Performance Tools Guide and Reference
Tunable File-Manipulation Commands ........................................ 175
Initial setup ........................................................................... 179
Reboot Tuning Procedure ....................................................... 179
Recovery Procedure ............................................................... 179
Kernel Tuning Using the SMIT Interface .................................. 180
Kernel Tuning using the Performance Plug-In for Web-based System Manager ........................................... 185
Files ..................................................................................... 196
Related Information .............................................................. 196

Chapter 8. The procmon tool .................................................. 197
Overview of the procmon tool .................................................. 197
Components of the procmon tool ............................................. 198
Filtering processes ............................................................... 200
Performing AIX commands on processes ............................... 200

Chapter 9. Profiling tools ....................................................... 201
The timing commands ........................................................... 201
The prof command ............................................................... 201
The gprof command ............................................................. 203
The tprof command .............................................................. 205

Appendix. Notices ............................................................... 211
Trademarks ........................................................................... 212

Index ..................................................................................... 213
About This Book

The Performance Tools Guide and Reference provides experienced system administrators, application programmers, service representatives, system engineers, end users, and system programmers with complete, detailed information about the various performance tools that are available for monitoring and tuning AIX® systems and applications running on those systems. This publication is also available on the documentation CD that is shipped with the operating system.

The information contained in this book pertains to systems running AIX 6.1 or later. Any content that is applicable to earlier releases will be noted as such.

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:

*Performance management*

*Performance Toolbox Version 2 and 3 for AIX: Guide and Reference*
Chapter 1. Introduction to Performance Tools and Application Program Interfaces (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 also takes 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. Throughout this book, there are sections that 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 might 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 permit 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.

**Note:** Beginning with AIX 5.3, you can generate a new format of the thread-level profiling `gmon.out` files. Xprofiler does not support this new format, so you must set the `GPROF` environment variable to ensure that you produce the previous format of the `gmon.out` files. For more information, please see the gprof Command.

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 might 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 might 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 might not be the same on the two processors. This situation might 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**

<table>
<thead>
<tr>
<th>Directory or file</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/usr/lib/nls/msg/En_US/xprofiler.cat</td>
<td>Message catalog for Xprofiler</td>
</tr>
<tr>
<td>/usr/lib/nls/msg/en_US/xprofiler.cat</td>
<td></td>
</tr>
<tr>
<td>/usr/lib/nls/msg/C/xprofiler.cat</td>
<td></td>
</tr>
<tr>
<td>/usr/xprofiler/defaults/Xprofiler.ad</td>
<td>Defaults file for X-Windows and Motif resource</td>
</tr>
<tr>
<td></td>
<td>variables</td>
</tr>
<tr>
<td>/usr/xprofiler/bin/.startup_script</td>
<td>Startup script for Xprofiler</td>
</tr>
<tr>
<td>/usr/xprofiler/bin/xprofiler</td>
<td>Xprofiler exec file</td>
</tr>
<tr>
<td>/usr/xprofiler/help/en_US/xprofiler.sdl</td>
<td>Online help</td>
</tr>
<tr>
<td>/usr/xprofiler/help/en_US/xprofiler_msg.sdl</td>
<td></td>
</tr>
<tr>
<td>/usr/xprofiler/help/en_US/graphics</td>
<td>Online help</td>
</tr>
<tr>
<td>/usr/xprofiler/READMES/xprofiler.README</td>
<td>Installation readme file</td>
</tr>
<tr>
<td>/usr/xprofiler/samples</td>
<td>Directory containing sample programs</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>This link</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>/usr/lpp/X11/lib/X11/app-defaults/Xprofiler</td>
<td>/usr/xprofiler/defaults/Xprofiler.ad</td>
</tr>
<tr>
<td>/usr/bin/xprofiler</td>
<td>/usr/xprofiler/bin.startup_script</td>
</tr>
</tbody>
</table>

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 might 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 14 for more information).

**Table 2. Xprofiler command-line flags**

<table>
<thead>
<tr>
<th>Use this flag:</th>
<th>To:</th>
<th>For example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</td>
<td>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.</td>
<td>To set an alternative file search path so that Xprofiler searches pathA, the default path, then pathB, type: xprofiler -a pathA:@:pathB</td>
</tr>
<tr>
<td>-b</td>
<td>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.</td>
<td>Type: xprofiler -b a.out gmon.out</td>
</tr>
<tr>
<td>-c</td>
<td>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.</td>
<td>To load the configuration file myfile.cfg, type: xprofiler a.out gmon.out -c myfile.cfg</td>
</tr>
<tr>
<td>-disp_max</td>
<td>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.</td>
<td>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</td>
</tr>
<tr>
<td>-e</td>
<td>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.</td>
<td>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</td>
</tr>
<tr>
<td>Use this flag</td>
<td>To:</td>
<td>For example:</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| -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 permitted 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)

<table>
<thead>
<tr>
<th>Use this flag:</th>
<th>To:</th>
<th>For example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>-F</td>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>

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 permitted 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. |


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

-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.

