHEDULE CONTRACT WITH IBM CORP.
/*
/* INSTRUCTIONS:
/* Before submitting this job, the JCL must be customized
/* for your installation. The following changes need to be
/* made:
/*
/* 1. Update the JOB card with the installation specific
/* parameters.
/* 2. If you chose to use a different prefix than the IBM supplied
/* one for the C/C++ Compiler, please change the value of CBC
/* to your chosen prefix on the // SET #CPP=CBC statement.
/* 3. If you chose to use a different prefix than the IBM supplied
/* one for the C/C++ Host Performance Analyzer, please change
/* the value of CBC to your chosen prefix on the // SET #PA=CBC
/* statement.
/* 4. If you chose to use a different prefix than the IBM supplied
/* one for the Language Environment, please change the value
/* of CEE to your chosen prefix on the // SET #LE=CEE
/* statement.
/* 5. If you have installed Kanji Messages for the C/C++ Compiler
/* on your system and want to enable it, uncomment the CRUN
/* line.
/* 6. You may have to change the unit TUNIT='VIO' to your
/* locally-defined esoteric name.
/*
/* REQUIRED ENVIRONMENT:
/* 1. C/C++ Compiler and Language Environment must be installed on
/* the system prior to execution of this JCL.
/*
```

```

/* INPUT:
/*   1. Input data set: CBC.SCBCSAM(CBC3GDC1).
/*
/* OUTPUT:
/*   1. Return code of zero for all steps.
/*   2. Function trace file yourid.CBC3GDC1.FUNCTION.TRC is
/*      generated.
/*   3. Output from the program:
/*      res_add =11.87655
/*      res_sub =0.34
/*      res_mul =-1.4814000
/*      res_div =1.12079927338782
/*-----
//PROFTST EXEC EDCCBG,
//      CPARM='OPTFILE(DD:OPTION)',
//      TUNIT='VIO',
//      LIBPRFX=&#LE.,
//      LNGPRFX=&#CPP.,
/*      CRUN='NATLANG(JPN)',
//      INFILE=&#CPP..SCBCSAM(CBC3GDC1),
//      OUTFILE='&&GSET(GO),DISP=(NEW,PASS),SPACE=(TRK,(7,7,1))',
//      GPARM='PROFILE(ON,"FUNC=A,T"),ENVAR("_CEE_ENVFILE=DD:MYVARS")/'
//OPTION DD *
LIST
TEST(HOOK)
NOGONUMBER
OPT(1)
/*
//BIND.SYSLMOD DD DSNAME=&OUTFILE,UNIT=&TUNIT.
//GO.MYVARS DD *
__PROF_FILE_NAME=CBC3GDC1.FUNCTION.TRC
__PROF_HOOKS=ALL
/*
//GO.STEPLIB DD
//      DD DSN=&#PA..SCTVMOD,DISP=SHR
/*      =====> END OF JOB PROFFUNC <=====

```

RELATED CONCEPTS

“Function trace” on page 15

RELATED TASKS

“Compiling your program” on page 27

“Setting environment variables for Performance Analyzer” on page 28

“Setting run-time option PROFILE for Performance Analyzer” on page 29

RELATED REFERENCES

“Sample TSO commands for creating trace files” on page 82

“Sample Unix system service commands for creating trace files” on page 80

“Sample trace file names from tracing a multiprocess program” on page 85

Sample trace file names from tracing a multiprocess program

In this example, program prog1 calls the fork function and the resulting child process calls the exec function to execute program prog2:

```
/* prog1.c */
int main()
{
    /* start of program */
    /* ...some code...*/
    if ((pid = fork()) < 0) {
        perror("fork failed");
        exit(2);
    }
    if (pid == (pid_t)0) { /* CHILD process */
        execl("./prog2", NULL);
        perror("The execl() call must have failed");
        exit(255); /* return failure to parent */
    }
    else { /* PARENT process */
        pid = wait(&c_status);
        if (WIFEXITED(c_status)) {
            printf("\nchild exited with code
%d\n", WEXITSTATUS(c_status));
        }
        else
            puts("\nchild did not exit successfully\n");
    }
    exit(0);
}
```

Output Trace Files Default Names

If `__PROF_FILE_NAME` is not set, the following trace files are produced:

- prog1.trc.11111111 - parent process (PID=11111111)
- prog1.trc.22222222 - child process (PID=22222222)
- prog2.trc - PID=22222222

Note:

1. Because the `__PROF_FILE_NAME` environment variable was not set, the trace file is named after the program that contains the main function, prog1, and the default file extension, .trc, is added.
2. With the fork function a new process is created, and to distinguish the trace file for the second process from the first one, the process ID (PID) number is appended at the end of each trace file's name.
3. Then, the exec function is executed and a new trace file is created. The exec function replaces the current process image with a new process image

without changing the PID number. The name of the new program that contains the main function, prog2, is used as the file name. Because the PID number did not change, the PID number is not appended to the new trace file's name.

Output Trace Files That Are Explicitly Named

If `__PROF_FILE_NAME=prog1.trace` is specified, the following trace files are produced:

- `prog1.trace.11111112` - parent process (PID=11111112)
- `prog1.trace.22222223` - child process (PID=22222223)
- `prog1.trace` - PID=22222223

Note:

1. Because the `__PROF_FILE_NAME` environment variable is set to `prog1.trace`, the `prog1.trace` is used for all processes, including the process that executes the prog2 program..
2. With the fork function a new process is created and to distinguish the trace file for the second process from the first one, the process ID (PID) number is appended at the end of each trace file's name.
3. Then, the exec function is executed and a new trace file is created. The exec function replaces the current process image with a new process image without changing the PID number. Because the PID number did not change, the PID number is not appended to the new trace file's name.

RELATED CONCEPTS

“Function trace” on page 15

“Multiple process support” on page 17

RELATED TASKS

“Setting environment variables for Performance Analyzer” on page 28

RELATED REFERENCES

“Sample JCL for creating trace files” on page 83

“Sample TSO commands for creating trace files” on page 82

“Sample Unix system service commands for creating trace files” on page 80

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
Intellectual Property and Licensing
IBM Corporation
North Castle Drive, MD-NC 119
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS DOCUMENT "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will

be incorporated in new editions of the document. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd.
1150 Eglinton Ave E
Toronto, Ontario, M3C 1H7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks and service marks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States, other countries, or both:

CICS	IMS
CICS/ESA	Language Environment
CICS/MVS	MVS/ESA
CICS/VSE	OS/390
DB2	S/390
IBM	

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark of The Open Group in the United States and/or other countries licensed exclusively through X/Open Company Limited.

Sun, SunLink, Solaris, SunOS, Java, all Java-based trademarks and logos, NFS, and Sun Microsystems are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP, AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARDS TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Other company, product, and service names may be trademarks or service marks of others.