AIX 5L Version 5.2

Performance Tools Guide and Reference

AIX 5L Version 5.2

Performance Tools Guide and Reference

Note
Before using this information and the product it supports, read the information in “Notices” on page 147.

First Edition (October 2002)

This edition applies to AIX 5L Version 5.2 and to all subsequent releases of this product until otherwise indicated in new editions.

A reader’s comment form is provided at the back of this publication. If the form has been removed, address comments to Information Development, Department H6DS-905-6C006, 11501 Burnet Road, Austin, Texas 78758-3493. To send comments electronically, use this commercial Internet address: aix6kpub@austin.ibm.com. Any information that you supply may be used without incurring any obligation to you.

Contents

About This Book

This book provides information on performance tools and application programming interfaces (APIs) for the AIX operating system.

The information contained in this book pertains to systems running AIX 5.2 or later. Any content that is applicable to earlier releases will be noted as such.

Who Should Use This Book

This book is intended for network administrators, system administrators, experienced system administrators, system engineers, and application programmers who are concerned with the performance of their system and the applications running on that system.

Highlighting

The following highlighting conventions are used in this book:

Bold	Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects.
Italics	Identifies parameters whose actual names or values are to be supplied by the user.
`Monospace`	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

Case-Sensitivity in AIX

Everything in the AIX operating system is case-sensitive, which means that it distinguishes between uppercase and lowercase letters. For example, you can use the **ls** command to list files. If you type `LS`, the system responds that the command is ″not found.″ Likewise, **FILEA**, **FiLea**, and **filea** are three distinct file names, even if they reside in the same directory. To avoid causing undesirable actions to be performed, always ensure that you use the correct case.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

Related Publications

The following books contain information about or related to performance monitoring:

AIX 5L Version 5.2 Performance Management Guide

Performance Toolbox Version 2 and 3 for AIX: Guide and Reference

Chapter 1. Introduction to Performance Tools and APIs

The performance of a computer system is based on human expectations and the ability of the computer system to fulfill these expectations. The objective for performance tuning is to make those expectations and their fulfillment match. The path to achieving this objective is a balance between appropriate expectations and optimizing the available system resources. The performance-tuning process demands great skill, knowledge, and experience, and cannot be performed by only analyzing statistics, graphs, and figures. If results are to be achieved, the human aspect of perceived performance must not be neglected. Performance tuning must also usually take into consideration problem-determination aspects as well as pure performance issues.

Expectations can often be classified as either of the following:

Throughput expectations	A measure of the amount of work performed over a period of time
Response time expectations	The elapsed time between when a request is submitted and when the response from that request is returned

The performance-tuning process can be initiated for a number of reasons:

- To achieve optimal performance in a newly installed system
- To resolve performance problems resulting from the design (sizing) phase
- To resolve performance problems occurring in the run-time (production) phase

Performance tuning on a newly installed system usually involves setting some base parameters for the operating system and applications. The sections in this chapter describe the characteristics of different system resources and provide guidelines regarding their base tuning parameters, if applicable.

Limitations originating from the sizing phase will either limit the possibility of tuning, or incur greater cost to overcome them. The system may not meet the original performance expectations because of unrealistic expectations, physical problems in the computer environment, or human error in the design or implementation of the system. In the worst case, adding or replacing hardware might be necessary. Be particularly careful when sizing a system to allow enough capacity for unexpected system loads. In other words, do not design the system to be 100 percent busy from the start of the project.

When a system in a productive environment still meets the performance expectations for which it was initially designed, but the demands and needs of the utilizing organization have outgrown the system's basic capacity, performance tuning is performed to delay or even to avoid the cost of adding or replacing hardware.

Many performance-related issues can be traced back to operations performed by a person with limited experience and knowledge who unintentionally restricted some vital logical or physical resource of the system.

Chapter 2. X-Windows Performance Profiler (Xprofiler)

The X-Windows Performance Profiler (Xprofiler) tool helps you analyze your parallel or serial application's performance. It uses procedure-profiling information to construct a graphical display of the functions within your application. Xprofiler provides quick access to the profiled data, which lets you identify the functions that are the most CPU-intensive. The graphical user interface (GUI) also lets you manipulate the display in order to focus on the application's critical areas.

The following Xprofiler topics are covered in this chapter:

- Before You Begin
- Xprofiler installation information
- Starting the Xprofiler GUI
- Customizing Xprofiler resources

The word *function* is used frequently throughout this chapter. Consider it to be synonymous with the terms *routine, subroutine,* and *procedure.*

Before you begin

About Xprofiler

Xprofiler lets you profile both serial and parallel applications. Serial applications generate a single profile data file, while a parallel application produces multiple profile data files. You can use Xprofiler to analyze the resulting profiling information.

Xprofiler provides a set of resource variables that let you customize some of the features of the Xprofiler window and reports.

Requirements and limitations

To use Xprofiler, your application must be compiled with the **-pg** flag. For more information, see "Compiling Applications to be Profiled" on page 4.

Like the **gprof** command, Xprofiler lets you analyze CPU (busy) usage only. It does not provide other kinds of information, such as CPU idle, I/O, or communication information.

If you compile your application on one processor, and then analyze it on another, you must first make sure that both processors have similar library configurations, at least for the system libraries used by the application. For example, if you run a High Performance Fortran application on a server, then try to analyze the profiled data on a workstation, the levels of High Performance Fortran run-time libraries must match and must be placed in a location on the workstation that Xprofiler recognizes. Otherwise, Xprofiler produces unpredictable results.

Because Xprofiler collects data by sampling, functions that run for a short amount of time may not show any CPU use.

Xprofiler does not give you information about the specific threads in a multi-threaded program. Xprofiler presents the data as a summary of the activities of all the threads.

Comparing Xprofiler and the gprof Command

With Xprofiler, you can produce the same tabular reports that you may be accustomed to seeing with the **gprof** command. As with **gprof**, you can generate the Flat Profile, Call Graph Profile, and Function Index reports.

Unlike **gprof**, Xprofiler provides a GUI that you can use to profile your application. Xprofiler generates a graphical display of your application's performance, as opposed to a text-based report. Xprofiler also lets you profile your application at the source statement level.

From the Xprofiler GUI, you can use all of the same command line flags as **gprof**, as well as some additional flags that are unique to Xprofiler.

Compiling Applications to be Profiled

To use Xprofiler, you must compile and link your application with the **-pg** flag of the compiler command. This applies regardless of whether you are compiling a serial or parallel application. You can compile and link your application all at once, or perform the compile and link operations separately. The following is an example of how you would compile and link all at once:

```
cc -pg -o foo foo.c
```

The following is an example of how you would first compile your application and then link it. To compile, do the following:

```
cc -pg -c foo.c
```

To link, do the following:

```
cc -pg -o foo foo.o
```

Notice that when you compile and link separately, you must use the **-pg** flag with *both* the compile and link commands.

The **-pg** flag compiles and links the application so that when you run it, the CPU usage data is written to one or more output files. For a serial application, this output consists of only one file called **gmon.out**, by default. For parallel applications, the output is written into multiple files, one for each task that is running in the application. To prevent each output file from overwriting the others, the task ID is appended to each **gmon.out** file (for example: **gmon.out.10**).

Note: The **-pg** flag is not a combination of the **-p** and the **-g** compiling flags.

To get a complete picture of your parallel application's performance, you must indicate all of its **gmon.out** files when you load the application into Xprofiler. When you specify more than one **gmon.out** file, Xprofiler shows you the sum of the profile information contained in each file.

The Xprofiler GUI lets you view included functions. Your application must also be compiled with the **-g** flag in order for Xprofiler to display the included functions.

In addition to the **-pg** flag, the **-g** flag is also required for source-statement profiling.

Xprofiler Installation Information

This section contains Xprofiler system requirements, limitations, and information about installing Xprofiler. It also lists the files and directories that are created by installing Xprofiler.

Preinstallation Information

The following are hardware and software requirements for Xprofiler:

Software requirements:

- X-Windows
- X11.Dt.lib 4.2.1.0 or later, if you want to run Xprofiler in the Common Desktop Environment (CDE)

Disk space requirements:

- 6500 512-byte blocks in the **/usr** directory

Limitations
Although it is not required to install Xprofiler on every node, it is advisable to install it on at least one node in each group of nodes that have the same software library levels.

If users plan to collect a **gmon.out** file on one processor and then use Xprofiler to analyze the data on another processor, they should be aware that some shared (system) libraries may not be the same on the two processors. This situation may result in different function-call tree displays for shared libraries.

Installing Xprofiler
There are two methods to install Xprofiler. One method is by using the **installp** command. The other is by using SMIT.

Using the installp Command
To install Xprofiler, type:

```
installp -a -I -X -d device_name xprofiler
```

Using SMIT
To install Xprofiler using SMIT, do the following:

1. Insert the distribution media in the installation device (unless you are installing over a network).
2. Enter the following:

   ```
   smit install_latest
   ```

   This command opens the SMIT panel for installing software.
3. Press **List**. A panel lists the available INPUT devices and directories for software.
4. Select the installation device or directory from the list of available INPUT devices. The original SMIT panel indicates your selection.
5. Press **Do**. The SMIT panel displays the default installation parameters.
6. Type:

   ```
   xprofiler
   ```

   in the **SOFTWARE to install** field and press **Enter**.
7. Once the installation is complete, press **F10** to exit SMIT.

Directories and Files Created by Xprofiler
Installing Xprofiler creates the directories and files shown in the following table:

Table 1. Xprofiler directories and files installed

Directory or file	Description
/usr/lib/nls/msg/En_US/xprofiler.cat **/usr/lib/nls/msg/en_US/xprofiler.cat** **/usr/lib/nls/msg/C/xprofiler.cat**	Message catalog for Xprofiler
/usr/xprofiler/defaults/Xprofiler.ad	Defaults file for X-Windows and Motif resource variables
/usr/xprofiler/bin/.startup_script	Startup script for Xprofiler
/usr/xprofiler/bin/xprofiler	Xprofiler exec file
/usr/xprofiler/help/en_US/xprofiler.sdl **/usr/xprofiler/help/en_US/xprofiler_msg.sdl** **/usr/xprofiler/help/en_US/graphics**	Online help

Table 1. Xprofiler directories and files installed (continued)

Directory or file	Description
/usr/xprofiler/README5/xprofiler.README	Installation readme file
/usr/xprofiler/samples	Directory containing sample programs

The following symbolic link is made during the installation process of Xprofiler:

This link:	To:
/usr/lpp/X11/lib/X11/app-defaults/Xprofiler	**/usr/xprofiler/defaults/Xprofiler.ad**
/usr/bin/xprofiler	**/usr/xprofiler/bin.startup_script**

Starting the Xprofiler GUI

To start Xprofiler, enter the **xprofiler** command on the command line. You must also specify the binary executable file, one or more profile data files, and optionally, one or more flags, which you can do in one of two ways. You can either specify the files and flags on the command line along with the **xprofiler** command, or you can enter the **xprofiler** command alone, then specify the files and flags from within the GUI.

You will have more than one **gmon.out** file if you are profiling a parallel application, because a **gmon.out** file is created for each task in the application when it is run. If you are running a serial application, there may be times when you want to summarize the profiling results from multiple runs of the application. In these cases, you must specify each of the profile data files you want to profile with Xprofiler.

To start Xprofiler and specify the binary executable file, one or more profile data files, and one or more flags, type:

```
xprofiler a.out gmon.out... [flag...]
```

where: **a.out** is the binary executable file, **gmon.out**... is the name of your profile data file (or files), and *flag*... is one or more of the flags listed in the following section on Xprofiler command-line flags.

Xprofiler Command-line Flags

You can specify the same command-line flags with the **xprofiler** command that you do with **gprof**, as well as one additional flag (**-disp_max**), which is specific to Xprofiler. The command-line flags let you control the way Xprofiler displays the profiled output.

You can specify the flags in Table 2 from the command line or from the Xprofiler GUI (see “Specifying command line options (from the GUI)” on page 13 for more information).

Table 2. Xprofiler command-line flags

Use this flag:	To:	For example:
-a	Add alternative paths to search for source code and library files, or changes the current path search order. When using this flag, you can use the ″at″ symbol (**@**) to represent the default file path, in order to specify that other paths be searched before the default path.	To set an alternative file search path so that Xprofiler searches **pathA**, the default path, then **pathB**, type: `xprofiler -a pathA:@:pathB`
-b	Suppress the printing of the field descriptions for the **Flat Profile**, **Call Graph Profile**, and **Function Index** reports when they are written to a file with the **Save As** option of the **File** menu.	Type: `xprofiler -b a.out gmon.out`

Table 2. Xprofiler command-line flags (continued)

Use this flag:	To:	For example:
-c	Load the specified configuration file. If this flag is used on the command line, the configuration file name specified with it will appear in the **Configuration File (-c):** text field in **Load Files Dialog** window and in the **Selection** field of the **Load Configuration File Dialog** window. When both the **-c** and **-disp_max** flags are specified on the command line, the **-disp_max** flag is ignored, but the value that was specified with it will appear in the **Initial Display (-disp_max):** field in the **Load Files Dialog** window the next time this window is opened.	To load the configuration file **myfile.cfg**, type: `xprofiler a.out gmon.out -c myfile.cfg`
-disp_max	Set the number of function boxes that Xprofiler initially displays in the function call tree. The value supplied with this flag can be any integer between 0 and 5000. Xprofiler displays the function boxes for the most CPU-intensive functions through the number you specify. For example, if you specify 50, Xprofiler displays the function boxes for the 50 functions in your program with the highest CPU usage. After this, you can change the number of function boxes that are displayed using the **Filter** menu options. This flag has no effect on the content of any of the Xprofiler reports.	To display the function boxes for the 50 most CPU-intensive functions in the function call tree, type: `xprofiler -disp_max 50 a.out gmon.out`
-e	Deemphasize the general appearance of the function box for the specified function in the function call tree, and limits the number of entries for this function in the **Call Graph Profile** report. This also applies to the specified function's descendants, as long as they have not been called by non-specified functions. In the function call tree, the function box for the specified function is made unavailable. The box size and the content of the label remain the same. This also applies to descendant functions, as long as they have not been called by non-specified functions. In the **Call Graph Profile** report, an entry for a specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. The information for this entry remains unchanged. Entries for descendants of the specified function do not appear unless they have been called by at least one non-specified function in the program.	To deemphasize the appearance of the function boxes for **foo** and **bar** and their qualifying descendants in the function call tree, and limit their entries in the **Call Graph Profile** report, type: `xprofiler -e foo -e bar a.out gmon.out`

Table 2. Xprofiler command-line flags (continued)

Use this flag:	To:	For example:
-E	Change the general appearance and label information of the function box for the specified function in the function call tree. This flag also limits the number of entries for this function in the **Call Graph Profile** report, and changes the CPU data associated with them. These results also apply to the specified function's descendants, as long as they have not been called by non-specified functions in the program. In the function call tree, the function box for the specified function is made unavailable, and the box size and shape also changes so that it appears as a square of the smallest allowable size. In addition, the CPU time shown in the function box label, appears as **0**. The same applies to function boxes for descendant functions, as long as they have not been called by non-specified functions. This flag also causes the CPU time spent by the specified function to be deducted from the CPU total on the left in the label of the function box for each of the specified function's ancestors. In the **Call Graph Profile** report, an entry for the specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. When this is the case, the time in the **self** and **descendants** columns for this entry is set to **0**. In addition, the amount of time that was in the **descendants** column for the specified function is subtracted from the time listed under the **descendants** column for the profiled function. As a result, be aware that the value listed in the **% time** column for most profiled functions in this report will change.	To change the display and label information for **foo** and **bar**, as well as their qualifying descendants in the function call tree, and limit their entries and data in the **Call Graph Profile** report, type: `xprofiler -E foo -E bar a.out gmon.out`
-f	Deemphasize the general appearance of all function boxes in the function call tree, *except* for that of the specified function and its descendants. In addition, the number of entries in the **Call Graph Profile** report for the non-specified functions and non-descendant functions is limited. The **-f** flag overrides the **-e** flag. In the function call tree, all function boxes *except* for that of the specified function and its descendants are made unavailable. The size of these boxes and the content of their labels remain the same. For the specified function and its descendants, the appearance of the function boxes and labels remain the same. In the **Call Graph Profile** report, an entry for a non-specified or non-descendant function only appears where it is a parent or child of a specified function or one of its descendants. All information for this entry remains the same.	To deemphasize the display of function boxes for all functions in the function call tree *except* for **foo**, **bar**, and their descendants, and limit their types of entries in the **Call Graph Profile** report, type: `xprofiler -f foo -f bar a.out gmon.out`

Table 2. Xprofiler command-line flags (continued)

Use this flag:	To:	For example:
-F	Change the general appearance and label information of all function boxes in the function call tree *except* for that of the specified function and its descendants. In addition, the number of entries in the **Call Graph Profile** report for the non-specified and non-descendant functions is limited, and the CPU data associated with them is changed. The **-F** flag overrides the **-E** flag. In the function call tree, the function box for the specified function are made unavailable, and its size and shape also changes so that it appears as a square of the smallest allowable size. In addition, the CPU time shown in the function box label, appears as **0**. In the **Call Graph Profile** report, an entry for a non-specified or non-descendant function only appears where it is a parent or child of a specified function or one of its descendants. When this is the case, the time in the **self** and **descendants** columns for this entry is set to **0**. As a result, be aware that the value listed in the **% time** column for most profiled functions in this report will change.	To change the display and label information of the function boxes for all functions *except* the functions **foo** and **bar** and their descendants, and limit their types of entries and data in the **Call Graph Profile** report, type: `xprofiler -F foo -F bar a.out gmon.out`
-h | **-?**	Display the **xprofiler** command's usage statement.	`xprofiler -h` Usage: `xprofiler [program] [-b] [-h] [-s] [-z] [-a path(s)] [-c file] [-L pathname] [[-e function]...] [[-E function]...] [[-f function]...] [[-F function]...] [-disp_max number_of_functions] [[gmon.out]...]`
-L	Specify an alternative path name for locating shared libraries. If you plan to specify multiple paths, use the **Set File Search Path** option of the **File** menu on the Xprofiler GUI. See "Setting the file search sequence" on page 19 for more information.	To specify **/lib/profiled/libc.a:shr.o** as an alternative path name for your shared libraries, type: `xprofiler -L /lib/profiled/libc.a:shr.o`
-s	Produce the **gmon.sum** profile data file (if multiple **gmon.out** files are specified when Xprofiler is started). The **gmon.sum** file represents the sum of the profile information in all the specified profile files. Note that if you specify a single **gmon.out** file, the **gmon.sum** file contains the same data as the **gmon.out** file.	To write the sum of the data from three profile data files, **gmon.out.1**, **gmon.out.2**, and **gmon.out.3**, into a file called **gmon.sum**, type: `xprofiler -s a.out gmon.out.1 gmon.out.2 gmon.out.3`
-z	Include functions that have both zero CPU usage and no call counts in the **Flat Profile**, **Call Graph Profile**, and **Function Index** reports. A function will not have a call count if the file that contains its definition was not compiled with the **-pg** flag, which is common with system library files.	To include all functions used by the application that have zero CPU usage and no call counts in the **Flat Profile**, **Call Graph Profile**, and **Function Index** reports, type: `xprofiler -z a.out gmon.out`

After you enter the **xprofiler** command, the Xprofiler main window appears and displays your application's data.

Loading files from the Xprofiler GUI

If you enter the **xprofiler** command on its own, you can then specify an executable file, one or more profile data file, and any flags, from within the Xprofiler GUI. You use the **Load File** option of the **File** menu to do this.

When you enter the **xprofiler** command alone, the Xprofiler main window appears. Because you did not load an executable file or specify a profile data file, the window will be empty.

If you enter the **xprofiler -h** or **xprofiler -?** command, Xprofiler displays the usage statement for the command and then exits.

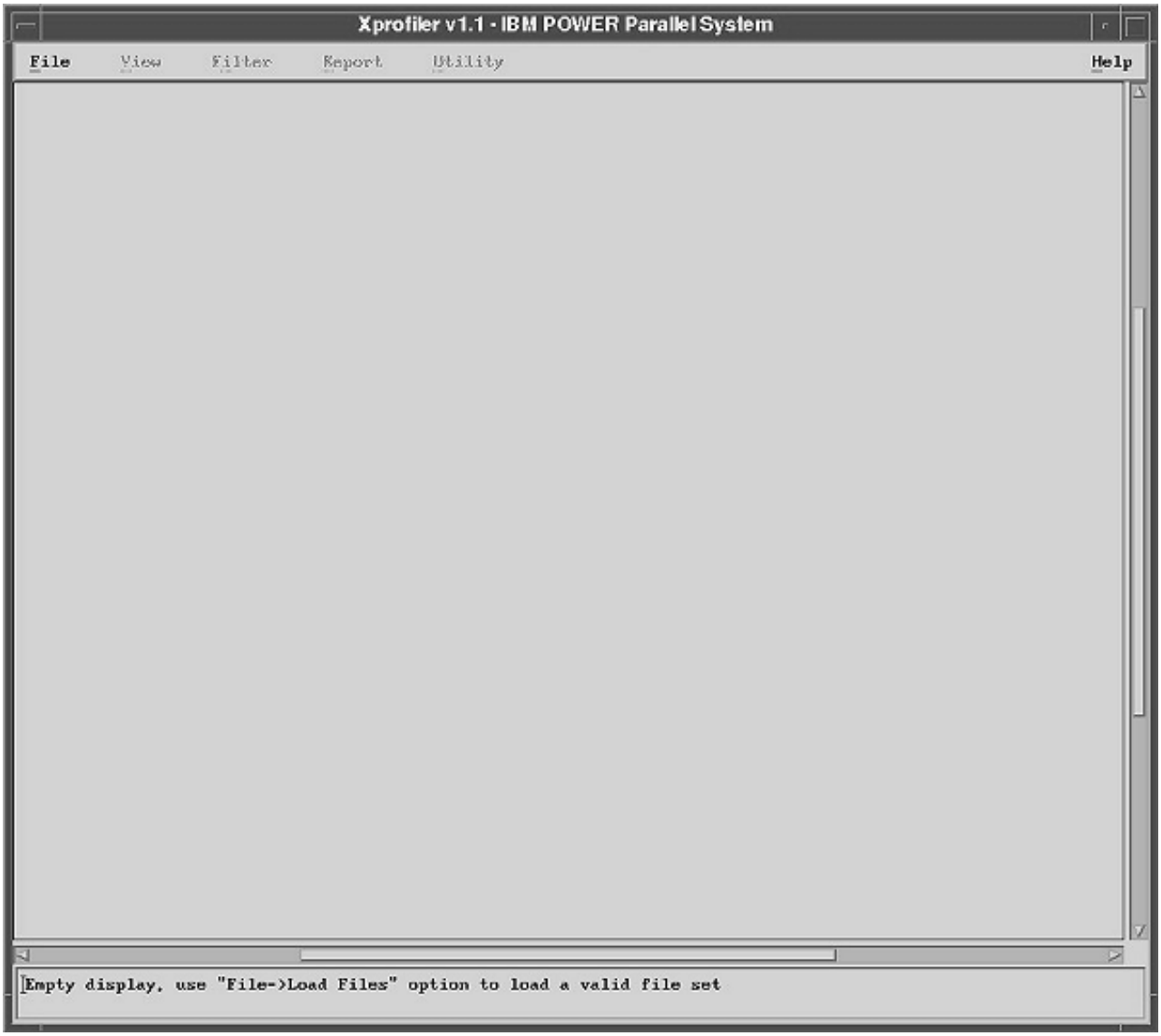


Figure 1. The Xprofiler main window.

From the Xprofiler GUI, select **File**, then **Load File** from the menu bar. The Load Files Dialog window will appear.

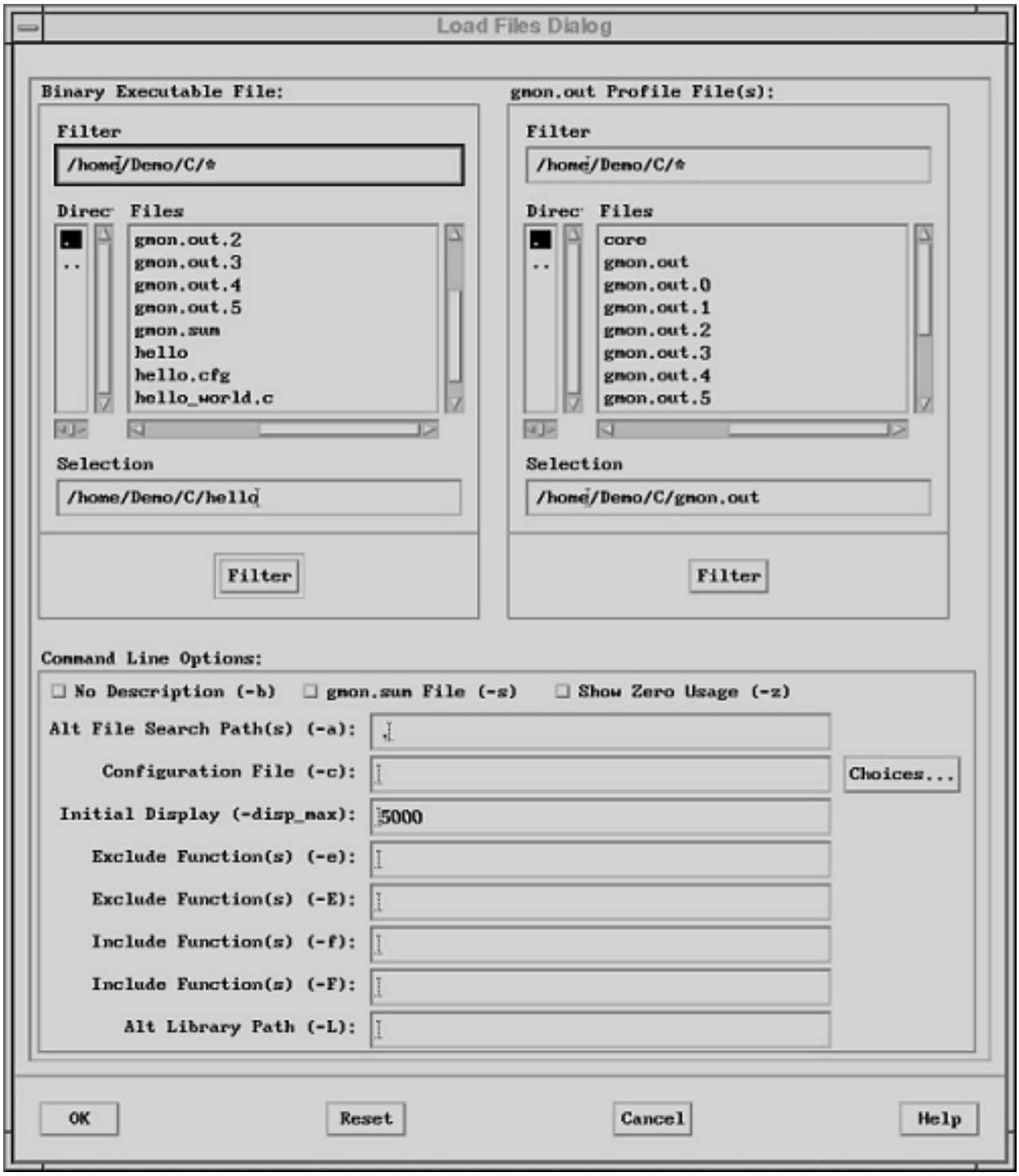


Figure 2. The Load Files Dialog window

The Load Files Dialog window lets you specify your application's executable file and its corresponding profile data (**gmon.out**) files. When you load a file, you can also specify the various command-line options that let you control the way Xprofiler displays the profiled data.

To load the files for the application you want to profile, you must specify the following:

- the binary executable file

- one or more profile data files

Optionally, you can also specify one or more command-line flags.

The binary executable file

You specify the binary executable file from the **Binary Executable File:** area of the Load Files Dialog window.

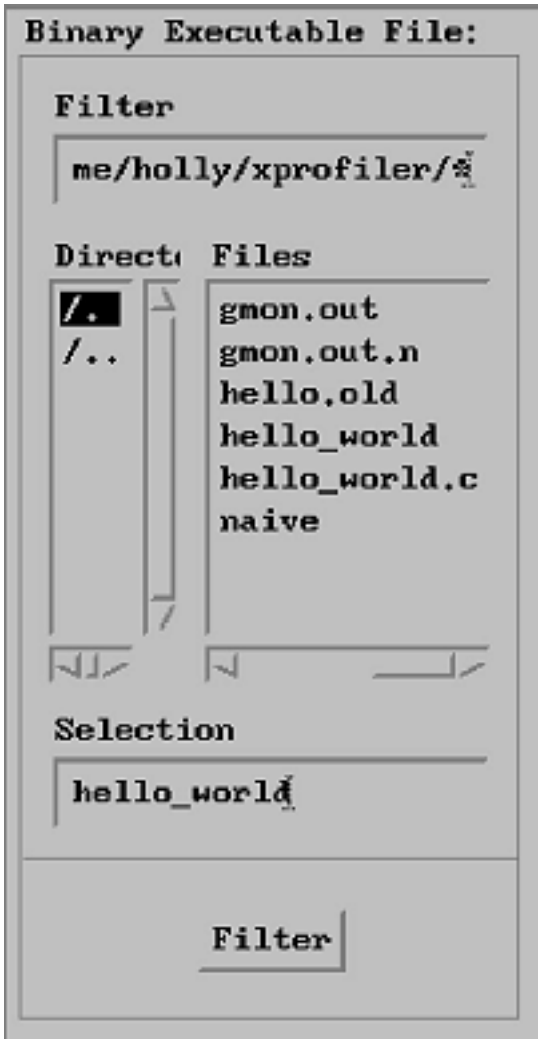


Figure 3. The Binary Executable File dialog

Use the scroll bars of the **Directories** and **Files** selection boxes to locate the executable file you want to load. By default, all of the files in the directory from which you called Xprofiler appear in the **Files** selection box.

To make locating your binary executable files easier, the **Binary Executable File:** area includes a **Filter** button. Filtering lets you limit the files that are displayed in the **Files** selection box to those of a specific directory or of a specific type. For information about filtering, see “Filtering what you see” on page 27.

Profile data files

You specify one or more profile data files from the **gmon.out Profile Data File(s)** area of the Load Files Dialog window.

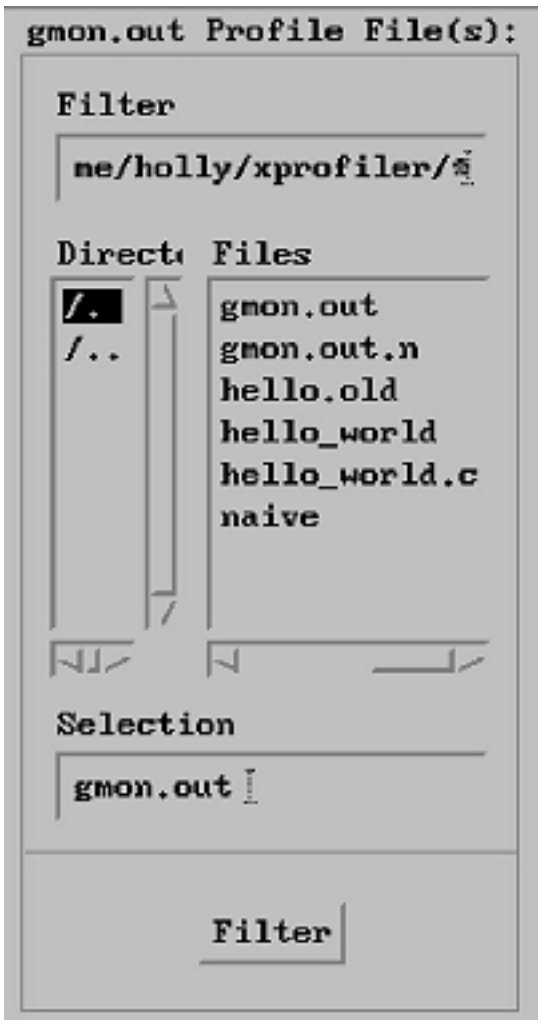


Figure 4. The gmon.out Profile Data File area

When you start Xprofiler using the **xprofiler** command, you are not required to indicate the name of the profile data file. If you do not specify a profile data file, Xprofiler searches your directory for the presence of a file named **gmon.out** and, if found, places it in the **Selection** field of the **gmon.out Profile Data File(s)** area, as the default. Xprofiler then uses this file as input, even if it is not related to the binary executable file you specify. Because this will cause Xprofiler to display incorrect data, it is important that you enter the correct file into this field. If the profile data file you want to use is named something other than what appears in the **Selection** field, you must replace it with the correct file name.

Use the scroll bars of the **Directories** and **Files** selection boxes to locate one or more of the profile data (**gmon.out**) files you want to specify. The file you use does not have to be named **gmon.out**, and you can specify more than one profile data file.

To make locating your output files easier, the **gmon.out Profile Data File(s)** area includes a **Filter** button. Filtering lets you limit the files that are displayed in the **Files** selection box to those in a specific directory or of a specific type. For information about filtering, see “Filtering what you see” on page 27.

Specifying command line options (from the GUI)

You specify command-line flags from the **Command Line Options** area of the Load Files Dialog window.

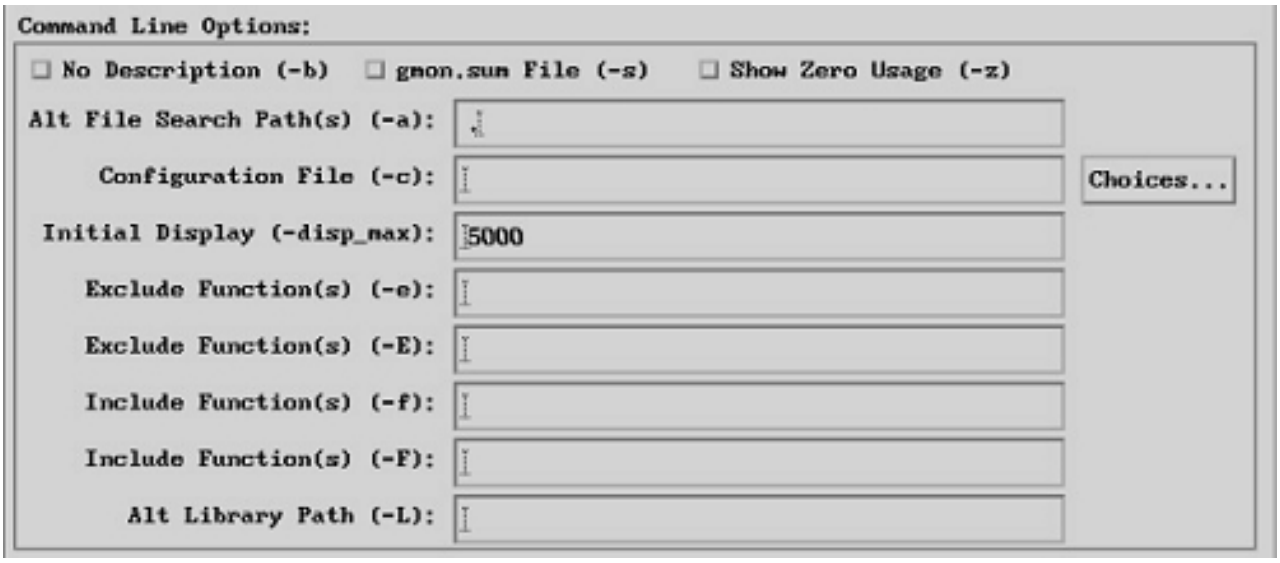


Figure 5. The Command Line Options area

You can specify one or more flags as follows:

Table 3. Xprofiler GUI command-line flags

Use this flag:	To:	For example:
-a (field)	Add alternative paths to search for source code and library files, or changes the current path search order. After clicking the **OK** button, any modifications to this field are also made to the **Enter Alt File Search Paths:** field of the Alt File Search Path Dialog window. If both the Load Files Dialog window and the Alt File Search Path Dialog window are opened at the same time, when you make path changes in the Alt File Search Path Dialog window and click **OK**, these changes are also made to the Load Files Dialog window. Also, when both of these windows are open at the same time, clicking the **OK** or **Cancel** buttons in the Load Files Dialog window causes both windows to close. If you want to restore the **Alt File Search Path(s) (-a):** field to the same state as when the Load Files Dialog window was opened, click the **Reset** button. You can use the “at” symbol (**@**) with this flag to represent the default file path, in order to specify that other paths be searched before the default path.	To set an alternative file search path so that Xprofiler searches **pathA**, the default path, then **pathB**, type `pathA:@:pathB` in the **Alt File Search Path(s) (-a)** field.
-b (button)	Suppress the printing of the field descriptions for the **Flat Profile**, **Call Graph Profile**, and **Function Index** reports when they are written to a file with the **Save As** option of the **File** menu.	To suppress printing of the field descriptions for the **Flat Profile**, **Call Graph Profile**, and **Function Index** reports in the saved file, set the **-b** button to the pressed-in position.

Table 3. Xprofiler GUI command-line flags (continued)

Use this flag:	To:	For example:
-c (field)	Load the specified configuration file. If the **-c** option was used on the command line, or a configuration file had been previously loaded with the Load Files Dialog window or the Load Configuration File Dialog window, the name of the most recently loaded file will appear in the **Configuration File (-c):** text field in the Load Files Dialog window, as well as the **Selection** field of Load Files Dialog window. If the Load Files Dialog window and the Load Files Dialog window are open at the same time, when you specify a configuration file in the Load Configuration File Dialog window and then click the **OK** button, the name of the specified file also appears in the Load Files Dialog window. Also, when both of these windows are open at the same time, clicking the **OK** or **Cancel** button in the Load Files Dialog window causes both windows to close. When entries are made to both the **Configuration File (-c):** and **Initial Display (-disp_max):** fields in the Load Files Dialog window, the value in the **Initial Display (-disp_max):** field is ignored, but is retained the next time this window is opened. If you want to retrieve the file name that was in the **Configuration File (-c):** field when the Load Files Dialog window was opened, click the **Reset** button.	To load the configuration file **myfile.cfg**, type `myfile.cfg` in the **Configuration File (-c)** field.
-disp_max (field)	Set the number of function boxes that Xprofiler initially displays in the function call tree. The value supplied with this flag can be any integer between 0 and 5000. Xprofiler displays the function boxes for the most CPU-intensive functions through the number you specify. For example, if you specify 50, Xprofiler displays the function boxes for the 50 functions in your program with the highest CPU usage. After this, you can change the number of function boxes that are displayed using the **Filter** menu options. This flag has no effect on the content of any of the Xprofiler reports.	To display the function boxes for the 50 most CPU-intensive functions in the function call tree, type `50` in the **Init Display (-disp_max)** field.

Table 3. Xprofiler GUI command-line flags (continued)

Use this flag:	To:	For example:
-e (field)	Deemphasize the general appearance of the function box for the specified function in the function call tree, and limits the number of entries for this function in the **Call Graph Profile** report. This also applies to the specified function's descendants, as long as they have not been called by non-specified functions. In the function call tree, the function box for the specified function is made unavailable. The box size and the content of the label remain the same. This also applies to descendant functions, as long as they have not been called by non-specified functions. In the **Call Graph Profile** report, an entry for a specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. The information for this entry remains unchanged. Entries for descendants of the specified function do not appear unless they have been called by at least one non-specified function in the program.	To deemphasize the appearance of the function boxes for **foo** and **bar** and their qualifying descendants in the function call tree, and limit their entries in the **Call Graph Profile** report, type `foo` and `bar` in the **Exclude Routines (-e)** field. Multiple functions are separated by a space.

Table 3. Xprofiler GUI command-line flags (continued)

Use this flag:	To:	For example:
-E (field)	Change the general appearance and label information of the function box for the specified function in the function call tree. This flag also limits the number of entries for this function in the **Call Graph Profile** report, and changes the CPU data associated with them. These results also apply to the specified function's descendants, as long as they have not been called by non-specified functions in the program. In the function call tree, the function box for the specified function appears greyed out, and the box size and shape also changes so that it appears as a square of the smallest allowable size. In addition, the CPU time shown in the function box label, appears as 0. The same applies to function boxes for descendant functions, as long as they have not been called by non-specified functions. This flag also causes the CPU time spent by the specified function to be deducted from the CPU total on the left in the label of the function box for each of the specified function's ancestors. In the **Call Graph Profile** report, an entry for the specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. When this is the case, the time in the self and descendants columns for this entry is set to 0. In addition, the amount of time that was in the descendants column for the specified function is subtracted from the time listed under the descendants column for the profiled function. As a result, be aware that the value listed in the % time column for most profiled functions in this report will change.	To change the display and label information for **foo** and **bar** and their qualifying descendants in the function call tree, and limit their entries and data in the **Call Graph Profile** report, type `foo bar` in the **Exclude Routines (-E)** field. You specify multiple functions by separating each one with a space.

Table 3. Xprofiler GUI command-line flags (continued)

Use this flag:	To:	For example:
-f (field)	Deemphasize the general appearance of all function boxes in the function call tree, *except* for that of the specified function and its descendants. In addition, the number of entries in the **Call Graph Profile** report for the non-specified functions and non-descendant functions is limited. The **-f** flag overrides the **-e** flag. In the function call tree, all function boxes *except* for that of the specified function and its descendants are made unavailable. The size of these boxes and the content of their labels remain the same. For the specified function and its descendants, the appearance of the function boxes and labels remain the same. In the **Call Graph Profile** report, an entry for a non-specified or non-descendant function only appears where it is a parent or child of a specified function or one of its descendants. All information for this entry remains the same.	To deemphasize the display of function boxes for all functions in the function call tree *except* for **foo** and **bar** and their descendants, and limit their types of entries in the **Call Graph Profile** report, type `foo bar` in the **Include Routines (-f)** field. You specify multiple functions by separating each one with a space.
-F (field)	Change the general appearance and label information of all function boxes in the function call tree *except* for that of the specified function and its descendants. In addition, the number of entries in the **Call Graph Profile** report for the non-specified and non-descendant functions is limited, and the CPU data associated with them is changed. The **-F** flag overrides the **-E** flag. In the function call tree, the function box for the specified function is made unavailable, and its size and shape also changes so that it appears as a square of the smallest allowable size. In addition, the CPU time shown in the function box label, appears as 0. In the **Call Graph Profile** report, an entry for a non-specified or non-descendant function only appears where it is a parent or child of a specified function or one of its descendants. When this is the case, the time in the self and descendants columns for this entry is set to 0. As a result, be aware that the value listed in the % time column for most profiled functions in this report will change.	To change the display and label information of the function boxes for all functions *except* the functions **foo** and **bar** and their descendants, and limit their types of entries and data in the **Call Graph Profile** report, type `foo bar` in the **Include Routines (-F)** field. You specify multiple functions by separating each one with a space.
-L (field)	Set the alternative path name for locating shared objects. If you plan to specify multiple paths, use the **Set File Search Path** option of the **File** menu on the Xprofiler GUI. See “Setting the file search sequence” on page 19 for information.	To specify **/lib/profiled/libc.a:shr.o** as an alternative path name for your shared libraries, type `/lib/profiled/libc.a:shr.o` in this field.

Table 3. Xprofiler GUI command-line flags (continued)

Use this flag:	To:	For example:
-s (button)	Produces the **gmon.sum** profile data file, if multiple **gmon.out** files are specified when Xprofiler is started. The **gmon.sum** file represents the sum of the profile information in all the specified profile files. Note that if you specify a single **gmon.out** file, the **gmon.sum** file contains the same data as the **gmon.out** file.	To write the sum of the data from three profile data files, **gmon.out.1**, **gmon.out.2**, and **gmon.out.3**, into a file called **gmon.sum**, set the **-s** button to the pressed-in position.
-z (button)	Includes functions that have both zero CPU usage and no call counts in the **Flat Profile**, **Call Graph Profile**, and **Function Index** reports. A function will not have a call count if the file that contains its definition was not compiled with the **-pg** flag, which is common with system library files.	To include all functions used by the application that have zero CPU usage and no call counts in the **Flat Profile**, **Call Graph Profile**, and **Function Index** reports, set the **-z** button to the pressed-in position.

After you have specified the binary executable file, one or more profile data files, and any command-line flags you want to use, click the **OK** button to save the changes and close the window. Xprofiler loads your application and displays its performance data.

Setting the file search sequence

You can specify where you want Xprofiler to look for your library files and source code files by using the **Set File Search Paths** option of the **File** menu. By default, Xprofiler searches the default paths first and then any alternative paths you specify.

Default paths

For library files, Xprofiler uses the paths recorded in the specified **gmon.out** files. If you use the **-L** flag, the path you specify with it will be used instead of those in the **gmon.out** files.

Note: The **-L** flag allows only one path to be specified, and you can use this flag only once.

For source code files, the paths recorded in the specified **a.out** file are used.

Alternative paths

You specify the alternative paths with the **Set File Search Paths** option of the **File** menu.

For library files, if everything else failed, the search will be extended to the path (or paths) specified by the LIBPATH environment variable associated with the executable file.

To specify alternative paths, do the following:

1. Select the **File** menu, and then the **Set File Search Paths** option. The Alt File Search Path Dialog window appears.
2. Enter the name of the path in the **Enter Alt File Search Path(s)** text field. You can specify more than one path by separating each path name with a colon (**:**) or a space.

   Notes:

   a. You can use the “at” symbol (**@**) with this option to represent the default file path, in order to specify that other paths be searched before the default path. For example, to set the alternative file search paths so that Xprofiler searches **pathA**, the default path, then **pathB**, type `pathA:@:pathB` in the **Alt File Search Path(s) (-a)** field.

   b. If **@** is used in the alternative search path, the two buttons in the Alt File Search Path Dialog window will be unavailable, and will have no effect on the search order.
3. Click the **OK** button. The paths you specified in the text field become the alternative paths.