-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.

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

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, as shown below.

![Xprofiler main window](image)

**Figure 1.** The Xprofiler main window. The screen capture below is an empty Xprofiler window. All that is visible is a menu bar at the top with dropdowns for File, View, Filter, Report, Utility, and Help. Also, there is a description box at the bottom that contains the following text: Empty display, use "File->Load Files" option to load a valid file set.

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

![Load Files Dialog](image)

Figure 2. The Load Files Dialog window. The screen capture below is a Load Files Dialog box that is split into three different sections. There are two boxes, side by side at the top, and one long box at the bottom that are described in more detail in the next three figures.

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. The screen capture below is the Binary Executable File dialog box of the Load Files Dialog window. There is a Filter box at the top that shows the path of the file to load. Underneath the Filter box, there are two selection boxes, side by side that are labeled Directory and Files. The one on the left is to select the Directory in which to locate the executable file, and the one on the right is a listing of the files that are contained in the directory that is selected in the Directory selection box. There is a Selection box that shows the file selected and at the bottom there is a Filter button.](image)

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. The screen capture below is the gmon.out Profile Data File(s) dialog box of the Load Files Dialog window. There is a Filter box at the top that shows the path of the file to use as input. Underneath the Filter box, there are two selection boxes, side by side that are labeled Directory and Files. The one on the left is to select the Directory in which to locate the profile file, and the one on the right is a listing of the files that are contained in the directory that is selected in the Directory selection box. There is a Selection box that shows the file selected and at the bottom there is a Filter button.

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)**
Specify command-line flags from the **Command Line Options** area of the Load Files Dialog window, which looks similar to the following:

![Command Line Options](image)

*Figure 5. The Command Line Options area. The screen capture below is the Command Line Options box of the Load Files Dialog window. There are three check boxes side by side at the top: No description (-b), gmon.sum File (-s), and Show Zero Usage (-z). Below that, there are eight boxes corresponding to the eight Xprofiler GUI command-line flags, Alt File Search Paths (-a), Configuration File (-c), Initial Display (-disp_max), Exclude Functions (-e), Exclude Functions (-E), Include Functions (-f), Include Functions (-F), and Alt Library Path (-L), that are described in great detail below. There is a Choices button next to the Configuration File (-c) box.*

You can specify one or more flags as follows:
### Table 3. Xprofiler GUI command-line flags

<table>
<thead>
<tr>
<th>Use this flag:</th>
<th>To:</th>
<th>For example:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>-a (field)</strong></td>
<td>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.</td>
<td>To set an alternative file search path so that Xprofiler searches <strong>pathA</strong>, the default path, then <strong>pathB</strong>, type <strong>pathA:@:pathB</strong> in the Alt File Search Path(s) (-a) field.</td>
</tr>
<tr>
<td><strong>-b (button)</strong></td>
<td>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.</td>
<td>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.</td>
</tr>
<tr>
<td><strong>-c (field)</strong></td>
<td>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.</td>
<td>To load the configuration file <strong>myfile.cfg</strong>, type <strong>myfile.cfg</strong> in the Configuration File (-c) field.</td>
</tr>
<tr>
<td>Use this flag:</td>
<td>To:</td>
<td>For example:</td>
</tr>
<tr>
<td>--------------</td>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>-disp_max</strong></td>
<td>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.</td>
<td>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.</td>
</tr>
<tr>
<td><strong>-e</strong></td>
<td>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.</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Table 3. Xprofiler GUI command-line flags (continued)

<table>
<thead>
<tr>
<th>Use this flag:</th>
<th>To:</th>
<th>For example:</th>
</tr>
</thead>
</table>
| -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 permitted 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. Multiple functions are separated by a space. |
<table>
<thead>
<tr>
<th>Use this flag:</th>
<th>To:</th>
<th>For example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>-f (field)</td>
<td>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 <code>-f</code> flag overrides the <code>-e</code> 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.</td>
<td>To deemphasize the display of function boxes for all functions in the function call tree except for <code>foo</code> and <code>bar</code> and their descendants, and limit their types of entries in the Call Graph Profile report, type <code>foo bar</code> in the Include Routines (<code>-f</code>) field. Multiple functions are separated by a space.</td>
</tr>
<tr>
<td>-F (field)</td>
<td>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 <code>-F</code> flag overrides the <code>-E</code> 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 permitted 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.</td>
<td>To change the display and label information of the function boxes for all functions except the functions <code>foo</code> and <code>bar</code> and their descendants, and limit their types of entries and data in the Call Graph Profile report, type <code>foo bar</code> in the Include Routines (<code>-F</code>) field. Multiple functions are separated by a space.</td>
</tr>
<tr>
<td>-L (field)</td>
<td>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.</td>
<td>To specify <code>/lib/profiled/libc.a:shr.o</code> as an alternative path name for your shared libraries, type <code>/lib/profiled/libc.a:shr.o</code> in this field.</td>
</tr>
</tbody>
</table>
Table 3. Xprofiler GUI command-line flags (continued)

<table>
<thead>
<tr>
<th>Use this flag:</th>
<th>To:</th>
<th>For example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>-s (button)</td>
<td>Produces the <code>gmon.sum</code> profile data file, if multiple <code>gmon.out</code> files are specified when Xprofiler is started. The <code>gmon.sum</code> file represents the sum of the profile information in all the specified profile files. Note that if you specify a single <code>gmon.out</code> file, the <code>gmon.sum</code> file contains the same data as the <code>gmon.out</code> file.</td>
<td>To write the sum of the data from three profile data files, <code>gmon.out.1</code>, <code>gmon.out.2</code>, and <code>gmon.out.3</code>, into a file called <code>gmon.sum</code>, set the -s button to the pressed-in position.</td>
</tr>
<tr>
<td>-z (button)</td>
<td>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 <code>-pg</code> flag, which is common with system library files.</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 enables 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:
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.

Figure 6. The Xprofiler main window with application loaded. The screen capture below shows one function box displaying a function call tree, with an arc pointing down to another function box displaying a function call tree in the Xprofiler main window.
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.
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, as shown below.
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 \texttt{gmon.out} files, used on the function itself. The width of each node represents the standard deviation of CPU time, among the \texttt{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 \texttt{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 \texttt{gmon.out} file is specified, \textbf{Summary Mode} and \textbf{Average Mode} will be unavailable, and the display is always in \textbf{Summary Mode}.

\section*{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.

\textbf{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.

\section*{Restoring the Status of the Function Call Tree}

Xprofiler enables 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 \textbf{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 \textbf{Filter} menu, and then the \textbf{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 \textbf{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 \textbf{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 \textbf{Undo} option include the following:

\begin{itemize}
  \item In the main \textbf{File} menu:
    \begin{itemize}
      \item Load Configuration
    \end{itemize}
  \item In the main \textbf{Filter} menu:
    \begin{itemize}
      \item Show Entire Call Tree
      \item Hide All Library Calls
      \item Add Library Calls
      \item Filter by Function Names
      \item Filter by CPU Time
      \item Filter by Call Counts
    \end{itemize}
  \item In the \textbf{Function} menu:
    \begin{itemize}
      \item Immediate Parents
      \item All Paths To
      \item Immediate Children
      \item All Paths From
      \item All Functions on The Cycle
      \item Show This Function Only
      \item Hide This Function
      \item Hide Descendant Functions
    \end{itemize}
\end{itemize}
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 can choose to filter out specific items from the window. However, there might 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 can 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 might 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 might 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 enable 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 following Filter By Function Names Dialog window appears:

![Filter By Function Names Dialog](image)

**Figure 9.** The Filter By Function Names Dialog window. The screen capture below shows the Filter By Function Names Dialog window. There are three check boxes: Add these functions to graph, Remove these functions from graph, and Display only these functions. There is an Enter Function Name box, where regular expressions are supported, and below it there are four buttons: OK, Apply, Cancel, and Help.

The Filter By Function Names Dialog window includes the following options:
• add these functions to graph
• remove these functions from the graph
• 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 following Filter By CPU Time Dialog window appears:
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.
The Filter By Call Counts Dialog window includes the following options:

1. show arcs with the most call counts
2. 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 might 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 might want to organize what you see in the Xprofiler window.
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. The following screen capture shows the hello_world application with the top-to-bottom view of its function boxes unclustered in the Xprofiler main window.](image)

### 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.

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 following 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. The following screen capture shows the function call tree of the hello program in the Xprofiler window with one library cluster box removed.](image)

**Locating Specific Objects in the Function Call Tree**

If you are interested in one or more specific functions in a complex program, you might 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, similar to the following:
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 \texttt{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 \texttt{gmon.out} files, used on the function itself
- The standard deviation of CPU time (in seconds), among all the input \texttt{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 might need to zoom in on it in order to see the labels. For information about how to do this, see “Information Boxes” on page 39.

**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 15. An example of a function box label. The following screen capture shows the details of a function box and in this example it is of the sub1 function. The following information is listed: The function label (sub1), the cycle it is associated with (1), and its index (5).]
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.