Changing the search sequence

You can change the order of the search sequence for library files and source code files using the **Set File Search Paths** option of the **File** menu. To change the search sequence:

1. Select the **File** menu, and then the **Set File Search Paths** option. The Alt File Search Path Dialog window appears.
2. To indicate that the file search should use alternative paths first, click the **Check alternative path(s) first** button.
3. Click **OK**. This changes the search sequence to the following:
   a. Alternative paths
   b. Default paths
   c. Paths specified in LIBPATH (library files only)

To return the search sequence back to its default order, repeat steps 1 through 3, but in step 2, click the **Check default path(s) first** button. When the action is confirmed (by clicking **OK**), the search sequence will start with the default paths again.

If a file is found in one of the alternative paths or a path in LIBPATH, this path now becomes the default path for this file throughout the current Xprofiler session (until you exit this Xprofiler session or load a new set of data).

Understanding the Xprofiler display

The primary difference between Xprofiler and the **gprof** command is that Xprofiler gives you a graphical picture of your application's CPU consumption in addition to textual data.

Xprofiler displays your profiled program in a single main window. It uses several types of graphical images to represent the relevant parts of your program. Functions appear as solid green boxes (called *function boxes*), and the calls between them appear as blue arrows (called *call arcs*). The function boxes and call arcs that belong to each library within your application appear within a fenced-in area called a *cluster box.*

Xprofiler main window

The Xprofiler main window contains a graphical representation of the functions and calls within your application as well as their interrelationships. The window provides six menus, including one for online help.

When an application has been loaded, the Xprofiler main window looks similar to the following:

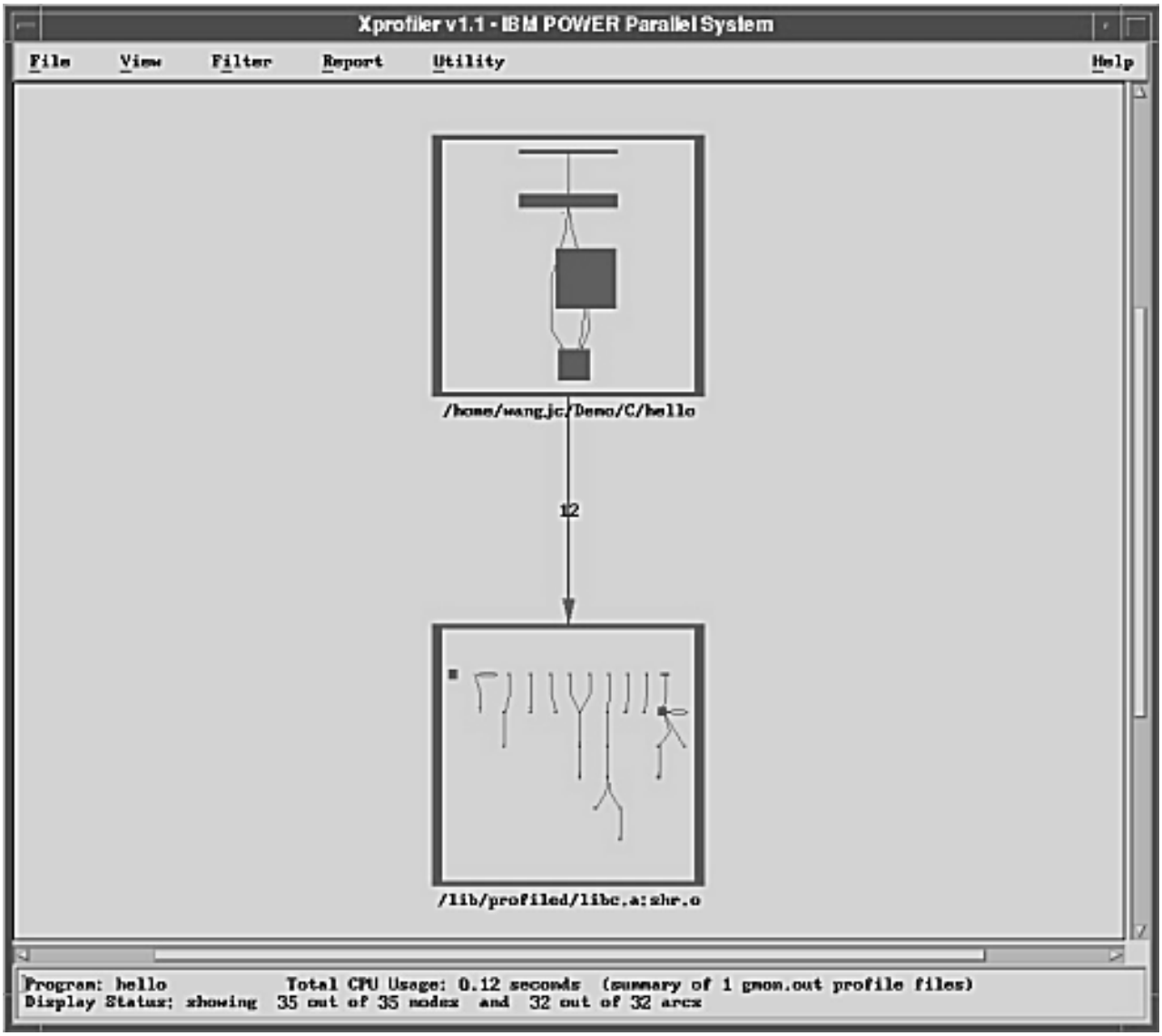


Figure 6. The Xprofiler main window with application loaded

In the main window, Xprofiler displays the *function call tree.* The function call tree displays the function boxes, call arcs, and cluster boxes that represent the functions within your application.

Note: When Xprofiler first opens, by default, the function boxes for your application will be *clustered* by library. A cluster box appears around each library, and the function boxes and arcs within the cluster box are reduced in size. To see more detail, you must uncluster the functions. To do this, select the **File** menu and then the **Uncluster Functions** option.

Xprofiler's main menus

The Xprofiler menus are as follows:

The File menu: The File menu lets you specify the executable (**a.out**) files and profile data (**gmon.out**) files that Xprofiler will use. It also lets you control how your files are accessed and saved.

The View menu: The View menu lets you focus on specific portions of the function call tree in order to get a better view of the application's critical areas.

The Filter menu: The Filter menu lets you add, remove, and change specific parts of the function call tree. By controlling what Xprofiler displays, you can focus on the objects that are most important to you.

The Report menu: The Report menu provides several types of profiled data in a textual and tabular format. In addition to presenting the profiled data, the options of the Report menu let you do the following:

- Display textual data
- Save it to a file
- View the corresponding source code
- Locate the corresponding function box or call arc in the function call tree

The Utility menu: The Utility menu contains one option, **Locate Function By Name**, which lets you highlight a particular function in the function call tree.

Xprofiler's hidden menus

The Function menu: The Function menu lets you perform a number of operations for any of the functions shown in the function call tree. You can access statistical data, look at source code, and control which functions are displayed.

The Function menu is not visible from the Xprofiler window. You access it by right-clicking on the function box of the function in which you are interested. By doing this, you open the Function menu, and select this function as well. Then, when you select actions from the Function menu, the actions are applied to this function.

The Arc menu: The Arc menu lets you locate the caller and callee functions for a particular *call arc*. A call arc is the representation of a call between two functions within the function call tree.

The Arc menu is not visible from the Xprofiler window. You access it by right-clicking on the call arc in which you are interested. By doing this, you open the Arc menu, and select that call arc as well. Then, when you perform actions with the Arc menu, they are applied to that call arc.

The Cluster Node menu: The Cluster Node menu lets you control the way your libraries are displayed by Xprofiler. To access the Cluster Node menu, the function boxes in the function call tree must first be clustered by library. For information about clustering and unclustering the function boxes of your application, see "Clustering libraries" on page 32. When the function call tree is clustered, all the function boxes within each library appear within a *cluster box*.

The Cluster Node menu is not visible from the Xprofiler window. You access it by right-clicking on the edge of the cluster box in which you are interested. By doing this, you open the Cluster Node menu, and select that cluster as well. Then, when you perform actions with the Cluster Node menu, they are applied to the functions within that library cluster.

The Display Status field

At the bottom of the Xprofiler window is a single field that provides the following information:

- Name of your application
- Number of **gmon.out** files used in this session
- Total amount of CPU used by the application
- Number of functions and calls in your application, and how many of these are currently displayed

How functions are represented

Functions are represented by solid green boxes in the function call tree. The size and shape of each function box indicates its CPU usage. The height of each function box represents the amount of CPU time it spent on executing itself. The width of each function box represents the amount of CPU time it spent executing itself, plus its descendant functions.

This type of representation is known as *summary mode*. In summary mode, the size and shape of each function box is determined by the total CPU time of multiple **gmon.out** files used on that function alone, and the total time used by the function and its descendant functions. A function box that is wide and flat represents a function that uses a relatively small amount of CPU on itself (it spends most of its time on its descendants). The function box for a function that spends most of its time executing only itself will be roughly square-shaped.

Functions can also be represented in *average mode*. In average mode, the size and shape of each function box is determined by the average CPU time used on that function alone, among all loaded **gmon.out** files, and the standard deviation of CPU time for that function among all loaded **gmon.out** files. The height of each function node represents the average CPU time, among all the input **gmon.out** files, used on the function itself. The width of each node represents the standard deviation of CPU time, among the **gmon.out** files, used on the function itself. The average mode representation is available only when more than one **gmon.out** file is entered. For more information about summary mode and average mode, see “Controlling the representation of the function call tree” on page 26.

Under each function box in the function call tree is a label that contains the name of the function and related CPU usage data. For information about the function box labels, see “Obtaining basic data” on page 37.

The following figure shows the function boxes for two functions, sub1 and printf, as they would appear in the Xprofiler display.

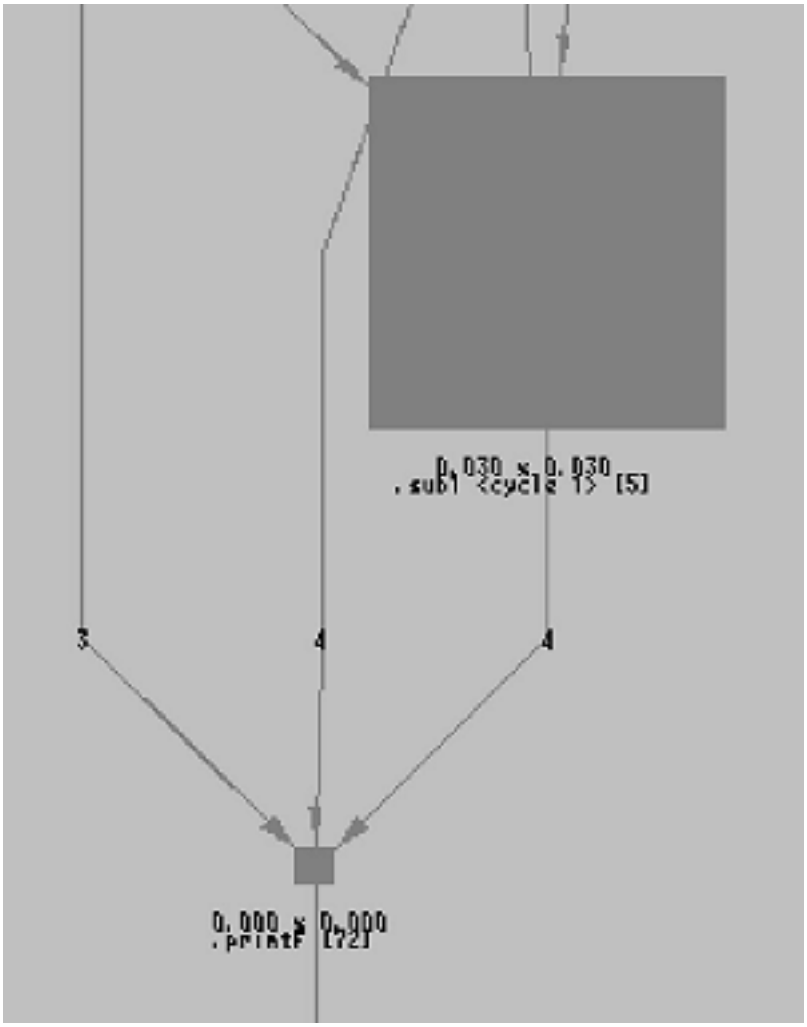


Figure 7. Function boxes and arcs in the Xprofiler display

Each function box has its own menu. To access it, place your mouse cursor over the function box of the function you are interested in and press the right mouse button. Each function also has an information box that lets you get basic performance numbers quickly. To access the information box, place your mouse cursor over the function box of the function you are interested in and press the left mouse button.

How calls between functions are depicted

The calls made between each of the functions in the function call tree are represented by blue arrows extending between their corresponding function boxes. These lines are called *call arcs*. Each call arc appears as a solid blue line between two functions. The arrowhead indicates the direction of the call; the function represented by the function box it points to is the one that receives the call. The function making the call is known as the *caller*, while the function receiving the call is known as the *callee*.

Each call arc includes a numeric label that indicates how many calls were exchanged between the two corresponding functions.

Each call arc has its own menu that lets you locate the function boxes for its caller and callee functions. To access it, place your mouse cursor over the call arc for the call in which you are interested, and press the right mouse button. Each call arc also has an information box that shows you the number of times the caller function called the callee function. To access the information box, place your mouse cursor over the call arc for the call in which you are interested, and press the left mouse button.

How library clusters are represented

Xprofiler lets you collect the function boxes and call arcs that belong to each of your shared libraries into *cluster boxes.*

Because there will be a box around each library, the individual function boxes and call arcs will be difficult to see. If you want to see more detail, you must uncluster the function boxes. To do this, select the Filter menu and then the **Uncluster Functions** option.

When viewing function boxes within a cluster box, note that the size of each function box is relative to those of the other functions within the same library cluster. On the other hand, when all the libraries are unclustered, the size of each function box is relative to all the functions in the application (as shown in the function call tree).

Each library cluster has its own menu that lets you manipulate the cluster box. To access it, place your mouse cursor over the edge of the cluster box you are interested in, and press the right mouse button. Each cluster also has an information box that shows you the name of the library and the total CPU usage (in seconds) consumed by the functions within it. To access the information box, place your mouse cursor over the edge of the cluster box you are interested in and press the left mouse button.

Controlling how the display is updated

The **Utility** menu of the Overview Window lets you choose the mode in which the display is updated. The default is the **Immediate Update** option, which causes the display to show you the items in the highlight area as you are moving it around. The **Delayed Update** option, on the other hand, causes the display to be updated only when you have moved the highlight area over the area in which you are interested, and released the mouse button. The **Immediate Update** option applies only to what you see when you move the highlight area; it has no effect on the resizing of items in highlight area, which is always delayed.

Other viewing options

Xprofiler lets you change the way it displays the function call tree, based on your personal preferences.

Controlling the graphic style of the function call tree

You can choose between two-dimensional and three-dimensional function boxes in the function call tree. The default style is two-dimensional. To change to three-dimensional, select the **View** menu, and then the **3-D Image** option. The function boxes in the function call tree now appear in three-dimensional format.

Controlling the orientation of the function call tree

You can choose to have Xprofiler display the function call tree in either top-to-bottom or left-to-right format. The default is top-to-bottom. To see the function call tree displayed in left-to-right format, select the **View** menu, and then the **Layout: Left→Right** option. The function call tree now displays in left-to-right format.

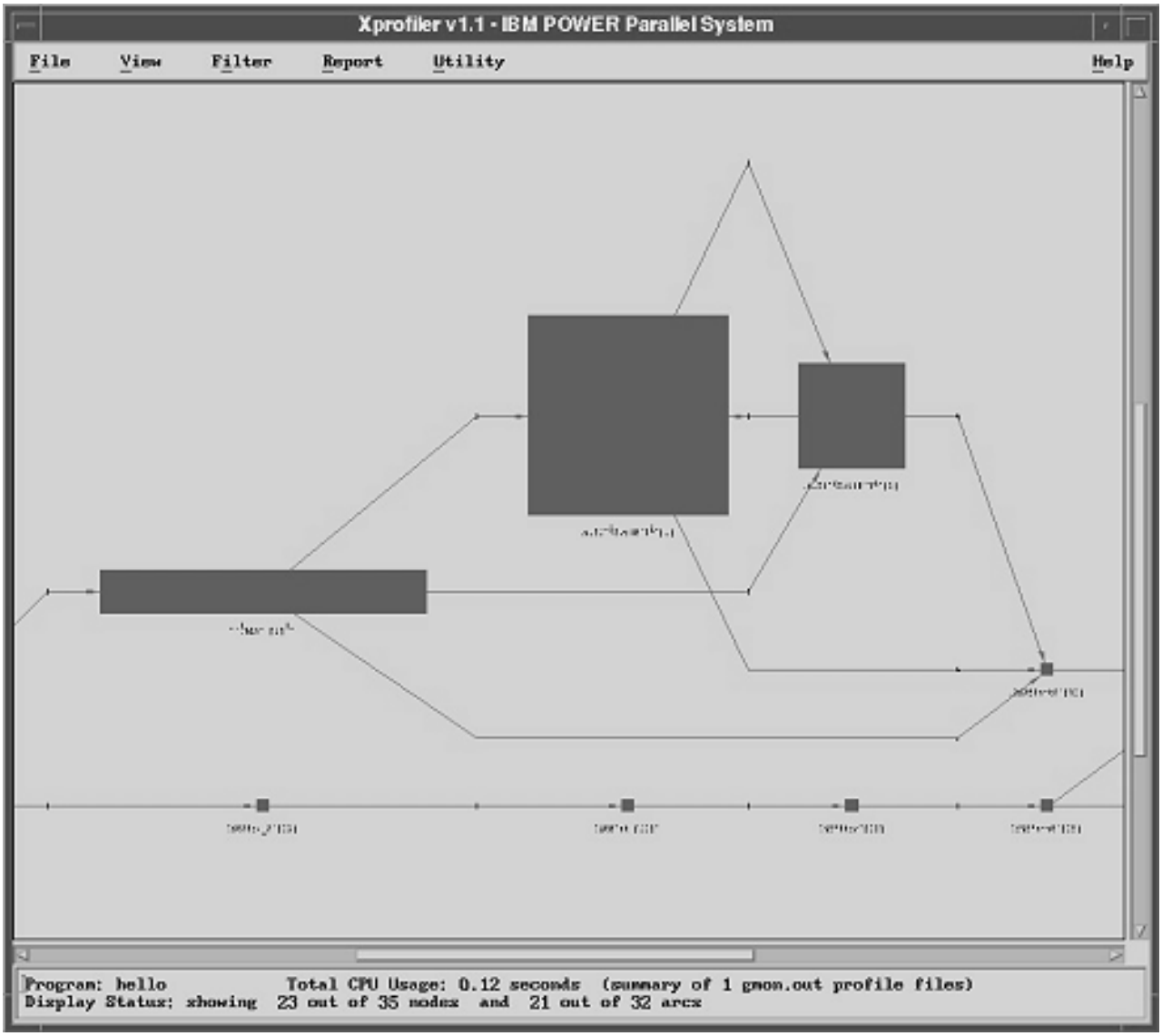


Figure 8. Left-to-right format

Controlling the representation of the function call tree

You can choose to have Xprofiler represent the function call tree in either *summary mode* or *average mode*.

When you select the **Summary Mode** option of the View menu, the size and shape of each function box is determined by the total CPU time of multiple **gmon.out** files used on that function alone, and the total time used by the function and its descendant functions. The height of each function node represents the total CPU time used on the function itself. The width of each node represents the total CPU time used on the function and its descendant functions. When the display is in summary mode, the **Summary Mode** option is unavailable and the **Average Mode** option is activated.

When you select the **Average Mode** option of the View menu, the size and shape of each function box is determined by the average CPU time used on that function alone, among all loaded **gmon.out** files, and the standard deviation of CPU time for that function among all loaded **gmon.out** files. The height of each

function node represents the average CPU time, among all the input **gmon.out** files, used on the function itself. The width of each node represents the standard deviation of CPU time, among the **gmon.out** files, used on the function itself.

The purpose of average mode is to reveal workload balancing problems when an application is involved with multiple **gmon.out** files. In general, a function node with large standard deviation has a wide width, and a node with small standard deviation has a slim width.

Both summary mode and average mode affect only the appearance of the function call tree and the labels associated with it. All the performance data in Xprofiler reports and code displays are always summary data. If only one **gmon.out** file is specified, **Summary Mode** and **Average Mode** will be unavailable, and the display is always in **Summary Mode**.

Filtering what you see

When Xprofiler first opens, the entire function call tree appears in the main window. This includes the function boxes and call arcs that belong to your executable file as well as the shared libraries that it uses. You can simplify what you see in the main window, and there are several ways to do this.

Note: Filtering options of the Filter menu let you change the appearance only of the function call tree. The performance data contained in the reports (through the Reports menu) is not affected.

Restoring the status of the function call tree

Xprofiler allows you to undo operations that involve adding or removing nodes and arcs from the function call tree. When you undo an operation, you reverse the effect of any operation which adds or removes function boxes or call arcs to the function call tree. When you select the **Undo** option, the function call tree is returned to its appearance just prior to the performance of the add or remove operation. To undo an operation, select the **Filter** menu, and then the **Undo** option. The function call tree is returned to its appearance just prior to the performance of the add or remove operation.

Whenever you invoke the **Undo** option, the function call tree loses its zoom focus and zooms all the way out to reveal the entire function call tree in the main display. When you start Xprofiler, the **Undo** option is unavailable. It is activated only after an add or remove operation involving the function call tree takes place. After you undo an operation, the option is made unavailable again until the next add or remove operation takes place.

The options that activate the **Undo** option include the following:

- In the main **File** menu:
  - Load Configuration
- In the main **Filter** menu:
  - Show Entire Call Tree
  - Hide All Library Calls
  - Add Library Calls
  - Filter by Function Names
  - Filter by CPU Time
  - Filter by Call Counts
- In the **Function** menu:
  - Immediate Parents
  - All Paths To
  - Immediate Children
  - All Paths From
  - All Functions on The Cycle
  - Show This Function Only
  - Hide This Function
  - Hide Descendant Functions

– Hide This & Descendant Functions

If a dialog, such as the Load Configuration Dialog or the Filter by CPU Time Dialog is invoked and then canceled immediately, the status of the **Undo** option is not affected. After the option is available, it stays that way until you invoke it, or a new set of files is loaded into Xprofiler through the Load Files Dialog window.

Displaying the entire function call tree

When you first open Xprofiler, by default, all the function boxes and call arcs of your executable and its shared libraries appear in the main window. After that, you may choose to filter out specific items from the window. However, there may be times when you want to see the entire function call tree again, without having to reload your application. To do this, select the **Filter** menu, and then the **Show Entire Call Tree** option. Xprofiler erases whatever is currently displayed in the main window and replaces it with the entire function call tree.

Excluding and including specific objects

There are a number of ways that Xprofiler lets you control the items that display in the main window. You will want to include or exclude certain objects so that you can more easily focus on the things that are of most interest to you.

Filtering shared library functions

In most cases, your application will call functions that are within shared libraries. By default, these shared libraries display in the Xprofiler window along with your executable file. As a result, the window may get crowded and obscure the items that you most need to see. If this is the case, you can filter the shared libraries from the display. To do this, select the **Filter** menu, and then the **Remove All Library Calls** option.

The shared library function boxes disappear from the function call tree, leaving only the function boxes of your executable file visible.

If you removed the library calls from the display, you may want to restore them. To do this, select the **File** menu and then the **Add Library Calls** option.

The function boxes again appear with the function call tree. Note, however, that all of the shared library calls that were in the initial function call tree may not be added back. This is because the **Add Library Calls** option only adds back in the function boxes for the library functions that were called by functions that are currently displayed in the Xprofiler window.

To add only specific function boxes back into the display, do the following:

1. Select the **Filter** menu, and then the **Filter by Function Names** option. The Filter By Function Names dialog window appears.
2. From the Filter By Function Names Dialog window, click the **add these functions to graph** button, and then type the name of the function you want to add in the **Enter function name** field. If you enter more than one function name, you must separate them with a blank space between each function name string.

   If there are multiple functions in your program that include the string you enter in their names, the filter applies to each one. For example, if you specified **sub** and **print**, and your program also included functions named **sub1**, **psub1**, and **printf**. The **sub**, **sub1**, **psub1**, **print**, and **printf** functions would all be added to the graph.
3. Click **OK**. One or more function boxes appears in the Xprofiler display with the function call tree.

Filtering by function characteristics

The Filter menu of Xprofiler offers the following options that allow you to add or subtract function boxes from the main window, based on specific characteristics:

- Filter by Function Names
- Filter by CPU Time
- Filter by Call Counts

Each option uses a different window to let you specify the criteria by which you want to include or exclude function boxes from the window.

To filter by function names, do the following:

1. Select the **Filter** menu and then the **Filter by Function Names** option. The Filter By Function Names Dialog window appears:

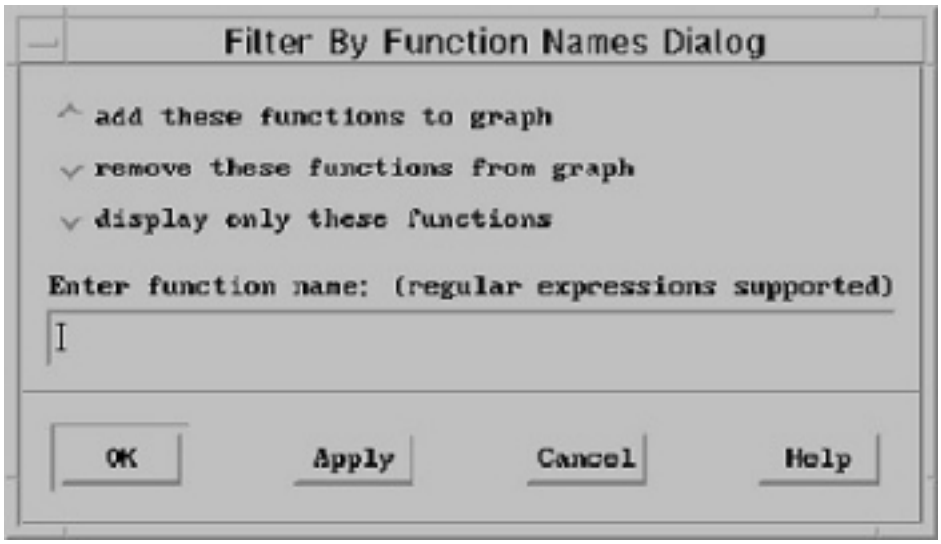


Figure 9. The Filter By Function Names Dialog window

   The Filter By Function Names Dialog window includes the following options:
   a. add these functions to graph
   b. remove these functions from the graph
   c. display only these functions
2. From the Filter By Function Names Dialog window, select the option, and then type the name of the function (or functions) to which you want it applied in the **Enter function name** field. For example, if you want to remove the function box for a function called **printf** from the main window, click the **remove this function from the graph** button, and type `printf` in the **Enter function name** field.

   You can enter more than one function name in this field. If there are multiple functions in your program that include the string you enter in their names, the filter will apply to each one. For example, if you specified **sub** and **print**, and your program also included functions named **sub1**, **psub1**, and **printf**, the option you chose would be applied to the **sub**, **sub1**, **psub1**, **print**, and **printf** functions.
3. Click **OK**. The contents of the function call tree now reflect the filtering options you specified.

To filter by CPU time, do the following:

1. Select the **Filter** menu and then the **Filter by CPU Time** option. The Filter By CPU Time Dialog window appears.

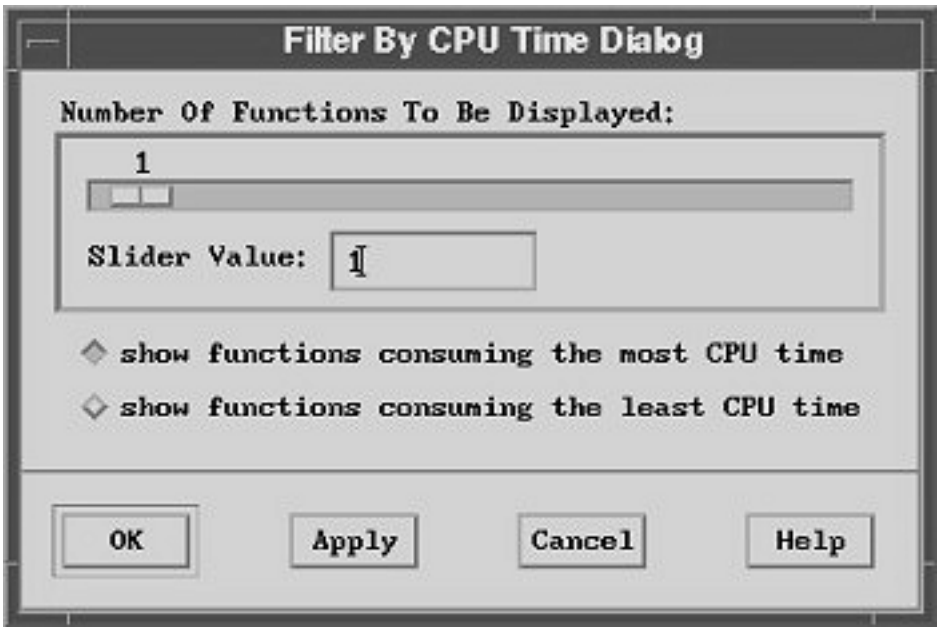


Figure 10. The Filter By CPU Time Dialog window

The Filter By CPU Time Dialog window includes the following options:
- show functions consuming the most CPU time
- show functions consuming the least CPU time

2. Select the option you want (**show functions consuming the most CPU time** is the default).
3. Select the number of functions to which you want it applied (1 is the default). You can move the slider in the **Functions** bar until the desired number appears, or you can enter the number in the **Slider Value** field. The slider and **Slider Value** field are synchronized so when the slider is updated, the text field value is updated also. If you enter a value in the text field, the slider is updated to that value when you click **Apply** or **OK**.

   For example, to display the function boxes for the 10 functions in your application that consumed the most CPU, you would select the **show functions consuming the most CPU** button, and specify 10 with the slider or enter the value 10 in the text field.
4. Click **Apply** to show the changes to the function call tree without closing the dialog. Click **OK** to show the changes and close the dialog.

To filter by call counts, do the following:

1. Select the **Filter** menu and then the **Filter by Call Counts** option. The Filter By Call Counts Dialog window appears.

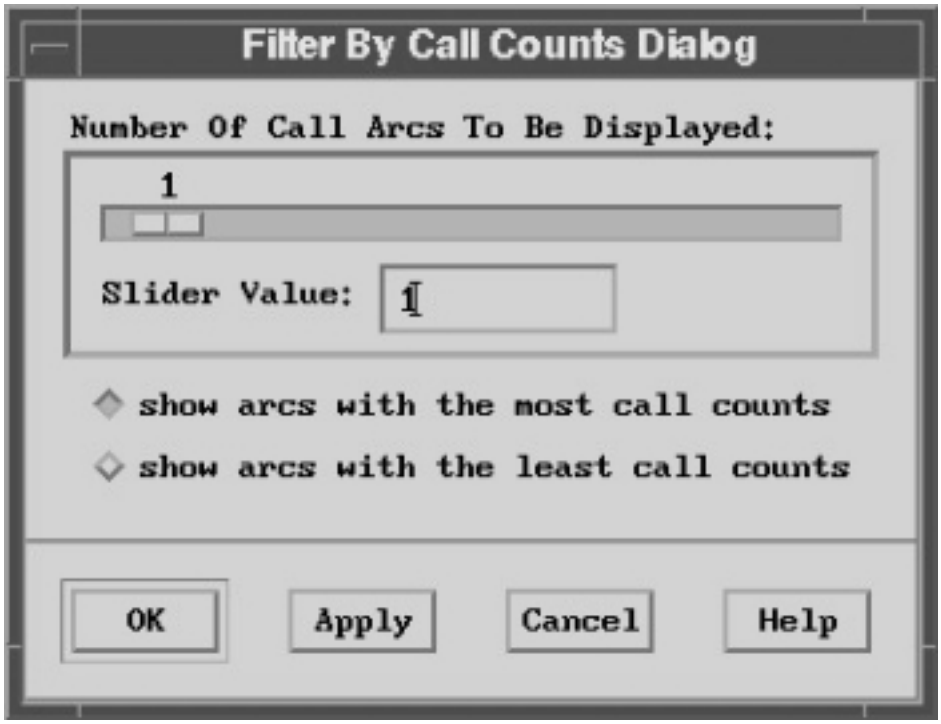


Figure 11. The Filter By Call Counts Dialog window

The **Filter By Call Counts Dialog** window includes the following options:
- show arcs with the most call counts
- show arcs with the least call counts

2. Select the option you want (**show arcs with the most call counts** is the default).
3. Select the number of call arcs to which you want it applied (1 is the default). If you enter a value in the text field, the slider is updated to that value when you click **Apply** or **OK**.

   For example, to display the 10 call arcs in your application that represented the least number of calls, you would select the **show arcs with the least call counts** button, and specify 10 with the slider or enter the value 10 in the text field.
4. Click **Apply** to show the changes to the function call tree without closing the dialog. Click **OK** to show the changes and close the dialog.

Including and excluding parent and child functions

When tuning the performance of your application, you will want to know which functions consumed the most CPU time, and then you will need to ask several questions in order to understand their behavior:

- Where did each function spend most of the CPU time?
- What other functions called this function? Were the calls made directly or indirectly?
- What other functions did this function call? Were the calls made directly or indirectly?

After you understand how these functions behave, and are able to improve their performance, you can proceed to analyzing the functions that consume less CPU.

When your application is large, the function call tree will also be large. As a result, the functions that are the most CPU-intensive may be difficult to see in the function call tree. To avoid this situation, use the **Filter by CPU** option of the Filter menu, which lets you display only the function boxes for the functions that consume the most CPU time. After you have done this, the Function menu for each function lets you add the parent and descendant function boxes to the function call tree. By doing this, you create a smaller, simpler function call tree that displays the function boxes associated with the most CPU-intensive area of the application.

A *child* function is one that is directly called by the function of interest. To see only the function boxes for the function of interest and its child functions, do the following:

1. Place your mouse cursor over the function box in which you are interested, and press the right mouse button. The Function menu appears.
2. From the Function menu, select the **Immediate Children** option, and then the **Show Child Functions Only** option.

   Xprofiler erases the current display and replaces it with only the function boxes for the function you chose, as well as its child functions.

A *parent* function is one that directly calls the function of interest. To see only the function box for the function of interest and its parent functions, do the following:

1. Place your mouse cursor over the function box in which you are interested, and press the right mouse button. The Function menu appears.
2. From the Function menu, select the **Immediate Parents** option, and then the **Show Parent Functions Only** option.

   Xprofiler erases the current display and replaces it with only the function boxes for the function you chose, as well as its parent functions.

You might want to view the function boxes for both the parent and child functions of the function in which you are interested, without erasing the rest of the function call tree. This is especially true if you chose to display the function boxes for two or more of the most CPU-intensive functions with the **Filter by CPU** option of the Filter menu (you suspect that more than one function is consuming too much CPU). Do the following:

1. Place your mouse cursor over the function box in which you are interested, and press the right mouse button. The Function menu appears.
2. From the Function menu, select the **Immediate Parents** option, and then the **Add Parent Functions to Tree** option.

   Xprofiler leaves the current display as it is, but adds the parent function boxes.
3. Place your mouse cursor over the same function box and press the right mouse button. The Function menu appears.
4. From the Function menu, select the **Immediate Children** option, and then the **Add Child Functions to Tree** option.

   Xprofiler leaves the current display as it is, but now adds the child function boxes in addition to the parents.

Clustering libraries

When you first open the Xprofiler window, by default, the function boxes of your executable file, and the libraries associated with it, are clustered. Because Xprofiler shrinks the call tree of each library when it places it in a cluster, you must uncluster the function boxes if you want to look closely at a specific function box label.

You can see much more detail for each function, when your display is in the unclustered or *expanded* state, than when it is in the clustered or *collapsed* state. Depending on what you want to do, you must cluster or uncluster (collapse or expand) the display.

The Xprofiler window can be visually crowded, especially if your application calls functions that are within shared libraries; function boxes representing your executable functions as well as the functions of the shared libraries are displayed. As a result, you may want to organize what you see in the Xprofiler window so you can focus on the areas that are most important to you. You can do this by collecting all the function boxes of each library into a single area, known as a library *cluster.*

The following figure shows the **hello_world** application with its function boxes unclustered.

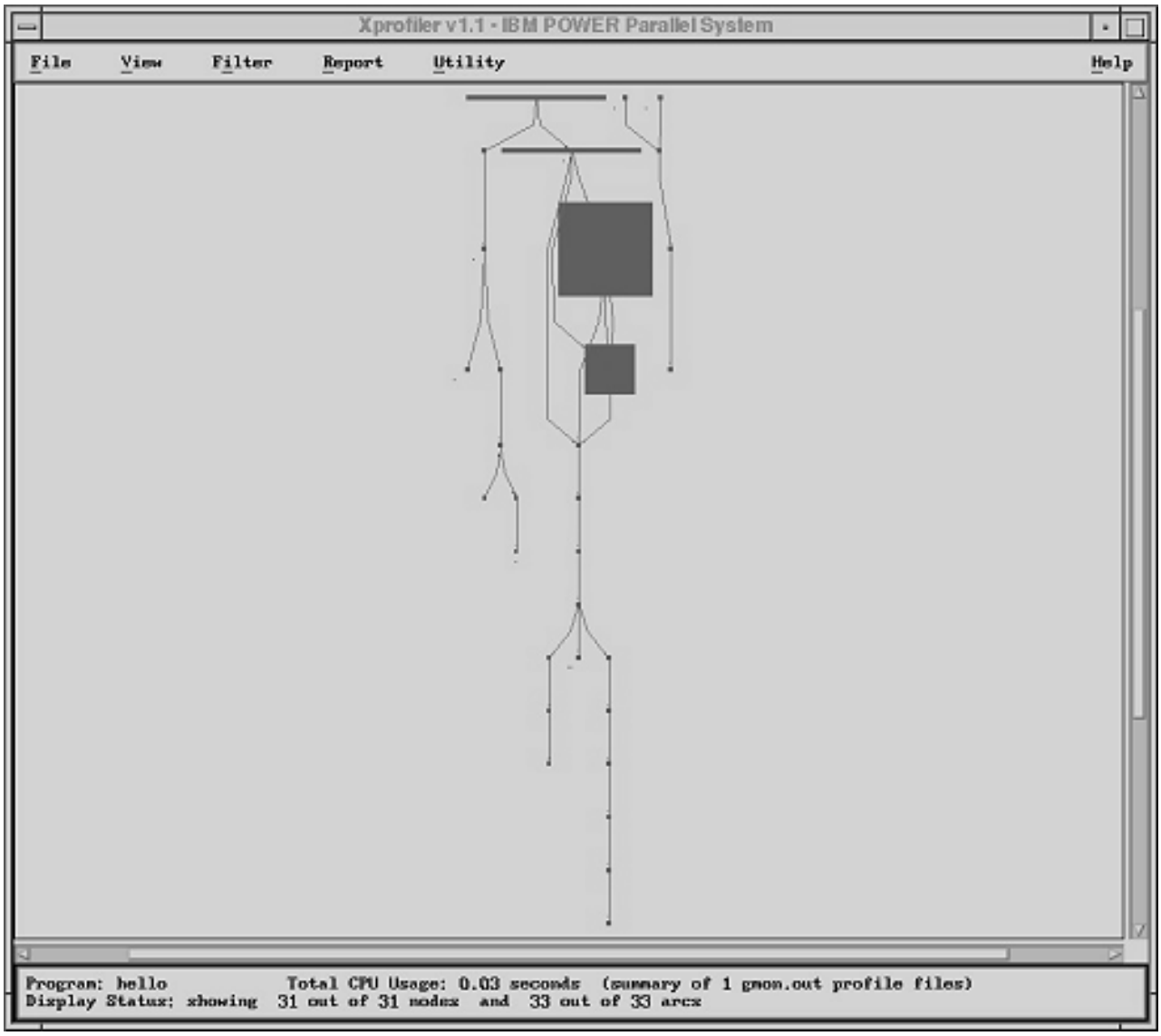


Figure 12. The Xprofiler window with function boxes unclustered

Clustering functions

If the functions within your application are unclustered, you can use an option of the **Filter** menu to cluster them. To do this, select the **Filter** menu and then the **Cluster Functions by Library** option. The libraries within your application appear within their respective cluster boxes.

After you cluster the functions in your application you can further reduce the size (also referred to as *collapse*) of each cluster box by doing the following:

1. Place your mouse cursor over the edge of the cluster box and press the right mouse button. The Cluster Node menu appears.
2. Select the **Collapse Cluster Node** option. The cluster box and its contents now appear as a small solid green box. In the following figure, the **/lib/profiled/libc.a:shr.o** library is collapsed.

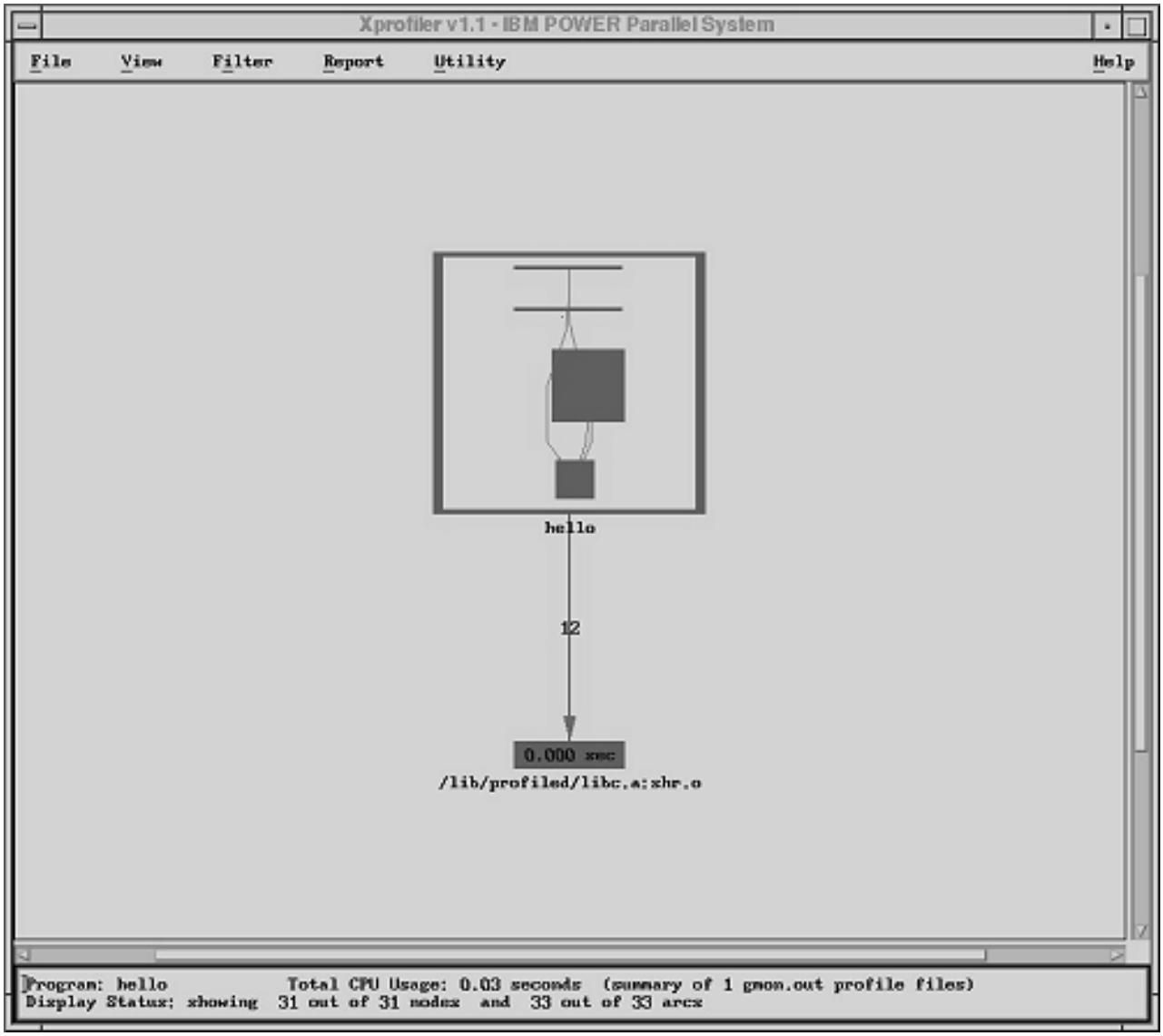


Figure 13. The Xprofiler window with one library cluster box collapsed

To return the cluster box to its original condition (*expand* it), do the following:

1. Place your mouse cursor over the collapsed cluster box and press the right mouse button. The **Cluster Node** menu appears.
2. Select the **Expand Cluster Node** option. The cluster box and its contents appear again.

Unclustering functions

If the functions within your application are clustered, you can use an option of the **Filter** menu to uncluster them. To do this, select the **Filter** menu, and then the **Uncluster Functions** option. The cluster boxes disappear and the functions boxes of each library expand to fill the Xprofiler window.

If your functions have been clustered, you can remove one or more (but not all) cluster boxes. For example, if you want to uncluster only the functions of your executable file, but keep its shared libraries within their cluster boxes, you would do the following:

1. Place your mouse cursor over the edge of the cluster box that contains the executable and press the right mouse button. The **Cluster Node** menu appears.

2. Select the **Remove Cluster Box** option. The cluster box is removed and the function boxes and call arcs that represent the executable functions, now appear in full detail. The function boxes and call arcs of the shared libraries remain within their cluster boxes, which now appear smaller to make room for the unclustered executable function boxes. The folowing figure shows the hello_world executable file with its cluster box removed. Its shared library remains within its cluster box.