![Function Level Statistics Report](image)

**Figure 17. The Function Level Statistics Report window. The screen capture below shows the Function Level Statistics Report window and shows the details of the main function. The specifics of a Function Level Statistics Report are detailed below the graphic.**

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 reports:

• 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:

```
File   Code Display   Utility
%time cumulative seconds self seconds calls ms/call ms/call name
54.5  0.06  0.06    2  30.00  30.00  hello_world.o .sub2 <cycle> [2]
27.3  0.09  0.03    2  15.00  15.00  hello_world.o .sub1 <cycle> [5]
9.1   0.10  0.01    1  10.00  100.00 hello_world.o main [3]
...

Figure 18. The Flat Profile report. The screen capture below shows an example of a Flat Profile report window. There is a menu bar at the top with the following options: File, Code Display, Utility, and Help. Below the menu bar is a list of statistics that are described below the graphic.

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:

![Call Graph Profile window](image)

*Figure 19. The Call Graph Profile report. The screen capture below shows an example of a Flat Profile report window. There is a menu bar at the top with the following options: File, and Help. Below the menu bar is a list of statistics that are described below the graphic.*

**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, called+self, called/total field. The screen capture below is an example of the called/total, called+self, called/total field of the Call Graph Profile report where sub2 is the function of interest, sub1 and main are its parents, and printf and sub1 are its children.](image)

- **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.
### 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:

![Function Index Report Example](image)

**Figure 21.** The name/index/parents/children field. The screen capture below is an example of the name/index/parents/children field of the Call Graph Profile report. 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.

- **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.
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:

```
[17] __ioctl [17]
[18] __findbuf [18]
[19] __write [19]
[8] __write [8]
[21] .exit [21]
[22] .expand_catname [22]
[16] .freem [16]
[10] .fwrite_unlocked [10]
[23] .getenv [23]
[26] .moncontrol [26]
[27] .monitor [27]
[28] .pre_ioctl [28]
```

Figure 22. The Function Index report. The following screen capture shows the Function Index Report window. There is a menu bar at the top with the following options: File, Code Display, Utility, and Help. Then, there is a list of the function names included in the function call tree, where to the left of each function name is its index, enclosed in brackets. An index also appears on the label of each corresponding function box in the function call tree.

**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:
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:
**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 that 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 14.
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 permits 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:

![Source Code window](image)

Figure 25. The Source Code window. The following screen capture shows an example of the Source Code window. There is a menu bar at the top with the following options: File, Utility, and Help. The fields of the Source Code window are described in greater detail below the graphic.

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 unavailable. 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 permit 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 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:
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.

![Screen Dump Options Dialog]

*Figure 27. The Screen Dump Options Dialog window. The screen capture below shows an example of the Screen Dump Options Dialog window. Each section of the Screen Dump Options Dialog window is described in greater detail below the graphic.*

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 grays 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 might 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 permits 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 might 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:

<table>
<thead>
<tr>
<th>Use this resource variable:</th>
<th>Specify this default, or a value of your choice:</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>narc</em>font</td>
<td>fixed</td>
</tr>
</tbody>
</table>

To specify the font used in textual reports:

<table>
<thead>
<tr>
<th>Use this resource variable:</th>
<th>Specify this default, or a value of your choice:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*fontList</td>
<td>rom10</td>
</tr>
</tbody>
</table>

**Controlling the Appearance of the Xprofiler Main Window**

To specify the size of the main window:

<table>
<thead>
<tr>
<th>Use this resource variable:</th>
<th>Specify this default, or a value of your choice:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*mainW.height</td>
<td>700</td>
</tr>
<tr>
<td>Xprofiler*mainW.width</td>
<td>900</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Use this resource variable:</th>
<th>Specify this default, or a value of your choice:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*foreground</td>
<td>black</td>
</tr>
<tr>
<td>Xprofiler*background</td>
<td>light grey</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Use this resource variable:</th>
<th>Specify this default, or a value of your choice:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*InitialDisplayGraph</td>
<td>5000</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Use this resource variable:</th>
<th>Specify this default, or a value of your choice:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*defaultNodeColor</td>
<td>forest green</td>
</tr>
<tr>
<td>Xprofiler*defaultArcColor</td>
<td>royal blue</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Use this resource variable:</th>
<th>Specify this default, or a value of your choice:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*HighlightNode</td>
<td>red</td>
</tr>
<tr>
<td>Xprofiler*HighlightArc</td>
<td>red</td>
</tr>
</tbody>
</table>
To specify the color in which de-emphasized function boxes appear:

<table>
<thead>
<tr>
<th>Use this resource variable</th>
<th>Specify this default, or a value of your choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*SuppressNode</td>
<td>grey</td>
</tr>
</tbody>
</table>

Function boxes are de-emphasized 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:

<table>
<thead>
<tr>
<th>Use this resource variable</th>
<th>Specify this default, or a value of your choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*loadFile.height</td>
<td>785</td>
</tr>
<tr>
<td>Xprofiler*loadFile.width</td>
<td>725</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Use this resource variable</th>
<th>Specify this default, or a value of your choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*OverwriteOK</td>
<td>False</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Use this resource variable</th>
<th>Specify this default, or a value of your choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*fileSearchPath</td>
<td>. (refers to the current working directory)</td>
</tr>
</tbody>
</table>

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):

<table>
<thead>
<tr>
<th>Use this resource variable</th>
<th>Specify this default, or a value of your choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*fileSearchDefault</td>
<td>True</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Use this resource variable</th>
<th>Specify this default, or a value of your choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*PrintToFile</td>
<td>True</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Use this resource variable</th>
<th>Specify this default, or a value of your choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*ColorPscript</td>
<td>False</td>
</tr>
</tbody>
</table>

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):

<table>
<thead>
<tr>
<th>Use this resource variable:</th>
<th>Specify this default, or a value of your choice:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*GreyShades</td>
<td>16</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Use this resource variable:</th>
<th>Specify this default, or a value of your choice:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*GrabDelay</td>
<td>1</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Use this resource variable:</th>
<th>Specify this default, or a value of your choice:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*grabDelayScale.maximum</td>
<td>30</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Use this resource variable:</th>
<th>Specify this default, or a value of your choice:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*Landscape</td>
<td>False</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Use this resource variable:</th>
<th>Specify this default, or a value of your choice:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*Annotate</td>
<td>False</td>
</tr>
</tbody>
</table>

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):

<table>
<thead>
<tr>
<th>Use this resource variable:</th>
<th>Specify this default, or a value of your choice:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*PrintFileName</td>
<td>/tmp/Xprofiler_screenDump.ps.0</td>
</tr>
</tbody>
</table>
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):

<table>
<thead>
<tr>
<th>Use this resource variable</th>
<th>Specify this default, or a value of your choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*PrintCommand</td>
<td>qprt -B ga -c -Pps</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Use this resource variable</th>
<th>Specify this default, or a value of your choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*overviewMain.height</td>
<td>300</td>
</tr>
<tr>
<td>Xprofiler*overviewMain.width</td>
<td>300</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Use this resource variable</th>
<th>Specify this default, or a value of your choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler<em>overviewGraph</em>defaultHighlightColor</td>
<td>sky blue</td>
</tr>
</tbody>
</table>

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):

<table>
<thead>
<tr>
<th>Use this resource variable</th>
<th>Specify this default, or a value of your choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*TrackImmed</td>
<td>True</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Use this resource variable</th>
<th>Specify this default, or a value of your choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*Shape2D</td>
<td>True</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Use this resource variable</th>
<th>Specify this default, or a value of your choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*LayoutTopDown</td>
<td>True</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Use this resource variable:</th>
<th>Specify this default, or a value of your choice:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xprofiler*ClusterArc</td>
<td>True</td>
</tr>
</tbody>
</table>

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, thread or pthread activity. These statistics ease the tracking of specific application activity. The curt command works with both uniprocessor and multiprocessor AIX Version 4 and AIX Version 5 traces.

Syntax for the curt Command

The syntax for the curt command is as follows:

```
```

Flags

- `-i inputfile` Specifies the input AIX trace file to be analyzed.
- `-o outputfile` Specifies an output file (default is stdout).
- `-n gensymsfile` Specifies a names file produced by gensyms.
- `-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.
- `-r PURR` Uses the PURR register to calculate CPU times.
- `-e` Outputs elapsed time information for system calls.
- `-h` Displays usage text (this information).
- `-p` Outputs detailed process information.
- `-s` Outputs information about errors returned by system calls.
- `-t` Outputs detailed thread information.
- `-P` Outputs detailed pthread information.

Parameters

- `gensymsfile` The names file as produced by the gensyms command.
- `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 the trcnm command.
- `PURR` The name of the register that is used to calculate CPU times.
**Measurement and Sampling**

A raw, or unformatted, system trace is read by the `curt` command to produce CPU utilization summaries. The summary information is useful for determining which application, system call, Network File System (NFS) operation, hypervisor call, pthread call, or interrupt handler is using most of the CPU time and is a candidate for optimization 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 might not be captured in this case. This is significant if you intend to analyze the trace in more detail.

<table>
<thead>
<tr>
<th>Hook ID</th>
<th>Event Name</th>
<th>Event Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>HKWD_KERN_FLIH</td>
<td>Occurrence of a first level interrupt, such as an I/O interrupt, a data access page fault, or a timer interrupt (scheduler).</td>
</tr>
<tr>
<td>101</td>
<td>HKWD_KERN_SVC</td>
<td>A thread has issued a system call.</td>
</tr>
<tr>
<td>102</td>
<td>HKWD_KERN_SLIH</td>
<td>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.</td>
</tr>
<tr>
<td>103</td>
<td>HKWD_KERN_SLIHRET</td>
<td>Return from a second level interrupt to the caller (usually a first level interrupt handler).</td>
</tr>
<tr>
<td>104</td>
<td>HKWD_KERN_SYSCRET</td>
<td>Return from a system call to the caller (usually a thread).</td>
</tr>
<tr>
<td>106</td>
<td>HKWD_KERN_DISPATCH</td>
<td>A thread has been dispatched from the run queue to a CPU.</td>
</tr>
<tr>
<td>10C</td>
<td>HKWD_KERN_IDLE</td>
<td>The idle process has been dispatched.</td>
</tr>
<tr>
<td>119</td>
<td>HKWD_KERN_PIDSIG</td>
<td>A signal has been sent to a process.</td>
</tr>
<tr>
<td>134</td>
<td>HKWD_SYSC_EXECVE</td>
<td>An exec supervisor call (SVC) has been issued by a (forked) process.</td>
</tr>
<tr>
<td>135</td>
<td>HKWD_SYSC__EXIT</td>
<td>An exit supervisor call (SVC) has been issued by a process.</td>
</tr>
<tr>
<td>139</td>
<td>HKWD_SYSC_FORK</td>
<td>A fork SVC has been issued by a process.</td>
</tr>
<tr>
<td>200</td>
<td>HKWD_KERN_RESUME</td>
<td>A dispatched thread is being resumed on the CPU.</td>
</tr>
<tr>
<td>210</td>
<td>HKWD_KERN_INITP</td>
<td>A kernel process has been created.</td>
</tr>
<tr>
<td>215</td>
<td>HKWD_NFS_DISPATCH</td>
<td>An entry or exit NFS V2 and V3 operation has been issued by a process.</td>
</tr>
<tr>
<td>38F</td>
<td>HKWD_DR</td>
<td>A processor has been added/removed.</td>
</tr>
<tr>
<td>419</td>
<td>HKWD_CPU_PREEMPT</td>
<td>A processor has been preempted.</td>
</tr>
<tr>
<td>465</td>
<td>HKWD_SYSC_CRTHREAD</td>
<td>A <strong>thread_create</strong> SVC has been issued by a process.</td>
</tr>
<tr>
<td>47F</td>
<td>HKWD_KERN_PHANTOM_EXTINT</td>
<td>A phantom interrupt has occurred.</td>
</tr>
<tr>
<td>488</td>
<td>HKWD_RFS4_VOPS</td>
<td>An entry or exit NFS V4 client operation (VOPS) has been issued by a process.</td>
</tr>
<tr>
<td>489</td>
<td>HKWD_RFS4_VFSOPS</td>
<td>An entry or exit NFS V4 client operation (VFSOPS) has been issued by a process.</td>
</tr>
<tr>
<td>48A</td>
<td>HKWD_RFS4_MISCOPS</td>
<td>An entry or exit NFS V4 client operation (MISCOPS) has been issued by a process.</td>
</tr>
<tr>
<td>48D</td>
<td>HKWD_RFS4</td>
<td>An entry or exit NFS V4 server operation has been issued by a process.</td>
</tr>
<tr>
<td>492</td>
<td>HKWD_KERN_HCALL</td>
<td>A hypervisor call has been issued by the kernel.</td>
</tr>
<tr>
<td>605</td>
<td>HKWD_PTHREAD_VPSLEEP</td>
<td>A <strong>pthread vp_sleep</strong> operation has been done by a pthread.</td>
</tr>
<tr>
<td>609</td>
<td>HKWD_PTHREAD_GENERAL</td>
<td>A general pthread operation has been done by a pthread.</td>
</tr>
</tbody>
</table>
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.

Trace hook 492 is used to report on the time spent in the hypervisor.

Trace hooks 605 and 609 are used to report on the time spent in the pthreads library.

To get the PTHREAD hooks in the trace, you must execute your pthread application using the instrumented `libpthreads.a` library.

---

**Examples of the curt command**

Preparing the `curt` command input is a three-stage process.

Trace and name files are generated using the following process:

1. **Build the raw trace.**
   
   On a 4-way machine, this will create files as listed in the example code below. 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.

   **Note:** If you want pthread information in the `curt` report, you must add the instrumented `libpthreads` directory to the library path, `LIBPATH`, when you build the trace. Otherwise, the `export LIBPATH` statement in the example below is unnecessary.

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 gensymsfile and trcnmfile files by running the gensyms and trcnm commands.**

   Neither the `gensymsfile` nor the `trcnmfile` file are necessary for the `curt` command to run. However, if you provide one or both of these files, or if you use the `trace` command with the `-n` option, the `curt` command outputs names for system calls and interrupt handlers instead of just addresses. The `gensyms` command output includes more information than the `trcnm` command output, and so, while the `trcnmfile` file will contain most of the important address to name mapping data, a `gensymsfile` file will enable the `curt` command to output more names, and is the preferred address to name mapping data collection command.

The following is an example of how to generate input files for the `curt` command:

```
# SIZE="1000000"
# export HOOKS SIZE
# trace -n -C all -d -j $HOOKS -L $SIZE -T $SIZE -afo trace.raw
# export LIBPATH=/usr/ccs/lib/perf:$LIBPATH
# trcon ; pthread.app ; 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.r
# gensyms > gensyms.out
# trcnm > trace.nm
```

---

**Overview of Information Generated by the curt Command**

The following is an overview of the content of the report that the `curt` command generates:

- A report header, including the trace file name, the trace size, and the date and time the trace was taken. The header also includes the command that was used when the trace was run. If the PURR register was used to calculate CPU times, this information is also included in the report header.
• 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.
• For each CPU (and a summary of all the CPUs), processing time expressed in milliseconds and as a percentage for CPU usage in application mode for various application usage categories.
• Average thread affinity across all CPUs and for each individual CPU.
• For each CPU (and for all the CPUs), the Physical CPU time spent and the percentage of total time this represents.
• Average physical CPU affinity across all CPUs and for each individual CPU.
• The physical CPU dispatch histogram of each CPU.
• The number of preemptions, and the number of H_CEDE and H_CONFER hypervisor calls for each individual CPU.
• The total number of idle and non-idle process dispatches for each individual CPU.
• Average pthread affinity across all CPUs and for each individual CPU.
• The total number of idle and non-idle pthread 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 the amount of CPU time spent executing calls to libpthread, expressed in milliseconds and as percentages of the total time and the total application 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 completed hypervisor calls that includes the name and address of the hypervisor call, the number of times the hypervisor call was executed, and the total CPU time expressed in milliseconds and as a percentage with average, minimum, and maximum time the hypervisor call was running.
• Information on pending hypervisor calls, which are hypervisor calls that were not completed by the end of the trace. The information includes the name and address of the hypervisor call, the thread or process which made the hypervisor call, and the accumulated CPU time the hypervisor call was running, expressed in milliseconds.
• Information on completed pthread calls that includes the name of the pthread call routine, the number of times the pthread call was executed, and the total CPU time expressed in milliseconds and the average, minimum, and maximum time the pthread call was running.
• Information on pending pthread calls, that is, pthread calls for which the pthread call return has not occurred at the end of the trace. The information includes the name of the pthread call, the process, the thread and the pthread which made the pthread call, and the accumulated CPU time the pthread call was running expressed in milliseconds.
• Information on completed NFS operations that includes the name of the NFS operation, the number of times the NFS operation was executed, and the total CPU time, expressed in milliseconds, and as a percentage with average, minimum, and maximum time the NFS operation call was running.
• Information on pending NFS operations, where the NFS operations did not complete before the end of the trace. The information includes the sequence number for NFS V2/V3, or opcode for NFS V4, the thread or process which made the NFS operation, and the accumulated CPU time that the NFS operation 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), the accumulated time and the pending SLIH name and address is reported for each CPU.

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

- **-e** Produces reports containing statistics and additional information on the System Calls Summary Report, Pending System Calls Summary Report, Hypervisor™ Calls Summary Report, System NFS Calls Summary Report, Pending NFS Calls Summary, Pthread Calls Summary, and the Pending Pthread Calls Summary. The additional information pertains to the total, average, maximum, and minimum elapsed times that a system call was running.

- **-s** Produces a report containing a list of errors returned by system calls.

- **-t** Produces a report containing 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 containing a detailed report on process status that includes the amount of CPU time the process was in application and system call mode, application time details, threads that were in the process, pthreads that were in the process, pthread calls that the process made and system calls that the process made.

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

### Default Report Generated by the curt Command

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