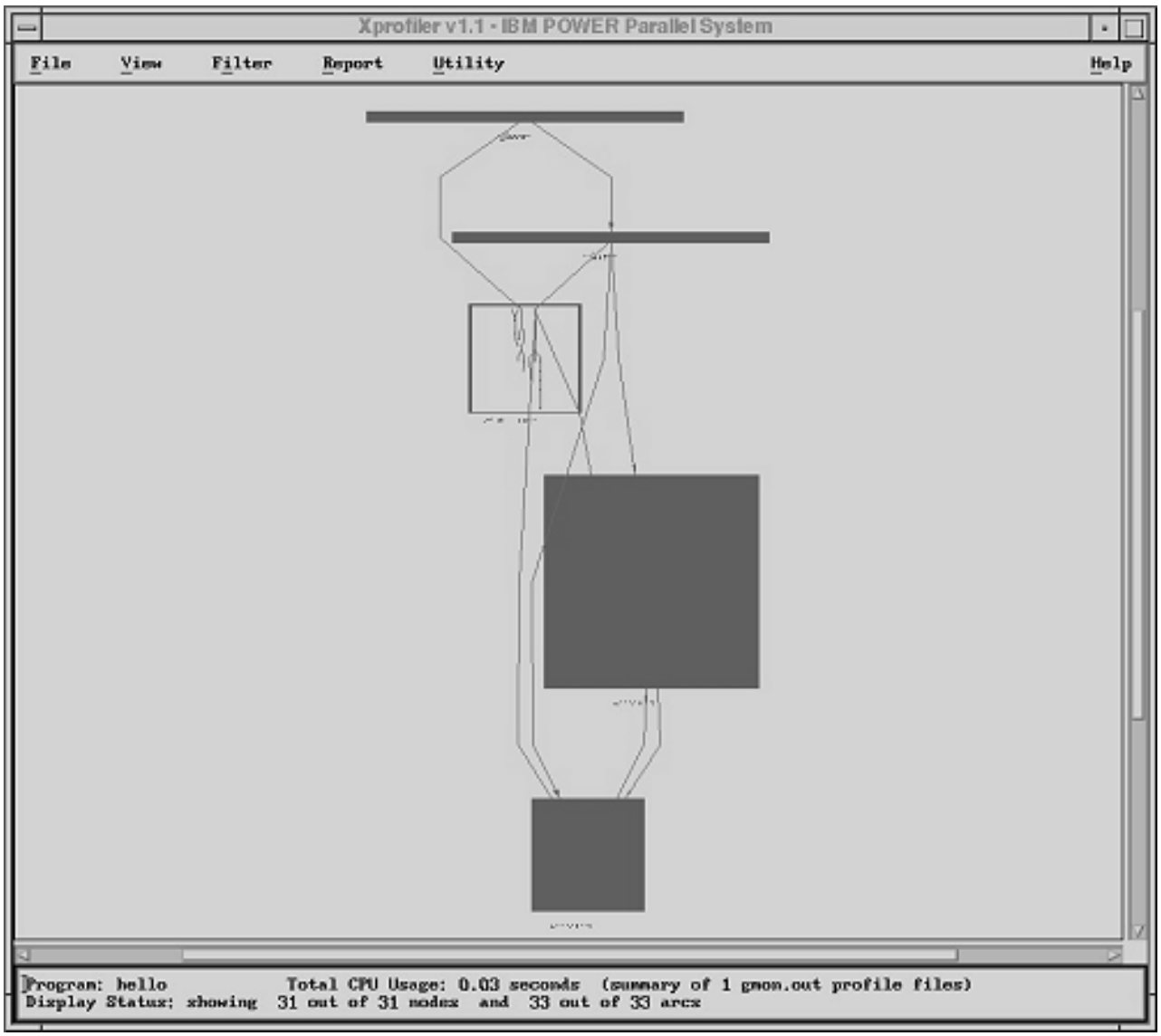


Figure 14. The Xprofiler window with one library cluster box removed

Locating specific objects in the function call tree

If you are interested in one or more specific functions in a complex program, you may need help locating their corresponding function boxes in the function call tree.

If you want to locate a single function, and you know its name, you can use the **Locate Function By Name** option of the **Utility** menu. To locate a function by name, do the following:

1. Select the **Utility** menu, and then the **Locate Function By Name** option. The **Search By Function Name Dialog** window appears.
2. Type the name of the function you want to locate in the **Enter Function Name** field. The function name you type here must be a continuous string (it cannot include blanks).

3. Click **OK** or **Apply**. The corresponding function box is highlighted (its color changes to red) in the function call tree and Xprofiler zooms in on its location.

   To display the function call tree in full detail again, go to the **View** menu and use the **Overview** option.

You might want to see only the function boxes for the functions that you are concerned with, in addition to other specific functions that are related to it. For example, if you want to see all the functions that directly called the function in which you are interested, it might not be easy to separate these function boxes when you view the entire call tree. You would want to display them, as well as the function of interest, alone.

Each function has its own menu. Through the Function menu, you can choose to see the following for the function you are interested in:

- Parent functions (functions that directly call the function of interest)
- Child functions (functions that are directly called by the function of interest)
- Ancestor functions (functions that can call, directly or indirectly, the function of interest)
- Descendant functions (functions that can be called, directly or indirectly, by the function of interest)
- Functions that belong to the same cycle

When you use these options, Xprofiler erases the current display and replaces it with only the function boxes for the function of interest and all the functions of the type you specified.

Locating and displaying parent functions

A *parent* is any function that directly calls the function in which you are interested. To locate the parent function boxes of the function in which you are interested:

1. Click the function box of interest with the right mouse button. The Function menu appears.
2. From the Function menu, select **Immediate Parents** then **Show Parent Functions Only**. Xprofiler redraws the display to show you only the function boxes for the function of interest and its parent functions.

Locating and displaying child functions

A *child* is any function that is directly called by the function in which you are interested. To locate the child functions boxes for the function in which you are interested:

1. Click the function box of interest with the right mouse button. The Function menu appears.
2. From the Function menu, select **Immediate Children** then **Show Child Functions Only**. Xprofiler redraws the display to show you only the function boxes for the function of interest and its child functions.

Locating and displaying ancestor functions

An *ancestor* is any function that can call, directly or indirectly, the function in which you are interested. To locate the ancestor functions:

1. Click the function box of interest with the right mouse button. The Function menu appears.
2. From the Function menu, select **All Paths To** then **Show Ancestor Functions Only**. Xprofiler redraws the display to show you only the function boxes for the function of interest and its ancestor functions.

Locating and displaying descendant functions

A *descendant* is any function that can be called, directly or indirectly, by the function in which you are interested. To locate the descendant functions (all the functions that the function of interest can reach, directly or indirectly):

1. Click the function box of interest with the right mouse button. The Function menu appears.

2. From the Function menu, select **All Paths From** then **Show Descendant Functions Only**. Xprofiler redraws the display to show you only the function boxes for the function of interest and its descendant functions.

Locating and displaying functions on a cycle

To locate the functions that are on the same cycle as the function in which you are interested:

1. Click the function box of interest with the right mouse button. The Function menu appears.
2. From the Function menu, select **All Functions on the Cycle** then **Show Cycle Functions Only**. Xprofiler redraws the display to show you only the function of interest and all the other functions on its cycle.

Obtaining performance data for your application

With Xprofiler, you can get performance data for your application on a number of levels, and in a number of ways. You can easily view data pertaining to a single function, or you can use the reports provided to get information on your application as a whole.

Obtaining basic data

Xprofiler makes it easy to get data on specific items in the function call tree. After you have located the item you are interested in, you can get data a number of ways. If you are having trouble locating a function in the function call tree, see “Locating specific objects in the function call tree” on page 35.

Basic function data

Below each function box in the function call tree is a label that contains basic performance data.

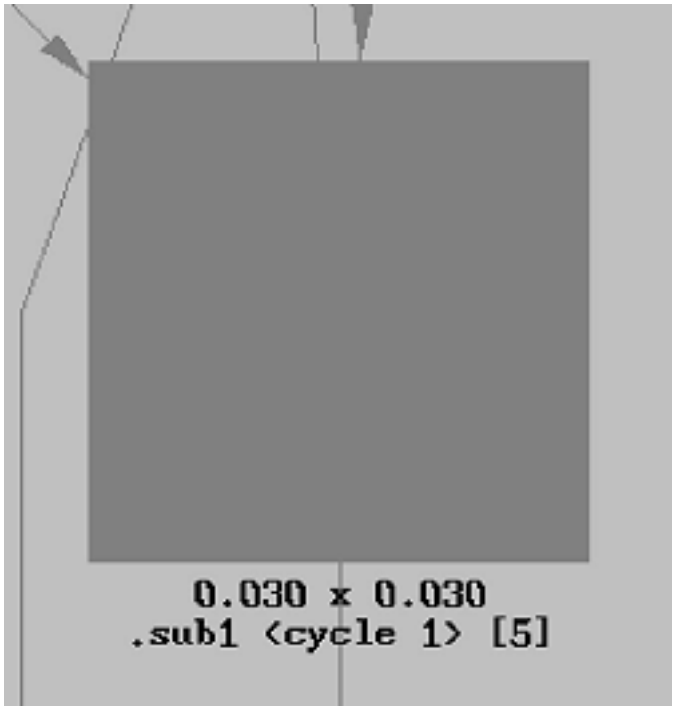


Figure 15. An example of a function box label

The label contains the name of the function, its associated cycle, if any, and its index. In the preceding figure, the name of the function is **sub1**. It is associated with cycle 1, and its index is 5. Also, depending on whether the function call tree is viewed in summary mode or average mode, the label will contain different information.

If the function call tree is viewed in summary mode, the label will contain the following information:

- The total amount of CPU time (in seconds) this function spent on itself plus the amount of CPU time it spent on its descendants (the number on the left of the *x*).
- The amount of CPU time (in seconds) this function spent only on itself (the number on the right of the *x*).

If the function call tree is viewed in average mode, the label will contain the following information:

- The average CPU time (in seconds), among all the input **gmon.out** files, used on the function itself
- The standard deviation of CPU time (in seconds), among all the input **gmon.out** files, used on the function itself

For more information about summary mode and average mode, see “Controlling the representation of the function call tree” on page 26.

Because labels are not always visible in the Xprofiler window when it is fully zoomed out, you may need to zoom in on it in order to see the labels. For information about how to do this, see “Information boxes”.

Basic call data

Call arc labels appear over each call arc. The label indicates the number of calls that were made between the two functions (from caller to callee). For example:

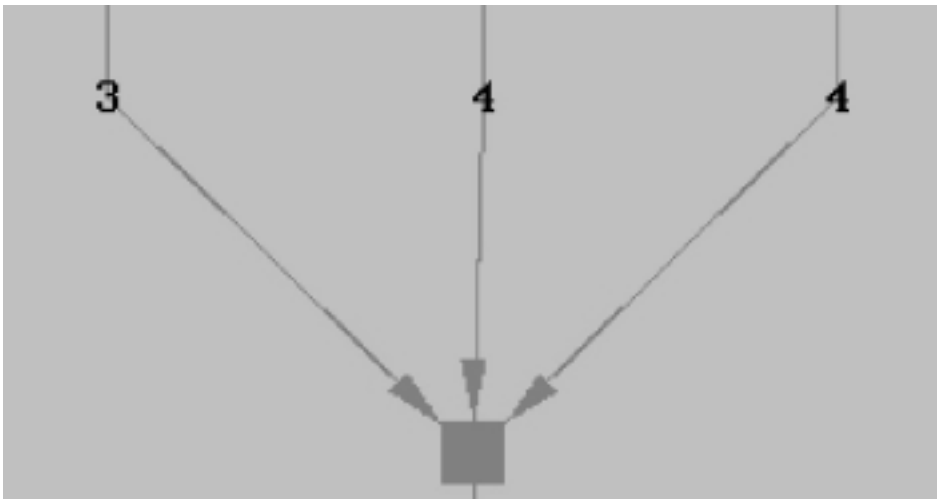


Figure 16. An example of a call arc label

To see a call arc label, you can zoom in on it. For information about how to do this, see “Information boxes”.

Basic cluster data

Cluster box labels indicate the name of the library that is represented by that cluster. If it is a shared library, the label shows its full path name.

Information boxes

For each function box, call arc, and cluster box, a corresponding information box gives you the same basic data that appears on the label. This is useful when the Xprofiler display is fully zoomed out and the labels are not visible. To access the information box, click on the function box, call arc, or cluster box (place the mouse pointer over the edge of the box) with the left mouse button. The information box appears.

For a function, the information box contains the following:

- The name of the function, its associated cycle, if any, and its index.

- The amount of CPU used by this function. There are two values supplied in this field. The first is the amount of CPU time spent on this function plus the time spent on its descendants. The second value represents the amount of CPU time this function spent only on itself.
- The number of times this function was called (by itself or any other function in the application).

  For a call, the information box contains the following:

  – The caller and callee functions (their names) and their corresponding indexes
  – The number of times the caller function called the callee

  For a cluster, the information box contains the following:

- The name of the library
- The total CPU usage (in seconds) consumed by the functions within it

Function menu Statistics Report option

You can get performance statistics for a single function through the **Statistics Report** option of the **Function** menu. This option lets you see data on the CPU usage and call counts of the selected function. If you are using more than one **gmon.out** file, the **Statistics Report** option breaks down the statistics for each **gmon.out** file you use.

When you select the **Statistics Report** menu option, the Function Level Statistics Report window appears.

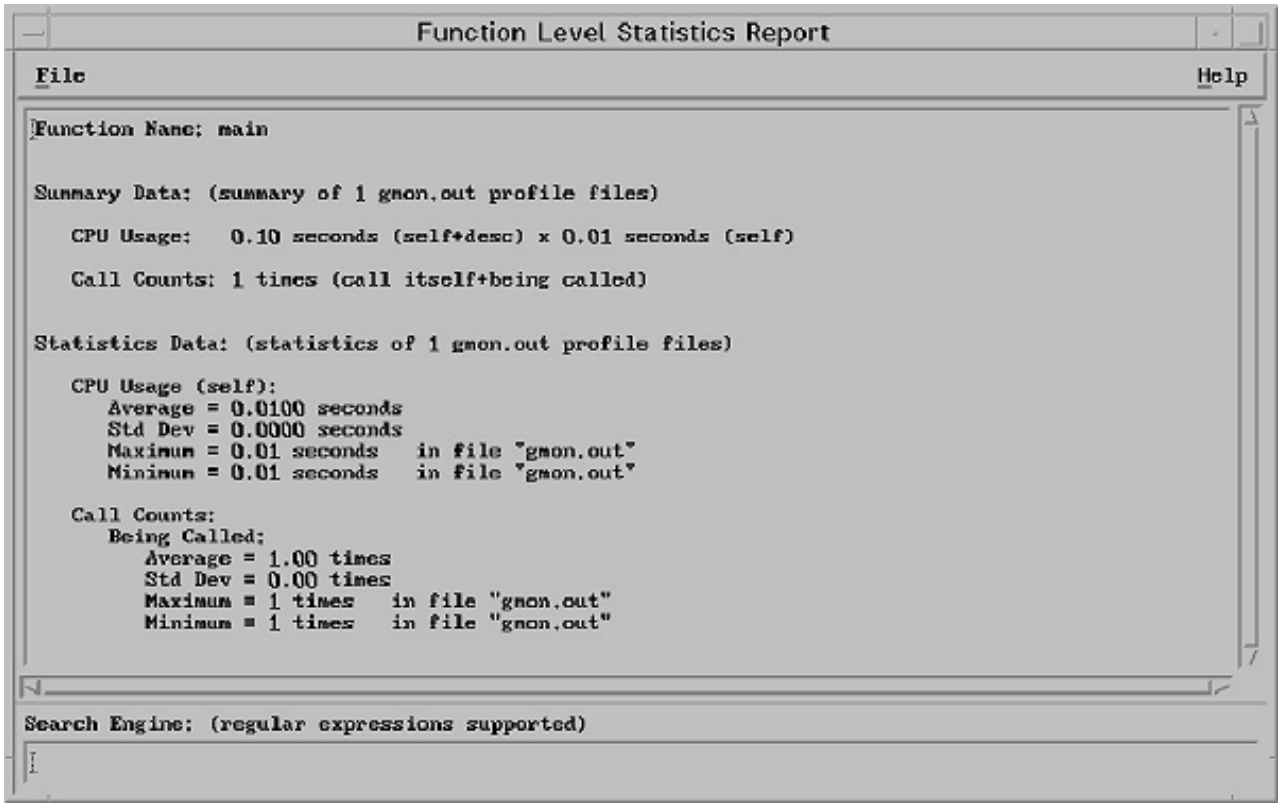


Figure 17. The Function Level Statistics Report window

The Function Level Statistics Report window provides the following information:

Function Name

The name of the function you selected.

Summary Data
The total amount of CPU used by this function. If you used multiple **gmon.out** files, the value shown here represents their sum.

The **CPU Usage** field indicates:

- The amount of CPU time used by this function. There are two values supplied in this field. The first is the amount of CPU time spent on this function plus the time spent on its descendants. The second value represents the amount of CPU time this function spent only on itself.

The **Call Counts** field indicates:

- The number of times this function called itself, plus the number of times it was called by other functions.

Statistics Data
The CPU usage and calls made to or by this function, broken down for each **gmon.out** file.

The **CPU Usage** field indicates:

- **Average**

  The average CPU time used by the data in each **gmon.out** file.
- **Std Dev**

  Standard deviation. A value that represents the difference in CPU usage samplings, per function, from one **gmon.out** file to another. The smaller the standard deviation, the more balanced the workload.
- **Maximum**

  Of all the **gmon.out** files, the maximum amount of CPU time used. The corresponding **gmon.out** file appears to the right.
- **Minimum**

  Of all the **gmon.out** files, the minimum amount of CPU time used. The corresponding **gmon.out** file appears to the right.

The **Call Counts** field indicates:

- **Average**

  The average number of calls made to this function or by this function, for each **gmon.out** file.
- **Std Dev**

  Standard deviation. A value that represents the difference in call count sampling, per function, from one **gmon.out** file to another. A small standard deviation value in this field means that the function was almost always called the same number of times in each **gmon.out** file.
- **Maximum**

  The maximum number of calls made to this function or by this function in a single **gmon.out** file. The corresponding **gmon.out** file appears to the right.
- **Minimum**

  The minimum number of calls made to this function or by this function in a single **gmon.out** file. The corresponding **gmon.out** file appears to the right.

Getting detailed data from reports
Xprofiler provides performance data in textual and tabular format. This data is provided in various tables called *reports.* Similar to the **gprof** command, Xprofiler generates the **Flat Profile**, **Call Graph Profile**, and **Function Index** reports, as well as two additional reports.

You can access the Xprofiler reports from the **Report** menu. The **Report** menu displays the following:

- Flat Profile
- Call Graph Profile

- Function Index
- Function Call Summary
- Library Statistics

Each report window includes a File menu. Under the File menu is the **Save As** option, which lets you save the report to a file. For information about using the **Save File Dialog** window to save a report to a file, see “Saving the Call Graph Profile, Function Index, and Flat Profile reports to a file” on page 49.

Note: If you select the **Save As** option from the **Flat Profile**, **Function Index**, or **Function Call Summary** report window, you must either complete the save operation or cancel it before you can select any other option from the menus of these reports. You can, however, use the other Xprofiler menus before completing the save operation or canceling it, with the exception of the **Load Files** option of the **File** menu, which remains unavailable.

Each of the Xprofiler reports are explained as follows.

Flat Profile report

When you select the **Flat Profile** menu option, the Flat Profile window appears. The Flat Profile report shows you the total execution times and call counts for each function (including shared library calls) within your application. The entries for the functions that use the greatest percentage of the total CPU usage appear at the top of the list, while the remaining functions appear in descending order, based on the amount of time used.

Unless you specified the **-z** flag, the **Flat Profile** report does not include functions that have no CPU usage and no call counts. The data presented in the **Flat Profile** window is the same data that is generated with the **gprof** command.

The Flat Profile report looks similar to the following:

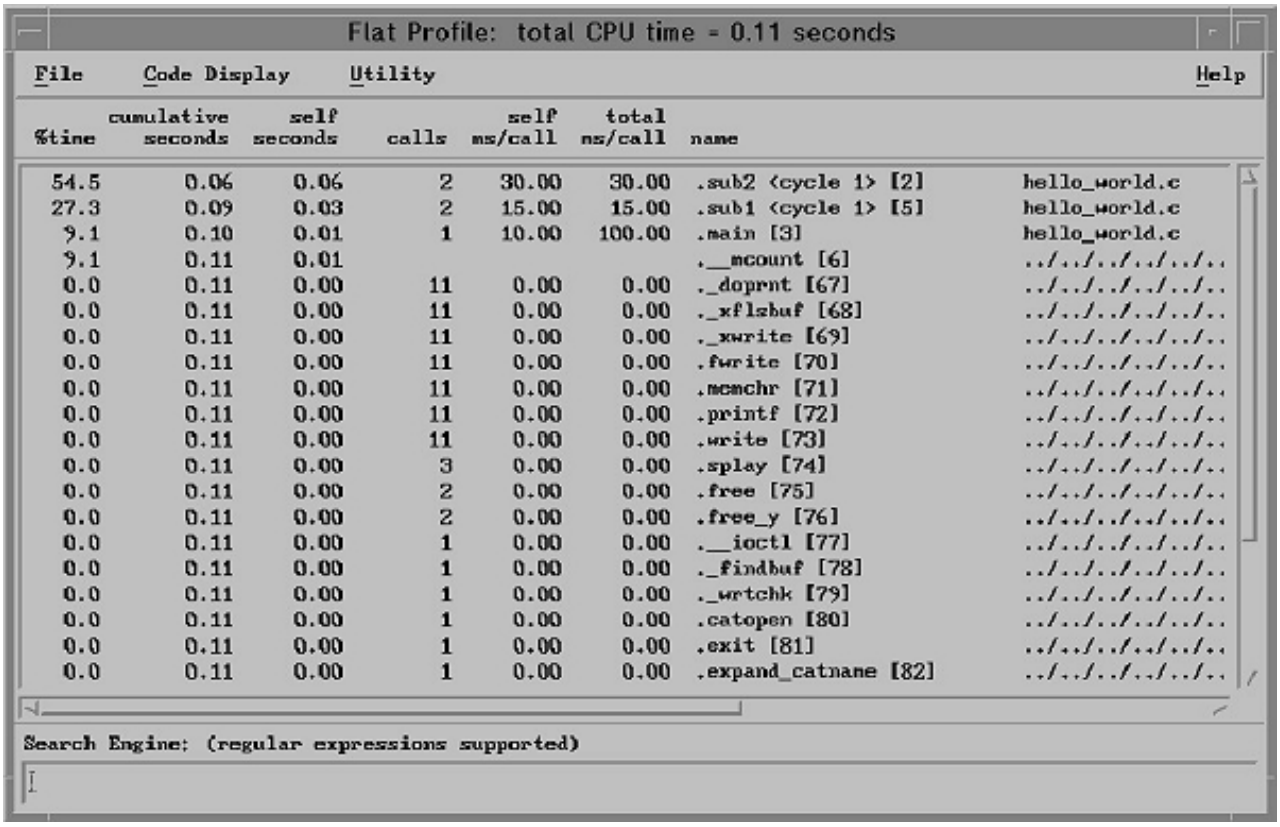


Figure 18. The Flat Profile report

Flat Profile window fields: The **Flat Profile** window contains the following fields:

- **%time**

  The percentage of the program's total CPU usage that is consumed by this function.
- **cumulative seconds**

  A running sum of the number of seconds used by this function and those listed above it.
- **self seconds**

  The number of seconds used by this function alone. Xprofiler uses the **self seconds** values to sort the functions of the **Flat Profile** report.
- **calls**

  The number of times this function was called (if this function is profiled). Otherwise, it is blank.
- **self ms/call**

  The average number of milliseconds spent in this function per call (if this function is profiled). Otherwise, it is blank.
- **total ms/call**

  The average number of milliseconds spent in this function and its descendants per call (if this function is profiled). Otherwise, it is blank.
- **name**

  The name of the function. The *index* appears in brackets (**[]**) to the right of the function name. The index serves as the function's identifier within Xprofiler. It also appears below the corresponding function in the function call tree.

Call Graph Profile report

The Call Graph Profile menu option lets you view the functions of your application, sorted by the percentage of total CPU usage that each function, and its descendants, consumed. When you select this option, the Call Graph Profile window appears.

Unless you specified the **-z** flag, the **Call Graph Profile** report does not include functions whose CPU usage is 0 (zero) and have no call counts. The data presented in the **Call Graph Profile** window is the same data that is generated with the **gprof** command.

The Call Graph Profile report looks similar to the following:

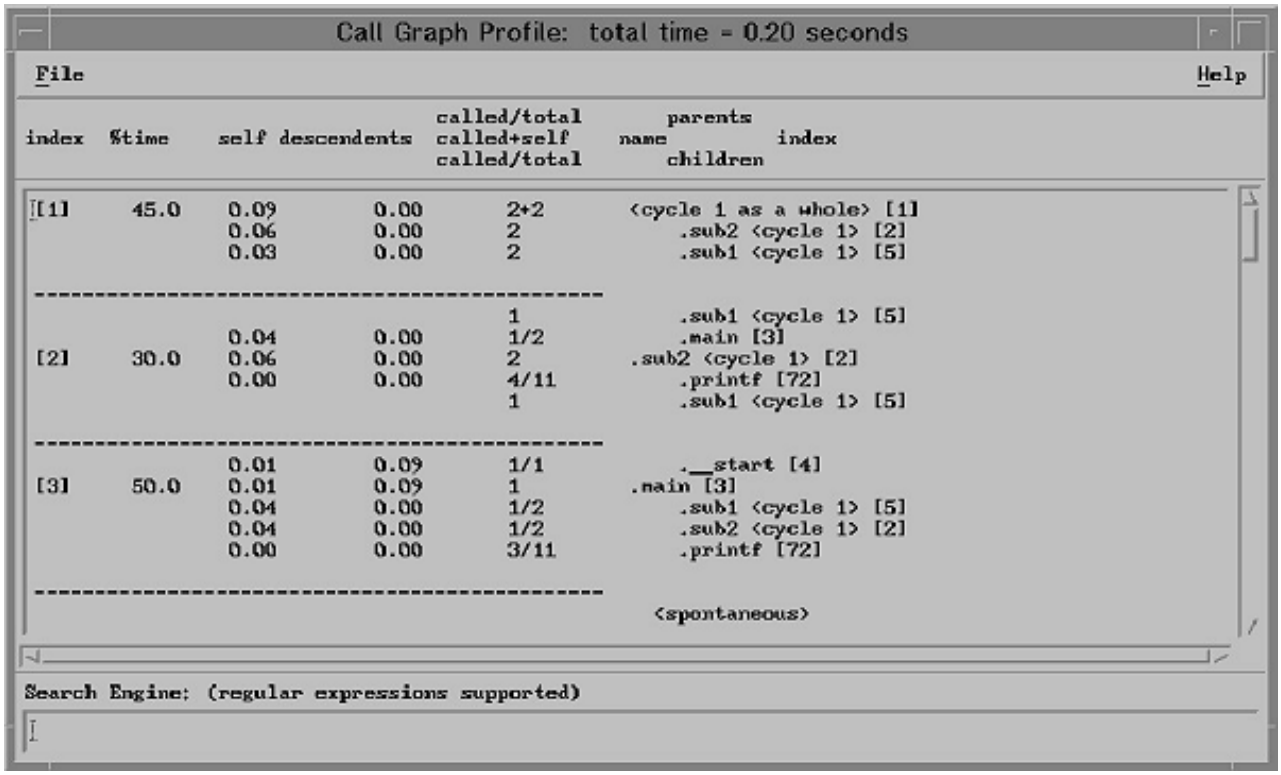


Figure 19. The Call Graph Profile report

Call Graph Profile window fields: The **Call Graph Profile** window contains the following fields:

- **index**

  The index of the function in the **Call Graph Profile**. Each function in the **Call Graph Profile** has an associated index number which serves as the function's identifier. The same index also appears with each function box label in the function call tree, as well as other Xprofiler reports.
- **%time**

  The percentage of the program's total CPU usage that was consumed by this function and its descendants.
- **self**

  The number of seconds this function spends within itself.
- **descendants**

  The number of seconds spent in the descendants of this function, on behalf of this function.
- **called/total, called+self, called/total**

  The heading of this column refers to the different kinds of calls that take place within your program. The values in this field correspond to the functions listed in the **name, index, parents, children** field to its

right. Depending on whether the function is a parent, a child, or the function of interest (the function with the index listed in the **index** field of this row), this value might represent the number of times that:

- a parent called the function of interest
- the function of interest called itself, recursively
- the function of interest called a child

In the following figure, **sub2** is the function of interest, **sub1** and **main** are its parents, and **printf** and **sub1** are its children.

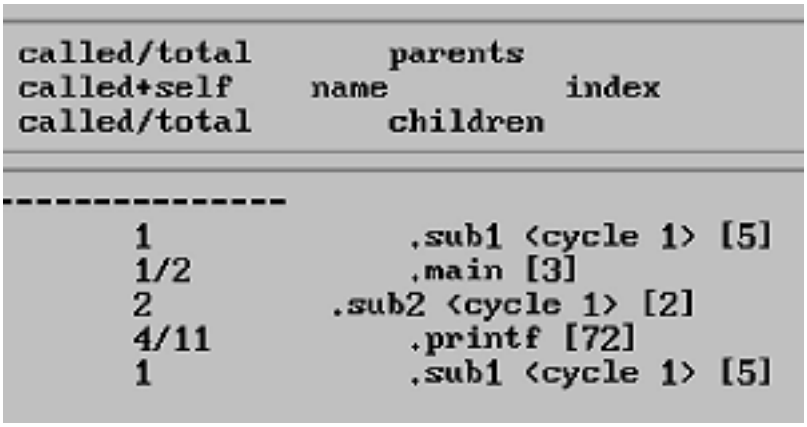


Figure 20. The called/total, call/self, called/total field

- **called/total**

  For a parent function, the number of calls made to the function of interest, as well as the total number of calls it made to all functions.
- **called+self**

  The number of times the function of interest called itself, recursively.
- **name, index, parents, children**

  The layout of the heading of this column indicates the information that is provided. To the left is the name of the function, and to its right is the function's index number. Appearing above the function are its parents, and below are its children.

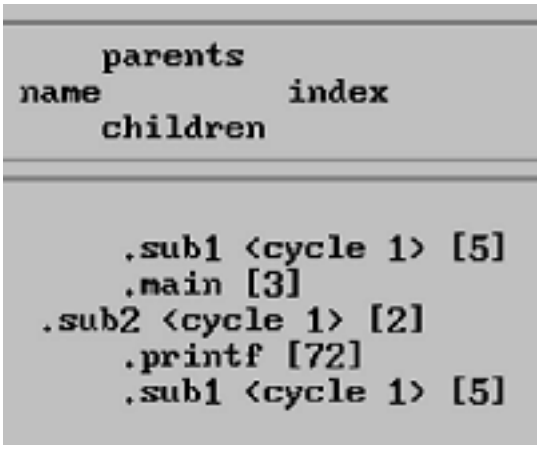


Figure 21. The name/index/parents/children field

- **name**

The name of the function, with an indication of its membership in a cycle, if any. The function of interest appears to the left, while its parent and child functions are indented above and below it.

- **index**

  The index of the function in the **Call Graph Profile**. This number corresponds to the index that appears in the *index* column of the **Call Graph Profile** and the on the function box labels in the function call tree.

- **parents**

  The parents of the function. A *parent* is any function that directly calls the function in which you are interested.

  If any portion of your application was not compiled with the **-pg** flag, Xprofiler cannot identify the parents for the functions within those portions. As a result, these parents will be listed as *spontaneous* in the **Call Graph Profile** report.

- **children**

  The children of the function. A *child* is any function that is directly called by the function in which you are interested.

Function Index report

The Function Index menu option lets you view a list of the function names included in the function call tree. When you select this option, the Function Index window appears and displays the function names in alphabetical order. To the left of each function name is its *index*, enclosed in brackets (**[]**). The index is the function's identifier, which is assigned by Xprofiler. An index also appears on the label of each corresponding function box in the function call tree, as well as on other reports.

Unless you specified the **-z** flag, the Function Index report does not include functions that have no CPU usage and no call counts.

Like the **Flat Profile** menu option, the **Function Index** menu option includes a **Code Display** menu, so you can view source code or disassembler code. See "Looking at your code" on page 50 for more information.

The **Function Index** report looks similar to the following:

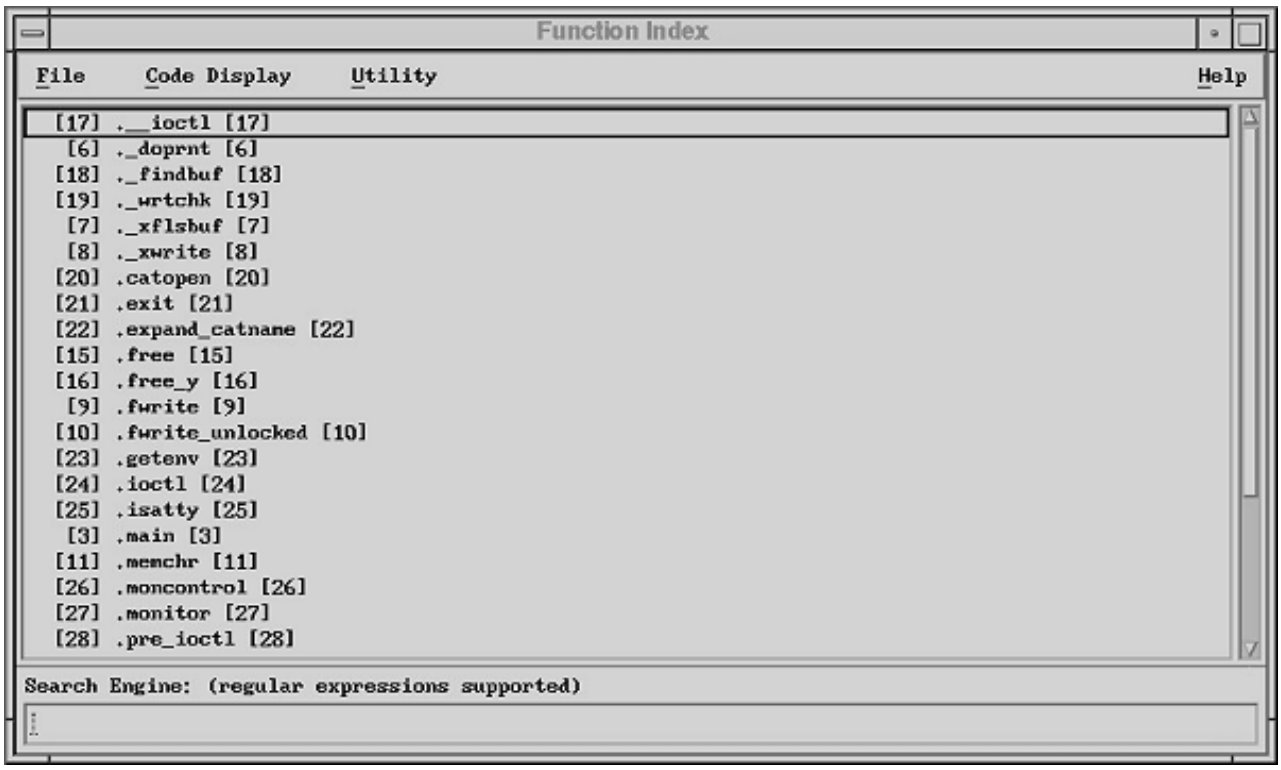


Figure 22. The Function Index report

Function Call Summary report

The Function Call Summary menu option lets you display all the functions in your application that call other functions. They appear as caller-callee pairs (call arcs, in the function call tree), and are sorted by the number of calls in descending order. When you select this option, the Function Call Summary window appears.

The **Function Call Summary** report looks similar to the following:

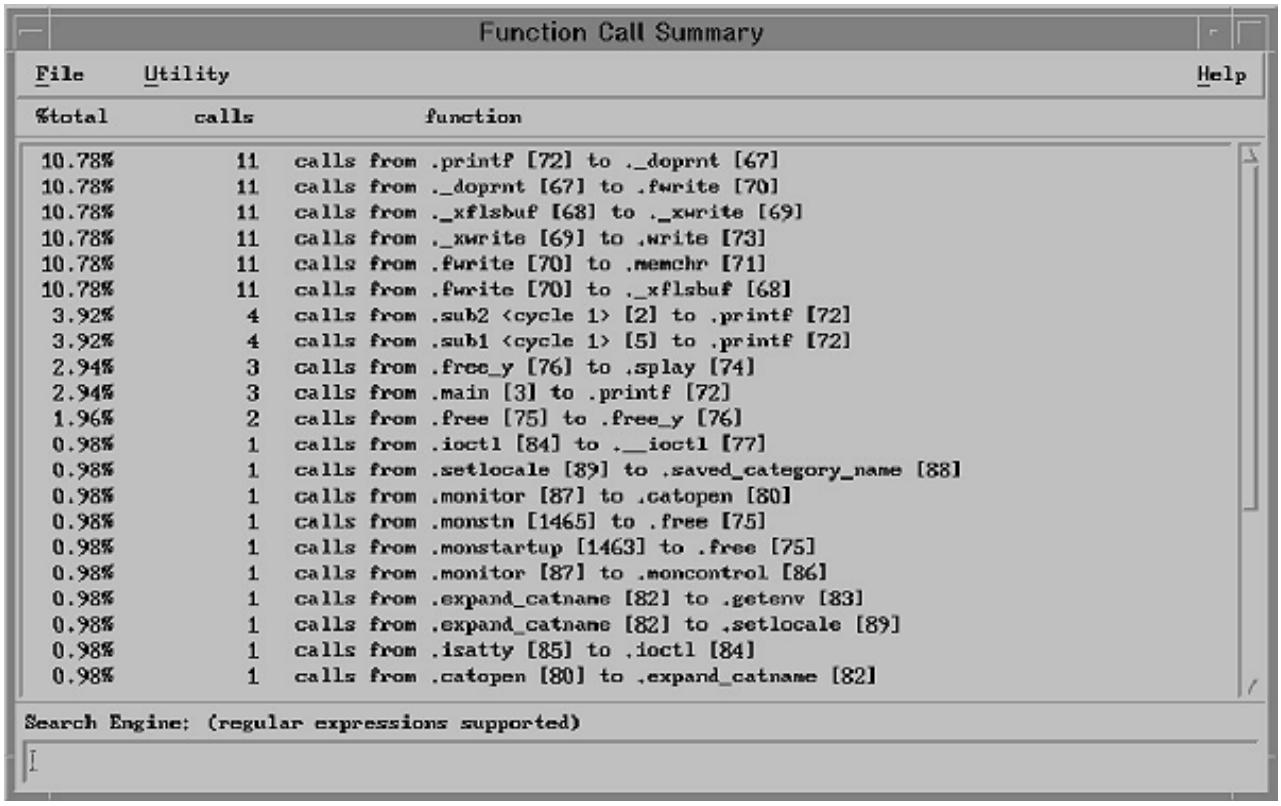


Figure 23. The Function Call Summary report

Function Call Summary window fields: The Function Call Summary window contains the following fields:

- **%total**

  The percentage of the total number of calls generated by this caller-callee pair
- **calls**

  The number of calls attributed to this caller-callee pair
- **function**

  The name of the caller function and callee function

Library Statistics report

The Library Statistics menu option lets you display the CPU time consumed and call counts of each library within your application. When you select this option, the Library Statistics window appears.

The **Library Statistics** report looks similar to the following:

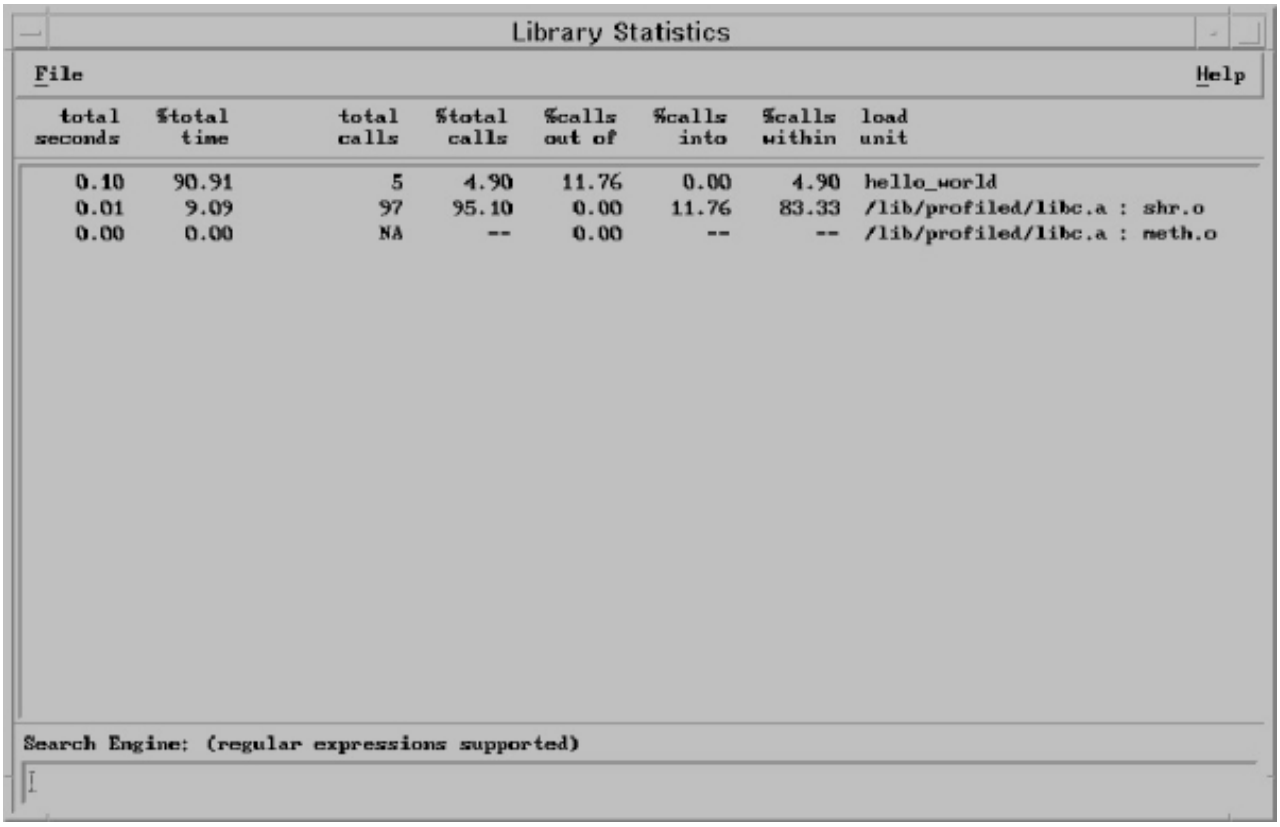


Figure 24. The Library Statistics report

Library Statistics window fields: The **Library Statistics** window contains the following fields:

- **total seconds**
  The total CPU usage of the library, in seconds
- **%total time**
  The percentage of the total CPU usage that was consumed by this library
- **total calls**
  The total number of calls that this library generated
- **%total calls**
  The percentage of the total calls that this library generated
- **%calls out of**
  The percentage of the total number of calls made from this library to other libraries
- **%calls into**
  The percentage of the total number of calls made from other libraries into this library
- **%calls within**
  The percentage of the total number of calls made between the functions within this library
- **load unit**
  The library's full path name

Saving reports to a file

Xprofiler lets you save any of the reports you generate with the **Report** menu to a file. You can do this using the **File** and **Report** menus of the Xprofiler GUI.

Saving a single report: To save a single report, go to the **Report** menu on the Xprofiler main window and select the report you want to save. Each report window includes a **File** menu. Select the **File** menu and then the **Save As** option to save the report. A **Save** dialog window appears, which is named according to the report from which you selected the **Save As** option. For example, if you chose **Save As** from the **Flat Profile** window, the save window is named **Save Flat Profile Dialog**.

Saving the Call Graph Profile, Function Index, and Flat Profile reports to a file: You can save the **Call Graph Profile**, **Function Index**, and **Flat Profile** reports to a single file through the **File** menu of the Xprofiler main window. The information you generate here is identical to the output of the **gprof** command. From the **File** menu, select the **Save As** option. The **Save File Dialog** window appears.

To save the reports, do the following:

1. Specify the file into which the profiled data should be placed. You can specify either an existing file or a new one. To specify an existing file, use the scroll bars of the **Directories** and **Files** selection boxes to locate the file. To make locating your files easier, you can also use the **Filter** button (see “Filtering what you see” on page 27 for more information). To specify a new file, type its name in the **Selection** field.
2. Click **OK**. A file that contains the profiled data appears in the directory you specified, under the name you gave it.

Note: After you select the **Save As** option from the **File** menu and the Save Profile Reports window opens, you must either complete the save operation or cancel it before you can select any other option from the menus of its parent window. For example, if you select the **Save As** option from the **Flat Profile** report and the Save File Dialog window appears, you cannot use any other option of the **Flat Profile** report window.

The **File Selection** field of the Save File Dialog window follows Motif standards.

Saving summarized data from multiple profile data files: If you are profiling a parallel program, you can specify more than one profile data (**gmon.out**) file when you start Xprofiler. The **Save gmon.sum As** option of the **File** menu lets you save a summary of the data in each of these files to a single file.

The Xprofiler **Save gmon.sum As** option produces the same result as the **xprofiler -s** command and the **gprof -s** command. If you run Xprofiler later, you can use the file you create here as input with the **-s** flag. In this way, you can accumulate summary data over several runs of your application.

To create a summary file, do the following:

1. Select the **File** menu, and then the **Save gmon.sum As** option. The **Save gmon.sum Dialog** window appears.
2. Specify the file into which the summarized, profiled data should be placed. By default, Xprofiler puts the data into a file called **gmon.sum**. To specify a new file, type its name in the selection field. To specify an existing file, use the scroll bars of the **Directories** and **Files** selection boxes to locate the file you want. To make locating your files easier, you can also use the **Filter** button (see “Filtering what you see” on page 27 for information).
3. Click **OK**. A file that contains the summary data appears in the directory you specified, under the name you specified.