```
# curt -i trace.r -n gensyms.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
- System Application Summary
- Processor Summary
- Processor Application Summary
- Application Summary by TID
- Application Summary by PID
- Application Summary by Process Type
- Kproc Summary
- Application Pthread Summary by PID
- System Calls Summary
Pending System Calls Summary
Hypervisor Calls Summary
Pending Hypervisor Calls Summary
System NFS Calls Summary
Pending NFS System Calls Summary
Pthread Calls Summary
Pending Pthread Calls Summary
FLIH Summary
SLIH Summary

General Information
The General Information section begins with the time and date when the report was generated. It is followed by the syntax of the `curt` command line that was used to produce the report.

This section also contains some information about the AIX trace file that was processed by the `curt` command. 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 the general information section:

```
Run on Wed Apr 26 10:51:33 2XXX
Command line was:
curt -i trace.raw -n gensyms.out -o curt.out
----
AIX trace file name = trace.raw
AIX trace file size = 787848
Wed Apr 26 10:50:11 2XXX
System: AIX 5.3 Node: bu Machine: 00CFEDAD4C00
AIX trace file created = Wed Apr 26 10:50:11 2XXX
Command used to gather AIX trace was:
trace -n -C all -d -j 100,101,103,104,106,10C,134,139,200,215,419,465,47F,488,489,48A,48D,492,605,609
-L 1000000 -T 1000000 -afo trace.raw
```

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

```
  System Summary
   ------------
  processing  percent  percent          processing category
    total time (msec) (incl. idle) (excl. idle)
  -----------  -----------  ------------
    4998.65     45.94      75.21  APPLICATION
      591.59     5.44       8.90  SYSCALL
       110.40     1.02       1.66  HCALL
        48.33     0.44       0.73  KPROC (excluding IDLE and NFS)
       352.23     3.24       5.30  NFS
      486.19     4.47       7.32  FLIH
       49.10     0.45       0.74  SLIH
        8.83     0.08       0.13  DISPATCH (all procs. incl. IDLE)
        1.04     0.01       0.02  IDLE DISPATCH (only IDLE proc.)
  -----------  -----------  -------------
     6646.36    61.08     100.00  CPU(s) busy time
     4234.76    38.92     100.00  IDLE
  -----------  -----------  -------------
     10881.12    100.00     100.00  TOTAL
```

Avg. Thread Affinity = 0.99

Total Physical CPU time (msec) = 20417.45
Physical CPU percentage = 100.00%
This portion of the report describes the time spent by the whole system (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.
- **Total Physical CPU time**: The real time that the virtual processor was running and not preempted.
- **Physical CPU percentage**: Gives the Physical CPU Time as a percentage of total time.