Saving a configuration file: The **Save Configuration** menu option lets you save the names of the functions that are displayed currently to a file. Later, in the same Xprofiler session or in a different session, you can read this configuration file in using the **Load Configuration** option. For more information, see “Loading a configuration file” on page 50.

To save a configuration file, do the following:

1. Select the **File** menu, and then the **Save Configuration** option. The **Save Configuration File Dialog** window opens with the *program***.cfg** file as the default value in the **Selection** field, where *program* is the name of the input **a.out** file.

   You can use the default file name, enter a file name in the **Selection** field, or select a file from the file list.
2. Specify a file name in the **Selection** field and click **OK**. A configuration file is created that contains the name of the program and the names of the functions that are displayed currently.
3. Specify an existing file name in the **Selection** field and click **OK**. An Overwrite File Dialog window appears so you can check the file before overwriting it.

If you selected the **Forced File Overwriting** option in the Runtime Options Dialog window, the Overwrite File Dialog window does not open and the specified file is overwritten without warning.

Loading a configuration file: The **Load Configuration** menu option lets you read in a configuration file that you saved. See "Saving a configuration file" on page 49 for more information. The **Load Configuration** option automatically reconstructs the function call tree according to the function names recorded in the configuration file.

To load a configuration file, do the following:

1. Select the File menu, and then the **Load Configuration** option. The **Load Configuration File Dialog** window opens. If configuration files were loaded previously during the current Xprofiler session, the name of the file that was most recently loaded will appear in the **Selection** field of this dialog.

   You can also load the file with the **-c** flag. For more information, see "Specifying command line options (from the GUI)" on page 13.
2. Select a configuration file from the dialog's **Files** list or specify a file name in the **Selection** field and click **OK**. The function call tree is redrawn to show only those function boxes for functions that are listed in the configuration file and are called within the program that is currently represented in the display. All corresponding call arcs are also drawn.

   If the **a.out** name, that is, the program name in the configuration file, is different from the **a.out** name in the current display, a confirmation dialog asks you whether you still want to load the file.
3. If after loading a configuration file, you want to return the function call tree to its previous state, select the **Filter** menu, and then the **Undo** option.

Looking at your code

Xprofiler provides several ways for you to view your code. You can view the source code or the disassembler code for your application, for each function. This also applies to any included function code that your application might use.

To view source or included function code, use the Source Code window. To view disassembler code, use the Disassembler Code window. You can access these windows through the Report menu of the Xprofiler GUI or the Function menu of the function you are interested in.

Viewing the source code

Both the Function menu and Report menu allow you to access the Source Code window, from which you can view your code.

To access the Source Code window through the Function menu:

1. Click the function box you are interested in with the right mouse button. The **Function** menu appears.
2. From the **Function** menu, select the **Show Source Code** option. The **Source Code** window appears.

To access the **Source Code** window through the Report menu:

1. Select the Report menu, and then the **Flat Profile** option. The **Flat Profile** window appears.

2. From the **Flat Profile** window, select the function you would like to view by clicking on its entry in the window. The entry is highlighted to show that it is selected.
3. Select the **Code Display** menu, and then the **Show Source Code** option. The **Source Code** window appears, containing the source code for the function you selected.

Using the Source Code window: The **Source Code** window shows you the source code file for the function you specified from the **Flat Profile** window or the **Function** menu. The **Source Code** window looks similar to the following:

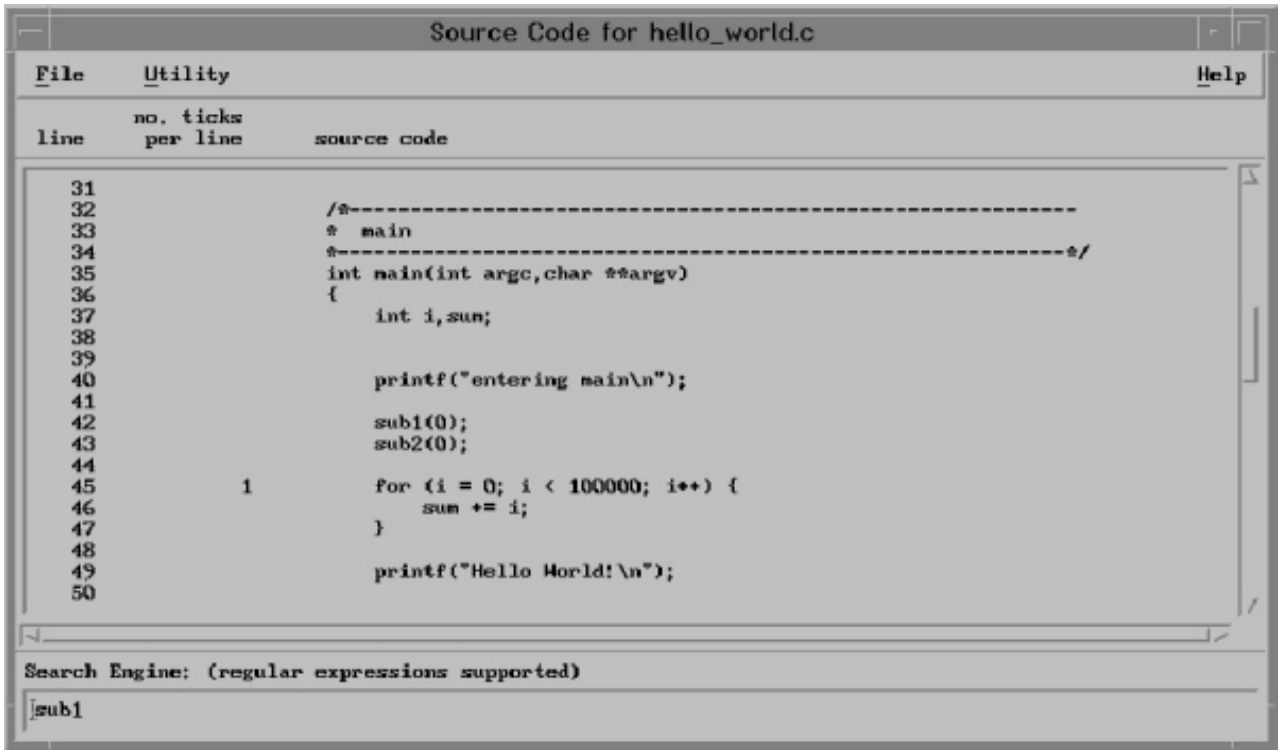


Figure 25. The Source Code window

The **Source Code** window contains information in the following fields:

- **line**

  The source code line number.
- **no. ticks per line**

  Each tick represents .01 seconds of CPU time used. The value in this field represents the number of ticks used by the corresponding line of code. For example, if the number 3 appeared in this field, for a source statement, this source statement would have used .03 seconds of CPU time. The CPU usage data only appears in this field if you used the **-g** flag when you compiled your application. Otherwise, this field is blank.
- **source code**

  The application's source code.

The **Source Code** window contains the following menus:

- **File**

  The **Save As** option lets you save the annotated source code to a file. When you select this option, the **Save File Dialog** window appears. For more information about using the **Save File Dialog** window, see "Saving the Call Graph Profile, Function Index, and Flat Profile reports to a file" on page 49.

  To close the **Source Code** window, select **Close**.

- **Utility**

  This menu contains the **Show Included Functions** option.

For C++ users, the **Show Included Functions** option lets you view the source code of included function files that are included by the application's source code.

If a selected function does not have an included function file associated with it or does not have the function file information available because the **-g** flag was not used for compiling, the **Utility** menu will be greyed out. The availability of the **Utility** menu indicates whether there is any included function-file information associated with the selected function.

When you select the **Show Included Functions** option, the **Included Functions Dialog** window appears, which lists all of the included function files. Specify a file by either clicking on one of the entries in the list with the left mouse button, or by typing the file name in the **Selection** field. Then click **OK** or **Apply**. After you select a file from the **Included Functions Dialog** window, the **Included Function File** window appears, displaying the source code for the file that you specified.

Viewing the disassembler code

Both the **Function** menu and **Report** menu allow you to access the Disassembler Code window, from which you can view your code.

To access the **Disassembler Code** window through the **Function** menu, do the following:

1. Click on the function you are interested in with the right mouse button. The **Function** menu appears.
2. From the **Function** menu, select the **Show Disassembler Code** option. The **Disassembler Code** window appears.

To access the Disassembler Code window through the **Report** menu, do the following:

1. Select the **Report** menu, and then the **Flat Profile** option. The **Flat Profile** window appears.
2. From the **Flat Profile** window, select the function you want to view by clicking on its entry in the window. The entry is highlighted to show that it is selected.
3. Select the **Code Display** menu, and then the **Show Disassembler Code** option. The Disassembler Code window appears, and contains the disassembler code for the function you selected.

Using the Disassembler Code window: The Disassembler Code window shows you only the disassembler code for the function you specified from the Flat Profile window. The **Disassembler Code** window looks similar to the following:

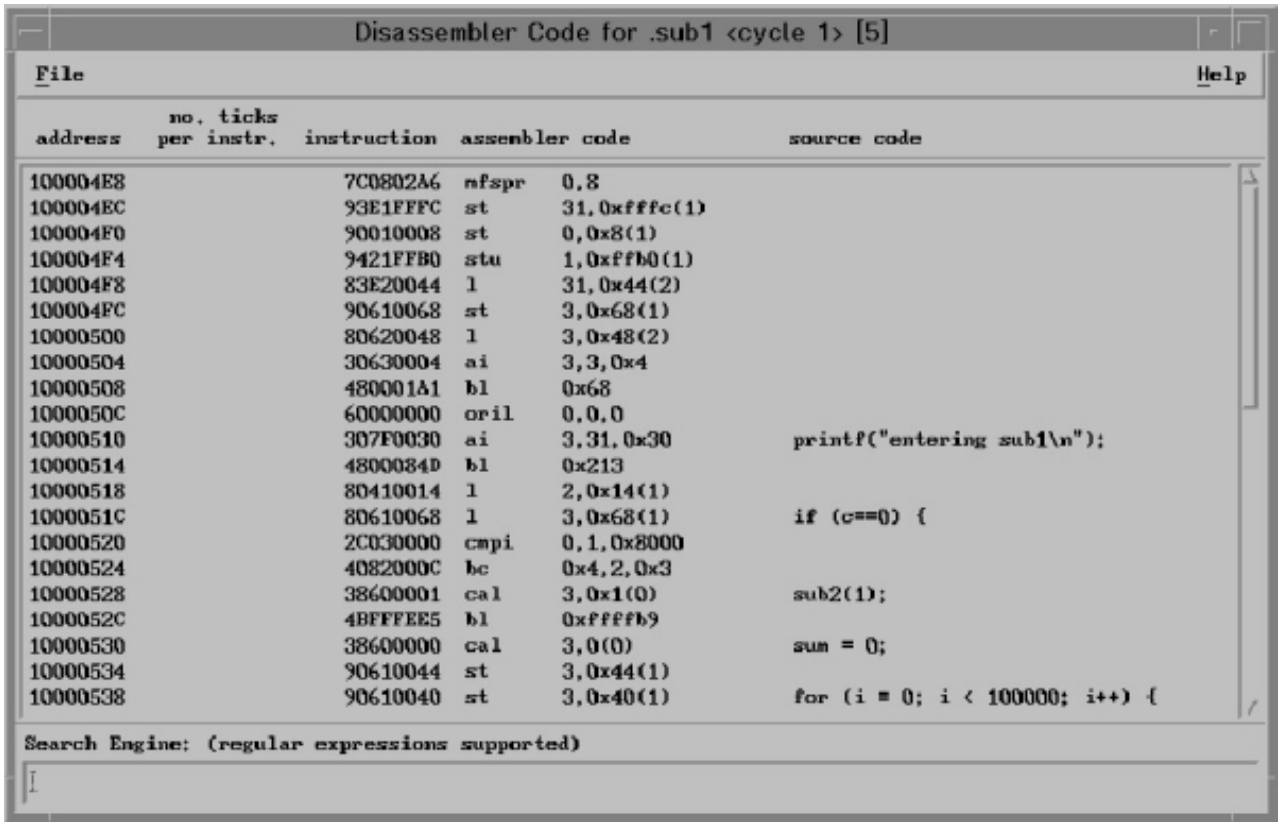


Figure 26. The Disassembler Code window

The Disassembler Code window contains information in the following fields:

- **address**

  The address of each instruction in the function you selected (from either the **Flat Profile** window or the function call tree).
- **no. ticks per instr.**

  Each tick represents .01 seconds of CPU time used. The value in this field represents the number of ticks used by the corresponding instruction. For instance, if the number 3 appeared in this field, this instruction would have used .03 seconds of CPU time.
- **instruction**

  The execution instruction.
- **assembler code**

  The execution instruction's corresponding assembler code.
- **source code**

  The line in your application's source code that corresponds to the execution instruction and assembler code. In order for information to appear in this field, you must have compiled your application with the **-g** flag.

The **Search Engine** field at the bottom of the **Disassembler Code** window lets you search for a specific string in your disassembler code.

The **Disassembler Code** window contains one menu:

- **File**

Select **Save As** to save the annotated disassembler code to a file. When you select this option, the **Save File Dialog** window appears. For information on using the **Save File Dialog** window, see “Saving the Call Graph Profile, Function Index, and Flat Profile reports to a file” on page 49.

To close the **Disassembler Code** window, select **Close**.

Saving screen images of profiled data

The **File** menu of the Xprofiler GUI includes an option called **Screen Dump** that lets you capture an image of the Xprofiler main window. This option is useful if you want to save a copy of the graphical display to refer to later. You can either save the image as a file in PostScript format, or send it directly to a printer.

To capture a window image, do the following:

1. Select **File** and then **Screen Dump**. The **Screen Dump** menu opens.
2. From the **Screen Dump** menu, select **Set Option**. The Screen Dump Options Dialog window appears.

Figure 27. The Screen Dump Options Dialog window

3. Make the appropriate selections in the fields of the **Screen Dump Options Dialog** window, as follows:
   - **Output To:**

This option lets you specify whether you want to save the captured image as a PostScript file or send it directly to a printer.

If you would like to save the image to a file, select the **File** button. This file, by default, is named **Xprofiler.screenDump.ps.0**, and is displayed in the **Default File Name** field of this dialog window. When you select the **File** button, the text in the **Print Command** field greys out.

To send the image directly to a printer, select the **Printer** button. The image is sent to the printer you specify in the **Print Command** field of this dialog window. When you specify the **Print** option, a file of the image is not saved. Also, selecting this option causes the text in the **Default File Name** field is made unavailable.

- **PostScript Output:**

  This option lets you specify whether you want to capture the image in shades of grey or in color.

  If you want to capture the image in shades of grey, select the **GreyShades** button. You must also select the number of shades you want the image to include with the **Number of Grey Shades** option, as discussed below.

  If you want to capture the image in color, select the **Color** button.

- **Number of Grey Shades**

  This option lets you specify the number of grey shades that the captured image will include. Select either the 2, 4, or 16 buttons, depending on the number of shades you want to use. Typically, the more shades you use, the longer it will take to print the image.

- **Delay Before Grab**

  This option lets you specify how much of a delay will occur between activating the capturing mechanism and when the image is actually captured. By default, the delay is set to one second, but you may need time to arrange the window the way you want it. Setting the delay to a longer interval gives you some extra time to do this. You set the delay with the slider bar of this field. The number above the slider indicates the time interval in seconds. You can set the delay to a maximum of thirty seconds.

- **Enable Landscape** (button)

  This option lets you specify that you want the output to be in landscape format (the default is portrait). To select landscape format, select the **Enable Landscape** button.

- **Annotate Output** (button)

  This option lets you specify that you would like information about how the file was created to be included in the PostScript image file. By default, this information is not included. To include this information, select the **Annotate Output** button.

- **Default File Name** (field)

  If you chose to put your output in a file, this field lets you specify the file name. The default file name is **Xprofiler.screenDump.ps.0**. If you want to change to a different file name, type it over the one that appears in this field.

  If you specify the output file name with an integer suffix (that is, the file name ends with *xxx.nn*, where *nn* is a non-negative integer), the suffix automatically increases by one every time a new output file is written in the same Xprofiler session.

- **Print Command** (field)

  If you chose to send the captured image directly to a printer, this field lets you specify the print command. The default print command is **qprt -B ga -c -Pps**. If you want to use a different command, type the new command over the one that appears in this field.

4. Click **OK**. The **Screen Dump Options Dialog** window closes.

After you have set your screen dump options, you need to select the window, or portion of a window, you want to capture. From the **Screen Dump** menu, select the **Select Target Window** option. A cursor that looks like a person's hand appears after the number of seconds you specified. To cancel the capture, click the right mouse button. The hand-shaped cursor will revert to normal and the operation will be terminated.

To capture the entire Xprofiler window, place the cursor in the window and then click the left mouse button.

To capture a portion of the Xprofiler window, do the following:

1. Place the cursor in the upper left corner of the area you want to capture.
2. Press and hold the middle mouse button and drag the cursor diagonally downward, until the area you want to capture is within the rubberband box.
3. Release the middle mouse button to set the location of the rubberband box.
4. Press the left mouse button to capture the image.

If you chose to save the image as a file, the file is stored in the directory that you specified. If you chose to print the image, the image is sent to the printer you specified.

Customizing Xprofiler resources

You can customize certain features of an X-Window. For example, you can customize its colors, fonts, and orientation. This section lists each of the resource variables you can set for Xprofiler.

You can customize resources by assigning a value to a resource name in a standard X-Windows format. Several resource files are searched according to the following X-Windows convention:

```
/usr/lib/X11/$LANG/app-defaults/Xprofiler
/usr/lib/X11/app-defaults/Xprofiler
$XAPPLRESDIR/Xprofiler
$HOME/.Xdefaults
```

Options in the **.Xdefaults** file take precedence over entries in the preceding files. This allows you to have certain specifications apply to all users in the **app-defaults** file, as well as user-specific preferences set for each user in their **$HOME/.Xdefaults** file.

You customize a resource by setting a value to a *resource variable* associated with that feature. You store these *resource settings* in a file called **.Xdefaults** in your home directory. You can create this file on a server, and so customize a resource for all users. Individual users may also want to customize resources. The resource settings are essentially your personal preferences for how the X-Windows should look.

For example, consider the following resource variables for a hypothetical X-Windows tool:

```
TOOL*MainWindow.foreground:
TOOL*MainWindow.background:
```

In this example, suppose the resource variable *TOOL*MainWindow.foreground* controls the color of text on the tool's main window. The resource variable *TOOL*MainWindow.background* controls the background color of this same window. If you wanted the tool's main window to have red lettering on a white background, you would insert these lines into the **.Xdefaults** file.

```
TOOL*MainWindow.foreground:    red
TOOL*MainWindow.background:    white
```

Customizable resources and instructions for their use for Xprofiler are defined in **/usr/lib/X11/app-defaults/Xprofiler** file, as well as **/usr/lpp/ppe.xprofiler/defaults/Xprofiler.ad** file. This file contains a set of X-Windows resources for defining graphical user interfaces based on the following criteria:

- Window geometry
- Window title
- Push button and label text
- Color maps
- Text font (in both textual reports and the graphical display)

Xprofiler resource variables

You can use the following resource variables to control the appearance and behavior of Xprofiler. The values listed in this section are the defaults; you can change these values to suit your preferences.

Controlling fonts

To specify the font for the labels that appear with function boxes, call arcs, and cluster boxes:

Use this resource variable:	Specify this default, or a value of your choice:
*narc*font	fixed

To specify the font used in textual reports:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*fontList	rom10

Controlling the appearance of the Xprofiler main window

To specify the size of the main window:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*mainW.height	700
Xprofiler*mainW.width	900

To specify the foreground and background colors of the main window:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*foreground	black
Xprofiler*background	light grey

To specify the number of function boxes that are displayed when you first open the Xprofiler main window:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*InitialDisplayGraph	5000

You can use the **-disp_max** flag to override this value.

To specify the colors of the function boxes and call arcs of the function call tree:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*defaultNodeColor	forest green
Xprofiler*defaultArcColor	royal blue

To specify the color in which a specified function box or call arc is highlighted:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*HighlightNode	red
Xprofiler*HighlightArc	red

To specify the color in which de-emphasized function boxes appear:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*SuppressNode	grey

Function boxes are deemphasized with the **-e**, **-E**, **-f**, and **-F** flags.

Controlling variables related to the File menu

To specify the size of the Load Files Dialog window, use the following:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*loadFile.height	785
Xprofiler*loadFile.width	725

The **Load Files Dialog** window is called by the **Load Files** option of the **File** menu.

To specify whether a confirmation dialog box should appear whenever a file will be overwritten:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*OverwriteOK	False

The value `True` would be equivalent to selecting the **Set Options** option from the File menu, and then selecting the **Forced File Overwriting** option from the Runtime Options Dialog window.

To specify the alternative search paths for locating source or library files:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*fileSearchPath	. (refers to the current working directory)

The value you specify for the search path is equivalent to the search path you would designate from the Alt File Search Path Dialog window. To get to this window, choose the **Set File Search Paths** option from the File menu.

To specify the file search sequence (whether the default or alternative path is searched first):

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*fileSearchDefault	True

The value `True` is equivalent to selecting the **Set File Search Paths** from the File menu, and then the **Check default path(s) first** option from the Alt File Search Path Dialog window.

Controlling variables related to the Screen Dump option: To specify whether a screen dump will be sent to a printer or placed in a file:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*PrintToFile	True

The value `True` is equivalent to selecting the **File** button in the **Output To** field of the Screen Dump Options Dialog window. You access the Screen Dump Options Dialog window by selecting **Screen Dump** and then **Set Option** from the File menu.

To specify whether the PostScript screen dump will created in color or in shades of grey:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*ColorPscript	False

The value `False` is equivalent to selecting the **GreyShades** button in the **PostScript Output** area of the Screen Dump Options Dialog window. You access the Screen Dump Options Dialog window by selecting **Screen Dump** and then **Set Option** from the File menu.

To specify the number of grey shades that the PostScript screen dump will include (if you selected **GreyShades** in the **PostScript Output** area):

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*GreyShades	16

The value `16` is equivalent to selecting the **16** button in the **Number of Grey Shades** field of the Screen Dump Options Dialog window. You access the Screen Dump Options Dialog window by selecting **Screen Dump** and then **Set Option** from the File menu.

To specify the number of seconds that Xprofiler waits before capturing a screen image:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*GrabDelay	1

The value `1` is the default for the **Delay Before Grab** option of the Screen Dump Options Dialog window, but you can specify a longer interval by entering a value here. You access the Screen Dump Options Dialog window by selecting **Screen Dump** and then **Set Option** from the File menu.

To set the maximum number of seconds that can be specified with the slider of the **Delay Before Grab** option:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*grabDelayScale.maximum	30

The value `30` is the maximum for the **Delay Before Grab** option of the Screen Dump Options Dialog window. This means that users cannot set the slider scale to a value greater than 30. You access the Screen Dump Options Dialog window by selecting **Screen Dump** and then **Set Option** from the File menu.

To specify whether the screen dump is created in landscape or portrait format:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*Landscape	False

The value `True` is the default for the **Enable Landscape** option of the Screen Dump Options Dialog window. You access the Screen Dump Options Dialog window by selecting **Screen Dump** and then **Set Option** from the File menu.

To specify whether you would like information about how the image was created to be added to the PostScript screen dump:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*Annotate	False

The value `False` is the default for the **Annotate Output** option of the Screen Dump Options Dialog window. You access the Screen Dump Options Dialog window by selecting **Screen Dump** and then **Set Option** from the File menu.

To specify the directory that will store the screen dump file (if you selected **File** in the **Output To** field):

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*PrintFileName	*/tmp/Xprofiler_screenDump.ps.0*

The value you specify is equivalent to the file name you would designate in the **File Name** field of the Screen Dump Dialog window. You access the Screen Dump Options Dialog window by selecting **Screen Dump** and then **Set Option** from the File menu.

To specify the printer destination of the screen dump (if you selected **Printer** in the **Output To** field):

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*PrintCommand	**qprt -B ga -c -Pps**

The value `qprt -B ga -c -Pps` is the default print command, but you can supply a different one.

Controlling variables related to the View menu

To specify the size of the **Overview** window:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*overviewMain.height	300
Xprofiler*overviewMain.width	300

To specify the color of the highlight area of the **Overview** window:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*overviewGraph*defaultHighlightColor	sky blue

To specify whether the function call tree is updated as the highlight area is moved (immediate) or only when it is stopped and the mouse button released (delayed):

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*TrackImmed	True

The value `True` is equivalent to selecting the **Immediate Update** option from the Utility menu of the Overview window. You access the Overview window by selecting the **Overview** option from the View menu.

To specify whether the function boxes in the function call tree appear in two-dimensional or three-dimensional format:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*Shape2D	True

The value `True` is equivalent to selecting the **2-D Image** option from the View menu.

To specify whether the function call tree appears in top-to-bottom or left-to-right format:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*LayoutTopDown	True

The value `True` is equivalent to selecting the **Layout: Top** and **Bottom** option from the View menu.

Controlling variables related to the Filter menu

To specify whether the function boxes of the function call tree are clustered or unclustered when the Xprofiler main window is first opened:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*ClusterNode	True

The value `True` is equivalent to selecting the **Cluster Functions by Library** option from the Filter menu.

To specify whether the call arcs of the function call tree are collapsed or expanded when the Xprofiler main window is first opened:

Use this resource variable:	Specify this default, or a value of your choice:
Xprofiler*ClusterArc	True

The value True is equivalent to selecting the **Collapse Library Arcs** option from the Filter menu.

Chapter 3. CPU Utilization Reporting Tool (curt)

The CPU Utilization Reporting Tool (**curt**) command converts an AIX trace file into a number of statistics related to CPU utilization and either process or thread activity. These statistics ease the tracking of specific application activity. **curt** works with both uniprocessor and multiprocessor AIX Version 4 and AIX Version 5 traces.

curt Command Syntax

The syntax for the **curt** command is as follows:

curt -i *inputfile* [**-o** *outputfile*] [**-n** *gennamesfile*] [**-m** *trcnmfile*] [**-a** *pidnamefile*] [**-f** *timestamp*] [**-l** *timestamp*] [**-ehpst**]

Flags

-i *inputfile*	Specifies the input AIX trace file to be analyzed.
-o *outputfile*	Specifies an output file (default is **stdout**).
-n *gennamesfile*	Specifies a names file produced by **gennames**.
-m *trcnmfile*	Specifies a names file produced by **trcnm**.
-a *pidnamefile*	Specifies a PID-to-process name mapping file.
-f *timestamp*	Starts processing trace at *timestamp* seconds.
-l *timestamp*	Stops processing trace at *timestamp* seconds.
-e	Outputs elapsed time information for system calls.
-h	Displays usage text (this information).
-p	Shows ticks as trace processing progresses.
-s	Outputs information about errors returned by system calls.
-t	Outputs detailed thread by thread information.

Parameters

gennamesfile	The names file as produced by **gennames**.
inputfile	The AIX trace file to be processed by the **curt** command.
outputfile	The name of the output file created by the **curt** command.
pidnamefile	If the trace process name table is not accurate, or if more descriptive names are desired, use the **-a** flag to specify a PID to process name mapping file. This is a file with lines consisting of a process ID (in decimal) followed by a space, then an ASCII string to use as the name for that process.
timestamp	The time in seconds at which to start and stop the trace file processing.
trcnmfile	The names file as produced by **trcnm**.

Measurement and Sampling

A raw (unformatted) system trace from AIX Version 4 or AIX Version 5 is read by the **curt** command to produce summaries, as well as first and second level interrupt handlers. This summary information is useful for determining which application, system call, or interrupt handler is using most of the CPU time and is a candidate to be optimized to improve system performance.

The following table lists the minimum trace hooks required for the **curt** command. Using only these trace hooks will limit the size of the trace file. However, other events on the system may not be captured in this case. This is significant if you intend to analyze the trace in more detail.

Hook ID	Event Name	Event Explanation
100	HKWD_KERN_FLIH	Occurrence of a first level interrupt, such as an I/O interrupt, a data access page fault, or a timer interrupt (scheduler).
101	HKWD_KERN_SVC	A thread has issued a system call.
102	HKWD_KERN_SLIH	Occurrence of a second level interrupt, that is, first level I/O interrupts are being passed on to the second level interrupt handler which then is working directly with the device driver.
103	HKWD_KERN_SLIHRET	Return from a second level interrupt to the caller (usually a first level interrupt handler).
104	HKWD_KERN_SYSCRET	Return from a system call to the caller (usually a thread).
106	HKWD_KERN_DISPATCH	A thread has been dispatched from the run queue to a CPU.
10C	HKWD_KERN_IDLE	The idle process has been dispatched.
119	HKWD_KERN_PIDSIG	A signal has been sent to a process.
134	HKWD_SYSC_EXECVE	An exec supervisor call (SVC) has been issued by a (forked) process.
135	HKWD_SYSC__EXIT	An exit supervisor call (SVC) has been issued by a process.
139	HKWD_SYSC_FORK	A fork SVC has been issued by a process.
200	HKWD_KERN_RESUME	A dispatched thread is being resumed on the CPU.
210	HKWD_KERN_INITP	A kernel process has been created.
38F	HKWD_DR	A processor has been added/removed.
465	HKWD_SYSC_CRTHREAD	A **thread_create** SVC has been issued by a process.

Trace hooks 119 and 135 are used to report on the time spent in the **exit** system call. Trace hooks 134, 139, 210, and 465 are used to keep track of TIDs, PIDs and process names.

Examples of the curt command

Preparing **curt** output is a four-stage process. The trace file used in the example was generated using the following process:

1. **Build the raw trace.**
   On a 4-way machine, this will create files as listed in the following example output. One raw trace file per CPU is produced. The files are named **trace.raw-0**, **trace.raw-1**, and so forth for each CPU. An additional file named **trace.raw** is also generated. This is a master file that has information that ties together the other CPU-specific traces.
2. **Merge the trace files.**
   To merge the individual CPU raw trace files to form one trace file, run the **trcrpt** command. If you are tracing a uniprocessor machine, this step is not necessary.
3. **Create the supporting files gennamesfile and/or trcnmfile.**
   Neither the **gennamesfile** nor the **trcnmfile** file are necessary for the **curt** command to run. However, if you provide one or both of those files, the **curt** command will output names for system calls and interrupt handlers instead of just addresses. The **gennames** command output includes more information than the **trcnm** command output, and so, while the **trcnmfile** will contain most of the important address to name mapping data, a **gennamesfile** will enable the **curt** command to output more names, especially interrupt handlers. The **gennames** command requires root authority to run. The **trcnm** command can be run by any user.
4. **Generate the curt command output.**

```
# HOOKS="100,101,102,103,104,106,10C,119,134,135,139,200,210,38F,465"
# SIZE="1000000"
# export HOOKS SIZE
# trace -n -C all -d -j $HOOKS -L $SIZE -T $SIZE -afo trace.raw
# trcon ; sleep 5 ; trcstop
# unset HOOKS SIZE
# ls trace.raw*
trace.raw   trace.raw-0  trace.raw-1  trace.raw-2  trace.raw-3
# trcrpt -C all -r trace.raw > trace.r
# rm trace.raw*
# ls trace*
trace.r
# gennames > gennames.out
# trcnm > trace.nm
```

Overview of the Reports Generated by the curt Command

The following is an overview of the reports that can be generated by the **curt** command.

- A report header with the trace file name, the trace size, the date and time the trace was taken. The header also includes the command used when the trace was run.
- For each CPU (and a summary of all the CPUs), processing time expressed in milliseconds and as a percentage (idle and non-idle percentages are included) for various CPU usage categories.
- Average thread affinity across all CPUs and for each individual CPU.
- The total number of process dispatches for each individual CPU.
- Information on the amount of CPU time spent in application and system call (**syscall**) mode expressed in milliseconds and as a percentage by thread, process, and process type. Also included are the number of threads per process and per process type.
- Information on the amount of CPU time spent executing each kernel process, including the idle process, expressed in milliseconds and as a percentage of the total CPU time.
- Information on completed system calls that includes the name and address of the system call, the number of times the system call was executed, and the total CPU time expressed in milliseconds and as a percentage with average, minimum, and maximum time the system call was running.
- Information on pending system calls, that is, system calls for which the system call return has not occurred at the end of the trace. The information includes the name and address of the system call, the thread or process which made the system call, and the accumulated CPU time the system call was running expressed in milliseconds.
- Information on the first level interrupt handlers (FLIHs) that includes the type of interrupt, the number of times the interrupt occurred, and the total CPU time spent handling the interrupt with average, minimum, and maximum time. This information is given for all CPUs and for each individual CPU. If there are any pending FLIHs (FLIHs for which the resume has not occurred at the end of the trace), for each CPU the accumulated time and the pending FLIH type is reported.
- Information on the second level interrupt handlers (SLIHs) which includes the interrupt handler name and address, the number of times the interrupt handler was called and the total CPU time spent handling the interrupt with average, minimum and maximum time. This information is given for all CPUs and for each individual CPU. If there are any pending SLIHs (SLIHs for which the return has not occurred at the end of the trace), for each CPU the accumulated time and the pending SLIH name and address is reported.

To create additional, specialized reports, run the **curt** command using the following flags:

-e Produces a report that includes statistics and additional information on the System Calls Summary Report. The additional information pertains to the total, average, maximum, and minimum elapsed times that a system call was running.

-s Produces a report that includes a list of errors returned by system calls.

-t Produces a report that includes a detailed report on thread status that includes the amount of CPU time the thread was in application and system call mode, what system calls the thread made, processor affinity, the number of times the thread was dispatched, and to which CPU(s) it was dispatched. The report also includes dispatch wait time and details of interrupts.

-p Produces a report that includes a detailed report on process status that includes the amount of CPU time the process was in application and system call mode, which threads were in the process, and what system calls the process made.

Default Report Generated by the curt Command

This section explains the default report created by the **curt** command, as follows:

```
# curt -i trace.r -m trace.nm -n gennames.out -o curt.out
```

The **curt** command output always includes this default report in its output, even if one of the flags described in the previous section is used.

The report is divided into the following sections:

- General Information
- System Summary
- Processor Summary
- Application Summary by TID
- Application Summary by PID
- Application Summary by Process Type
- Kproc Summary
- System Calls Summary
- Pending System Calls Summary
- FLIH Summary
- SLIH Summary

General Information

The first information in the report is the time and date when this particular **curt** command was run, including the syntax of the **curt** command line that produced the report.

The General Information section also contains some information about the AIX **trace** file that was processed by **curt**. This information consists of the **trace** file's name, size, and its creation date. The command used to invoke the AIX trace facility and gather the trace file is displayed at the end of the report.

The following is a sample of this output:

```
Run on Fri May 25 11:08:46 2001
Command line was:
curt -i trace.r -m trace.nm -n gennames.out -o curt.out
----
AIX trace file name = trace.r
AIX trace file size = 1632496
AIX trace file created = Fri May 25 11:04:33 2001


Command used to gather AIX trace was:
trace -n -C all -d -j 100,101,102,103,104,106,10C,134,139,200,465 -L 1000000 -T 1000000 -afo trace.raw
```

System Summary

The next part of the default report is the System Summary produced by the **curt** command. The following is a sample:

```
               System Summary
               --------------
 processing       percent       percent
 total time     total time     busy time
     (msec)   (incl. idle)   (excl. idle)  processing category
===========    ===========    ===========  ==================
   14998.65          73.46          92.98  APPLICATION
     591.59           2.90           3.66  SYSCALL
      48.33           0.24           0.30  KPROC
     486.19           2.38           3.00  FLIH
      49.10           0.24           0.30  SLIH
       8.83           0.04           0.05  DISPATCH (all procs. incl. IDLE)
       1.04           0.01           0.01  IDLE DISPATCH (only IDLE proc.)
-----------    ----------        -------
   16182.69          79.26         100.00  CPU(s) busy time
    4234.76          20.74                 IDLE
-----------    ----------
   20417.45                                TOTAL

Avg. Thread Affinity =        0.99
```

This portion of the report describes the time spent by the system as a whole (all CPUs) in various execution modes.

The System Summary has the following fields:

processing total time Total time in milliseconds for the corresponding processing category.

percent total time Time from the first column as a percentage of the sum of total trace elapsed time for all processors. This includes whatever amount of time each processor spent running the IDLE process.

percent busy time Time from the first column as a percentage of the sum of total trace elapsed time for all processors without including the time each processor spent executing the IDLE process.

Avg. Thread Affinity Probability that a thread was dispatched to the same processor on which it last executed.

The possible execution modes or processing categories are interpreted as follows:

APPLICATION The sum of times spent by all processors in User (that is, non-privileged) mode.

SYSCALL The sum of times spent by all processors doing System Calls. This is the portion of time that a processor spends executing in the kernel code providing services directly requested by a user process.

KPROC The sum of times spent by all processors executing kernel processes other than the IDLE process. This is the portion of time that a processor spends executing specially created dispatchable processes that only execute kernel code.

FLIH The sum of times spent by all processors executing FLIHs (First Level Interrupt Handlers).

SLIH The sum of times spent by all processors executing SLIHs (Second Level Interrupt Handlers).

DISPATCH The sum of times spent by all processors executing the AIX dispatch code. This sum includes the time spent dispatching all threads (that is, it includes dispatches of the IDLE process).

IDLE DISPATCH The sum of times spent by all processors executing the AIX dispatch code where the process being dispatched was the IDLE process. Because the DISPATCH category includes the IDLE DISPATCH category's time, the IDLE DISPATCH category's time is not separately added to calculate either CPU(s) busy time or TOTAL (see below).

CPU(s) busy time The sum of times spent by all processors executing in APPLICATION, SYSCALL, KPROC, FLIH, SLIH, and DISPATCH modes.

IDLE The sum of times spent by all processors executing the IDLE process.

TOTAL The sum of CPU(s) busy time and IDLE.

The System Summary example indicates that the CPU is spending most of its time in application mode. We still have 4234.76 ms of IDLE time so we have enough CPU to run our applications. If there was insufficient CPU power, we would not expect to see any IDLE time. The Avg. Thread Affinity value is 0.99 showing good processor affinity; that is, threads returning to the same processor when they are ready to be rerun.

Processor Summary

This part of the **curt** output follows the System Summary and is essentially the same information but presented on a processor-by-processor basis. The same description that was given for the System Summary applies here, except that this report covers only one processor.

Below is a sample of this output:

```
             Processor Summary  processor number 0
             ---------------------------------------
 processing       percent       percent
 total time    total time     busy time
     (msec)  (incl. idle)  (excl. idle)  processing category
==========   ==========   ==========  ==================
      45.07         0.88         5.16  APPLICATION
     591.39        11.58        67.71  SYSCALL
      47.83         0.94         5.48  KPROC
     173.78         3.40        19.90  FLIH
       9.27         0.18         1.06  SLIH
       6.07         0.12         0.70  DISPATCH (all procs. incl. IDLE)
       1.04         0.02         0.12  IDLE DISPATCH (only IDLE proc.)
----------    ----------       -------
     873.42        17.10       100.00  CPU(s) busy time
    4232.92        82.90               IDLE
----------    ----------
    5106.34                            TOTAL

Avg. Thread Affinity =        0.98

Total number of process dispatches  = 1620
Total number of idle dispatches = 782

               Processor Summary  processor number 1
               ---------------------------------------
 processing       percent       percent
 total time    total time     busy time
     (msec)  (incl. idle)  (excl. idle)  processing category
==========   ==========   ==========  ==================
    4985.81        97.70        97.70  APPLICATION
       0.09         0.00         0.00  SYSCALL
       0.00         0.00         0.00  KPROC
     103.86         2.04         2.04  FLIH
      12.54         0.25         0.25  SLIH
       0.97         0.02         0.02  DISPATCH (all procs. incl. IDLE)
       0.00         0.00         0.00  IDLE DISPATCH (only IDLE proc.)
----------    ----------       -------
    5103.26       100.00       100.00  CPU(s) busy time
       0.00         0.00               IDLE
----------    ----------
    5103.26                            TOTAL

Avg. Thread Affinity =        0.99

Total number of process dispatches  = 516
```

```
Total number of idle dispatches = 0


 ...(lines omitted)...
```

The Total number of process dispatches refers to how many times AIX dispatched any non-IDLE process on this processor. The Total number of idle dispatches gives the count of IDLE process dispatches.

Application Summary by Thread ID (TID)

The Application Summary (by Tid) shows an output of all the threads that were running on the system during the time of trace collection and their CPU consumption. The thread that consumed the most CPU time during the time of the trace collection is at the top of the list.

```
          Application Summary (by Tid)
          ----------------------------
  -- processing total (msec) --   -- percent of total processing time --
 combined  application  syscall  combined  application      syscall  name (Pid  Tid)
 ========  ===========  =======  ========  ===========      =======  ==================
4986.2355    4986.2355   0.0000   24.4214      24.4214       0.0000  cpu(18418  32437)
4985.8051    4985.8051   0.0000   24.4193      24.4193       0.0000  cpu(19128  33557)
4982.0331    4982.0331   0.0000   24.4009      24.4009       0.0000  cpu(18894  28671)
  83.8436       2.5062  81.3374    0.4106       0.0123       0.3984  disp+work(20390  28397)
  72.5809       2.7269  69.8540    0.3555       0.0134       0.3421  disp+work(18584  32777)
  69.8023       2.5351  67.2672    0.3419       0.0124       0.3295  disp+work(19916  33033)
  63.6399       2.5032  61.1368    0.3117       0.0123       0.2994  disp+work(17580  30199)
  63.5906       2.2187  61.3719    0.3115       0.0109       0.3006  disp+work(20154  34321)
  62.1134       3.3125  58.8009    0.3042       0.0162       0.2880  disp+work(21424  31493)
  60.0789       2.0590  58.0199    0.2943       0.0101       0.2842  disp+work(21992  32539)

 ...(lines omitted)...
```

The output is divided into two main sections, of which one shows the total processing time of the thread in milliseconds (processing total (msec)), and the other shows the CPU time the thread has consumed, expressed as a percentage of the total CPU time (percent of total processing time).

The Application Summary (by Tid) has the following fields:

name (Pid Tid) The name of the process associated with the thread, its process id, and its thread id.

processing total (msec)

combined The total amount of CPU time, expressed in milliseconds, that the thread was running in either application mode or system call mode.

application The amount of CPU time, expressed in milliseconds, that the thread spent in application mode.

syscall The amount of CPU time, expressed in milliseconds, that the thread spent in system call mode.

percent of total processing time

combined The amount of CPU time that the thread was running, expressed as percentage of the total processing time.

application The amount of CPU time that the thread the thread spent in application mode, expressed as percentage of the total processing time.

syscall The amount of CPU time that the thread spent in system call mode, expressed as percentage of the total processing time.

In this example, we can investigate why the system is spending so much time in application mode by looking at the Application Summary (by Tid), where we can see the top three processes of the report are

named **cpu**, a test program that uses a great deal of CPU time. The report shows again that the CPU spent most of its time in application mode running the ″cpu″ process. Therefore the ″cpu″ process is a candidate to be optimized to improve system performance.

Application Summary by Process ID (PID)

The Application Summary (by Pid) has the same content as the Application Summary (by Tid), except that the threads that belong to each process are consolidated and the process that consumed the most CPU time during the monitoring period is at the beginning of the list.

The column `name (PID) (Thread Count)` shows the process name, its process ID, and the number of threads that belong to this process and that have been accumulated for this line of data.

```
                        Application Summary (by Pid)
                        ----------------------------
  -- processing total (msec) --   -- percent of total processing time --
 combined  application  syscall  combined  application  syscall  name (Pid)(Thread Count)
 ========  ===========  =======  ========  ===========  =======  ==================
4986.2355    4986.2355   0.0000   24.4214      24.4214   0.0000  cpu(18418)(1)
4985.8051    4985.8051   0.0000   24.4193      24.4193   0.0000  cpu(19128)(1)
4982.0331    4982.0331   0.0000   24.4009      24.4009   0.0000  cpu(18894)(1)
  83.8436       2.5062  81.3374    0.4106       0.0123   0.3984  disp+work(20390)(1)
  72.5809       2.7269  69.8540    0.3555       0.0134   0.3421  disp+work(18584)(1)
  69.8023       2.5351  67.2672    0.3419       0.0124   0.3295  disp+work(19916)(1)
  63.6399       2.5032  61.1368    0.3117       0.0123   0.2994  disp+work(17580)(1)
  63.5906       2.2187  61.3719    0.3115       0.0109   0.3006  disp+work(20154)(1)
  62.1134       3.3125  58.8009    0.3042       0.0162   0.2880  disp+work(21424)(1)
  60.0789       2.0590  58.0199    0.2943       0.0101   0.2842  disp+work(21992)(1)

...(lines omitted)...
```

Application Summary (by process type)

The Application Summary (by process type) consolidates all processes of the same name and sorts them in descending order of combined processing time.

The name (thread count) column shows the name of the process, and the number of threads that belong to this process name (type) and were running on the system during the monitoring period.

```
                  Application Summary (by process type)
                ------------------------------------------------
   -- processing total (msec) --    -- percent of total processing time --
  combined  application   syscall  combined  application  syscall  name (thread count)
  ========  ===========   =======  ========  ===========  =======  ==================
14954.0738   14954.0738    0.0000   73.2416      73.2416   0.0000  cpu(3)
  573.9466      21.2609  552.6857    2.8111       0.1041   2.7069  disp+work(9)
   20.9568       5.5820   15.3748    0.1026       0.0273   0.0753  trcstop(1)
   10.6151       2.4241    8.1909    0.0520       0.0119   0.0401  i4llmd(1)
    8.7146       5.3062    3.4084    0.0427       0.0260   0.0167  dtgreet(1)
    7.6063       1.4893    6.1171    0.0373       0.0073   0.0300  sleep(1)

...(lines omitted)...
```

Kproc Summary by Thread ID (TID)

The Kproc Summary (by Tid) shows an output of all the kernel process threads that were running on the system during the time of trace collection and their CPU consumption. The thread that consumed the most CPU time during the time of the trace collection is at the beginning of the list.

```
                        Kproc Summary  (by Tid)
                        -----------------------
  -- processing total (msec) --   -- percent of total time --
 combined    operation     kernel  combined  operation   kernel  name (Pid  Tid Type)
 ========    =========     ======  ========  =========   ======  ====================
4232.9216       0.0000  4232.9216   20.7319     0.0000  20.7319  wait(516 517 W)
  30.4374       0.0000    30.4374    0.1491     0.0000   0.1491  lrud(1548 1549 -)

...(lines omitted)...
```

```
                              Kproc Types
                              -----------
Type Function                       Operation
==== ============================   =========================
 W   idle thread                    -
```

The Kproc Summary has the following fields:

name (Pid Tid Type) The name of the kernel process associated with the thread, its process ID, its thread ID, and its type. The **kproc** type is defined in the Kproc Types listing following the Kproc Summary.

processing total (msec)

combined The total amount of CPU time, expressed in milliseconds, that the thread was running in either operation or kernel mode.

operation The amount of CPU time, expressed in milliseconds, that the thread spent in operation mode.

kernel The amount of CPU time, expressed in milliseconds, that the thread spent in kernel mode.

percent of total time

combined The amount of CPU time that the thread was running, expressed as percentage of the total processing time.

operation The amount of CPU time that the thread the thread spent in operation mode, expressed as percentage of the total processing time.

kernel The amount of CPU time that the thread spent in kernel mode, expressed as percentage of the total processing time.

Kproc Types

Type A single letter to be used as an index into this listing.

Function A description of the nominal function of this type of kernel process.

Operation A description of the operation mode for this type of kernel process.

System Calls Summary

The System Calls Summary provides a list of all the system calls that have completed execution on the system during the monitoring period. The list is sorted by the total CPU time in milliseconds consumed by each type of system call.

```
                          System Calls Summary
                          --------------------
    Count   Total Time   % sys  Avg Time  Min Time  Max Time  SVC (Address)
                (msec)    time    (msec)    (msec)    (msec)
========  ===========  ======  ========  ========  ========  ================
      605     355.4475   1.74%    0.5875    0.0482    4.5626  kwrite(4259c4)
      733     196.3752   0.96%    0.2679    0.0042    2.9948  kread(4259e8)
        3       9.2217   0.05%    3.0739    2.8888    3.3418  execve(1c95d8)
       38       7.6013   0.04%    0.2000    0.0051    1.6137  __loadx(1c9608)
     1244       4.4574   0.02%    0.0036    0.0010    0.0143  lseek(425a60)
       45       4.3917   0.02%    0.0976    0.0248    0.1810  access(507860)
       63       3.3929   0.02%    0.0539    0.0294    0.0719  _select(4e0ee4)
        2       2.6761   0.01%    1.3380    1.3338    1.3423  kfork(1c95c8)
      207       2.3958   0.01%    0.0116    0.0030    0.1135  _poll(4e0ecc)
```

```
      228       1.1583   0.01%    0.0051    0.0011    0.2436  kioctl(4e07ac)
        9       0.8136   0.00%    0.0904    0.0842    0.0988  .smtcheckinit(1b245a8)
        5       0.5437   0.00%    0.1087    0.0696    0.1777  open(4e08d8)
       15       0.3553   0.00%    0.0237    0.0120    0.0322  .smtcheckinit(1b245cc)
        2       0.2692   0.00%    0.1346    0.1339    0.1353  statx(4e0950)
       33       0.2350   0.00%    0.0071    0.0009    0.0210  _sigaction(1cada4)
        1       0.1999   0.00%    0.1999    0.1999    0.1999  kwaitpid(1cab64)
      102       0.1954   0.00%    0.0019    0.0013    0.0178  klseek(425a48)

 ...(lines omitted)...
```

The System Calls Summary has the following fields:

Count The number of times that a system call of a certain type (see SVC (Address)) has been used (called) during the monitoring period.

Total Time (msec) The total CPU time that the system spent processing these system calls, expressed in milliseconds.

% sys time The total CPU time that the system spent processing these system calls, expressed as a percentage of the total processing time.

Avg Time (msec) The average CPU time that the system spent processing one system call of this type, expressed in milliseconds.

Min Time (msec) The minimum CPU time that the system needed to process one system call of this type, expressed in milliseconds.

Max Time (msec) The maximum CPU time that the system needed to process one system call of this type, expressed in milliseconds.

SVC (Address) The name of the system call and its kernel address.

Pending System Calls Summary

The Pending System Calls Summary provides a list of all the system calls that have been executed on the system during the monitoring period but have not completed. The list is sorted by TID.

```
                    Pending System Calls Summary
                    ----------------------------
Accumulated   SVC (Address)               Procname (Pid  Tid)
Time (msec)
============  ========================  ========================
      0.0656  _select(4e0ee4)           sendmail(7844 5001)
      0.0452  _select(4e0ee4)           syslogd(7514 8591)
      0.0712  _select(4e0ee4)           snmpd(5426 9293)
      0.0156  kioctl(4e07ac)            trcstop(47210 18379)
      0.0274  kwaitpid(1cab64)          ksh(20276 44359)
      0.0567  kread4259e8)              ksh(23342 50873)

 ...(lines omitted)...
```

The Pending System Calls Summary has the following fields:

Accumulated Time(msec) The accumulated CPU time that the system spent processing the pending system call, expressed in milliseconds.

SVC (Address) The name of the system call and its kernel address.

Procname (Pid Tid) The name of the process associated with the thread which made the system call, its process ID, and the thread ID.

FLIH Summary

The FLIH (First Level Interrupt Handler) Summary lists all first level interrupt handlers that were called during the monitoring period.

The Global Flih Summary lists the total of first level interrupts on the system, while the Per CPU Flih Summary lists the first level interrupts per CPU.

```
                          Global Flih Summary
                          -------------------
       Count  Total Time    Avg Time    Min Time    Max Time  Flih Type
                 (msec)      (msec)      (msec)      (msec)
      ====== =========== =========== =========== ===========  =========
        2183    203.5524      0.0932      0.0041      0.4576   31(DECR_INTR)
         946    102.4195      0.1083      0.0063      0.6590    3(DATA_ACC_PG_FLT)
          12      1.6720      0.1393      0.0828      0.3366   32(QUEUED_INTR)
        1058    183.6655      0.1736      0.0039      0.7001    5(IO_INTR)


                          Per CPU Flih Summary
                          --------------------
CPU Number 0:
       Count  Total Time    Avg Time    Min Time    Max Time  Flih Type
                 (msec)      (msec)      (msec)      (msec)
      ====== =========== =========== =========== ===========  =========
          635     39.8413      0.0627      0.0041      0.4576   31(DECR_INTR)
          936    101.4960      0.1084      0.0063      0.6590    3(DATA_ACC_PG_FLT)
            9      1.3946      0.1550      0.0851      0.3366   32(QUEUED_INTR)
          266     33.4247      0.1257      0.0039      0.4319    5(IO_INTR)

CPU Number 1:
       Count  Total Time    Avg Time    Min Time    Max Time  Flih Type
                 (msec)      (msec)      (msec)      (msec)
      ====== =========== =========== =========== ===========  =========
            4      0.2405      0.0601      0.0517      0.0735    3(DATA_ACC_PG_FLT)
          258     49.2098      0.1907      0.0060      0.5076    5(IO_INTR)
          515     55.3714      0.1075      0.0080      0.3696   31(DECR_INTR)

                   Pending Flih Summary
                   --------------------
Accumulated Time (msec)    Flih Type
=======================    ================
                0.0123     5(IO_INTR)

...(lines omitted)...
```

The FLIH Summary report has the following fields:

Count The number of times that a first level interrupt of a certain type (see Flih Type) occurred during the monitoring period.

Total Time (msec) The total CPU time that the system spent processing these first level interrupts, expressed in milliseconds.

Avg Time (msec) The average CPU time that the system spent processing one first level interrupt of this type, expressed in milliseconds.

Min Time (msec) The minimum CPU time the system needed to process one first level interrupt of this type, expressed in milliseconds.

Max Time (msec) The maximum CPU time that the system needed to process one first level interrupt of this type, expressed in milliseconds.

Flih Type The number and name of the first level interrupt.

Accumulated Time (msec) The accumulated CPU time that the system spent processing the pending first level interrupt, expressed in milliseconds.

FLIH types in the example

The following are FLIH types that were depicted in the above example:

DATA_ACC_PG_FLT Data access page fault

QUEUED_INTR	Queued interrupt
DECR_INTR	Decrementer interrupt
IO_INTR	I/O interrupt

SLIH Summary

The Second level interrupt handler (SLIH) Summary lists all second level interrupt handlers that were called during the monitoring period.

The Global Slih Summary lists the total of second level interrupts on the system, while the Per CPU Slih Summary lists the second level interrupts per CPU.

```
                           Global Slih Summary
                           -------------------
       Count  Total Time    Avg Time    Min Time    Max Time Slih Name(Address)
                  (msec)      (msec)      (msec)      (msec)
      ====== =========== =========== =========== =========== =================
          43      7.0434      0.1638      0.0284      0.3763 s_scsiddpin(1a99104)
        1015     42.0601      0.0414      0.0096      0.0913 ssapin(1990490)


                           Per CPU Slih Summary
                           --------------------
CPU Number 0:
       Count  Total Time    Avg Time    Min Time    Max Time Slih Name(Address)
                  (msec)      (msec)      (msec)      (msec)
      ====== =========== =========== =========== =========== =================
           8      1.3500      0.1688      0.0289      0.3087 s_scsiddpin(1a99104)
         258      7.9232      0.0307      0.0096      0.0733 ssapin(1990490)
CPU Number 1:
       Count  Total Time    Avg Time    Min Time    Max Time Slih Name(Address)
                  (msec)      (msec)      (msec)      (msec)
      ====== =========== =========== =========== =========== =================
          10      1.2685      0.1268      0.0579      0.2818 s_scsiddpin(1a99104)
         248     11.2759      0.0455      0.0138      0.0641 ssapin(1990490)

...(lines omitted)...
```

The SLIH Summary report has the following fields:

Count	The number of times that each second level interrupt handler was called during the monitoring period.
Total Time (msec)	The total CPU time that the system spent processing these second level interrupts, expressed in milliseconds.
Avg Time (msec)	The average CPU time that the system spent processing one second level interrupt of this type, expressed in milliseconds.
Min Time (msec)	The minimum CPU time that the system needed to process one second level interrupt of this type, expressed in milliseconds.
Max Time (msec)	The maximum CPU time that the system needed to process one second level interrupt of this type, expressed in milliseconds.
Slih Name (Address)	The module name and kernel address of the second level interrupt.

Reports Generated with the -e Flag

The report generated with the **-e** flag includes the data shown in the default report, and also includes additional information in the System Calls Summary and the Pending System Calls Summary. The additional information in the System Calls Summary includes the total, average, maximum, and minimum elapsed time that a system call was running. The additional information in the Pending System Calls Summary is the accumulated elapsed time for the pending system calls. The following is an example of the additional information reported by using the **-e** flag:

```
# curt -e -i trace.r -m trace.nm -n gennames.out -o curt.out
# cat curt.out

 ...(lines omitted)...

                                  System Calls Summary
                                  --------------------
 Count  Total   % sys    Avg    Min    Max       Tot       Avg       Min      Max       SVC (Address)
        Time    time    Time   Time   Time     ETime     ETime     ETime    ETime
       (msec)          (msec) (msec) (msec)    (msec)    (msec)    (msec)   (msec)
 ===== ======== ===== ====== ====== ====== ========== ========= ========= ========= ======================
   605 355.4475 1.74% 0.5875 0.0482 4.5626 31172.7658   51.5252    0.0482  422.2323 kwrite(4259c4)
   733 196.3752 0.96% 0.2679 0.0042 2.9948 12967.9407   17.6916    0.0042  265.1204 kread(4259e8)
     3   9.2217 0.05% 3.0739 2.8888 3.3418    57.2051   19.0684    4.5475   40.0557 execve(1c95d8)
    38   7.6013 0.04% 0.2000 0.0051 1.6137    12.5002    0.3290    0.0051    3.3120 __loadx(1c9608)
  1244   4.4574 0.02% 0.0036 0.0010 0.0143     4.4574    0.0036    0.0010    0.0143 lseek(425a60)
    45   4.3917 0.02% 0.0976 0.0248 0.1810     4.6636    0.1036    0.0248    0.3037 access(507860)
    63   3.3929 0.02% 0.0539 0.0294 0.0719  5006.0887   79.4617    0.0294  100.4802 _select(4e0ee4)
     2   2.6761 0.01% 1.3380 1.3338 1.3423    45.5026   22.7513    7.5745   37.9281 kfork(1c95c8)
   207   2.3958 0.01% 0.0116 0.0030 0.1135  4494.9249   21.7146    0.0030  499.1363 _poll(4e0ecc)
   228   1.1583 0.01% 0.0051 0.0011 0.2436     1.1583    0.0051    0.0011    0.2436 kioctl(4e07ac)
     9   0.8136 0.00% 0.0904 0.0842 0.0988  4498.7472  499.8608  499.8052  499.8898 .smtcheckinit(1b245a8)
     5   0.5437 0.00% 0.1087 0.0696 0.1777     0.5437    0.1087    0.0696    0.1777 open(4e08d8)
    15   0.3553 0.00% 0.0237 0.0120 0.0322     0.3553    0.0237    0.0120    0.0322 .smtcheckinit(1b245cc)
     2   0.2692 0.00% 0.1346 0.1339 0.1353     0.2692    0.1346    0.1339    0.1353 statx(4e0950)
    33   0.2350 0.00% 0.0071 0.0009 0.0210     0.2350    0.0071    0.0009    0.0210 _sigaction(1cada4)
     1   0.1999 0.00% 0.1999 0.1999 0.1999  5019.0588 5019.0588 5019.0588 5019.0588 kwaitpid(1cab64)
   102   0.1954 0.00% 0.0019 0.0013 0.0178     0.5427    0.0053    0.0013    0.3650 klseek(425a48)

 ...(lines omitted)...

                         Pending System Calls Summary
                         ----------------------------
Accumulated   Accumulated   SVC (Address)              Procname (Pid  Tid)
Time (msec)   ETime (msec)
============  ============  =========================  ========================
      0.0855       93.6498  kread(4259e8)              oracle(143984 48841)

 ...(lines omitted)...
```

The System Calls Summary in the preceding example has the following fields in addition to the default System Calls Summary:

Tot ETime (msec) The total amount of time from when each instance of the system call was started until it completed. This time will include any time spent servicing interrupts, running other processes, and so forth.

Avg ETime (msec) The average amount of time from when the system call was started until it completed. This time will include any time spent servicing interrupts, running other processes, and so forth.

Min ETime (msec) The minimum amount of time from when the system call was started until it completed. This time will include any time spent servicing interrupts, running other processes, and so forth.

Max ETime (msec) The maximum amount of time from when the system call was started until it completed. This time will include any time spent servicing interrupts, running other processes, and so forth.

Accumulated ETime (msec) The total amount of time from when the pending system call was started until the end of the trace. This time will include any time spent servicing interrupts, running other processes, and so forth.

The preceding example report shows that the maximum elapsed time for the **kwrite** system call was 422.2323 msec, but the maximum CPU time was 4.5626 msec. If this amount of overhead time is unusual for the device being written to, further analysis is needed.

Reports Generated with the -s Flag

The report generated with the **-s** flag includes the data shown in the default report, plus it includes data on errors returned by system calls as shown by the following:

```
# curt -s -i trace.r -m trace.nm -n gennames.out -o curt.out
# cat curt.out

...(lines omitted)...

                      Errors Returned by System Calls
                      -------------------------------

Errors (errno : count : description) returned for System Call: kioctl(4e07ac)
   25 :       15 : "Not a typewriter"
Errors (errno : count : description) returned for System Call: execve(1c95d8)
    2 :        2 : "No such file or directory"

...(lines omitted)...
```

If a large number of errors of a specific type or on a specific system call point to a system or application problem, other debug measures can be used to determine and fix the problem.

Reports Generated with the -t Flag

The report generated with the **-t** flag includes the data shown in the default report, and also includes a detailed report on thread status that includes the amount of time the thread was in application and system call mode, what system calls the thread made, processor affinity, the number of times the thread was dispatched, and to which CPU(s) it was dispatched. The report also includes dispatch wait time and details of interrupts:

```
...(lines omitted)...
--------------------------------------------------------------------------------
Report for Thread Id: 48841 (hex bec9) Pid: 143984 (hex 23270)
Process Name: oracle
---------------------
 Total Application Time (ms):  70.324465
 Total System Call Time (ms):  53.014910

                             Thread System Call Data
   Count   Total Time     Avg Time     Min Time     Max Time     SVC (Address)
               (msec)       (msec)       (msec)       (msec)
========  ===========  ===========  ===========  ===========  ================
      69      34.0819       0.4939       0.1666       1.2762  kwrite(169ff8)
      77      12.0026       0.1559       0.0474       0.2889  kread(16a01c)
     510       4.9743       0.0098       0.0029       0.0467  times(f1e14)
      73       1.2045       0.0165       0.0105       0.0306  select(1d1704)
      68       0.6000       0.0088       0.0023       0.0445  lseek(16a094)
      12       0.1516       0.0126       0.0071       0.0241  getrusage(f1be0)

  No Errors Returned by System Calls

                     Pending System Calls Summary
                     ----------------------------
Accumulated   SVC (Address)
Time (msec)
============  =========================
      0.1420  kread(16a01c)

 processor affinity:  0.583333

Dispatch Histogram for thread (CPUid : times_dispatched).
    CPU 0 : 23
    CPU 1 : 23
    CPU 2 : 9
    CPU 3 : 9
    CPU 4 : 8
    CPU 5 : 14
```

```
   CPU 6 : 17
   CPU 7 : 19
   CPU 8 : 1
   CPU 9 : 4
   CPU 10 : 1
   CPU 11 : 4

total number of dispatches:  131
total number of redispatches due to interupts being disabled:  1
avg. dispatch wait time (ms):  8.273515

     Data on Interrupts that Occured while Thread was Running
          Type of Interrupt       Count
  ============================== ============================
   Data Access Page Faults (DSI): 115
  Instr. Fetch Page Faults (ISI): 0
         Align. Error Interrupts: 0
        IO (external) Interrupts: 0
        Program Check Interrupts: 0
       FP Unavailable Interrupts: 0
         FP Imprecise Interrupts: 0
              RunMode Interrupts: 0
          Decrementer Interrupts: 18
  Queued (Soft level) Interrupts: 15

...(lines omitted)...
```

The information in the threads summary includes the following:

Thread ID	The Thread ID of the thread.
Process ID	The Process ID that the thread belongs to.
Process Name	The process name, if known, that the thread belongs to.
Total Application Time (ms)	The amount of time, expressed in milliseconds, that the thread spent in application mode.
Total System Call Time (ms)	The amount of time, expressed in milliseconds, that the thread spent in system call mode.
Thread System Call Data	A system call summary for the thread; this has the same fields as the global System Call Summary. It also includes elapsed time if the **-e** flag is specified and error information if the **-s** flag is specified.
Pending System Calls Summary	If the thread was executing a system call at the end of the trace, a pending system call summary will be printed. This has the Accumulated Time and Supervisor Call (SVC Address) fields. It also includes elapsed time if the **-e** flag is specified.
processor affinity	The process affinity, which is the probability that, for any dispatch of the thread, the thread was dispatched to the same processor on which it last executed.
Dispatch Histogram for thread	Shows the number of times the thread was dispatched to each CPU in the system.
total number of dispatches	The total number of times the thread was dispatched (not including redispatches).
total number of redispatches due to interrupts being disabled	The number of redispatches due to interrupts being disabled, which is when the dispatch code is forced to dispatch the same thread that is currently running on that particular CPU because the thread had disabled some interrupts. This total is only reported if the value is non-zero.
avg. dispatch wait time (ms)	The average dispatch wait time is the average elapsed time for the thread from being undispatched and its next dispatch.
Data on Interrupts that occurred while Thread was Running	Count of how many times each type of FLIH occurred while this thread was executing.

Reports Generated with the -p Flag

When a report is generated using the **-p** flag, it gives detailed information about each process found in the trace. The following example shows the report generated for the router process (PID 129190).

```
...(lines omitted)...

Process Details for Pid: 129190
    Process Name: router
    7 Tids for this Pid: 245889 245631 244599 82843 78701 75347
    28941
Total Application Time (ms):  124.023749
Total System Call Time (ms):  8.948695

                          Process System Call Data
    Count   Total Time   % sys  Avg Time  Min Time  Max Time  SVC (Address)
                (msec)    time    (msec)    (msec)    (msec)
 ========  ===========  ======  ========  ========  ========  ================
       93       3.6829   0.05%    0.0396    0.0060    0.3077  kread(19731c)
       23       2.2395   0.03%    0.0974    0.0090    0.4537  kwrite(1972f8)
       30       0.8885   0.01%    0.0296    0.0073    0.0460  select(208c5c)
        1       0.5933   0.01%    0.5933    0.5933    0.5933  fsync(1972a4)
      106       0.4902   0.01%    0.0046    0.0035    0.0105  klseek(19737c)
       13       0.3285   0.00%    0.0253    0.0130    0.0387  semctl(2089e0)
        6       0.2513   0.00%    0.0419    0.0238    0.0650  semop(2089c8)
        3       0.1223   0.00%    0.0408    0.0127    0.0730  statx(2086d4)
        1       0.0793   0.00%    0.0793    0.0793    0.0793  send(11e1ec)
        9       0.0679   0.00%    0.0075    0.0053    0.0147  fstatx(2086c8)
        4       0.0524   0.00%    0.0131    0.0023    0.0348  kfcntl(22aa14)
        5       0.0448   0.00%    0.0090    0.0086    0.0096  yield(11dbec)
        3       0.0444   0.00%    0.0148    0.0049    0.0219  recv(11e1b0)
        1       0.0355   0.00%    0.0355    0.0355    0.0355  open(208674)
        1       0.0281   0.00%    0.0281    0.0281    0.0281  close(19728c)

                    Pending System Calls Summary
                    ----------------------------
Accumulated    SVC (Address)                  Tid
Time (msec)
============  ========================  ================
      0.0452   select(208c5c)             245889
      0.0425   select(208c5c)             78701
      0.0285   select(208c5c)             82843
      0.0284   select(208c5c)             245631
      0.0274   select(208c5c)             244599
      0.0179   select(208c5c)             75347

 ...(lines omitted)...
```

The **-p** flag process information includes the Process ID and name, and a count and list of the thread IDs belonging to the process. The total application and system call time for all the threads of the process is given. Lastly, it includes summary reports of all the completed and pending system calls for the threads of the process.

Chapter 4. Simple Performance Lock Analysis Tool (splat)

The Simple Performance Lock Analysis Tool (splat) is a software tool that generates reports on the use of synchronization locks. These include the simple and complex locks provided by the AIX kernel as well as user-level mutexes, read and write locks, and condition variables provided by the **PThread** library. The **splat** tool is not currently equipped to analyze the behavior of the Virtual Memory Manager (VMM) and PMAP locks used in the AIX kernel.

Splat Command Syntax

The syntax for the **splat** command is as follows:

splat [**-i** *file*] [**-n** *file*] [**-o** *file*] [**-k** *kexList*] [**-d** [**bfta**]] [**-l** *address*][**-c** *class*] [**-s** [**acelmsS**]] [**-C**#] [**-S**#] [**-t** *start*] [**-T** *stop*]

splat -h [*topic*]

splat -j

Flags

-i *inputfile*	Specifies the AIX trace log file input.
-n *namefile*	Specifies the file containing output of **gennames** command.
-o *outputfile*	Specifies an output file (default is **stdout**).
-k kex[,kex]*	Specifies the kernel extensions for which the lock activities will be reported. This flag is valid only if a name file is provided with **-n**.
-d *detail*	Specifies the level of detail of the report.
-c *class*	Specifies class of locks to be reported.
-l *address*	Specifies the address for which activity on the lock will be reported.
-s *criteria*	Specifies the sort order of the lock, function, and thread.
-C *CPUs*	Specifies the number of processors on the MP system that the trace was drawn from. The default is 1. This value is overridden if more processors are observed to be reported in the trace.
-S *count*	Specifies the number of items to report on for each section. The default is 10. This gives the number of locks to report in the Lock Summary and Lock Detail reports, as well as the number of functions to report in the Function Detail and threads to report in the Thread detail (the **-s** option specifies how the most significant locks, threads, and functions are selected).
-t *starttime*	Overrides the start time from the first event recorded in the trace. This flag forces the analysis to begin an event that occurs *starttime* seconds after the first event in the trace.
-T *stoptime*	Overrides the stop time from the last event recorded in the trace. This flag forces the analysis to end with an event that occurs *stoptime* seconds after the first event in the trace.
-j	Prints the list of IDs of the trace hooks used by the **splat** command.
-h *topic*	Prints a help message on usage or a specific topic.

Parameters

inputfile	The AIX trace log file input. This file can be a merge trace file generated using the **trcrpt -r** command.
namefile	File containing output of the **gennames** command.
outputfile	File to write reports to.

kex[,kex]*
List of kernel extensions.

detail
The detail level of the report, it can be one of the following:

basic Lock summary plus lock detail (the default)

function
Basic plus function detail

thread Basic plus thread detail

all Basic plus function plus thread detail

class
Activity classes, which is a decimal value found in the **/usr/include/sys/lockname.h** file .

address
The address to be reported, given in hexadecimal.

criteria
Order the lock, function, and thread reports by the following criteria:

a Acquisitions

c Percent processor time held

e Percent elapsed time held

l Lock address, function address, or thread ID

m Miss rate

s Spin count

S Percent processor spin hold time (the default)

CPUs
The number of processors on the MP system that the trace was drawn from. The default is 1. This value is overridden if more processors are observed to be reported in the trace.

count
The number of locks to report in the Lock Summary and Lock Detail reports, as well as the number of functions to report in the Function Detail and threads to report in the Thread detail. (The **-s** option specifies how the most significant locks, threads, and functions are selected).

starttime
The number of seconds after the first event recorded in the trace that the reporting starts.

stoptime
The number of seconds after the first event recorded in the trace that the reporting stops.

topic
Help topics, which are:
all
overview
input
names
reports
sorting

Measurement and Sampling

The **splat** tool takes as input an AIX trace log file or (for an SMP trace) a set of log files, and preferably a **names** file produced by the **gennames** command. The procedure for generating these files is shown in the **trace** section. When you run **trace**, you will usually use the flag **-J splat** to capture the events analyzed by **splat** (or without the **-J** flag, to capture all events). The significant trace hooks are shown in the following table:

Hook ID	Event name	Event explanation
106	HKWD_KERN_DISPATCH	The thread is dispatched from the run queue to a processor.
10C	HKWD_KERN_IDLE	The idle process is been dispatched.
10E	HKWD_KERN_RELOCK	One thread is suspended while another is dispatched; the ownership of a **RunQ** lock is transferred from the first to the second.
112	HKWD_KERN_LOCK	The thread attempts to secure a kernel lock; the sub-hook shows what happened.

Hook ID	Event name	Event explanation
113	HKWD_KERN_UNLOCK	A kernel lock is released.
46D	HKWD_KERN_WAITLOCK	The thread is enqueued to wait on a kernel lock.
606	HKWD_PTHREAD_COND	Operations on a Condition Variable.
607	HKWD_PTHREAD_MUTEX	Operations on a Mutex.
608	HKWD_PTHREAD_RWLOCK	Operations on a Read/Write Lock.
609	HKWD_PTHREAD_GENERAL	Operations on a **PThread**.

Execution, Trace, and Analysis Intervals

In some cases, you can use the **trace** tool to capture the entire execution of a workload, while in other cases you will capture only an interval of the execution. The *execution interval* is the entire time that a workload runs. This interval is arbitrarily long for server workloads that run continuously. The *trace interval* is the time actually captured in the trace log file by **trace**. The length of this trace interval is limited by how large a trace log file will fit on the file system.

In contrast, the analysis interval is the portion of the trace interval that is analyzed by the **splat** command. The **-t** and **-T** options indicate to the **splat** command to start and finish analysis some number of seconds after the first event in the trace. By default, the **splat** command analyzes the entire trace, so this analysis interval is the same as the trace interval.

Note: As an optimization, the **splat** command stops reading the trace when it finishes its analysis, so it indicates that the trace and analysis intervals end at the same time even if they do not.

To most accurately estimate the effect of lock activity on the computation, you will usually want to capture the longest trace interval that you can, and analyze that entire interval with the **splat** command. The **-t** and **-T** flags are usually used for debugging purposes to study the behavior of the **splat** command across a few events in the trace.

As a rule, either use large buffers when collecting a trace, or limit the captured events to the ones you need to run the **splat** command.

Trace Discontinuities

The **splat** command uses the events in the trace to reconstruct the activities of threads and locks in the original system. If part of the trace is missing, it is because one of the following situations exists:

- Tracing was stopped at one point and restarted at a later point.
- One processor fills its trace buffer and stops tracing, while other processors continue tracing.
- Event records in the trace buffer were overwritten before they could be copied into the trace log file.

In any of the above cases, the **splat** command will not be able to correctly analyze all the events across the trace interval. The policy of **splat** is to finish its analysis at the first point of discontinuity in the trace, issue a warning message, and generate its report. In the first two cases, the message is as follows:

```
TRACE OFF record read at 0.567201 seconds. One or more of the CPUs has
stopped tracing. You may want to generate a longer trace using larger
buffers and re-run splat.
```

In the third case, the message is as follows:

```
TRACEBUFFER WRAPAROUND record read at 0.567201 seconds. The input trace
has some records missing; splat finishes analyzing at this point. You
may want to re-generate the trace using larger buffers and re-run splat.
```

Some versions of the AIX kernel or **PThread** library may be incompletely instrumented, so the traces will be missing events. The **splat** command may not provide correct results in this case.

Address-to-Name Resolution in the splat Command

The lock instrumentation in the kernel and **PThread** library is what captures the information for each lock event. Data addresses are used to identify locks; instruction addresses are used to identify the point of execution. These addresses are captured in the event records in the trace, and used by the **splat**command to identify the locks and the functions that operate on them.

However, these addresses are not of much use to the programmer, who would rather know the names of the lock and function declarations so that they can be located in the program source files. The conversion of names to addresses is determined by the compiler and loader, and can be captured in a file using the **gennames** utility. The **gennames** utility also captures the contents of the **/usr/include/sys/lockname.h** file, which declares classes of kernel locks.

This **gennames** output file is passed to the **splat** command with the **-n** flag. When **splat** reports on a kernel lock, it provides the best identification that it can.

Kernel locks that are declared are resolved by name. Locks that are created dynamically are identified by class if their class name is given when they are created. The **libpthreads.a** instrumentation is not equipped to capture names or classes of **PThread** synchronizers, so they are always identified by address only.

Examples of Generated Reports

The report generated by the **splat** command consists of an execution summary, a gross lock summary, and a per-lock summary, followed by a list of lock detail reports that optionally includes a function detail or a thread detail report.

Execution Summary

The following example shows a sample of the Execution summary. This report is generated by default when using the **splat** command.

```
********************************************************************************
splat Cmd: splat -sa -da -S100 -i trace.cooked -n gennames -o splat.out

Trace Cmd:   trace -C all -aj 600,603,605,606,607,608,609 -T 20000000 -L 200000000 -o CONDVAR.raw
Trace Host:  darkwing (0054451E4C00) AIX 5.2
Trace Date:  Thu Sep 27 11:26:16 2002


Elapsed Real Time:        0.098167
Number of CPUs Traced:    1           (Observed):0
Cumulative CPU Time:      0.098167


                                              start                  stop
                                    --------------------   --------------------
trace interval     (absolute tics)             967436752              969072535
                   (relative tics)                     0                1635783
                   (absolute secs)              58.057947              58.156114
                   (relative secs)               0.000000               0.098167
analysis interval  (absolute tics)             967436752              969072535
                   (trace-relative tics)               0                1635783
                   (self-relative tics)                0                1635783
                   (absolute secs)              58.057947              58.156114
                   (trace-relative secs)         0.000000               0.098167
                   (self-relative secs)          0.000000               0.098167
********************************************************************************
```

The execution summary consists of the following elements:

- The **splat** version and build information, disclaimer, and copyright notice.

- The command used to run **splat**.
- The **trace** command used to collect the trace.
- The host on which the trace was taken.
- The date that the trace was taken.
- The real-time duration of the trace expressed in seconds.
- The maximum number of processors that were observed in the trace (the number specified in the trace conditions information, and the number specified on the **splat** command line).
- The cumulative processor time, equal to the duration of the trace in seconds times the number of processors that represents the total number of seconds of processor time consumed.
- A table containing the start and stop times of the trace interval, measured in tics and seconds, as absolute timestamps, from the trace records, as well as relative to the first event in the trace
- The start and stop times of the analysis interval, measured in tics and seconds, as absolute timestamps as well as relative to the beginning of the trace interval and the beginning of the analysis interval.

Gross Lock Summary

The following example shows a sample of the gross lock summary report. This report is generated by default when using the **splat** command.

```
*******************************************************************************
                        Unique       Acquisitions     Acq. or Passes   Total System
                 Total  Addresses     (or Passes)         per Second   Time
               --------- ---------    ------------     -------------   ------------
AIX (all) Locks:     523       523         1323045         72175.7768       0.003986
          RunQ:        2         2          487178         26576.9121       0.000000
       Simple:      480       480          824898         45000.4754       0.003986
      Complex:       41        41           10969           598.3894       0.000000
 PThread CondVar:     7         6          160623          8762.4305       0.000000
        Mutex:      128       116         1927771        105165.2585      10.280745 *
       RWLock:        0         0               0             0.0000       0.000000

 ( spin time goal )
*******************************************************************************
```

The gross lock summary report table consists of the following columns:

Total — The number of AIX Kernel locks, followed by the number of each type of AIX Kernel lock; RunQ, Simple, and Complex. Under some conditions, this will be larger than the sum of the numbers of RunQ, Simple, and Complex locks because we may not observe enough activity on a lock to differentiate its type. This is followed by the number of PThread condition-variables, the number of PThread Mutexes, and the number of PThread Read/Write.

Unique Addresses — The number of unique addresses observed for each synchronizer type. Under some conditions, a lock will be destroyed and re-created at the same address; **splat** produces a separate lock detail report for each instance because the usage may be different.

Acquisitions (or Passes) — For locks, the total number of times acquired during the analysis interval; for PThread condition-variables, the total number of times the condition passed during the analysis interval.

Acq. or Passes (per Second) — Acquisitions or passes per second, which is the total number of acquisitions or passes divided by the elapsed real time of the trace.

% Total System spin Time — The cumulative time spent spinning on each synchronizer type, divided by the cumulative processor time, times 100 percent. The general goal is to spin for less than 10 percent of the processor time; a message to this effect is printed at the bottom of the table. If any of the entries in this column exceed 10 percent, they are marked with an astrisk (*).

Per-lock Summary

The following example shows a sample of the per-lock summary report. This report is generated by default when using the **splat** command.

```
*********************************************************************************************************
100 max entries, Summary sorted by Acquisitions:

                      T Acqui-
                      y sitions                                  Locks or      Percent Holdtime
Lock Names,           p or                                        Passes    Real      Real      Comb      Kernel
Class, or Address     e Passes Spins Wait %Miss   %Total   / CSec    CPU      Elapse    Spin      Symbol
********************* * ****** ***** **** *****   ******  *********  *******  ******   *******   *******
PROC_INT_CLASS.0003   Q 486490 0     0    0.0000  36.7705 26539.380  5.3532  100.000   0.0000   unix
THREAD_LOCK_CLASS.0012 S 323277 0     0    0.0000  24.4343 17635.658  6.8216    6.8216  0.0000   libc
THREAD_LOCK_CLASS.0118 S 323094 0     0    0.0000  24.4205 17625.674  6.7887    6.7887  0.0000   libc
ELIST_CLASS.003C      S 80453  0     0    0.0000  6.0809   4388.934  1.0564    1.0564  0.0000   unix
ELIST_CLASS.0044      S 80419  0     0    0.0000  6.0783   4387.080  1.1299    1.1299  0.0000   unix
tod_lock              C 10229  0     0    0.0000  0.7731    558.020  0.2212    0.2212  0.0000   unix
LDATA_CONTROL_LOCK.0000 S 1833   0     0    0.0000  0.1385     99.995  0.0204    0.0204  0.0000   unix
U_TIMER_CLASS.0014    S 1514   0     0    0.0000  0.1144     82.593  0.0536    0.0536  0.0000   netinet

( ... lines omitted ... )

000000002FF22B70      L 368838 0    N/A   0.0000  100.000    9622.964 99.9865  99.9865  0.0000
00000000F00C3D74      M 160625 0     0    0.0000   14.2831   8762.540 99.7702  99.7702  0.0000
00000000200017E8      M 160625 175   0    0.1088   14.2831   8762.540 42.9371  42.9371  0.1487
0000000020001820      V 160623 0    624   0.0000  100.000    1271.728  N/A       N/A     N/A
00000000F00C3750      M     37 0     0    0.0000    0.0033      2.018  0.0037   0.0037  0.0000
00000000F00C3800      M     30 0     0    0.0000    0.0027      1.637  0.0698   0.0698  0.0000

 ( ... lines omitted ... )
*********************************************************************************************************
```

The first line indicates the maximum number of locks to report (100 in this case, but we show only 14 of the entries here) as specified by the **-S 100** flag. The report also indicates that the entries are sorted by the total number of acquisitions or passes, as specified by the **-sa** flag. The various Kernel locks and **PThread** synchronizers are treated as two separate lists in this report, so the report would produce the top 100 Kernel locks sorted by acquisitions, followed by the top 100 **PThread** synchronizers sorted by acquisitions or passes.

The per-lock summary table consists of the following columns:

Lock Names, Class, or Address
The name, class, or address of the lock, depending on whether the **splat** command could map the address from a name file.

Type
The type of the lock, identified by one of the following letters:

- **Q** A RunQ lock
- **S** A simple kernel lock
- **C** A complex kernel lock
- **M** A **PThread** mutex
- **V** A **PThread** condition-variable
- **L** A **PThread** read/write lock

Acquisitions or Passes
The number of times that the lock was acquired or the condition passed, during the analysis interval.

Spins
The number of times the lock (or condition-variable) was spun on during the analysis interval.

Wait
The number of times that a thread was driven into a wait state for that lock or condition-variable during the analysis interval.

%Miss The percentage of access attempts that resulted in a spin as opposed to a successful acquisition or pass.

%Total The percentage of all acquisitions that were made to this lock, out of all acquisitions to all locks of this type. All AIX locks (RunQ, simple, and complex) are treated as being the same type for this calculation. The **PThread** synchronizers mutex, condition-variable, and read/write lock are all distinct types.

Locks or Passes / CSec The number of times that the lock (or condition-variable) was acquired (or passed) divided by the cumulative processor time. This is a measure of the acquisition frequency of the lock.

Real Elapse The percentage of the elapsed real time that the lock was held by any thread at all, whether running or suspended. Note that this definition is not applicable to condition-variables because they are not held.

Comb Spin The percentage of the cumulative processor time that executing threads spent spinning on the lock. The **PThreads** library uses waiting for condition-variables, so there is no time actually spent spinning.

Kernel Symbol The name of the kernel-extension or library (or /unix for the kernel) that the lock was defined in. This information is not recoverable for **PThreads**.

AIX Kernel Lock Details

By default, the **splat** command prints a lock detail report for each entry in the summary report. The AIX Kernel locks can be either simple or complex.

The RunQ lock is a special case of the simple lock, although its pattern of usage will differ markedly from other lock types. The **splat** command distinguishes it from the other simple locks to ease its analysis.

Simple and RunQ Lock Details

In an AIX SIMPLE lock report, the first line starts with either [AIX SIMPLE Lock] or [AIX RunQ lock]. If the **gennames** output file allows, the ADDRESS is also converted into a lock NAME and CLASS, and the containing kernel extension (KEX) is identified as well. The CLASS is printed with an eight hex-digit extension indicating how many locks of this class were allocated prior to it.

The SIMPLE lock report fields are as follows:

Acquisitions The number of times that the lock was acquired in the analysis interval (this includes successful **simple_lock_try** calls).

Miss Rate The percentage of attempts that failed to acquire the lock.

Spin Count The number of unsuccessful attempts to acquire the lock.

Wait Count The number of times that a thread was forced into a suspended wait state, waiting for the lock to come available.

Busy Count The number of **simple_lock_try** calls that returned busy.

Seconds Held This field contains the following sub-fields:

CPU The total number of processor seconds that the lock was held by an executing thread.

Elapsed
The total number of elapsed seconds that the lock was held by any thread, whether running or suspended.

Percent Held This field contains the following sub-fields:

Real CPU
The percentage of the cumulative processor time that the lock was held by an executing thread.

Real Elapsed
The percentage of the elapsed real time that the lock was held by any thread at all, either running or suspended.

Comb(ined) Spin
The percentage of the cumulative processor time that running threads spent spinning while trying to acquire this lock.

Real Wait
The percentage of elapsed real time that any thread was waiting to acquire this lock. If two or more threads are waiting simultaneously, this wait time will only be charged once. To determine how many threads were waiting simultaneously, look at the WaitQ Depth statistics.

%Enabled The percentage of acquisitions of this lock that occurred while interrupts were enabled. In parentheses is the total number of acquisitions made while interrupts were enabled.

%Disabled The percentage of acquisitions of this lock that occurred while interrupts were disabled. In parentheses is the total number of acquisitions made while interrupts were disabled.

SpinQ The minimum, maximum, and average number of threads spinning on the lock, whether executing or suspended, across the analysis interval.

WaitQ The minimum, maximum, and average number of threads waiting on the lock, across the analysis interval.

The Lock Activity with Interrupts Enabled (milliseconds) and Lock Activity with Interrupts Disabled (milliseconds) sections contain information on the time that each lock state is used by the locks.

The states that a thread can be in (with respect to a given simple or complex lock) are as follows:

(no lock reference) The thread is running, does not hold this lock, and is not attempting to acquire this lock.

LOCK The thread has successfully acquired the lock and is currently executing.

SPIN The thread is executing and unsuccessfully attempting to acquire the lock.

UNDISP The thread has become undispatched while unsuccessfully attempting to acquire the lock.

WAIT The thread has been suspended until the lock comes available. It does not necessarily acquire the lock at that time, but instead goes back to a SPIN state.

PREEMPT The thread is holding this lock and has become undispatched.

The Lock Activity sections of the report measure the intervals of time (in milliseconds) that each thread spends in each of the states for this lock. The columns report the number of times that a thread entered the given state, followed by the maximum, minimum, and average time that a thread spent in the state once entered, followed by the total time that all threads spent in that state. These sections distinguish whether interrupts were enabled or disabled at the time that the thread was in the given state.

A thread can acquire a lock prior to the beginning of the analysis interval and release the lock during the analysis interval. When the **splat** command observes the lock being released, it recognizes that the lock had been held during the analysis interval up to that point and counts the time as part of the state-machine statistics. For this reason, the state-machine statistics can report that the number of times that the lock state was entered may actually be larger than the number of acquisitions of the lock that were observed in the analysis interval.

RunQ locks are used to protect resources in the thread management logic. These locks are acquired a large number of times and are only held briefly each time. A thread need not be executing to acquire or release a RunQ lock. Further, a thread may spin on a RunQ lock, but it will not go into an UNDISP or WAIT state on the lock. You will see a dramatic difference between the statistics for RunQ versus other simple locks.

Functional Detail

The functional detail report is obtained by using the **-df** or **-da** options of **splat**.

The columns are defined as follows:

Function Name The name of the function that acquired or attempted to acquire this lock (with a call to one of the functions **simple_lock**, **simple_lock_try**, **simple_unlock**, **disable_lock**, or **unlock_enable**), if it could be resolved.

Acquisitions The number of times that the function was able to acquire this lock.

Miss Rate The percentage of acquisition attempts that failed.

Spin Count The number of unsuccessful attempts by the function to acquire this lock.

Wait Count The number of times that any thread was forced to wait on the lock, using a call to this function to acquire the lock.

Busy Count The number of times the function tried to acquire the lock without success (that is, calls to **simple_lock_try** that were returned busy).

Percent Held of Total Time Contains the following sub-fields:

- **CPU** Percentage of the cumulative processor time that the lock was held by an executing thread that had acquired the lock through a call to this function.
- **Elapse(d)** The percentage of the elapsed real time that the lock was held by any thread at all, whether running or suspended, that had acquired the lock through a call to this function.
- **Spin** The percentage of cumulative processor time that executing threads spent spinning on the lock while trying to acquire the lock through a call to this function.
- **Wait** The percentage of elapsed real time that executing threads spent waiting for the lock while trying to acquire the lock through a call to this function.

Return Address The return address to this calling function, in hexadecimal.

Start Address The start address to this calling function, in hexadecimal.

Offset The offset from the function start address to the return address, in hexadecimal.

The functions are ordered by the same sorting criterion as the locks, controlled by the **-s** option of **splat**. Further, the number of functions listed is controlled by the **-S** parameter. The default is the top ten functions.

Thread Detail

The Thread Detail report is obtained by using the **-dt** or **-da** options of **splat**.

At any point in time, a single thread is either running or it is not. When a single thread runs, it only runs on one processor. Some of the composite statistics are measured relative to the cumulative processor time when they measure activities that can happen simultaneously on more than one processor, and the magnitude of the measurements can be proportional to the number of processors in the system. In contrast, the thread statistics are generally measured relative to the elapsed real time, which is the amount of time that a single processor spends processing and the amount of time that a single thread spends in an executing or suspended state.

The Thread Detail report columns are defined as follows:

ThreadID The thread identifier.

Acquisitions The number of times that this thread acquired the lock.

Miss Rate The percentage of acquisition attempts by the thread that failed to secure the lock.

Spin Count The number of unsuccessful attempts by this thread to secure the lock.

Wait Count The number of times that this thread was forced to wait until the lock came available.

Busy Count The number of **simple_lock_try()** calls that returned busy.

Percent Held of Total Time Consists of the following sub-fields:

- **CPU** The percentage of the elapsed real time that this thread executed while holding the lock.
- **Elapse(d)** The percentage of the elapsed real time that this thread held the lock while running or suspended.
- **Spin** The percentage of elapsed real time that this thread executed while spinning on the lock.
- **Wait** The percentage of elapsed real time that this thread spent waiting on the lock.

Complex-Lock Report

AIX Complex lock supports recursive locking, where a thread can acquire the lock more than once before releasing it, as well as differentiating between write-locking, which is exclusive, from read-locking, which is not exclusive.

This report begins with [AIX COMPLEX Lock]. Most of the entries are identical to the simple lock report, while some of them are differentiated by read/write/upgrade. For example, the SpinQ and WaitQ statistics include the minimum, maximum, and average number of threads spinning or waiting on the lock. They also include the minimum, maximum, and average number of threads attempting to acquire the lock for reading versus writing. Because an arbitrary number of threads can hold the lock for reading, the report includes the minimum, maximum, and average number of readers in the LockQ that holds the lock.

A thread may hold a lock for writing; this is exclusive and prevents any other thread from securing the lock for reading or for writing. The thread downgrades the lock by simultaneously releasing it for writing and acquiring it for reading; this allows other threads to also acquire the lock for reading. The reverse of this operation is an upgrade; if the thread holds the lock for reading and no other thread holds it as well, the thread simultaneously releases the lock for reading and acquires it for writing. The upgrade operation may require that the thread wait until other threads release their read-locks. The downgrade operation does not.

A thread may acquire the lock to some recursive depth; it must release the lock the same number of times to free it. This is useful in library code where a lock must be secured at each entry-point to the library; a thread will secure the lock once as it enters the library, and internal calls to the library entry-points simply re-secure the lock, and release it when returning from the call. The minimum, maximum, and average recursion depths of any thread holding this lock are reported in the table.

A thread holding a recursive write-lock is not allowed to downgrade it because the downgrade is intended to apply to only the last write-acquisition of the lock, and the prior acquisitions had a real reason to keep the acquisition exclusive. Instead, the lock is marked as being in the downgraded state, which is erased when the this latest acquisition is released or upgraded. A thread holding a recursive read-lock can only upgrade the latest acquisition of the lock, in which case the lock is marked as being upgraded. The thread will have to wait until the lock is released by any other threads holding it for reading. The minimum, maximum, and average recursion-depths of any thread holding this lock in an upgraded or downgraded state are reported in the table.

The Lock Activity report also breaks the time down based on what task the lock is being secured for (reading, writing, or upgrading).

No time is reported to perform a downgrade because this is performed without any contention. The upgrade state is only reported for the case where a recursive read-lock is upgraded. Otherwise, the thread activity is measured as releasing a read-lock and acquiring a write-lock.

The function and thread details also break down the acquisition, spin, and wait counts by whether the lock is to be acquired for reading or writing.

PThread Synchronizer Reports

By default, **splat** prints a detailed report for each **PThread** entry in the summary report. The **PThread** synchronizers are of the following types; mutex, read/write lock, and condition-variable. The mutex and read/write lock are related to the AIX complex lock. You can view the similarities in the lock detail reports. The condition-variable differs significantly from a lock, and this is reflected in the report details.

The **PThread** library instrumentation does not provide names or classes of synchronizers, so the addresses are the only way we have to identify them. Under certain conditions, the instrumentation can capture the return addresses of the function call stack, and these addresses are used with the **gennames** output to identify the call chains when these synchronizers are created. The creation and deletion times of the synchronizer can sometimes be determined as well, along with the ID of the **PThread** that created them.

Mutex Reports

The **PThread** mutex is similar to an AIX simple lock in that only one thread can acquire the lock, and is like an AIX complex lock in that it can be held recursively.

In addition to the common header information and the [**PThread** MUTEX] identifier, this report lists the following lock details:

Acquisitions The number of times that the lock was acquired in the analysis interval.

Miss Rate The percentage of attempts that failed to acquire the lock.

Spin Count The number of unsuccessful attempts to acquire the lock.

Wait Count The number of times that a thread was forced into a suspended wait state waiting for the lock to come available.

Busy Count The number of **trylock** calls that returned busy.

Seconds Held This field contains the following sub-fields:

CPU The total number of processor seconds that the lock was held by an executing thread.

Elapse(d)
The total number of elapsed seconds that the lock was held, whether the thread was running or suspended.

Percent Held This field contains the following sub-fields:

Real CPU
The percentage of the cumulative processor time that the lock was held by an executing thread.

Real Elapsed
The percentage of the elapsed real time that the lock was held by any thread, either running or suspended.

Comb(ined) Spin
The percentage of the cumulative processor time that running threads spent spinning while trying to acquire this lock.

Real Wait
The percentage of elapsed real time that any thread was waiting to acquire this lock. If two or more threads are waiting simultaneously, this wait time will only be charged once. To how many threads were waiting simultaneously, look at the WaitQ Depth statistics.

Depth This field contains the following sub-fields:

SpinQ The minimum, maximum, and average number of threads spinning on the lock, whether executing or suspended, across the analysis interval.

WaitQ The minimum, maximum, and average number of threads waiting on the lock, across the analysis interval.

Recursion
The minimum, maximum, and average recursion depth to which each thread held the lock.

Note: If the **-dt** or **-da** options are used, **splat** reports the thread detail.

The columns are defined as follows:

PThreadID The **PThread** identifier.

Acquisitions The number of times that this thread acquired the lock.

Miss Rate The percentage of acquisition attempts by the thread that failed to secure the lock.

Spin Count The number of unsuccessful attempts by this thread to secure the lock.

Wait Count The number of times that this thread was forced to wait until the lock came available.

Busy Count The number of times this thread used try to acquire the lock without success (calls to **simple_lock_try** that returned busy).

Percent Held of Total Time This field contains the following sub-fields:

CPU The percentage of the elapsed real time that this thread executed while holding the lock.

Elapse(d)
The percentage of the elapsed real time that this thread held the lock while running or suspended.

Spin The percentage of elapsed real time that this thread executed while spinning on the lock.

Wait The percentage of elapsed real time that this thread spent waiting on the lock.

Read/Write Lock Reports

The **PThread** read/write lock is similar to an AIX complex lock in that it can be acquired for reading or writing; writing is exclusive in that a single thread can only acquire the lock for writing, and no other thread

can hold the lock for reading or writing at that point. Reading is not exclusive, so more than one thread can hold the lock for reading. Reading is recursive in that a single thread can hold multiple read-acquisitions on the lock. Writing is not recursive.

In addition to the common header information and the [**PThread** RWLock] identifier, this report lists the following lock details:

Acquisitions The number of times that the lock was acquired in the analysis interval.

Miss Rate The percentage of attempts that failed to acquire the lock.

Spin Count The number of unsuccessful attempts to acquire the lock.

Wait Count The current **PThread** implementation does not force threads to wait for read/write locks. This reports the number of times a thread, spinning on this lock, is undispatched.

Seconds Held This field contains the following sub-fields:

CPU The total number of processor seconds that the lock was held by an executing thread. If the lock is held multiple times by the same thread, only one hold interval is counted.

Elapse(d)
The total number of elapsed seconds that the lock was held by any thread, whether the thread was running or suspended.

Percent Held This field contains the following sub-fields:

Real CPU
The percentage of the cumulative processor time that the lock was held by any executing thread.

Real Elapsed
The percentage of the elapsed real time that the lock was held by any thread, either running or suspended.

Comb(ined) Spin
The percentage of the cumulative processor time that running threads spent spinning while trying to acquire this lock.

Real Wait
The percentage of elapsed real time that any thread was waiting to acquire this lock. If two or more threads are waiting simultaneously, this wait time will only be charged once. To learn how many threads were waiting simultaneously, look at the WaitQ Depth statistics.

Depth This field contains the following sub-fields:

LockQ The minimum, maximum, and average number of threads holding the lock, whether executing or suspended, across the analysis interval. This is broken down by read-acquisitions, write-acquisitions, and total acquisitions.

SpinQ The minimum, maximum, and average number of threads spinning on the lock, whether executing or suspended, across the analysis interval. This is broken down by read-acquisitions, write-acquisitions, and total acquisitions.

WaitQ The minimum, maximum, and average number of threads in a timed-wait state for the lock, across the analysis interval. This is broken down by read-acquisitions, write-acquisitions, and and total acquisitions.

Note: If the **-dt** or **-da** options are used, **splat** reports the thread as detailed below:

PThreadID The **PThread** identifier.

Acquisitions The number of times that this thread acquired the lock.

Miss Rate The percentage of acquisition attempts by the thread that failed to secure the lock.

Spin Count The number of unsuccessful attempts by this thread to secure the lock.

Wait Count The number of times this thread was forced to wait until the lock came available.

Busy Count The number of times this thread used try to acquire the lock without success (calls to **simple_lock_try** that returned busy).

Percent Held of Total Time This field contains the following sub-fields:

- **CPU** The percentage of the elapsed real time that this thread executed while holding the lock.
- **Elapse(d)** The percentage of the elapsed real time that this thread held the lock while running or suspended.
- **Spin** The percentage of elapsed real time that this thread executed while spinning on the lock.
- **Wait** The percentage of elapsed real time that this thread spent waiting for the lock.

Condition-Variable Report

The **PThread** condition-variable is a synchronizer, but not a lock. A **PThread** is suspended until a signal indicates that the condition now holds.

In addition to the common header information and the [**PThread** CondVar] identifier, this report lists the following details:

Passes The number of times that the condition was signaled to hold during the analysis interval.

Fail Rate The percentage of times that the condition was tested and was not found to be true.

Spin Count The number of times that the condition was tested and was not found to be true.

Wait Count The number of times that a thread was forced into a suspended wait state waiting for the condition to be signaled.

Spin / Wait Time This field contains the following sub-fields:

- **Comb Spin** The total number of processor seconds that threads spun while waiting for the condition.
- **Comb Wait** The total number of elapsed seconds that threads spent in a wait state for the condition.

Depth This field contains the following sub-fields:

- **SpinQ** The minimum, maximum, and average number of threads spinning while waiting for the condition, across the analysis interval.
- **WaitQ** The minimum, maximum, and average number of threads waiting for the condition, across the analysis interval.

If the **-dt** or **-da** options are used, **splat** reports the thread detail as described below:

PThreadID The **PThread** identifier.

Passes The number of times that this thread was notified that the condition passed.

Fail Rate The percentage of times the thread checked the condition and did not find it to be true.

Spin Count The number of times the thread checked the condition and did not find it to be true.

Wait Count The number of times this thread was forced to wait until the condition became true.

Percent Total Time This field contains the following sub-fields:

Spin The percentage of elapsed real time that this thread spun while testing the condition.

Wait The percentage of elapsed real time that this thread spent waiting for the condition to hold.

Chapter 5. Performance Monitor API Programming

The **libpmapi** library contains a set of application programming interfaces (APIs) that are designed to provide access to some of the counting facilities of the Performance Monitor feature included in selected IBM microprocessors. Those APIs include the following:

- A set of system-level APIs to allow counting of the activity of a whole machine or of a set of processes with a common ancestor.
- A set of first party kernel-thread-level APIs to allow threads running in 1:1 mode to count their own activity.
- A set of third party kernel-thread-level APIs to allow a debug program to count the activity of target threads running in 1:1 mode.

Note: The APIs and the events available on each of the supported processors have been completely separated by design. The events available, their descriptions, and their current testing status (which are different on each processor) are in separately installable tables, and are not described here because none of the API calls depend on the availability or status of any of the events.

The status of an event, as returned by the **pm_init** API initialization routine, can be *verified*, *unverified*, or works with some *caveat* (see “Performance Monitor Accuracy” about testing status and event accuracy).

An event filter (which is any combination of the status bits) must be passed to the **pm_init** routine to force the return of events with status matching the filter. If no filter is passed to the **pm_init** routine, no events will be returned.

The following topics discuss programming the Performance Monitor API:

Performance Monitor Accuracy

Only events marked *verified* have gone through full verification. Events marked *caveat* have been verified within the limitations documented in the event description returned by the **pm_init** routine.

Events marked *unverified* have undefined accuracy. Use caution with *unverified* events. The Performance Monitor API is essentially providing a service to read hardware registers that may not have any meaningful content.

Users may experiment with *unverified* event counters and determine for themselves if they can be used for specific tuning situations.

Performance Monitor Context and State

To provide Performance Monitor data access at various levels, the AIX operating system supports optional performance monitoring contexts. These contexts are an extension to the regular processor and thread contexts and include one 64-bit counter per hardware counter and a set of control words. The control words define which events are counted and when counting is on or off.

System-Level Context and Accumulation

For the system-level APIs, optional Performance Monitor contexts can be associated with each of the processors. When installed, the Performance Monitor kernel extension automatically handles 32-bit Performance Monitor hardware counter overflows. It also maintains per-processor sets of 64-bit accumulation counters (one per 32-bit hardware Performance Monitor counter).

Thread Context

Optional Performance Monitor contexts can also be associated with each kernel thread. The AIX operating system and the Performance Monitor kernel extension automatically maintain sets of 64-bit counters for each of these contexts.

Thread Counting-Group and Process Context

The concept of thread counting-group is optionally supported by the thread-level APIs. All the threads within a group, in addition to their own Performance Monitor context, share a group accumulation context. A thread group is defined as all the threads created by a common ancestor thread. By definition, all the threads in a thread group count the same set of events, and, with one exception described below, the group must be created before any of the descendant threads are created. This restriction is due to the fact that, after descendant threads are created, it is impossible to determine a list of threads with a common ancestor.

One special case of a group is the collection of all the threads belonging to a process. Such a group can be created at any time regardless of when the descendant threads are created, because a list of threads belonging to a process can be generated. Multiple groups can coexist within a process, but each thread can be a member of only one Performance Monitor counting-group. Because all the threads within a group must be counting the same events, a process group creation will fail if any thread within the process already has a context.

Performance Monitor State Inheritance

The PM state is defined as the combination of the Performance Monitor programmation (the events being counted), the counting state (on or off), and the optional thread group membership. A counting state is associated with each group. When the group is created, its counting state is inherited from the initial thread in the group. For thread members of a group, the effective counting state is the result of AND-ing their own counting state with the group counting state. This provides a way to effectively control the counting state for all threads in a group. Simply manipulating the group-counting state will affect the effective counting state of all the threads in the group. Threads inherit their complete Performance Monitor state from their parents when the thread is created. A thread Performance Monitor context data (the value of the 64-bit counters) is not inherited, that is, newly created threads start with counters set to zero.

Thread Accumulation and Thread Group Accumulation

When a thread gets suspended (or redispatched), its 64-bit accumulation counters are updated. If the thread is member of a group, the group accumulation counters are updated at the same time.

Similarly, when a thread stops counting or reads its Performance Monitor data, its 64 bit accumulation counters are also updated by adding the current value of the Performance Monitor hardware counters to them. Again, if the thread is a member of a group, the group accumulation counters are also updated, regardless of whether the counter read or stop was for the thread or for the thread group.

The group-level accumulation data is kept consistent with the individual Performance Monitor data for the thread members of the group, whenever possible. When a thread voluntarily leaves a group, that is, deletes its Performance Monitor context, its accumulated data is automatically subtracted from the group-level accumulated data. Similarly, when a thread member in a group resets its own data, the data in question is subtracted from the group level accumulated data. When a thread dies, no action is taken on the group-accumulated data.

The only situation where the group-level accumulation is not consistent with the sum of the data for each of its members is when the group-level accumulated data has been reset, and the group has more than one member. This situation is detected and marked by a bit returned when the group data is read.

Security Considerations

The system-level APIs calls are only available from the root user except when the process tree option is used. In that case, a locking mechanism prevents calls being made from more than one process. This mechanism ensures ownership of the API and exclusive access by one process from the time that the system-level contexts are created until they are deleted.

Enabling the process tree option results in counting for only the calling process and its descendants; the default is to count all activities on each processor.

Because the system-level APIs would report bogus data if thread contexts where in use, system-level API calls are not allowed at the same time as thread-level API calls. The allocation of the first thread context will take the system-level API lock, which will not be released until the last context has been deallocated.

When using first party calls, a thread is only allowed to modify its own Performance Monitor context. The only exception to this rule is when making group level calls, which obviously affect the group context, but can also affect other threads' context. Deleting a group deletes all the contexts associated with the group, that is, the caller context, the group context, and all the contexts belonging to all the threads in the group.

Access to a Performance Monitor context not belonging to the calling thread or its group is available only from the target process's debugger program. The third party API calls only succeed when the target process is being **ptraced** by the API caller, that is, the caller is already attached to the target process, and the target process is stopped.

The fact that the debugger program must already have been attached to the debugged thread before any third party call to the API can be made, ensures that the security level of the API will be the same as the one used between debugger programs and process being debugged.

Common Rules

The following rules are common to the Performance Monitor APIs:

- The **pm_init** routine must be called before any other API call can be made, and only events returned by a given **pm_init** call with its associated filter setting can be used in subsequent **pm_set_program** calls.
- PM contexts cannot be reprogrammed or reused at any time. This means that none of the APIs support more than one call to a **pm_set_program** interface without a call to a **pm_delete_program** interface. This also means that when creating a process group, none of the threads in the process is allowed to already have a context.
- All the API calls return 0 when successful or a positive error code (which can be decoded using **pm_error**) otherwise.

The pm_init API Initialization Routine

The **pm_init** routine returns (in a structure of type **pm_info_t** pointed to by its second parameter) the processor name, the number of counters available, the list of available events for each counter, and the threshold multipliers supported. Some processor support two threshold multipliers, others none, meaning that thresholding is not supported at all.

For each event returned, in addition to the testing status, the **pm_init** routine also returns the identifier to be used in subsequent API calls, a short name, and a long name. The short name is a mnemonic name in the form PM_MNEMONIC. Events that are the same on different processors will have the same mnemonic name. For instance, PM_CYC and PM_INST_CMPL are respectively the number of processor cycles and

instruction completed and should exist on all processors. For each event returned, a thresholdable flag is also returned. This flag indicates whether an event can be used with a threshold. If so, then specifying a threshold defers counting until a number of cycles equal to the threshold multiplied by the processor's selected threshold multiplier has been exceeded.

Beginning with AIX level 5.1.0.15, the Performance Monitoring API enables the specification of event groups instead of individual events. Event groups are predefined sets of events. Rather than each event being individually specified, a single group ID is specified. The interface to the **pm_init** routine has been enhanced to return the list of supported event groups in a structure of type **pm_groups_info_t** pointed to by a new optional third parameter. To preserve binary compatibility, the third parameter must be explicitly announced by OR-ing the PM_GET_GROUPS bitflag into the filter. Some events on some platforms can only be used from within a group. This is indicated in the threshold flag associated with each event returned. The following convention is used:

y A thresholdable event

g An event that can only be used in a group

G A thresholdable event that can only be used in a group

n A non-thresholdable event that is usable individually

On some platforms, use of event groups is required because all the events are marked **g** or **G**. Each of the event groups that are returned includes a short name, a long name, and a description similar to those associated with events, as well as a group identifier to be used in subsequent API calls and the events contained in the group (in the form of an array of event identifiers).

The testing status of a group is defined as the lowest common denominator among the testing status of the events that it includes. If at least one event has a testing status of **caveat**, the group testing status is at best **caveat**, and if at least one event has a status of **unverified**, then the group status is **unverified**. This is not returned as a group characteristic, but it is taken into account by the filter. Like events, only groups with status matching the filter are returned.

Eight Basic API Calls

Each of the eight sections below describes a system-wide API call that has variations for first-party kernel thread or group counting, and third-party kernel thread or group counting. Variations are indicated by suffixes to the function call names, such as **pm_set_program**, **pm_set_program_mythread**, and **pm_set_program_group**.

pm_set_program
Sets the counting configuration. Use this call to specify the events (as a list of event identifiers, one per counter, or as a single event-group identifier) to be counted, and a mode in which to count. The list of events to choose from is returned by the **pm_init** routine. If the list includes a thresholdable event, you can also use this call to specify a threshold, and a threshold multiplier.

The mode in which to count can include user-mode and kernel-mode counting, and whether to start counting immediately. For the system-wide API call, the mode also includes whether to turn counting on only for a process and its descendants or for the whole system. For counting group API calls, the mode includes the type of counting group to create, that is, a group containing the initial thread and its future descendants, or a process-level group, which includes all the threads in a process.

pm_get_program
Retrieves the current Performance Monitor settings. This includes mode information and the list of events (or the event group) being counted. If the list includes a thresholdable event, this call also returns a threshold and the multiplier used.

pm_delete_program
Deletes the Performance Monitor configuration. Use this call to undo **pm_set_program**.

pm_start
Starts Performance Monitor counting.

pm_stop
Stops Performance Monitor counting.

pm_get_data
Returns Performance Monitor counting data. The data is a set of 64-bit values, one per hardware counter. For the counting group API calls, the group information is also returned. (See “Thread Counting-Group Information”.)

The **pm_get_data_cpu** interface returns the Performance Monitor counting data for a single processor.

pm_get_tdata
Same as **pm_get_data**, but includes a timestamp that indicates the last time that the hardware Performance Monitoring counters were read. This is a timebase value that can be converted to time by using **time_base_to_time**.

The **pm_get_tdata_cpu** interface returns the Performance Monitor counting data for a single processor accompanied with a timestamp.

pm_reset_data
Resets Performance Monitor counting data. All values are set to 0.

Thread Counting-Group Information

The following information is associated with each thread counting-group:

member count
The number of threads that are members of the group. This includes deceased threads that were members of the group when running.

If the consistency flag is on, the count will be the number of threads that have contributed to the group-level data.

process flag
Indicates that the group includes all the threads in the process.

consistency flag
Indicates that the group PM data is consistent with the sum of the individual PM data for the thread members.

This information is returned by the **pm_get_data_mygroup** and **pm_get_data_group** interfaces in a **pm_groupinfo_t** structure.

Examples

The following are examples of using Performance Monitor APIs in pseudo-code. Functional sample code is available in the **/usr/samples/pmapi** directory.

Simple Single-Threaded Program:

```
# include <pmapi.h>
main()
{
       pm_info_t pminfo;
       pm_prog_t prog;
       pm_data_t data;
       int filter = PM_VERIFIED; /* use only verified events */

       pm_init(filter, &pminfo)
```

```
        prog.mode.w       = 0;  /* start with clean mode */
        prog.mode.b.user  = 1;  /* count only user mode */

        for (i = 0; i < pminfo.maxpmcs; i++)
                 prog.events[i] = COUNT_NOTHING;

        prog.events[0]    = 1;  /* count event 1 in first counter */
        prog.events[1]    = 2;  /* count event 2 in second counter */

        pm_program_mythread(&prog);
        pm_start_mythread();

(1)     ... usefull work ....

        pm_stop_mythread();
        pm_get_data_mythread(&data);

        ... print results ...
}
```

Initialization Example Using an Event Group:

```
# include <pmapi.h>
main()
{
        pm_info_t        pminfo;
        pm_prog_t        prog;
        pm_groups_info_t pmginfo;

        int filter = PM_VERIFIED|PM_GET_GROUPS;  /* get list of verified events and groups */

        pm_init(filter, &pminfo;, &pmginfo;)

        prog.mode.w           = 0;  /* start with clean mode */
        prog.mode.b.user      = 1;  /* count only user mode */
        prog.mode.b.is_group  = 1;  /* specify event group */

        for (i = 0; i < pminfo.maxpmcs; i++)
                 prog.events[i] = COUNT_NOTHING;

        prog.events[0]    = 1;  /* count events in group 1 */
        .....
}
```

Debugger Program Example for Initialization Program:

The following illustrates how to look at the Performance Monitor data while the program is executing:

```
from a debugger at breakpoint (1)

        pm_init(filter);
(2)     pm_get_program_thread(pid, tid, &prog);
        ... display PM programmation ...

(3)     pm_get_data_thread(pid, tid);
        ... display PM data ...

        pm_delete_program_thread(pid, tid);
        prog.events[0] = 2; /* change counter 1 to count event number 2 */
        pm_set_program_thread(pid, tid, &prog);

continue program
```

The preceding scenario would also work if the program being executed under the debugger did not have any embedded Performance Monitor API calls. The only difference would be that the calls at (2) and (3) would fail, and that when the program continues, it will be counting only event number 2 in counter 1, and nothing in other counters.

Simple Multi-Threaded Example:

The following is a simple multi-threaded example with independent threads counting the same set of events.

```
# include <pmapi.h>
pm_data_t data2;

void *
doit(void *)
{

(1)    pm_start_mythread();

       ... usefull work ....

       pm_stop_mythread();
       pm_get_data_mythread(&data2);
}

main()
{
       pthread_t threadid;
       pthread_attr_t attr;
       pthread_addr_t status;

       ... same initialization as in previous example ...

       pm_program_mythread(&prog);

       /* setup 1:1 mode, M:N not supported by APIs */
       pthread_attr_init(&attr);
       pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
       pthread_create(&threadid, &attr, doit, NULL);

(2)    pm_start_mythread();

       ... usefull work ....

       pm_stop_mythread();
       pm_get_data_mythread(&data);

       ... print main thread results (data )...

       pthread_join(threadid, &status);

       ... print auxiliary thread results (data2) ...
}
```

In the preceding example, counting starts at (1) and (2) for the main and auxiliary threads respectively because the initial counting state was off and it was inherited by the auxiliary thread from its creator.

Simple Thread Counting-Group Example:

The following example has two threads in a counting-group. The body of the auxiliary thread's initialization routine is the same as in the previous example.

```
main()
{
        ... same initialization as in previous example ...
```

```
        pm_program_mygroup(&prog); /* create counting group */
(1)     pm_start_mygroup()

        pthread_create(&threadid, &attr, doit, NULL)

(2)     pm_start_mythread();

        ... usefull work ....

        pm_stop_mythread();
        pm_get_data_mythread(&data)


        ... print main thread results ...

        pthread_join(threadid, &status);

        ... print auxiliary thread results ...

        pm_get_data_mygroup(&data)


        ... print group results ...
}
```

In the preceding example, the call in (2) is necessary because the call in (1) only turns on counting for the group, not the individual threads in it. At the end, the group results are the sum of both threads results.

Thread Counting Example with Reset:

The following example with a reset call illustrates the impact on the group data. The body of the auxiliary thread is the same as before, except for the **pm_start_mythread** call, which is not necessary in this case.

```
main()
{
        ... same initialization as in previous example...

        prog.mode.b.count  = 1;  /* start counting immediately */
        pm_program_mygroup(&prog);

        pthread_create(&threadid, pthread_attr_default, doit, NULL)

        ... usefull work ....

        pm_stop_mythread()
        pm_reset_data_mythread()

        pthread_join(threadid, &status);

        ...print auxiliary thread results...

        pm_get_data_mygroup(&data)


        ...print group results...
}
```

In the preceding example, the main thread and the group counting state are both on before the auxiliary thread is created, so the auxiliary thread will inherit that state and start counting immediately.

At the end, **data1** is equal to **data** because the **pm_reset_data_mythread** automatically subtracted the main thread data from the group data to keep it consistent. In fact, the group data remains equal to the sum of the auxiliary and the main thread data, but in this case, the main thread data is null.

Chapter 6. Perfstat API Programming

The **perfstat** application programming interface (API) is a collection of C programming language subroutines that execute in user space and uses the **perfstat** kernel extension to extract various AIX performance metrics. System component information is also retrieved from the Object Data Manager (ODM) and returned with the performance metrics.

The **perfstat** API is both a 32–bit and a 64–bit API, is thread–safe, and does not require root authority.

The API supports extensions so binary compatibility is maintained across all releases. This is accomplished by using one of the parameters in all the API calls to specify the size of the data structure to be returned. This allows the library to easily determine which version is in use, as long as the structures are only growing, which is guaranteed. This releases the user from version dependencies. For the list of extensions made in earlier versions of AIX, see the Change History section.

The **perfstat** API subroutines reside in the **libperfstat.a** library and are part of the **bos.perf.libperfstat** fileset, which is installable from the AIX base installation media and requires that the **bos.perf.perfstat** fileset is installed. The later contains the kernel extension and is automatically installed with AIX.

The **/usr/include/libperfstat.h** file contains the interface declarations and type definitions of the data structures to use when calling the interfaces. This **include** file is also part of the **bos.perf.libperfstat** fileset. Sample source code is provided with **bos.perf.libperfstat** and resides in the **/usr/samples/libperfstat** directory. Detailed information for the individual interfaces and the data structures used can be found in the **libperfstat.h** file in the *AIX 5L Version 5.2 Files Reference*.

API Characteristics

Two types of APIs are available. Global types return global metrics related to a set of components, while the individual types return metrics related to individual components. Both types of interfaces have similar signatures, but slightly different behavior.

All the interfaces return raw data; that is, values of running counters. Multiple calls must be made at regular intervals to calculate rates.

Several interfaces return data retrieved from the ODM (object data manager) database. This information is automatically cached into a dictionary that is assumed to be ″frozen″ after it is loaded. The **perfstat_reset** subroutine must be called to clear the dictionary whenever the machine configuration has changed.

Most types returned are unsigned long long; that is, unsigned 64-bit data. This provides complete kernel independence. Some kernel internal metrics are in fact 32-bit wide in the 32-bit kernel, and 64-bit wide in the 64-bit kernel. The corresponding **libperfstat** APIs data type is always unsigned 64-bit.

All of the examples presented in this chapter can be compiled in AIX 5.2 using **cc** with **-lperfstat**.

Global Interfaces

Global interfaces report metrics related to a set of components on a system (such as processors, disks, memory).

All of the following AIX 5.2 interfaces use the naming convention **perfstat**_*subsystem*_**total**, and use a common signature:

perfstat_cpu_total	Retrieves global CPU usage metrics
perfstat_memory_total	Retrieves global memory usage metrics

perfstat_disk_total	Retrieves global disk usage metrics
perfstat_netinterface_total	Retrieves global network interfaces metrics

The common signature used by all of the global interfaces is as follows:

```
int perfstat_subsystem_total(perfstat_id_t *name,
                             perfstat_subsystem_total_t *userbuff,
                             int sizeof_struct,
                             int desired_number);
```

The usage of the parameters for all of the interfaces is as follows:

perfstat_id_t *name	Reserved for future use, should be NULL
perfstat_subsystem_total_t *userbuff	A pointer to a memory area with enough space for the returned structure
int sizeof_struct	Should be set to **sizeof(perfstat_subsystem_t)**
int desired_number	Reserved for future use, must be set to 0 or 1

The return value will be -1 in case of errors. Otherwise, the number of structures copied is returned. This is always 1.

An exception to this scheme is: when **name=NULL**, **userbuff=NULL** and **desired_number=0**, the total number of structures available is returned. This is always 1.

The following sections provide examples of the type of data returned and code using each of the interfaces.

perfstat_cpu_total Interface

The **perfstat_cpu_total** function returns a **perfstat_cpu_total_t** structure, which is defined in the **libperfstat.h** file. Selected fields from the **perfstat_cpu_total_t** structure include:

processorHz	Processor speed in Hertz (from ODM)
description	Processor type (from ODM)
ncpus	Current number of active CPUs
ncpus_cfg	Number of configured CPUs; that is, the maximum number of processors that this copy of AIX can handle simultaneously
ncpus_high	Maximum number of active CPUs; that is, the maximum number of active processors since the last reboot
user	Total number of clock ticks spent in user mode
sys	Total number of clock ticks spent in system (kernel) mode
idle	Total number of clock ticks spent idle with no I/O pending
wait	Total number of clock ticks spent idle with I/O pending

Several other processor-related counters (such as number of system calls, number of reads, write, forks, execs, and load average) are also returned. For a complete list, see the **perfstat_cpu_total_t** section of the **libperfstat.h** header file in *AIX 5L Version 5.2 Files Reference*.

The following code shows an example of how **perfstat_cpu_total** is used:

```
#include <stdio.h>
#include <sys/time.h>
#include <libperfstat.h>
```

```
unsigned long long last_tot, last_user, last_sys, last_idle, last_wait;

int
main(int argc, char *argv[]) {
   perfstat_cpu_total_t cpu_total_buffer;
   unsigned long long cur_tot;
   unsigned long long delt_tot, delt_user, delt_sys, delt_idle, delt_wait;

   /* get initial set of data */
   perfstat_cpu_total(NULL, &cpu_total_buffer, sizeof(perfstat_cpu_total_t), 1);

   /* print general processor information */
   printf("Processors: (%d:%d) %s running at %llu MHz\n",
          cpu_total_buffer.ncpus, cpu_total_buffer.ncpus_cfg,
          cpu_total_buffer.description, cpu_total_buffer.processorHZ/1000000);

   /* save values for delta calculations */
   last_tot = cpu_total_buffer.user + cpu_total_buffer.sys  +
              cpu_total_buffer.idle + cpu_total_buffer.wait;

   last_user  = cpu_total_buffer.user;
   last_sys   = cpu_total_buffer.sys;
   last_idle  = cpu_total_buffer.idle;
   last_wait  = cpu_total_buffer.wait;

   printf("\n User    Sys   Idle   Wait  Total  Rate\n");

   while(1 == 1) {
      sleep(1);

      /* get new values after one second */
      perfstat_cpu_total(NULL, &cpu_total_buffer, sizeof(perfstat_cpu_total_t), 1);


      /* calculate current total number of ticks */
      cur_tot = cpu_total_buffer.user + cpu_total_buffer.sys  +
                cpu_total_buffer.idle + cpu_total_buffer.wait;

      delt_user = cpu_total_buffer.user - last_user;
      delt_sys  = cpu_total_buffer.sys  - last_sys;
      delt_idle = cpu_total_buffer.idle - last_idle;
      delt_wait = cpu_total_buffer.wait - last_wait;
      delt_tot  = cur_tot - last_tot;

      /* print percentages, total delta ticks and tick rate per cpu per sec */
      printf("%#5.1f  %#5.1f  %#5.1f  %#5.1f  %-5llu  %llu\n",
             100.0 * (double) delt_user / (double) delt_tot,
             100.0 * (double) delt_sys  / (double) delt_tot,
             100.0 * (double) delt_idle / (double) delt_tot,
             100.0 * (double) delt_wait / (double) delt_tot,
             delt_tot, delt_tot/cpu_total_buffer.ncpus);

      /* save current value for next time */
      last_tot  = cur_tot;
      last_user = cpu_total_buffer.user;
      last_sys  = cpu_total_buffer.sys;
      last_idle = cpu_total_buffer.idle;
      last_wait = cpu_total_buffer.wait;
   }
}
```

The preceding program produces (on a single PowerPc 604e microprocessor-based machine) output similar to the following:

```
Processors: (1:1) PowerPC_604e running at 375 MHz

 User    Sys   Idle   Wait  Total  Rate
 19.0   31.0    1.0   49.0  100    100
 20.8   34.7    0.0   44.6  101    101
 35.0   30.0    0.0   35.0  100    100
 12.0   20.0    0.0   68.0  100    100
 19.0   33.0    0.0   48.0  100    100
 29.0   43.0   11.0   17.0  100    100
 23.0   30.0   25.0   22.0  100    100
 24.0   25.0   15.0   36.0  100    100
 26.0   27.0   25.0   22.0  100    100
 20.0   32.0   37.0   11.0  100    100
 16.0   22.0   49.0   13.0  100    100
 16.0   33.0   18.0   33.0  100    100
```

perfstat_memory_total Interface

The **perfstat_memory_total** function returns a **perfstat_memory_total_t structure**, which is defined in the **libperfstat.h** file. Selected fields from the **perfstat_memory_total_t** structure include:

virt_total	Amount of virtual memory (in units of 4 KB pages)
real_total	Amount of real memory (in units of 4 KB pages)
real_free	Amount of free real memory (in units of 4 KB pages)
real_pinned	Amount of pinned memory (in units of 4 KB pages)
pgins	Number of pages paged in
pgouts	Number of pages paged out
pgsp_total	Total amount of paging space (in units of 4 KB pages)
pgsp_free	Amount of free paging space (in units of 4 KB pages)
pgsp_rsvd	Amount of reserved paging space (in units of 4 KB pages)

Several other memory related metrics (such as number of paging space paged in and out, and amount of system memory) are also returned. For a complete list, see the **perfstat_memory_total_t** section in **libperfstat.h**.

The following code shows an example of how **perfstat_memory_total** is used:

```
#include <stdio.h>
#include <libperfstat.h>

int main(int argc, char* argv[]) {
    perfstat_memory_total_t minfo;

    perfstat_memory_total(NULL, &minfo, sizeof(perfstat_memory_total_t), 1);

    printf("Memory statistics\n");
    printf("-----------------\n");
    printf("real memory size                : %llu MB\n",
           minfo.real_total*4096/1024/1024);
    printf("reserved paging space           : %llu MB\n",minfo.pgsp_rsvd);
    printf("virtual memory size             : %llu MB\n",
           minfo.virt_total*4096/1024/1024);
    printf("number of free pages            : %llu\n",minfo.real_free);
    printf("number of pinned pages          : %llu\n",minfo.real_pinned);
    printf("number of pages in file cache   : %llu\n",minfo.numperm);
    printf("total paging space pages        : %llu\n",minfo.pgsp_total);
    printf("free paging space pages         : %llu\n", minfo.pgsp_free);
    printf("used paging space               : %3.2f%%\n",
        (float)(minfo.pgsp_total-minfo.pgsp_free)*100.0/
        (float)minfo.pgsp_total);
```

```
    printf("number of paging space page ins  : %llu\n",minfo.pgspins);
    printf("number of paging space page outs : %llu\n",minfo.pgspouts);
    printf("number of page ins               : %llu\n",minfo.pgins);
    printf("number of page outs              : %llu\n",minfo.pgouts);
}
```

The preceding program produces output similar to the following:

```
Memory statistics
-----------------
real memory size                : 256 MB
reserved paging space           : 512 MB
virtual memory size             : 768 MB
number of free pages            : 32304
number of pinned pages          : 6546
number of pages in file cache   : 12881
total paging space pages        : 131072
free paging space pages         : 129932
used paging space               : 0.87%
number of paging space page ins  : 0
number of paging space page outs : 0
number of page ins              : 20574
number of page outs             : 92508
```

perfstat_disk_total Interface

The **perfstat_disk_total** function returns a **perfstat_disk_total_t** structure, which is defined in the **libperfstat.h** file. Selected fields from the **perfstat_disk_total_t** structure include:

number	Number of disks
size	Total disk size (in MB)
free	Total free disk space (in MB)
xfers	Total transfers to/from disk (in KB)

Several other disk-related metrics, such as number of blocks read from and written to disk, are also returned. For a complete list, see the **perfstat_disk_total_t** section in **libperfstat.h**.

The following code shows an example of how **perfstat_disk_total** is used:

```
#include <stdio.h>
#include <libperfstat.h>

int main(int argc, char* argv[]) {
    perfstat_disk_total_t dinfo;

    perfstat_disk_total(NULL, &dinfo, sizeof(perfstat_disk_total_t), 1);

    printf("Total disk statistics\n");
    printf("---------------------\n");
    printf("number of  disks         : %d\n",   dinfo.number);
    printf("total disk space         : %llu\n", dinfo.size);
    printf("total free space         : %llu\n", dinfo.free);
    printf("number of transfers      : %llu\n", dinfo.xfers);
    printf("number of blocks written : %llu\n", dinfo.wblks);
    printf("number of blocks read    : %llu\n", dinfo.rblks);
}
```

This program produces output similar to the following:

```
Total disk statistics
---------------------
number of  disks         : 3
total disk space         : 4296
total free space         : 2912
number of transfers      : 77759
number of blocks written : 738016
number of blocks read    : 363120
```

perfstat_netinterface_total Interface

The **perfstat_netinterface_total** function returns a **perfstat_netinterface_total_t** structure, which is defined in the **libperfstat.h** file. Selected fields from the **perfstat_netinterface_total_t** structure include:

number	Number of network interfaces
ipackets	Total number of input packets received on all network interfaces
opackets	Total number of output packets sent on all network interfaces
ierror	Total number of input errors on all network interfaces
oerror	Total number of output errors on all network interfaces

Several other network interface related metrics (such as number of bytes sent and received). For a complete list, see the **perfstat_netinterface_total_t** section in **libperfstat.h** .

The following code shows an example of how **perfstat_netinterface_total** is used:

```
#include <stdio.h>
#include <libperfstat.h>

int main(int argc, char* argv[]) {
    perfstat_netinterface_total_t ninfo;

    perfstat_netinterface_total(NULL, &ninfo, sizeof(perfstat_netinterface_total_t), 1);

    printf("Network interfaces statistics\n");
    printf("-----------------------------\n");
    printf("number of interfaces : %d\n",   ninfo.number);
    printf("\ninput statistics:\n");
    printf("number of packets    : %llu\n", ninfo.ipackets);
    printf("number of errors     : %llu\n", ninfo.ierrors);
    printf("number of bytes      : %llu\n", ninfo.ibytes);
    printf("\noutput statistics:\n");
    printf("number of packets    : %llu\n", ninfo.opackets);
    printf("number of bytes      : %llu\n", ninfo.obytes);
    printf("number of errors     : %llu\n", ninfo.oerrors);
}
```

The program above produces output similar to this:

```
Network interfaces statistics
-----------------------------
number of interfaces : 2

input statistics:
number of packets    : 306688
number of errors     : 0
number of bytes      : 24852688

output statistics:
```

```
number of packets    : 63005
number of bytes      : 11518591
number of errors     : 0
```

Component-Specific Interfaces

Component-specific interfaces report metrics related to individual components on a system (such as a processor, disk, network interface, or paging space).

All of the following interfaces use the naming convention **perfstat_subsystem**, and use a common signature:

perfstat_cpu	Retrieves individual CPU usage metrics
perfstat_disk	Retrieves individual disk usage metrics
perfstat_diskadapter	Retrieves individual disk adapter metrics
perfstat_netinterface	Retrieves individual network interfaces metrics
perfstat_protocol	Retrieves individual network protocol related metrics
perfstat_netbuffer	Retrieves individual network buffer allocation metrics
perfstat_pagingspace	Retrieves individual paging space metrics

The common signature used by all the component interfaces is as follows:

```
int perfstat_subsystem(perfstat_id *name,
                       perfstat_subsystem_t * userbuff,
                       int sizeof_struct,
                       int desired_number);
```

The usage of the parameters for all of the interfaces is as follows:

perfstat_id_t *name	The name of the first component (for example **hdisk2** for **perfstat_disk()**) for which statistics are desired. A structure containing a char * field is used instead of directly passing a char * argument to the function to avoid allocation errors and to prevent the user from giving a constant string as parameter. To start from the first component of a subsystem, set the char* field of the name parameter to "" (empty string). You can also use the macros such as **FIRST**_*SUBSYSTEM* (for example,**FIRST_CPU**) defined in the **libperfstat.h** file.
perfstat_subsystem_total_t *userbuff	A pointer to a memory area with enough space for the returned structure(s).
int sizeof_struct	Should be set to **sizeof(perfstat_subsystem_t)**.
int desired_number	The number of structures of type **perfstat_subsystem_t** to return in userbuff.

The return value will be -1 in case of error. Otherwise, the number of structures copied is returned. The field name is either set to NULL or to the name of the next structure available.

An exception to this scheme is when **name=NULL**, **userbuff=NULL** and **desired_number=0**, the total number of structures available is returned.

To retrieve all structures of a given type, either ask first for their number, allocate enough memory to hold them all at once, then call the appropriate API to retrieve them all in one call. Otherwise, allocate a fixed set of structures and repeatedly call the API to get the next such number of structures, each time passing the name returned by the previous call. Start the process with the name set to "" or **FIRST_SUBSYSTEM**, and repeat the process until the name returned is equal to "".

Minimizing the number of API calls, and therefore the number of system calls, will always lead to more efficient code, so the two-call approach should be preferred. Some of the examples shown in the following sections illustrate the API usage using the two-call approach. Because the two-call approach can lead to a lot of memory being allocated, the multiple-call approach must sometime be used and is illustrated in the following examples.

The following sections provide examples of the type of data returned and code using each of the interfaces.

perfstat_cpu interface

The **perfstat_cpu** function returns a set of structures of type **perfstat_cpu_t**, which is defined in the **libperfstat.h** file. Selected fields from the **perfstat_cpu_t** structure include:

name	Logical CPU name (cpu0, cpu1, ...)
user	Number of clock ticks spent in user mode
sys	Number of clock ticks spent in system (kernel) mode
idle	Number of clock ticks spent idle with no I/O pending
wait	Number of clock ticks spent idle with I/O pending
syscall	Number of system call executed

Several other CPU related metrics (such as number of forks, read, write, and execs) are also returned. For a complete list, see the **perfstat_cpu_t** section in the **libperfstat.h** file.

The following code shows an example of how **perfstat_cpu** is used:

```
#include <stdio.h>
#include <stdlib.h>
#include <libperfstat.h>

int main(int argc, char *argv[]) {
   int i, retcode, cputotal;
   perfstat_id_t firstcpu;
   perfstat_cpu_t *statp;

   /* check how many perfstat_cpu_t structures are available */
   cputotal =  perfstat_cpu(NULL, NULL, sizeof(perfstat_cpu_t), 0);

   printf("number of perfstat_cpu_t available : %d\n", cputotal);

   /* allocate enough memory for all the structures */
   statp = calloc(cputotal,sizeof(perfstat_cpu_t));

   /* set name to first cpu */
   strcpy(firstcpu.name, FIRST_CPU);

   /* ask to get all the structures available in one call */
   retcode = perfstat_cpu(&firstcpu, statp, sizeof(perfstat_cpu_t), cputotal);

   /* return code is number of structures returned */
   printf("number of perfstat_cpu_t returned  : %d\n", retcode);

   for (i = 0; i < retcode; i++) {
      printf("\nStatistics for CPU : %s\n", statp[i].name);
      printf("------------------\n");
      printf("CPU user time (raw ticks)  : %llu\n", statp[i].user);
      printf("CPU sys  time (raw ticks)  : %llu\n", statp[i].sys);
      printf("CPU idle time (raw ticks)  : %llu\n", statp[i].idle);
      printf("CPU wait time (raw ticks)  : %llu\n", statp[i].wait);
      printf("number of syscalls         : %llu\n", statp[i].syscall);
      printf("number of readings         : %llu\n", statp[i].sysread);
```

```
        printf("number of writings         : %llu\n", statp[i].syswrite);
        printf("number of forks            : %llu\n", statp[i].sysfork);
        printf("number of execs            : %llu\n", statp[i].sysexec);
        printf("number of char read        : %llu\n", statp[i].readch);
        printf("number of char written     : %llu\n", statp[i].writech);
        }
}
```

On a single processor machine, the preceding program produces output similar to the following:

```
number of perfstat_cpu_t available : 1
number of perfstat_cpu_t returned  : 1

Statistics for CPU : cpu0
------------------
CPU user time (raw ticks)  : 1336297
CPU sys  time (raw ticks)  : 111958
CPU idle time (raw ticks)  : 57069585
CPU wait time (raw ticks)  : 19545
number of syscalls         : 4734311
number of readings         : 562121
number of writings         : 323367
number of forks            : 6839
number of execs            : 7257
number of char read        : 753568874
number of char written     : 132494990
```

In an environment where dynamic logical partitioning is used, the number of **perfstat_cpu_t** structures available will always be equal to the **ncpus_high** field in the **perfstat_cpu_total_t**. This number represents the highest index of any active processor since the last reboot. Kernel data structures holding performance metrics for processors are not deallocated when processors are turned offline or moved to a different partition. They simply stop being updated. The **ncpus** field of the **perfstat_cpu_total_t** structure always represents the number of active processors, but the **perfstat_cpu** interface will always return **ncpus_high** structures.

Applications can detect offline or moved processors by checking clock-tick increments. If the sum of the user, sys, idle and wait fields is identical for a given processor between two **perfstat_cpu** calls, that processor has been offline for the complete interval. If the sum multiplied by 10 ms (the value of a clock tick) does not match the time interval, the processor has not been online for the complete interval.

perfstat_disk Interface

The **perfstat_disk** function returns a set of structures of type **perfstat_disk_t**, which is defined in the **libperfstat.h** file. Selected fields from the **perfstat_disk_t** structure include:

name	Disk name (from ODM)
description	Disk description (from ODM)
vgname	Volume group name (from ODM)
size	Disk size (in MB)
free	Free space (in MB)
xfers	Transfers to/from disk (in KB)

Several other disk related metrics (such as number of blocks read from and written to disk, and adapter names) are also returned. For a complete list, see the **perfstat_disk_t** section in the **libperfstat.h** header file.

The following code shows an example of how **perfstat_disk** is used:

```
#include <stdio.h>
#include <stdlib.h>
#include <libperfstat.h>

int main(int argc, char* argv[]) {
    int i, ret, tot;
    perfstat_disk_t *statp;
    perfstat_id_t first;

    /* check how many perfstat_disk_t structures are available */
    tot =  perfstat_disk(NULL, NULL, sizeof(perfstat_disk_t), 0);

    /* allocate enough memory for all the structures */
    statp = calloc(tot, sizeof(perfstat_disk_t));

    /* set name to first interface */
    strcpy(first.name, FIRST_DISK);

    /* ask to get all the structures available in one call */
    /* return code is number of structures returned */
    ret = perfstat_disk(&first, statp,
                        sizeof(perfstat_disk_t), tot);



    /* print statistics for each of the disks */
    for (i = 0; i < ret; i++) {
        printf("\nStatistics for disk : %s\n", statp[i].name);
        printf("-------------------\n");
        printf("description              : %s\n", statp[i].description);
        printf("volume group name        : %s\n", statp[i].vgname);
        printf("adapter name             : %s\n", statp[i].adapter);
        printf("size                     : %llu MB\n", statp[i].size);
        printf("free space               : %llu MB\n", statp[i].free);
        printf("number of blocks read    : %llu\n", statp[i].rblks);
        printf("number of blocks written : %llu\n", statp[i].wblks);
        }
}
```

The preceding program produces output similar to the following:

```
Statistics for disk : hdisk1
-------------------
description              : 16 Bit SCSI Disk Drive
volume group name        : rootvg
adapter name             : scsi0
size                     : 4296 MB
free space               : 2912 MB
number of blocks read    : 403946
number of blocks written : 768176

Statistics for disk : hdisk0
-------------------
description              : 16 Bit SCSI Disk Drive
volume group name        : None
adapter name             : scsi0
size                     : 0 MB
free space               : 0 MB
number of blocks read    : 0
number of blocks written : 0

Statistics for disk : cd0
-------------------
description              : SCSI Multimedia CD-ROM Drive
volume group name        : not available
adapter name             : scsi0
```

```
size                     : 0 MB
free space               : 0 MB
number of blocks read    : 3128
number of blocks written : 0
```

perfstat_diskadapter Interface

The **perfstat_diskadapter** function returns a set of structures of type **perfstat_diskadapter_t**, which is defined in the **libperfstat.h** file. Selected fields from the **perfstat_diskadapter_t** structure include:

name	Adapter name (from ODM)
description	Adapter description (from ODM)
size	Total disk size connected to this adapter (in MB)
free	Total free space on disks connected to this adapter (in MB)
xfers	Total transfers to/from this adapter (in KB)

Several other disk adapter related metrics (such as the number of blocks read from and written to the adapter) are also returned. For a complete list, see the **perfstat_diskadapter_t** section in **libperfstat.h**.

The following code shows an example of how **perfstat_diskadapter** is used:

```
#include <stdio.h>
#include <stdlib.h>
#include <libperfstat.h>

int main(int argc, char* argv[]) {
   int i, ret, tot;
   perfstat_diskadapter_t *statp;
   perfstat_id_t first;

   /* check how many perfstat_diskadapter_t structures are available */
   tot = perfstat_diskadapter(NULL, NULL, sizeof(perfstat_diskadapter_t), 0);

   /* allocate enough memory for all the structures */
   statp = calloc(tot, sizeof(perfstat_diskadapter_t));

   /* set name to first interface */
   strcpy(first.name, FIRST_DISK);

   /* ask to get all the structures available in one call */
   /* return code is number of structures returned */
   ret = perfstat_diskadapter(&first, statp, sizeof(perfstat_diskadapter_t), tot);

   /* print statistics for each of the disk adapters */
   for (i = 0; i < ret; i++) {
       printf("\nStatistics for adapter : %s\n", statp[i].name);
       printf("----------------------\n");
       printf("description               : %s\n", statp[i].description);
       printf("number of disks connected : %d\n", statp[i].number);
       printf("total disk size           : %llu MB\n", statp[i].size);
       printf("total disk free space     : %llu MB\n", statp[i].free);
       printf("number of blocks read     : %llu\n", statp[i].rblks);
       printf("number of blocks written  : %llu\n", statp[i].wblks);
       }
}
```

The preceding program produces output similar to the following:

```
Statistics for adapter : scsi0
----------------------
description               : Wide/Fast-20 SCSI I/O Controller
number of disks connected : 3
```

```
total disk size          : 4296 MB
total disk free space    : 2912 MB
number of blocks read    : 411284
number of blocks written : 768256
```

perfstat_netinterface Interface

The **perfstat_netinterface** function returns a set of structures of type **perfstat_netinterface_t**, which is defined in the **libperfstat.h** file. Selected fields from the **perfstat_netinterface_t** structure include:

name	Interface name (from ODM)
description	Interface description (from ODM)
ipackets	Total number of input packets received on this network interface
opackets	Total number of output packets sent on this network interface
ierror	Total number of input errors on this network interface
oerror	Total number of output errors on this network interface

Several other network interface related metrics (such as number of bytes sent and received, type, and bitrate) are also returned. For a complete list, see the **perfstat_netinterface_t** section in the **libperfstat.h** file.

The following code shows an example of how **perfstat_netinterface**is used:

```
#include <stdio.h>
#include <stdlib.h>
#include <libperfstat.h>
#include <net/if_types.h>

char *
decode(uchar type) {

    switch(type) {

    case IFT_LOOP:
        return("loopback");

    case IFT_ISO88025:
        return("token-ring");

    case IFT_ETHER:
        return("ethernet");
    }

    return("other");
}

int main(int argc, char* argv[]) {
   int i, ret, tot;
   perfstat_netinterface_t *statp;
   perfstat_id_t first;

   /* check how many perfstat_netinterface_t structures are available */
   tot = perfstat_netinterface(NULL, NULL, sizeof(perfstat_netinterface_t), 0);

   /* allocate enough memory for all the structures */
   statp = calloc(tot, sizeof(perfstat_netinterface_t));

   /* set name to first interface */
   strcpy(first.name, FIRST_NETINTERFACE);

   /* ask to get all the structures available in one call */
   /* return code is number of structures returned */
```

```
   ret = perfstat_netinterface(&first, statp, sizeof(perfstat_netinterface_t), tot);

   /* print statistics for each of the interfaces */
   for (i = 0; i < ret; i++) {
       printf("\nStatistics for interface : %s\n", statp[i].name);
       printf("-----------------------\n");
       printf("type : %s\n", decode(statp[i].type));
       printf("\ninput statistics:\n");
       printf("number of packets : %llu\n", statp[i].ipackets);
       printf("number of errors  : %llu\n", statp[i].ierrors);
       printf("number of bytes   : %llu\n", statp[i].ibytes);
       printf("\noutput statistics:\n");
       printf("number of packets : %llu\n", statp[i].opackets);
       printf("number of bytes   : %llu\n", statp[i].obytes);
       printf("number of errors  : %llu\n", statp[i].oerrors);
       }
}
```

The preceding program produces output similar to the following:

```
Statistics for interface : tr0
-----------------------
type : token-ring

input statistics:
number of packets : 306352
number of errors  : 0
number of bytes   : 24831776

output statistics:
number of packets : 62669
number of bytes   : 11497679
number of errors  : 0

Statistics for interface : lo0
-----------------------
type : loopback

input statistics:
number of packets : 336
number of errors  : 0
number of bytes   : 20912

output statistics:
number of packets : 336
number of bytes   : 20912
number of errors  : 0
```

perfstat_protocol Interface

The **perfstat_protocol** function returns a set of structures of type **perfstat_protocol_t**, which consists of a set of unions to accommodate the different sets of fields needed for each protocol, as defined in the **libperfstat.h** file. Selected fields from the **perfstat_protocol_t** structure include:

name	protocol name: **ip**, **ip6**, **icmp**, **icmp6**, **udp**, **tcp**, **rpc**, **nfs**, **nfsv2** or **nfsv3**.
ipackets	Number of input packets received using this protocol. This field exists only for protocols **ip**, **ipv6**, **udp**, and **tcp**.
opackets	Number of output packets sent using this protocol. This field exists only for protocols **ip**, **ipv6**, **udp**, and **tcp**.
received	Number of packets received using this protocol. This field exists only for protocols **icmp** and **icmpv6**.
calls	Number of calls made to this protocol. This field exists only for protocols **rpc**, **nfs**, **nfsv2**, and **nfsv3**.

Many other network protocol related metrics are also returned. The complete set of metrics printed by **nfsstat** is returned for instance. For a complete list, see the **perfstat_protocol_t** section in the **libperfstat.h** file.

The following code shows an example of how **perfstat_protocol** is used:

```
#include <stdio.h>
#include <string.h>
#include <libperfstat.h>

int main(int argc, char* argv[]) {
    int ret, tot, retrieved = 0;
    perfstat_protocol_t pinfo;
    perfstat_id_t protid;

    /* check how many perfstat_protocol_t structures are available */
    tot = perfstat_protocol(NULL, NULL, sizeof(perfstat_protocol_t), 0);

    printf("number of protocol usage structures available : %d\n", tot);

    /* set name to first protocol */
    strcpy(protid.name, FIRST_PROTOCOL);

    /* retrieve first protocol usage information */
    ret = perfstat_protocol(&protid, &pinfo, sizeof(perfstat_protocol_t), 1);
    retrieved += ret;

    do {
       printf("\nStatistics for protocol : %s\n", pinfo.name);
       printf("-----------------------\n");

       if (!strcmp(pinfo.name,"ip")) {
           printf("number of input packets  : %llu\n", pinfo.ip.ipackets);
           printf("number of input errors   : %llu\n", pinfo.ip.ierrors);
           printf("number of output packets : %llu\n", pinfo.ip.opackets);
           printf("number of output errors  : %llu\n", pinfo.ip.oerrors);
       } else if (!strcmp(pinfo.name,"ipv6")) {
           printf("number of input packets  : %llu\n", pinfo.ipv6.ipackets);
           printf("number of input errors   : %llu\n", pinfo.ipv6.ierrors);
           printf("number of output packets : %llu\n", pinfo.ipv6.opackets);
           printf("number of output errors  : %llu\n", pinfo.ipv6.oerrors);
       } else if (!strcmp(pinfo.name,"icmp")) {
           printf("number of packets received : %llu\n", pinfo.icmp.received);
           printf("number of packets sent     : %llu\n", pinfo.icmp.sent);
           printf("number of errors           : %llu\n", pinfo.icmp.errors);
       } else if (!strcmp(pinfo.name,"icmpv6")) {
           printf("number of packets received : %llu\n", pinfo.icmpv6.received);
           printf("number of packets sent     : %llu\n", pinfo.icmpv6.sent);
           printf("number of errors           : %llu\n", pinfo.icmpv6.errors);
       } else if (!strcmp(pinfo.name,"udp")) {
           printf("number of input packets  : %llu\n", pinfo.udp.ipackets);
           printf("number of input errors   : %llu\n", pinfo.udp.ierrors);
           printf("number of output packets : %llu\n", pinfo.udp.opackets);
       } else if (!strcmp(pinfo.name,"tcp")) {
           printf("number of input packets  : %llu\n", pinfo.tcp.ipackets);
           printf("number of input errors   : %llu\n", pinfo.tcp.ierrors);
           printf("number of output packets : %llu\n", pinfo.tcp.opackets);
       } else if (!strcmp(pinfo.name,"rpc")) {
           printf("client statistics:\n");
           printf("number of connection-oriented RPC requests  : %llu\n",
                  pinfo.rpc.client.stream.calls);
           printf("number of rejected connection-oriented RPCs : %llu\n",
                  pinfo.rpc.client.stream.badcalls);
           printf("number of connectionless RPC requests       : %llu\n",
                  pinfo.rpc.client.dgram.calls);
           printf("number of rejected connectionless RPCs      : %llu\n",
                  pinfo.rpc.client.dgram.badcalls);
```

```
            printf("\nserver statistics:\n");
            printf("number of connection-oriented RPC requests  : %llu\n",
                   pinfo.rpc.server.stream.calls);
            printf("number of rejected connection-oriented RPCs : %llu\n",
                   pinfo.rpc.server.stream.badcalls);
            printf("number of connectionless RPC requests       : %llu\n",
                   pinfo.rpc.server.dgram.calls);
            printf("number of rejected connectionless RPCs      : %llu\n",
                   pinfo.rpc.server.dgram.badcalls);
        } else if (!strcmp(pinfo.name,"nfs")) {
            printf("total number of NFS client requests        : %llu\n",
                   pinfo.nfs.client.calls);
            printf("total number of NFS client failed calls    : %llu\n",
                   pinfo.nfs.client.badcalls);
            printf("total number of NFS server requests        : %llu\n",
                   pinfo.nfs.server.calls);
            printf("total number of NFS server failed calls    : %llu\n",
                   pinfo.nfs.server.badcalls);
            printf("total number of NFS version 2 server calls : %llu\n",
                   pinfo.nfs.server.public_v2);
            printf("total number of NFS version 3 server calls : %llu\n",
                   pinfo.nfs.server.public_v3);
        } else if (!strcmp(pinfo.name,"nfsv2")) {
            printf("number of NFS V2 client requests : %llu\n",
                   pinfo.nfsv2.client.calls);
            printf("number of NFS V2 server requests : %llu\n",
                   pinfo.nfsv2.server.calls);
        } else if (!strcmp(pinfo.name,"nfsv3")) {
            printf("number of NFS V3 client requests : %llu\n",
                   pinfo.nfsv3.client.calls);
            printf("number of NFS V3 server requests : %llu\n",
                   pinfo.nfsv3.server.calls);
        }

        /* make sure we stop after the last protocol */
        if (ret = strcmp(protid.name, "")) {
            printf("\nnext protocol name : %s\n", protid.name);

            /* retrieve information for next protocol */
            ret = perfstat_protocol(&protid, &pinfo, sizeof(perfstat_protocol_t), 1);
            retrieved += ret;
        }
     } while (ret == 1);

   printf("\nnumber of protocol usage structures retrieved : %d\n", retrieved);
}
```

The preceding program produces output similar to the following:

```
number of protocol usage structures available : 10

Statistics for protocol : ip
----------------------
number of input packets  : 142839
number of input errors   : 54665
number of output packets : 63974
number of output errors  : 55878

next protocol name : ipv6

Statistics for protocol : ipv6
----------------------
number of input packets  : 0
number of input errors   : 0
number of output packets : 0
number of output errors  : 0
```

```
next protocol name : icmp

Statistics for protocol : icmp
----------------------
number of packets received : 35
number of packets sent     : 1217
number of errors           : 0

next protocol name : icmpv6

Statistics for protocol : icmpv6
----------------------
number of packets received : 0
number of packets sent     : 0
number of errors           : 0

next protocol name : udp

Statistics for protocol : udp
----------------------
number of input packets  : 4316
number of input errors   : 0
number of output packets : 308

next protocol name : tcp

Statistics for protocol : tcp
----------------------
number of input packets  : 82604
number of input errors   : 0
number of output packets : 62250

next protocol name : rpc

Statistics for protocol : rpc
----------------------
client statistics:
number of connection-oriented RPC requests  : 375
number of rejected connection-oriented RPCs : 0
number of connectionless RPC requests       : 20
number of rejected connectionless RPCs      : 0

server statistics:
number of connection-oriented RPC requests  : 32
number of rejected connection-oriented RPCs : 0
number of connectionless RPC requests       : 0
number of rejected connectionless RPCs      : 0

next protocol name : nfs

Statistics for protocol : nfs
----------------------
total number of NFS client requests        : 375
total number of NFS client failed calls    : 0
total number of NFS server requests        : 32
total number of NFS server failed calls    : 0
total number of NFS version 2 server calls : 0
total number of NFS version 3 server calls : 0

next protocol name : nfsv2

Statistics for protocol : nfsv2
----------------------
number of NFS V2 client requests : 0
number of NFS V2 server requests : 0

next protocol name : nfsv3
```

```
Statistics for protocol : nfsv3
-----------------------
number of NFS V3 client requests : 375
number of NFS V3 server requests : 32

number of protocol usage structures retrieved : 10
```

perfstat_netbuffer Interface

The **perfstat_netbuffer** function returns a set of structures of type **perfstat_netbuffer_t**, which is defined in the **libperfstat.h**file. Selected fields from the **perfstat_netbuffer_t** structure include:

size	Size of the allocation (string expressing size in bytes)
inuse	Current allocation of this size
failed	Failed allocation of this size
free	Free list for this size

Several other allocation related metrics (such as high-water mark and freed) are also returned. For a complete list, see the **perfstat_netbuffer_t** section in the **libperfstat.h** file.

The following code shows an example of how **perfstat_netbuffer** is used:

```
#include <stdio.h>
#include <stdlib.h>
#include <libperfstat.h>

int main(int argc, char* argv[]) {
   int i, ret, tot;
   perfstat_netbuffer_t *statp;
   perfstat_id_t first;

   /* check how many perfstat_netbuffer_t structures are available */
   tot = perfstat_netbuffer(NULL, NULL, sizeof(perfstat_netbuffer_t), 0);

   /* allocate enough memory for all the structures */
   statp = calloc(tot, sizeof(perfstat_netbuffer_t));

   /* set name to first interface */
   strcpy(first.name, FIRST_NETBUFFER);

   /* ask to get all the structures available in one call */
   /* return code is number of structures returned */
   ret = perfstat_netbuffer(&first, statp,
                            sizeof(perfstat_netbuffer_t), tot);

   /* print info in netstat -m format */
   printf("%-12s %10s %9s %6s %9s %7s %7s %7s\n",
          "By size", "inuse", "calls", "failed",
          "delayed", "free", "hiwat", "freed");
   for (i = 0; i < ret; i++) {
       printf("%-12s %10llu %9llu %6llu %9llu %7llu %7llu %7llu\n",
           statp[i].name,
           statp[i].inuse,
           statp[i].calls,
           statp[i].delayed,
           statp[i].free,
           statp[i].failed,
           statp[i].highwatermark,
           statp[i].freed);
       }
}
```

The preceding program produces output similar to the following:

```
By size           inuse     calls failed   delayed    free   hiwat   freed
32                  199      4798      0        57       0     826       0
64                   96      8121      0        32       0     413       0
128                 110     50156      0       146       0     206       2
256                 279  20313587      0       361       0     496       0
512                 156      5298      0        12       0      51       0
1024                 38      1038      0         6       0     129       0
2048                  1      6946      0       129       0     129    1024
4096                 67    276102      0       132       0     155       0
8192                  4       123      0         4       0      12       0
16384                 1         1      0        15       0      31       0
65536                 1         1      0         0       0     512       0
```

perfstat_pagingspace Interface

The **perfstat_pagingspace** function returns a set of structures of type **perfstat_pagingspace_t**, which is defined in the **libperfstat.h** file. Selected fields from the **perfstat_pagingspace_t** structure include:

mb_size	Size of the paging space in MB
lp_size	Size of the paging space in logical partitions
mb_used	Portion of the paging space used in MB

Several other paging space related metrics (such as name, type, and active) are also returned. For a complete list, see the **perfstat_pagingspace_t** section in the **libperfstat.h** file.

The following code shows an example of how **perfstat_pagingspace** is used:

```
#include <stdio.h>
#include <stdlib.h>
#include <libperfstat.h>

int main(int argc, char agrv[]) {
    int i, ret, tot;
    perfstat_id_t first;
    perfstat_pagingspace_t *pinfo;

    tot = perfstat_pagingspace(NULL, NULL, sizeof(perfstat_pagingspace_t), 0);

    pinfo = calloc(tot,sizeof(perfstat_pagingspace_t));

    strcpy(first.name, FIRST_PAGINGSPACE);

    ret = perfstat_pagingspace(&first, pinfo, sizeof(perfstat_pagingspace_t), tot);
    for (i = 0; i < ret; i++) {
        printf("\nStatistics for paging space : %s\n", pinfo[i].name);
        printf("----------------------------\n");
        printf("type         : %s\n",
               pinfo[i].type == LV_PAGING ? "logical volume" : "NFS file");
        if (pinfo[i].type == LV_PAGING) {
         printf("volume group : %s\n", pinfo[i].lv_paging.vgname);
        }
        else {
         printf("hostname : %s\n", pinfo[i].nfs_paging.hostname);
         printf("filename : %s\n", pinfo[i].nfs_paging.filename);
        }
        printf("size (in LP) : %llu\n", pinfo[i].lp_size);
        printf("size (in MB) : %llu\n", pinfo[i].mb_size);
        printf("used (in MB) : %llu\n", pinfo[i].mb_used);
    }
}
```

The preceding program produces output similar to the following:

```
Statistics for paging space : hd6
--------------------------
type         : logical volume
volume group : rootvg
size (in LP) : 64
size (in MB) : 512
used (in MB) : 4
```

Change History

The following additions have been made to the perfstat tool:

Interface Additions

The following interfaces were added in AIX 5.2:

- perfstat_netbuffer
- perfstat_protocol
- perfstat_pagingspace
- perfstat_diskadapter
- perfstat_reset

Field Additions

The following additions have been made to the specified AIX release.

AIX Release 5.1.0.15

The following fields were added to **perfstat_cpu_total_t**:

```
u_longlong_t bread
u_longlong_t bwrite
u_longlong_t lread
u_longlong_t lwrite
u_longlong_t phread
u_longlong_t phwrite
```

Support for C++ was added in this AIX level.

Note that the AIX 4.3.3 version of **libperfstat** is synchronized with this level. No binary or source compatibility is provided between the AIX 4.3.3 version and any AIX 5.1 version prior to 5.1.0.15.

AIX Release 5.1.0.25

The following fields were added to **perfstat_cpu_t**:

```
u_longlong_t bread
u_longlong_t bwrite
u_longlong_t lread
u_longlong_t lwrite
u_longlong_t phread
u_longlong_t phwrite
```

AIX Release 5.2.0

The following fields were added to **perfstat_cpu_t**:

```
u_longlong_t iget
u_longlong_t namei
u_longlong_t dirblk
u_longlong_t msg
u_longlong_t sema
```

The **name** field which returns the logical processor name is now of the form *cpu0*, *cpu1*, ... instead of *proc0*, *proc1*, ... as it was in previous releases.

The following fields were added to **perfstat_cpu_total_t**:

```
u_longlong_t runocc
u_longlong_t swpocc
u_longlong_t iget
u_longlong_t namei
u_longlong_t dirblk
u_longlong_t msg
u_longlong_t sema
u_longlong_t rcvint
u_longlong_t xmtint
u_longlong_t mdmint
u_longlong_t tty_rawinch
u_longlong_t tty_caninch
u_longlong_t tty_rawoutch
u_longlong_t ksched
u_longlong_t koverf
u_longlong_t kexit
u_longlong_t rbread
u_longlong_t rcread
u_longlong_t rbwrt
u_longlong_t rcwrt
u_longlong_t traps
int ncpus_high
```

The following field was added to **perfstat_disk_t**:

```
char adapter[IDENTIFIER_LENGTH]
```

The following field was added to **perfstat_netinterface_t**:

```
u_longlong_t bitrate
```

The following fields were added to **perfstat_memory_total_t**:

```
u_longlong_t real_system
u_longlong_t real_user
u_longlong_t real_process
```

The following defines were added to **libperfstat.h**:

```
#define FIRST_CPU          ""
#define FIRST_DISK         ""
#define FIRST_DISKADAPTER  ""
#define FIRST_NETINTERFACE ""
#define FIRST_PAGINGSPACE  ""
#define FIRST_PROTOCOL     ""
#define FIRST_ALLOC        ""
```

Related Information

The libperfstat.h file

Chapter 7. Kernel Tuning

Beginning with AIX 5.2, you can make permanent kernel-tuning changes without having to edit any **rc** files. This is achieved by centralizing the reboot values for all tunable parameters in the **/etc/tunables/nextboot** stanza file. When a system is rebooted, the values in the **/etc/tunables/nextboot** file are automatically applied.

The following commands are used to manipulate the **nextboot** file and other files containing a set of tunable parameter values:

- The **tunsave** command is used to save values to a stanza file.
- The **tunrestore** is used to apply a file; that is, to change all tunables parameter values to those listed in a file.
- The **tuncheck** command must be used to validate a file created manually.
- The **tundefault** is available to reset tunable parameters to their default values.

The preceding commands work on both current and reboot values.

All five tuning commands (**no**, **nfso**, **vmo**, **ioo**, and **schedo**) use a common syntax and are available to directly manipulate the tunable parameter values. Available options include making permanent changes and displaying detailed help on each of the parameters the command manages.

SMIT panels and Web-based System Manager plug-ins are also available to manipulate current and reboot values for all tuning parameters, as well as the files in the **/etc/tunables** directory.

The following topics are covered in this chapter:

Migration and Compatibility

When machines are migrated to AIX 5.2 from a previous release of AIX, the tuning commands are automatically set to run in compatibility mode. Most the information in this section does not apply to compatibility mode. For more information, see Tuning Enhancements for AIX 5.2 in the *AIX 5L Version 5.2 Performance Management Guide*.

When a machine is initially installed with AIX 5.2, it is automatically set to run in the mode described in this article. The mode is controlled by the **sys0** attribute called **pre520tune**, which can be set to enable to run in compatibility mode and disable to run in AIX 5.2 mode.

To retrieve the current setting of the **pre520tune** attribute, run the following command:

```
lsattr -E -l sys0
```

To change the current setting of the **pre520tune** attribute, run the following command:

```
chdev -l sys0 -a pre520tune=enable
```

OR use SMIT or Web-based System Manager.

Tunables File Directory

Information about tunable parameter values is located in the **/etc/tunables** directory. Except for a log file created during each reboot, this directory only contains ASCII stanza files with sets of tunable parameters. These files contain **parameter=value** pairs specifying tunable parameter changes, classified in five stanzas corresponding to the five tuning commands : **schedo**, **vmo**, **ioo**, **no**, and **nfso**. Additional information about the level of AIX, when the file was created, and a user-provided description of file usage is stored in a special stanza in the file. For detailed information on the file's format, see the **tunables** file.

The main file in the tunables directory is called **nextboot**. It contains all the tunable parameter values to be applied at the next reboot. The **lastboot** file in the tunables directory contains all the tunable values that were set at the last machine reboot, a *timestamp* for the last reboot, and *checksum* information about the matching **lastboot.log** file, which is used to log any changes made, or any error messages encountered, during the last rebooting. The **lastboot** and **lastboot.log** files are set to be read-only and are owned by the root user, as are the directory and all the other files.

Users can create as many **/etc/tunables** files as needed, but only the **nextboot** file is ever automatically applied. Manually created files must be validated using the **tuncheck** command. Parameters and stanzas can be missing from a file. Only tunable parameters present in the file will be changed when the file is applied with the **tunrestore** command. Missing tunables will simply be left at their current or default values. To force resetting of a tunable to its default value with **tunrestore** (presumably to force other tunables to known values, otherwise **tundefault**, which sets all parameters to their default value, could have been used), `DEFAULT` can be specified. Specifying `DEFAULT` for a tunable in the **nextboot** file is the same as not having it listed in the file at all because the reboot tuning procedure enforces default values for missing parameters. This will guarantee to have all tunables parameters set to the values specified in the **nextboot** file after each reboot.

Tunable files can have a special stanza named **info** containing the parameters **AIX_level**, **Kernel_type** and **Last_validation**. Those parameters are automatically set to the level of AIX and to the type of kernel (UP, MP, or MP64) running when the **tuncheck** or **tunsave** is run on the file. Both commands automatically update those fields. However, the **tuncheck** command, will only update if no error was detected.

The **lastboot** file always contains values for every tunable parameters. Tunables set to their default value will be marked with the comment `DEFAULT VALUE`. The **Logfile_checksum** parameter only exists in that file and is set by the tuning reboot process (which also sets the rest of the info stanza) after closing the log file.

Tunable files can be created and modified using one of the following options:

- Using SMIT or Web-based System Manager, to modify the next reboot value for tunable parameters, or to ask to save all current values for next boot, or to ask to use an existing tunable file at the next reboot. All those actions will update the **/etc/tunables/nextboot** file. A new file in the **/etc/tunables** directory can also be created to save all current or all **nextboot** values.
- Using the tuning commands (**ioo**, **vmo**, **schedo**, **no** or **nfso**) with the **-p** or **-r** options, which will update the **/etc/tunables/nexboot** file.
- A new file in the **/etc/tunables** directory can also be created directly with an editor or copied from another machine. Running **tuncheck** [**-r** | **-p**] **-f** must then be done on that file.
- Using the **tunsave** command. To create or overwrite files in the **/etc/tunables** directory
- Using the **tunrestore -r** command. To update the **nextboot** file.

Tunable Parameters Type

All the tunable parameters manipulated by the tuning commands (**no**, **nfso**, **vmo**, **ioo**, and **schedo**) have been classified into six categories:

- **Dynamic**: if the parameter can be changed at any time
- **Static**: if the parameter can never be changed
- **Reboot**: if the parameter can only be changed during reboot
- **Bosboot**: if the parameter can only be changed by running **bosboot** and rebooting the machine
- **Mount**: if changes to the parameter are only effective for future file systems or directory mounts
- **Incremental**: if the parameter can only be incremented, except at boot time

The manual page for each of the five tuning commands contains the complete list of all the parameter manipulated by each of the commands and for each parameter, its type, range, default value, and any dependencies on other parameters.

Common Syntax for Tuning Commands

The **no**, **nfso**, **vmo**, **ioo**, and **schedo** tuning commands all support the following syntax:

```
command [-p|-r] {-o tunable[=newvalue]}
command [-p|-r] {-d tunable}
command [-p|-r] -D
command [-p|-r] -a
command -h tunable
command -L [tunable]
```

-a Displays current, reboot (when used in conjunction with **-r**) or permanent (when used in conjunction with **-p**) value for all tunable parameters, one per line in pairs tunable = value. For the permanent options, a value is displayed for a parameter only if its reboot and current values are equal. Otherwise, NONE is displayed as the value.

-d *tunable* Resets tunable to default value. If a tunable needs to be changed (that is, it is currently not set to its default value) and is of type **Bosboot** or **Reboot**, or if it is of type Incremental and has been changed from its default value, and **-r** is not used in combination, it is not changed, but a message displays instead.

-D Resets all tunables to their default value. If tunables needing to be changed are of type **Bosboot** or **Reboot**, or are of type Incremental and have been changed from their default value, and **-r** is not used in combination, they are not changed, but a message displays instead.

-h *tunable* Displays help about tunable parameter.

-o tunable[=*newvalue*] Displays the value or sets tunable to *newvalue*. If a tunable needs to be changed (the specified value different than current value), and is of type **Bosboot** or **Reboot**, or if it is of type Incremental and its current value is bigger than the specified value, and **-r** is not used in combination, it is not changed, but a message displays instead.

When **-r** is used in combination without a new value, the **nextboot** value for tunable is displayed. When **-p** is used in combination without a new value, a value is displayed only if the current and next boot values for tunable are the same. Otherwise, NONE is displayed as the value.

-p When used in combination with **-o**, **-d** or **-D**, makes changes apply to both current and reboot values; that is, turns on the updating of the **/etc/tunables/nextboot** file in addition to the updating of the current value. This flag cannot be used on **Reboot** and **Bosboot** type parameters because their current value cannot be changed.

When used with **-a** or **-o** flag without specifying a new value, values are displayed only if the current and next boot values for a parameter are the same. Otherwise, NONE is displayed as the value.

-r When used in combination with **-o**, **-d** or **-D** flags, makes changes apply to reboot values only; that is, turns on the updating of the **/etc/tunables/nextboot** file. If any parameter of type **Bosboot** is changed, the user will be prompted to run **bosboot**.

When used with **-a** or **-o** without specifying a new value, next boot values for tunables are displayed instead of current values.

-L [*tunable*] Lists the characteristics of one or all tunables, one per line, using the following format:

```
Name   Current  Default  Reboot   Minimum  Maximum  Unit  Type Dependencies
       value    value    value    value    value
--------------------------------------------------------------------------

param1 5        2        4        1        10       MB/s     I  param2
                                                               param3
Parameters types are encoded with one character using the following values:
D for Dynamic
S for Static
R for Reboot
B for Bosboot
M for Mount
I for Incremental
```

Any change (with **-o**, **-d** or **-D** flags) to a parameter of type **Mount** will result in a message displays to warn the user that the change is only effective for future mountings.

Any attempt to change (with **-o**, **-d** or **-D** flags) a parameter of type **Bosboot** or **Reboot** without **-r**, will result in an error message.

Any attempt to change (with **-o**, **-d** or **-D** flags but without **-r**) the current value of a parameter of type **Incremental** with a new value smaller than the current value, will result in an error message.

Tunable File-Manipulation Commands

The following commands can only manipulate files in the **/etc/tunables** directory. Therefore, all the file names specified are expanded to **/etc/tunables/filename**. To guarantee the consistency of their content, all the files are locked before any updates are made. The commands **tunsave**, **tuncheck** (only if successful), and **tundefault -r** all update the info stanza.

tuncheck Command

The **tuncheck** command is used to validate a file. Its syntax is as follows:

```
tuncheck [-r|-p] -f filename
```

For example, the following validates the file **/etc/tunables/mytunable** for usage on current values.

```
tuncheck -f mytunable
```

For example, the following validates the **/etc/tunables/nextboot** file for usage during reboot. Note that **-r** is the only valid option when the file to check is the **nextboot** file.

```
tuncheck -r -f nextboot
```

All parameters in the file are checked for range, and dependencies, and if a problem is detected, a message similar to: `"Parameter X is out of range"` or `"Dependency problem between parameter A and B"` is issued. The **-r** and **-p** options control the values used in dependency checking for parameters not listed in the file and the handling of proposed changes to parameters of type **Incremental**, **Bosboot**, and **Reboot**.

Except when used with the **-r** option, checking is performed on parameter of type **Incremental** to make sure the value in the file is not less than the current value. If one or more parameter of type **Bosboot** are listed in the file with a different value than its current value, the user will either be prompted to run **bosboot** (when **-r** is used) or an error message will display.

Parameters having dependencies are checked for compatible values. When one or more parameters in a set of interdependent parameters is not listed in the file being checked, their values are assumed to either be set at their current value (when the **tuncheck** command is called without **-p** or **-r**), or their default value. This is because when called without **-r**, the file is validated to be applicable on the current values, while with **-r**, it is validated to be used during reboot when parameters not listed in the file will be left at their default value. Calling this command with **-p** is the same as calling it twice; once with no argument, and once with the **-r** flag. This checks if a file can be used both immediately, and at reboot time.

Note: Users creating a file with an editor, or copying a file from another machine, must run the **tuncheck** command to validate their file.

tunrestore Command

The **tunrestore** command is used to restore all the parameters from a file. Its syntax is as follows:

```
tunrestore -R | [-r] -f filename
```

For example, the following will change the current values for all tunable parameters present in the file if ranges, dependencies, and incremental parameter rules are all satisfied.

```
tunrestore -f mytunable
```

In case of problems, only the changes possible will be made.

For example, the following will change the **reboot** values for all tunable parameters present in the file if ranges and dependencies rules are all satisfied. In other words, they will be copied to the **/etc/tunables/nextboot** file.

```
tunrestore -r -f mytunable
```

If changes to parameters of type **Bosboot** are detected, the user will be prompted to run the **bosboot** command.

The following can only be called from **/etc/inittab** and changes tunable parameters to values from the **/etc/tunables/nextboot** file.

```
tunrestore -R
```

Any problem found or change made is logged in the **/etc/tunables/lastboot.log** file. A new **/etc/tunables/lastboot** file is always created with the list of current values for all parameters.

If *filename* does not exist, an error message displays. If the **nextboot** file does not exist, an error message displays if **-r** was used. If **-R** was used, all the tuning parameters of a type other than **Bosboot** will be set to their default value, and a **nextboot** file containing only an info stanza will be created. A warning will also be logged in the **lastboot.log** file.

Except when **-r** is used, parameters requiring a call to **bosboot** and a **reboot** are not changed, but an error message is displayed to indicate they could not be changed. When **-r** is used, if any parameter of type **Bosboot** needs to be changed, the user will be prompted to run **bosboot**. Parameters missing from the file are simply left unchanged, except when **-R** is used, in which case missing parameters are set to their default values. If the file contains multiple entries for a parameter, only the first entry will be applied, and a warning will be displayed or logged (if called with **-R**).

tunsave Command

The **tunsave** command is used to save current tunable parameter values into a file. Its syntax is as follows:

```
tunsave [-a|-A] -f|-F filename
```

For example, the following saves all the current tunable parameter values different from their default into the file **/etc/tunables/mytunable**.

```
tunsave -f mytunable
```

If the file already exists, an error message is printed instead. Option **-F** must be used to overwrite an existing file.

For example, the following saves all the current tunable parameter values different from their default into the file **/etc/tunables/nextboot**.

```
tunsave -f nextboot
```

If necessary, **tunsave** will prompt the user to run **bosboot**.

For example, the following saves all current tunable parameters values (including parameters which value is their default value) into the **mytunable** file.

```
tunsave -A -f mytunable
```

This allows you to save the current setting. This setting can be reproduced at a later time, even if the default values have changed (default values can change when the file is used on another machine or when running another version of AIX).

For example, the following saves all current tunable parameters values into the **mytunable** file.

```
tunsave -a -f mytunable
```

For the parameters that are set to default values, a line using the keyword `DEFAULT` will be put in the file. This essentially saves only the current changed values, while forcing all the other parameters to their default values. This allows you to return to a known setup later using **tunrestore**.

tundefault Command

The **tundefault** command is used to force all tuning parameters to be reset to their default value. The **-p** flag makes changes permanent, while the **-r** flag defers changes until the next reboot. The command syntax is as follows:

```
tundefault [-p|-r]
```

For example, the following resets all tunable parameters to their default value, except the parameters of type **Bosboot** and **Reboot**, and parameters of type **Incremental** set at values bigger than their default value.

```
tundefault
```

Error messages will be displayed for any parameter change that is not permitted.

For example, the following resets all the tunable parameters to their default value. It also updates the **/etc/tunables/nextboot** file, and if necessary, offers to run **bosboot**, and displays a message warning that rebooting is needed for all the changes to be effective.

```
tundefault -p
```

This command permanently resets *all* tunable parameters to their default values, returning the system to a consistent state and making sure the state is preserved after the next reboot.

For example, the following clears all the command stanzas in the **/etc/tunables/nextboot** file, and proposes **bosboot** if necessary.

```
tundefault -r
```

Initial setup

Installing the **bos.perf.tune** fileset automatically creates an initial **/etc/tunables/nextboot** file and adds the following line at the end of the **/etc/inittab** file:

```
tunable:23456789:once:/usr/bin/tunrestore -R
```

This entry sets the **reboot** value of all tunable parameters to their default. For more information about migration from a previous version of AIX and the compatibility mode automatically setup in case of migration, read ″Introduction to AIX 5.2 Tunable Parameter Settings″ in the *AIX 5L Version 5.2 Performance Management Guide*.

Reboot Tuning Procedure

Parameters of type **Bosboot** are set by the **bosboot** command, which retrieves their values from the **nextboot** file when creating a new boot image. Parameters of type **Reboot** are set during the reboot process by the appropriate configuration methods, which also retrieve the necessary values from the **nextboot** file. In both cases, if there is no **nextboot** file, the parameters will be set to their default values. All other parameters are set using the following process:

1. When **tunrestore -R** is called, any tunable changed from its default value is logged in the **lastboot.log** file. The parameters of type **Reboot** and **Bosboot** present in the **nextboot** file, and which should already have been changed by the time **tunrestore -R** is called, will be checked against the value in the file, and any difference will also be logged.
2. The **lastboot** file will record all the tunable parameter settings, including default values, which will be flagged as explained above, and the **AIX_level, Kernel_type**, **Last_validation**, and **Logfile_checksum** fields will be set appropriately.
3. If there is no **/etc/tunables/nextboot** file, all tunable parameters, except those of type **Bosboot**, will be set to their default value, a **nextboot** file with only an info stanza will be created, and the following warning: ″cannot access the /etc/tunables/nextboot file″ will be printed in the log file. The **lastboot** file will be created as described above.
4. If the desired value for a parameter is found to be out of range, the parameter will be left to its default value, and a message similar to the following: ″Parameter A could not be set to X, which is out of range, and was left to its current value (Y) instead″ will be printed in the log file. Similarly, if a set of interdependent parameters have values incompatible with each other, they will all be left at their default values and a message similar to the following: ″Dependent parameter A, B and C could not be set to X, Y and Z because those values are incompatible with each other. Instead, they were left to their current values (T, U and V)″ will be printed in the log file.

   Both these conditions could exist if a user modified the **/etc/tunables/nextboot** file with an editor or copied it from another machine, possibly running a different version of AIX with different valid ranges, and did not run **tuncheck -r -f** on the file. Alternatively, **tuncheck -r -f** prompted the user to run **bosboot**, but this was not done.

Recovery Procedure

If the machine becomes unstable with a given **nextboot** file, users should put the system into maintenance mode, make sure the **sys0 pre520tune** attribute is set to disable, delete the **nextboot** file, run the **bosboot** command, move the tunable line to be last in **/etc/inittab** and reboot. This action will guarantee that all tunables are set to their default value.

Kernel Tuning Using the SMIT Interface

To start the SMIT panels that manage AIX kernel tuning parameters, use the SMIT fast path **smitty tuning**. The following is a view of the tuning panel:

```
   Tuning Kernel Parameters

Save/Restore All Kernel & Network Parameters
Tuning Scheduler and Memory Load Control Parameters
Tuning Virtual Memory Manager Parameters
Tuning Network Parameters
Tuning NFS Parameters
Tuning I/O Parameters
```

Select **Save/Restore All Kernel & Network Parameters** to manipulate all tuning parameter values at the same time. To individually change tuning parameters managed by one of the tuning commands, select any of the other lines.

Global Manipulation of Tuning Parameters

The main panel to manipulate all tunable parameters by sets looks similar to the following:

```
     Save/Restore All Kernel Tuning Parameters

View Last Boot Parameters
View Last Boot Log File

Save All Current Parameters for Next Boot
Save All Current Parameters
Restore All Current Parameters from Last Boot Values
Restore All Current Parameters from Saved Values
Reset All Current Parameters To Default Value

Save All Next Boot Parameters
Restore All Next Boot Parameters from Last Boot Values
Restore All Next Boot Parameters from Saved Values
Reset All Next Boot Parameters To Default Value
```

Each of the options in this panel are explained in the following sections.

1. View Last Boot Parameters
   All last boot parameters are listed stanza by stanza, retrieved from the **/etc/tunables/lastboot** file.
2. View Last Boot Log File
   Displays the content of the file **/etc/tunables/lastboot.log**.
3. Save All Current Parameters for Next Boot

   ```
      Save All Current Kernel Tuning Parameters for Next Boot

   ARE YOU SURE ?
   ```

   After selecting **yes** and pressing **ENTER**, all the current tuning parameter values are saved in the **/etc/tunables/nextboot** file. **Bosboot** will be offered if necessary.
4. Save All Current Parameters

```
    Save All Current Kernel Tuning Parameters

File name                                       []
Description                                     []
```

Type or select values for the two entry fields:

- **File name**: F4 will show the list of existing files. This is the list of all files in the **/etc/tunables** directory except the files **nextboot**, **lastboot** and **lastboot.log** which all have special purposes. File names entered cannot be any of the above three reserved names.
- **Description**: This field will be written in the info stanza of the selected file.

After pressing **ENTER**, all of the current tuning parameter values will be saved in the selected stanza file of the **/etc/tunables** directory.

5. Restore All Current Parameters from Last Boot Values

```
    Restore All Current Parameters from Last Boot Values

ARE YOU SURE ?
```

After selecting **yes** and pressing **ENTER**, all the tuning parameters will be set to values from the **/etc/tunables/lastboot** file. Error messages will be displayed if any parameter of type **Bosboot** or **Reboot** would need to be changed, which can only be done when changing reboot values.

6. Restore All Current Parameters from Saved Values

```
    Restore Saved Kernel Tuning Parameters

Move cursor to desired item and press Enter.

  mytunablefile    Description field of mytunable file
  tun1             Description field of lastweek file
```

A select menu shows existing files in the **/etc/tunables** directory, except the files **nextboot**, **lastboot** and **lastboot.log** which all have special purposes.
After pressing **ENTER**, the parameters present in the selected file in the **/etc/tunables** directory will be set to the value listed if possible. Error messages will be displayed if any parameter of type **Bosboot** or **Reboot** would need to be changed, which can't be done on the current values. Error messages will also be displayed for any parameter of type **Incremental** when the value in the file is smaller than the current value, and for out of range and incompatible values present in the file. All possible changes will be made.

7. Reset All Current Parameters To Default Value

```
    Reset All Current Kernel Tuning Parameters To Default Value

ARE YOU SURE ?
```

After pressing **ENTER**, each tunable parameter will be reset to its default value. Parameters of type **Bosboot** and **Reboot**, are never changed, but error messages are displayed if they should have been changed to get back to their default values.

8. Save All Next Boot Parameters

```
    Save All Next Boot Kernel Tuning Parameters

File name                                   []
```

Type or a select values for the entry field. F4 will show the list of existing files. This is the list of all files in the **/etc/tunables** directory except the files **nextboot**, **lastboot** and **lastboot.log** which all have special purposes. File names entered cannot be any of those three reserved names.
After pressing **ENTER**, the **nextboot** file, is copied to the specified **/etc/tunables** file if it can be successfully **tuncheck**ed.

9. Restore All Next Boot Parameters from Last Boot Values

```
    Restore All Next Boot Kernel Tuning Parameters from Last Boot Values

ARE YOU SURE ?
```

After selecting **yes** and pressing **ENTER**, all values from the **lastboot** file will be copied to the **nextboot** file. If necessary, the user will be prompted to run **bosboot**, and warned that for all the changes to be effective, the machine must be rebooted.

10. Restore All Next Boot Parameters from Saved Values

```
    Restore All Next Boot Kernel Tuning Parameters from Saved Values

Move cursor to desired item and press Enter.

  mytunablefile    Description field of mytunablefile file
  tun1             Description field of tun1 file
```

A select menu shows existing files in the **/etc/tunables** directory, except the files **nextboot**, **lastboot** and **lastboot.log** which all have special purposes.
After selecting a file and pressing **ENTER**, all values from the selected file will be copied to the **nextboot** file, if the file was successfully **tuncheck**ed first. If necessary, the user will be prompted to run **bosboot**, and warned that for all the changes to be effective, rebooting the machine is necessary.

11. Reset All Next Boot Parameters To Default Value

```
    Reset All Next Boot Kernel Tuning Parameters To Default Value

ARE YOU SURE ?
```

After hitting **ENTER**, the **/etc/tunables/nextboot** file will be cleared. If necessary **bosboot** will be proposed and a message indicating that a reboot is needed will be displayed.

Changing individual parameters managed by a tuning command

All the panels for all five commands behave the same way. In the following sections, we will use the example of the Scheduler and Memory Load Control (i.e. **schedo**) panels to explain the behavior. Here is the main panel to manipulate parameters managed by the **schedo** command:

```
    Tuning Scheduler and Memory Load Control Parameters

List All Characteristics of Current Parameters
Change / Show Current Parameters
Change / Show Parameters for next boot
Save Current Parameters for Next Boot
Reset Current Parameters to Default value
Reset Next Boot Parameters To Default Value
```

Interaction between parameter types and the different SMIT sub-panels

The following table contains the different parameters and how they relate to the different SMIT sub-panels:

List All Characteristics of Current Parameters	Lists current, default, reboot, limit values, unit, type and dependencies. This is the output of a tuning command called with the **-L** option.
Change / Show Current Parameters	Displays and changes current parameter value, except for parameter of type Static, Bosboot and Reboot which are displayed without surrounding square brackets to indicate that they cannot be changed.
Change / Show Parameters for Next Boot	Displays values from and rewrite updated values to the **nextboot** file. If necessary, **bosboot** will be proposed. Only parameters of type **Static** cannot be changed (no brackets around their value).
Save Current Parameters for Next Boot	Writes current parameters in the **nextboot** file, **bosboot** will be proposed if any parameter of type **Bosboot** was changed.
Reset Current Parameters to Default value	Resets current parameters to default values, except those which need a **bosboot** plus reboot or a reboot (B and R type).
Reset Next Boot Parameters to Default value	Clears values in **nextboot** file, and propose **bosboot** if any parameter of type **Bosboot** was different from its default value.

Each of the sub-panels behavior is explained in the following sections:

1. List All Characteristics of Tuning Parameters
   The output of **schedo -L** is displayed.
2. Change/Show Current Scheduler and Memory Load Control Parameters

```
Change / Show Current Scheduler and Memory Load Control Parameters

                                    [Entry Field]


  affinity_lim                          [7]
  idle_migration_barrier                [4]
  fixed_pri_global                      [0]
  maxspin                               [1]
  pacefork                              [10]
  sched_D                               [16]
  sched_R                               [16]
  timeslice                             [1]
  %usDelta                              [100]
  v_exempt_secs                         [2]
  v_min_process                         [2]
  v_repage_hi                           [2]
  v_repage_proc                         [6]
  v_sec_wait                            [4]
```

This panel is initialized with the current **schedo** values (output from the **schedo -a** command). Any parameter of type **Bosboot**, **Reboot** or **Static** is displayed with no surrounding square bracket indicating that it cannot be changed.
From the F4 list, type or select values for the entry fields corresponding to parameters to be changed. Clearing a value results in resetting the parameter to its default value. The F4 list also shows minimum, maximum, and default values, the unit of the parameter and its type. Selecting F1 displays the help associated with the selected parameter. The text displayed will be identical to what is displayed by the tuning commands when called with the **-h** option.
Press **ENTER** after making all the desired changes. Doing so will launch the **schedo** command to make the changes. Any error message generated by the command, for values out of range, incompatible values, or lower values for parameter of type **Incremental**, will be displayed to the user.

3. Change / Show Scheduler and Memory Load Control Parameters for next boot

```
Change / Show Scheduler and Memory Load Control Parameters for next boot

                                    [Entry Field]

  affinity_lim                          [7]
  idle_migration_barrier                [4]
  fixed_pri_global                      [0]
  maxpin                                [1]
  pacefork                              [10]
  sched_D                               [16]
  sched_R                               [16]
  timeslice                             [1]
  %usDelta                              [100]
  v_exempt_secs                         [2]
  v_min_process                         [2]
  v_repage_hi                           [2]
  v_repage_proc                         [6]
  v_sec_wait                            [4]
```

This panel is similar to the previous panel, in that, any parameter value can be changed except for parameters of type **Static**. It is initialized with the values listed in the **/etc/tunables/nextboot** file, completed with default values for the parameter not listed in the file.
Type or select (from the F4 list) values for the entry field corresponding to the parameters to be changed. Clearing a value results in resetting the parameter to its default value. The F4 list also shows minimum, maximum, and default values, the unit of the parameter and its type. Pressing F1 displays the help associated with the selected parameter. The text displayed will be identical to what is displayed by the tuning commands when called with the **-h** option.
Press **ENTER** after making all desired changes. Doing so will result in the**/etc/tunables/nextboot** file

being updated with the values modified in the panel, except for out of range, and incompatible values for which an error message will be displayed instead. If necessary, the user will be prompted to run **bosboot**.

4. Save Current Scheduler and Memory Load Control Parameters for Next Boot

```
    Save Current Scheduler and Memory Load Control Parameters for Next Boot

ARE YOU SURE ?
```

After pressing **ENTER** on this panel, all the current **schedo** parameter values will be saved in the **/etc/tunables/nextboot** file . If any parameter of type **Bosboot** needs to be changed, the user will be prompted to run **bosboot**.

5. Reset Current Scheduler and Memory Load Control Parameters to Default Values

```
    Reset Current Scheduler and Memory Load Control Parameters to Default Value

 ARE YOU SURE ?
```

After selecting **yes** and pressing **ENTER** on this panel, all the tuning parameters managed by the **schedo** command will be reset to their default value. If any parameter of type **Incremental**, **Bosboot** or **Reboot** should have been changed, and error message will be displayed instead.

6. Reset Scheduler and Memory Load Control Next Boot Parameters To Default Values

```
    Reset Next Boot Parameters To Default Value

 ARE YOU SURE ?
```

After pressing **ENTER**, the **schedo** stanza in the **/etc/tunables/nextboot** file will be cleared. This will defer changes until next reboot. If necessary, **bosboot** will be proposed.

Kernel Tuning using the Performance Plug-In for Web-based System Manager

AIX kernel tuning parameters can be managed using the Web-based System Manager System Tuning Plug-in, which is a sub-plugin of the Web-based System Manager2000 Performance plug-in. The Performance Plug-in is available from the Web-based System Manager main console which looks similar to the following:

Figure 28. Performance Plug-in shown in Web-based System Manager main console

The Performance plug-in is organized into the following sub-plugins:

- Performance Monitoring plug-in
- System Tuning plug-in

The Performance Monitoring sub-plugin gives access to a variety of performance-monitoring and report-generation tools. The System Tuning sub-plugin consists of CPU, Memory, Disk I/O, and Network I/O sub-plugins, which present tuning tables from which AIX tuning parameters can be visualized and changed.

The Navigation Area for the System Tuning plug-in contains three levels of sub-plugins as seen in the following:

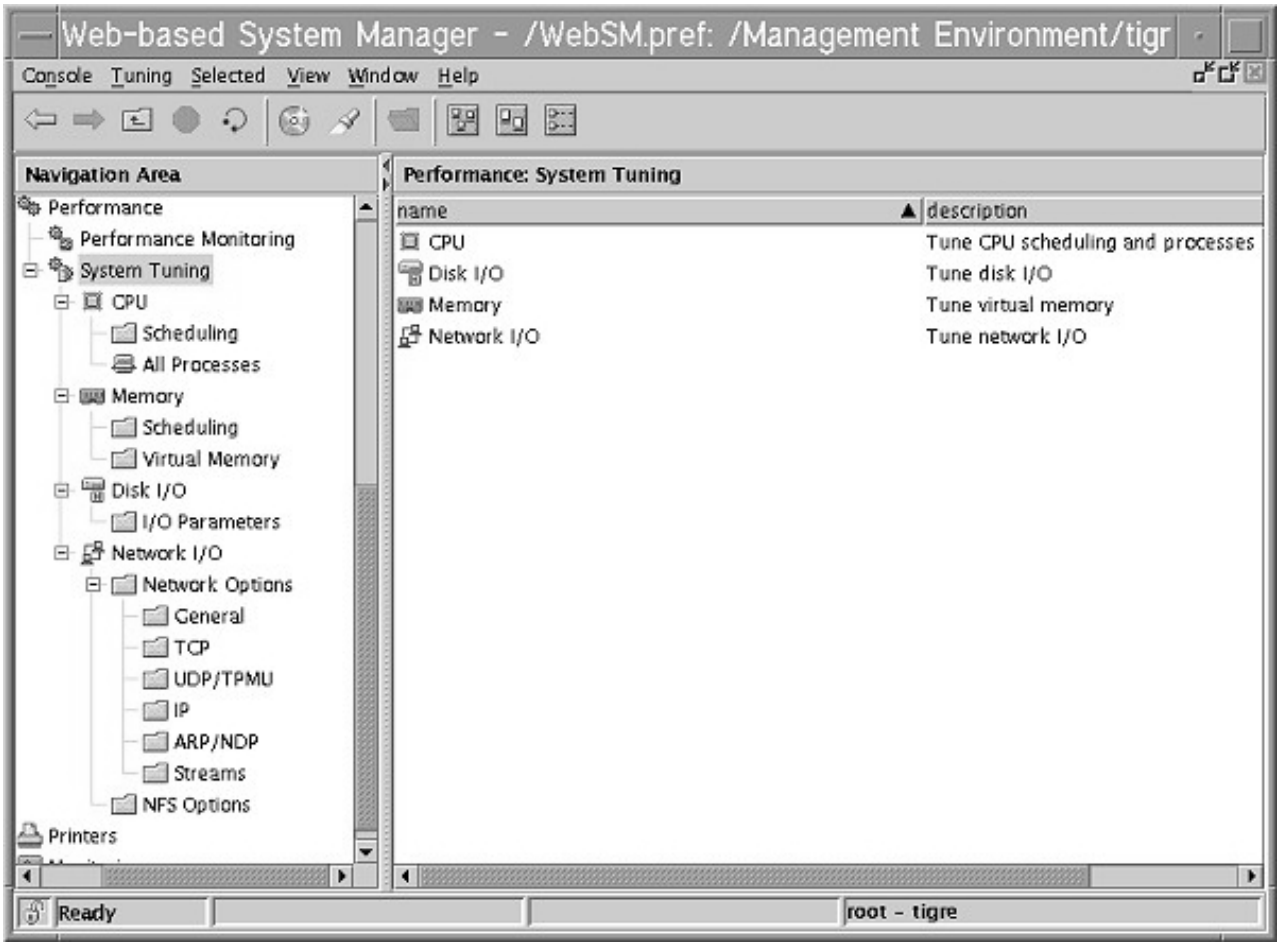


Figure 29. System Tuning plug-in Performance window

These intermediate levels represent tuning resources. They are further split into sub-plugins but have no specific actions associated with them and only exist to group access to tunable parameters in a logical way. Actions on tunable parameters can be applied at the following levels:

System-Tuning level
Global actions applicable to all tunable parameters are provided at this level.

Leaf Levels
Leaves are represented by a folder icon (see navigation area in Figure 29). When selecting a leaf, a tuning table is displayed in the content area. A table represents a logical group of tunable parameters, all managed by one of the tunable commands (**schedo**, **vmo**, **ioo**, **no**, and **nfso**). Specific actions provided at this level apply only to the tunable parameters displayed in the current table.

The **CPU/All Processes** sub-plugin is a link to the **All Processes** sub-plugin of the Processes application. Its purpose is not to manipulate tuning parameters and will not be discussed.

Global Actions on Tunable Parameters

Only the Web-based System Manager **Tuning** menu has specific actions associated with it.

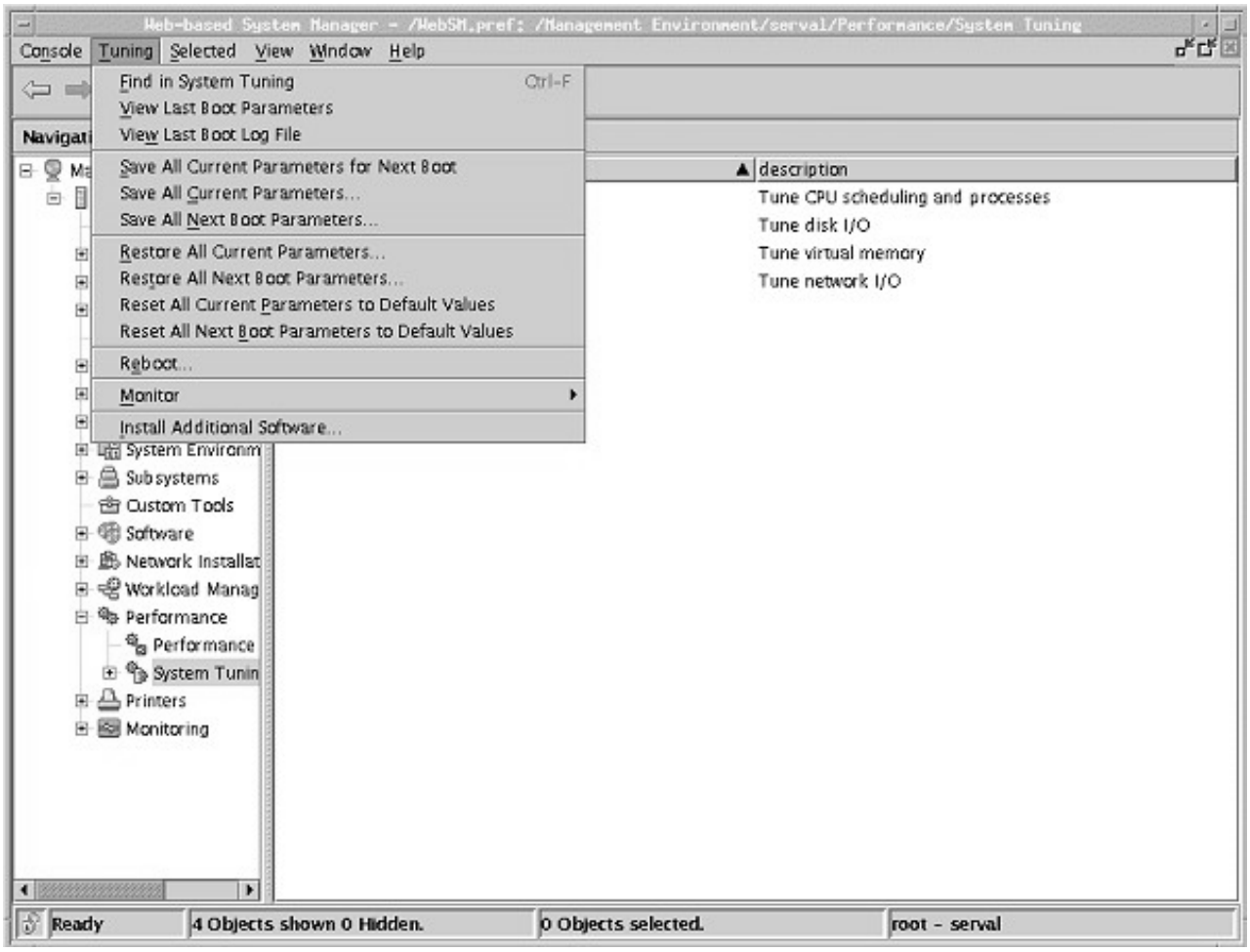


Figure 30. Web-based System Manager Tuning menu

The specific actions available at this level are global, in that they apply to all the performance tunable parameters.

1. **View Last Boot Parameters**
   This action displays the **/etc/tunables/lastboot** file in an open working dialog.
2. **View Last Boot Log File**
   This action displays the **/etc/tunables/lastboot.log** file in an open working dialog.
3. **Save All Current Parameters for Next Boot**
   The Save All Current Parameters warning dialog is opened.

Figure 31. Save All Current Parameters for next boot dialog

After clicking **Yes**, all the current tuning parameter values will be saved in the **/etc/tunables/nextboot** file. **Bosboot** will be offered if necessary.

4. **Save All Current Parameters**
   The Save All Current Parameters dialog with a Filename field and a Description field is opened.

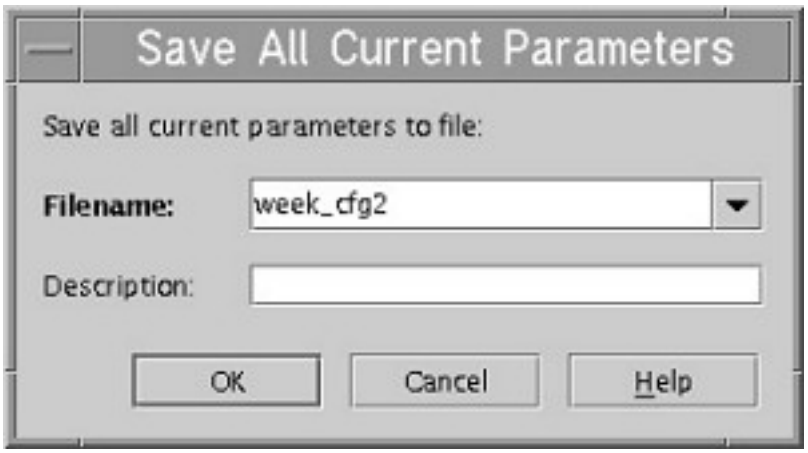


Figure 32. Save All Current Parameters to file dialog

The **Filename** editable combobox, lists all the tunable files present in the **/etc/tunables** directory, except the **nextboot**, **lastboot** and **lastboot.log** files, which all have special purposes. If no file is present, the combobox list is empty. The user can choose an existing file, or create a new file by entering a new name. File names entered cannot be any of the three reserved names. The **Description** field will be written in the info stanza of the selected file. After clicking **OK**, all the current tuning parameter values will be saved in the selected file in the **/etc/tunables** directory.

5. **Save All Next Boot Parameters**
   This action opens an editable combobox which lists all the tunable files present in the **/etc/tunables**

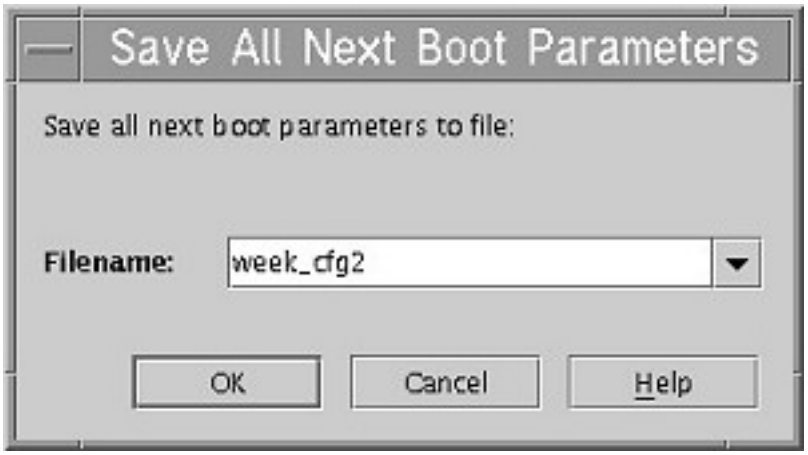


Figure 33. Save All Next Boot Parameters to file dialog

directory, except the **nextboot**, **lastboot** and **lastboot.log** files, which all have special purposes. If no file is present, the combobox list is empty. The user can choose an existing file, or create a new file by entering a new name. File names entered cannot be any of the three reserved names. After clicking **OK**, the **nextboot** file, is copied to the specified **/etc/tunables** file it it can be successfully checked using the **tuncheck** command.

6. **Restore All Current Parameters**
   This action opens an editable combobox showing the list of all existing files in the **/etc/tunables** directory, except the files **nextboot**, and **lastboot.log** which have special purposes.

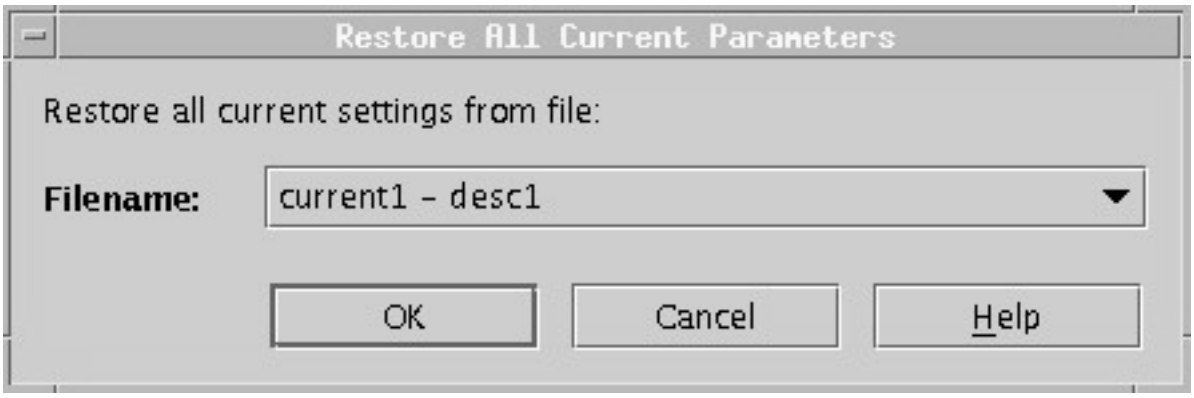


Figure 34. Restore All Current Parameters dialog

The user selects the file to use for restoring the current values of tuning parameters. The **lastboot** file is proposed as the default (first element of the combo list). Files can have a description which is displayed after the name in the combobox items, separated from the file name by a dash character. After clicking OK, the parameters present in the selected file in the **/etc/tunables** directory will be set to the value listed if possible. Error messages will be displayed if any parameter of type **Bosboot** or **Reboot** would need to be changed, which cannot be done on the current values. Error messages will also be displayed for any parameter of type **Incremental** when the value in the file is smaller than the current value, and for out of range and incompatible values present in the file. All possible changes will be made.

7. **Restore All Next Boot Parameters**
   A combobox is opened to display the list of all existing files in the **/etc/tunables** directory, except the files **nextboot**, and **lastboot.log** which have special purposes.

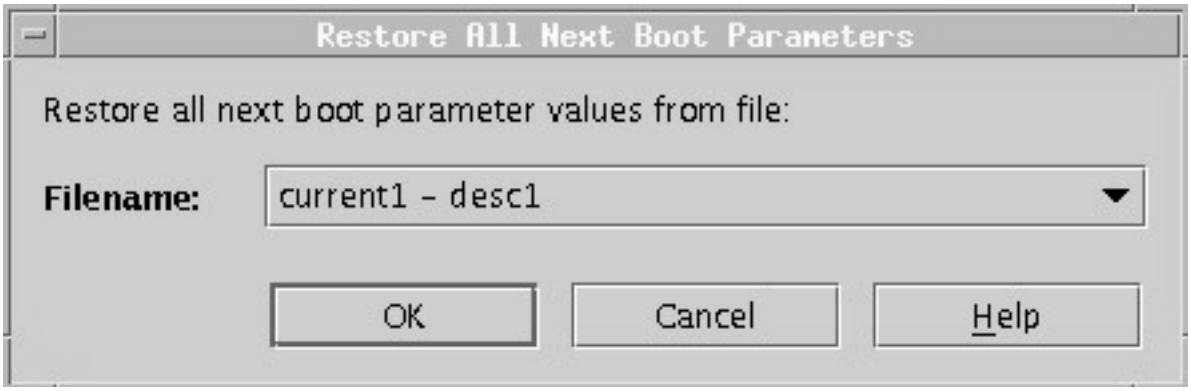


Figure 35. Restore All Next Boot Parameters dialog

The user selects the file to use for restoring the **nextboot** values of tuning parameters. The **lastboot** file is proposed as the default (first element of the combo list). Files can have a description which is displayed after the name in the combobox items, separated from the file name by a dash character. After clicking **OK**, all values from the selected file will be copied to the **/etc/tunables/nextboot** file. Incompatible dependent parameter values or out of range values will not be copied to the file (this could happen if the file selected was not previously **tuncheck**ed). Error messages will be displayed instead. If necessary, the user will be prompted to run **bosboot**, and warned that for all the changes to be effective, rebooting the machine is necessary.

8. **Reset All Current Parameters to Default Values**
   A warning dialog is opened and after clicking **Yes**, a working dialog is displayed. Each tunable parameter is reset to its default value. Parameters of type **Incremental**, **Bosboot** and **Reboot**, are never changed, but error messages are displayed if they should have been changed to revert to default values.
9. **Reset All Next Boot Parameters to Default Values**
   A warning dialog is opened and after clicking **Yes**, an interactive working dialog is displayed and the **/etc/tunables/nextboot** file is cleared. If necessary **bosboot** will be proposed and a message indicating that a reboot is needed will be displayed.

Using Tuning Tables to Change Individual Parameter Values

Each tuning table in the content area has the same structure. It allows all the characteristics of the tunable parameters to be viewed at a glance. The table is editable using the **New Value** column. Each cell in the **New Value** column, is an editable combobox, with only one predefined value of **Default**, for the capture of new value for a parameter. Data entered in the New Value column is validated when pressing **ENTER**.

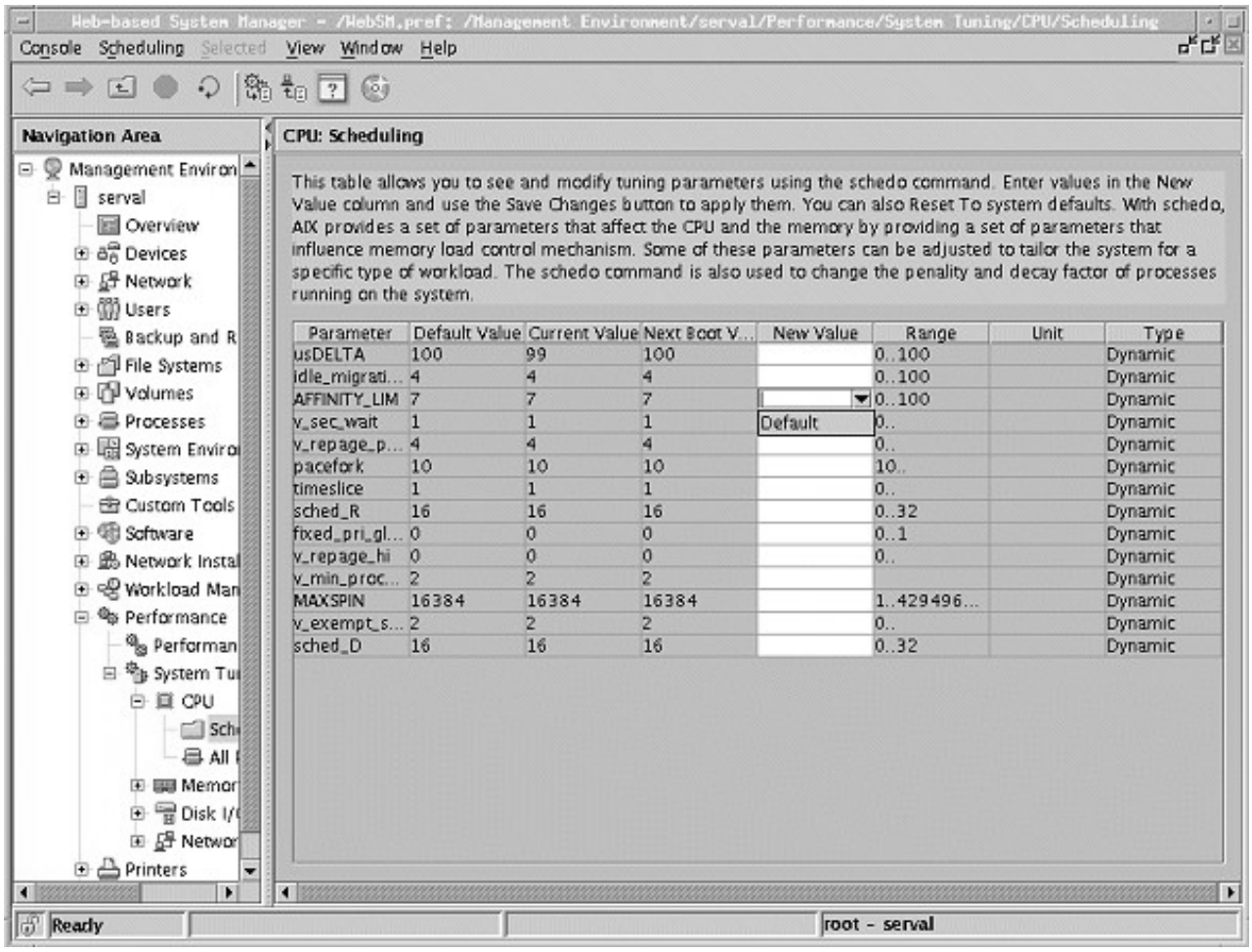


Figure 36. CPU Scheduling window

The parameters are grouped as they are in the SMIT panels with two small exceptions. First, the Network related parameters are all presented in one SMIT panel, but subdivided in six sections. The Web-based System Manager interface uses six separate tables instead.

Lastly, the parameters managed by the **schedo** command are available from two sub-plugins: CPU/scheduling and memory/scheduling.

Actions allowed vary according to parameter types:

- Static parameters do not have an editable cell in the **New Value** column.
- New values for Dynamic parameters can be applied now or saved for next boot.
- New values for **Reboot** parameters can only be saved for next boot.
- New values for **Bosboot** parameters can only be saved for next boot, and users are prompted to run bosboot.
- New values for **Mount** parameters can be applied now or saved for next boot, but when applied immediately, a warning will be displayed to tell the user that changes will only be effective for future file systems or directory mountings.
- New values for **Incremental** parameters can be applied now or saved for next boot. If applied now, they will only be accepted if the new value is bigger than the current value.

The following section explains in detail the behavior of the tables.

Tunable Tables Actions

The actions available for each tunable table are **Save Changes**, **Save Current Parameters for Next Boot**, **Reset Parameters to System Default**, **Parameter Details**, and **Monitor**. The **Monitor** action enables related monitoring tools to start from each of the plug-ins and is not discussed in this section.

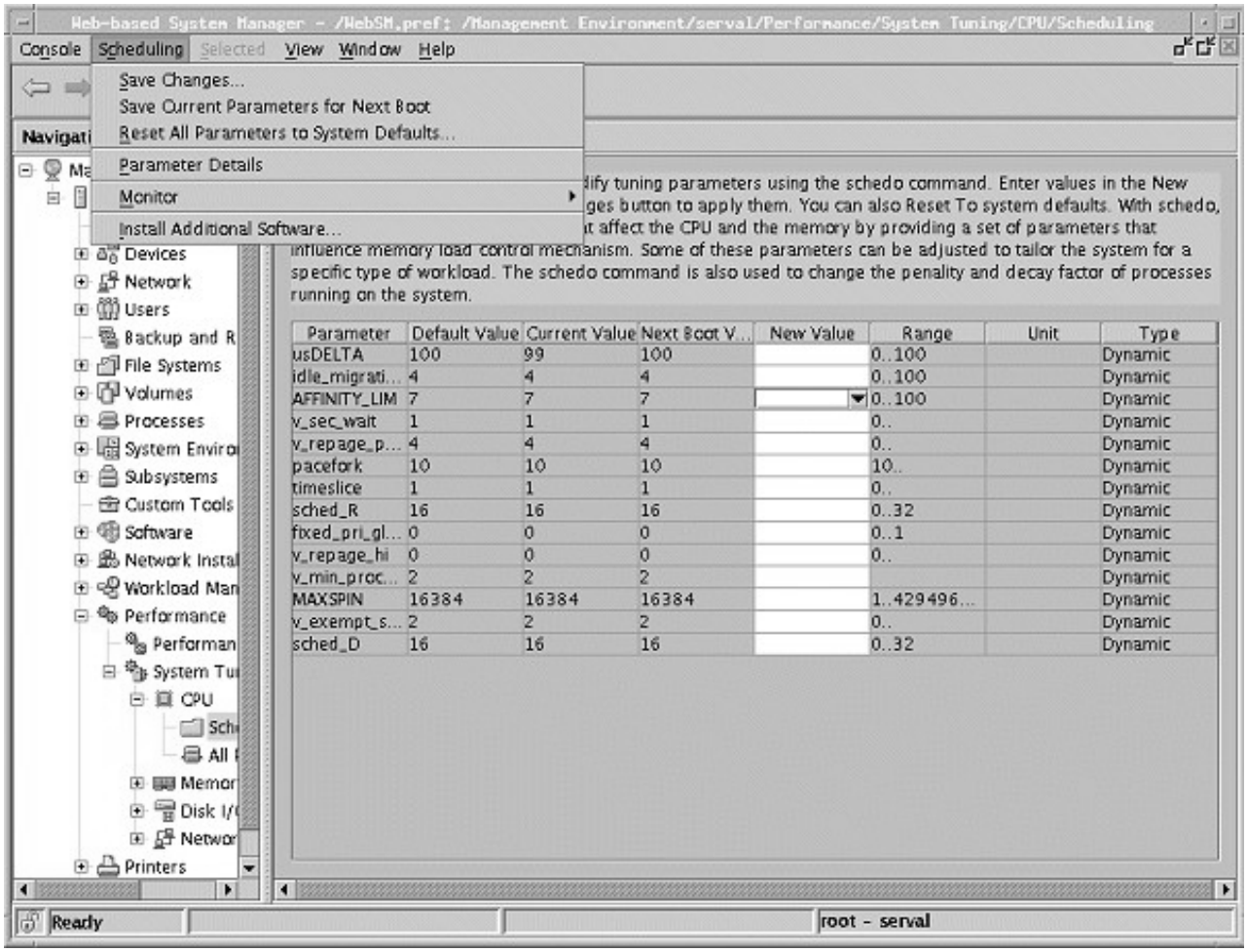


Figure 37. Tables Menus window

1. **Save Changes**
   This option opens a dialog allowing the saving of new values for the parameters listed in the table. The two options are:

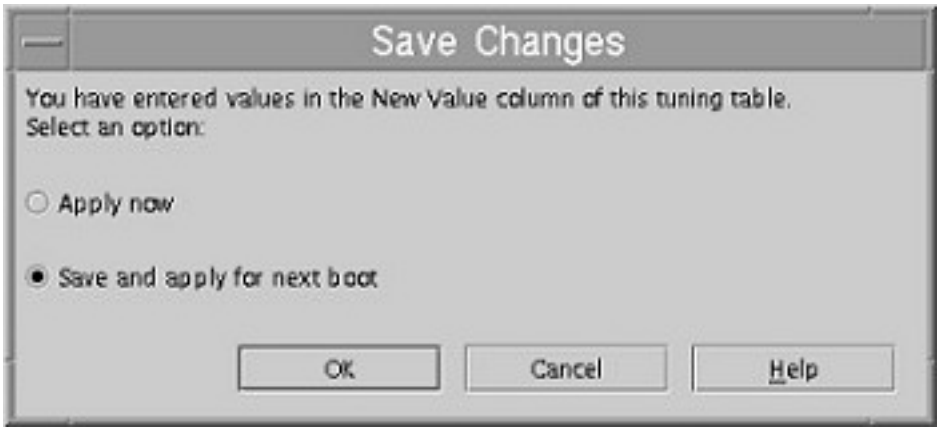


Figure 38. Save Changes dialog

   - Selecting **Apply Now** and clicking **OK**, launches the tuning command corresponding to the parameters shown in the table to make all the desired changes. Selecting Default in the combobox as the new value resets the parameter to its default value. If any parameter of type **Bosboot** or **Reboot** has a new value, or a parameter of type **Incremental** has a new value smaller that its current value, an error message will be displayed. If incompatible dependent parameter values or out of range values have been entered, an error message will also be displayed. All the acceptable changes will be made.
   - Selecting **Save and apply for next boot** and clicking **OK**, writes the desired changes to the **/etc/tunables/nextboot** file. If necessary, the user will be prompted to run **bosboot**. If incompatible dependent parameter values or out of range values have been entered, an error message will be displayed, and those parameter values will not be copied to the **nextboot** file.

2. **Save Current Parameters for Next Boot**
   A warning dialog is opened.

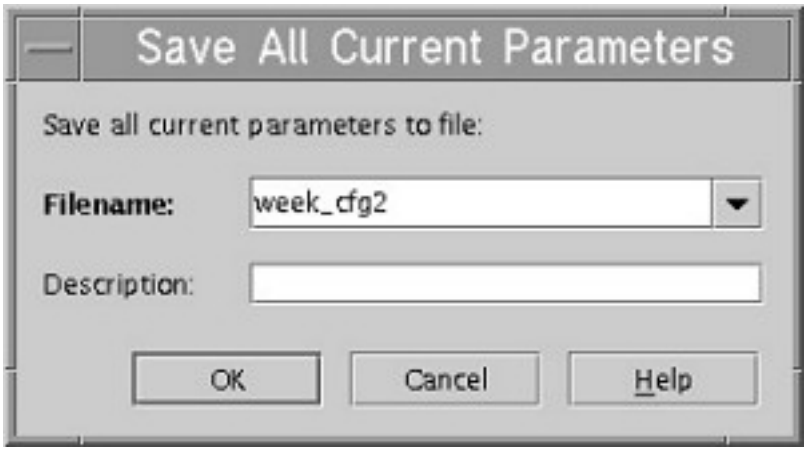


Figure 39. Save All Current Parameters to file dialog

   After clicking **Yes**, all the current parameter values listed in the table will be saved in the **/etc/tunables/nextboot** file. If any parameter of type **Bosboot** needs to be changed, the user will be prompted to run **bosboot**.

3. **Reset Parameters to System Default**
   This dialog allows resetting of current or next boot values for all the parameters listed in the table to their default value. Two options are available:

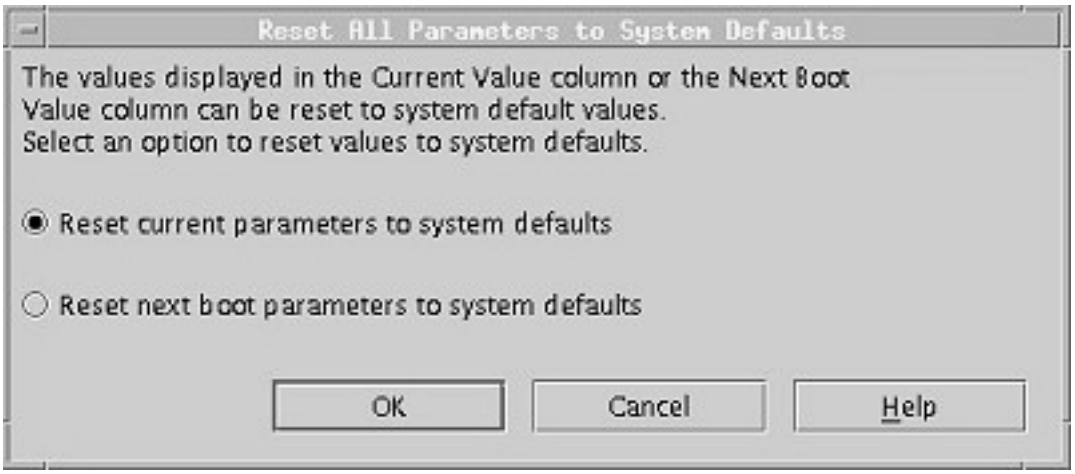


Figure 40. Reset All Parameters to System Defaults dialog

- Selecting **Reset current parameters to system default** and clicking **OK**, will reset all the tuning parameters listed in the table to their default value. If any parameter of type **Incremental**, **Bosboot** or **Reboot** should have been changed, an error message will be displayed and the parameter will not be changed.
- Selecting **Reset next boot parameters to system default** and clicking **OK** deletes the parameter listed in the table from the **/etc/tunables/nextboot** file. This action will defer changes until next reboot. If necessary, **bosboot** will be proposed.

Parameter Details

Clicking on **Parameter Details** in the toolbar or selecting the equivalent menu item, followed by a click on a parameter in the table will display the help information available in a help dialog..

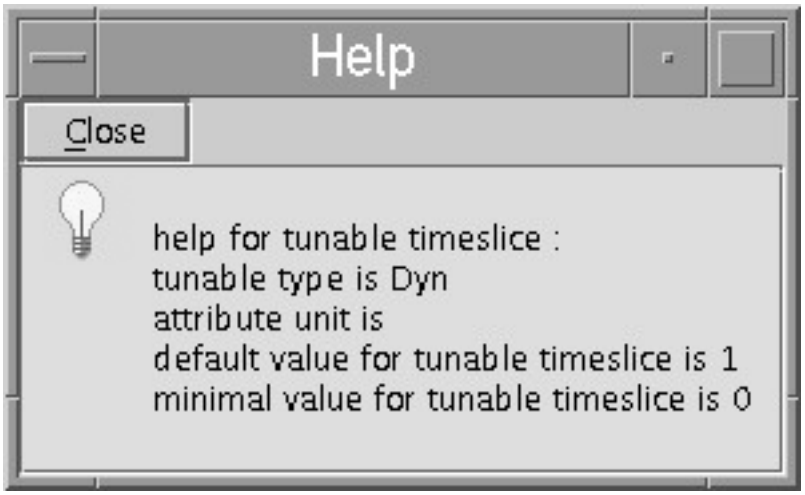


Figure 41. Help dialog

Files

/etc/tunables/lastboot	Contains tuning parameter stanzas from the last boot.
/etc/tunables/lastboot.log	Contains logging information from the last boot.
/etc/tunables/nextboot	Contains tuning parameter stanzas for the next system boot.

Related Information

The **bosboot**, **ioo**, **nfso**, **no**, **schedo**, **tunsave**, **tunrestore**, **tuncheck**, **tundefault**, and **vmo** commands.

The **tunables** file.

Appendix. 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
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

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 PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES 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 publication. 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.

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:

IBM Corporation
Dept. LRAS/Bldg. 003
11400 Burnet Road
Austin, TX 78758-3498
U.S.A.

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.

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

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

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.

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.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples 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.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX

AIX 5L

IBM

Microsoft, Windows 3.1, Windows 95, Windows 98, Windows NT, Windows 2000, and Windows for Workgroups are all registered trademarks of the Microsoft Corporation in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

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

Index

G

I

K

L

N

O

P

R

S

T

U

V

W

X

Readers' Comments — We'd Like to Hear from You

AIX 5L Version 5.2
Performance Tools Guide and Reference

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name _____ Address _____

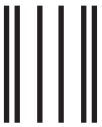
Company or Organization _____

Phone No. _____

Readers' Comments — We'd Like to Hear from You

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Information Development
Department H6DS-905-6C006
11501 Burnet Road
Austin, TX
78758-3493

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line