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.
- **HCALL**: The sum of times spent by all processors doing Hypervisor Calls. This is the portion of time that a processor spends executing in the hypervisor code providing services directly requested by the kernel.
- **KPROC**: The sum of times spent by all processors executing kernel processes other than IDLE and NFS processes. This is the portion of time that a processor spends executing specially created dispatchable processes that only execute kernel code.
- **NFS**: The sum of times spent by all processors executing NFS operations. This is the portion of time that a processor spends executing in the kernel code providing NFS services directly requested by a kernel process.
- **FLIH**: The sum of times spent by all processors executing FLIHs.
- **SLIH**: The sum of times spent by all processors executing SLIHs.
- **DISPATCH**: The sum of times spent by all processors executing 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. There is still 4234.76 ms of IDLE time so there is enough CPU to run applications. If there is insufficient CPU power, do 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 run again.
**System Application Summary**

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

<table>
<thead>
<tr>
<th>System Application Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>processing</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>3.95</td>
</tr>
<tr>
<td>4.69</td>
</tr>
<tr>
<td>0.13</td>
</tr>
<tr>
<td>5356.99</td>
</tr>
<tr>
<td>5365.77</td>
</tr>
</tbody>
</table>

Avg. Pthread Affinity = 0.84

This portion of the report describes the time spent by the system as a whole (all CPUs) in various execution modes. The System Application 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 application time**: Time from the first column as a percentage of the sum of total trace elapsed application time for all processors.
- **Avg. Pthread Affinity**: Probability that a pthread was dispatched on the same kernel thread on which it last executed.

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

- **PTHREAD**: The sum of times spent by all pthreads on all processors in traced pthread library calls.
- **PDISPATCH**: The sum of times spent by all pthreads on all processors executing the libpthreads dispatch code.
- **PIDLE**: The sum of times spent by all kernel threads on all processors executing the libpthreads `vp_sleep` code.
- **OTHER**: The sum of times spent by all pthreads on all processors in non-traced user mode.
- **APPLICATION**: The sum of times spent by all processors in User (that is, non-privileged) mode.

**Processor Summary and Processor Application Summary**

This part of the `curt` command output follows the System Summary and System Application 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 and System Application Summary applies here, except that this report covers each processor rather than the whole system.

Below is a sample of this output:

<table>
<thead>
<tr>
<th>Processor Summary</th>
<th>processor number 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>processing total time (msec)</td>
<td>percent</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------</td>
</tr>
<tr>
<td>45.07</td>
<td>0.88</td>
</tr>
<tr>
<td>591.39</td>
<td>11.58</td>
</tr>
</tbody>
</table>
0.00  0.00  0.00  HCALL
47.83  0.94  5.48  KPROC (excluding IDLE and NFS)
0.00  0.00  0.00  NFS
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  100.00 IDLE
--------- --------- -------
  5106.34  TOTAL

Avg. Thread Affinity =  0.98

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

Total Physical CPU time (msec) = 3246.25
Physical CPU percentage = 63.57%
Physical processor affinity = 0.50
Dispatch Histogram for processor (PHYSICAL CPUid : times_dispatched).

PROC 0 : 15
PROC 24 : 15

Total number of preemptions = 30
Total number of H_CEDE = 6  with preemption = 3
Total number of H_CONFER = 3  with preemption = 2

Processor Application Summary processor 0
-----------------------------------------------

<table>
<thead>
<tr>
<th>processing</th>
<th>percent</th>
<th>percent</th>
<th>processing category</th>
</tr>
</thead>
<tbody>
<tr>
<td>total time</td>
<td>total time (incl. idle)</td>
<td>time processing category</td>
<td></td>
</tr>
<tr>
<td>(msec)</td>
<td>(incl. idle)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.66</td>
<td>0.04</td>
<td>0.06</td>
<td>PTHREAD</td>
</tr>
<tr>
<td>2.61</td>
<td>0.05</td>
<td>0.10</td>
<td>PDISPATCH</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>PIDLE</td>
</tr>
<tr>
<td>2685.12</td>
<td>56.67</td>
<td>99.84</td>
<td>OTHER</td>
</tr>
</tbody>
</table>

2689.39  56.76  100.00  APPLICATION
--------- --------- -------

Avg. Pthread Affinity =  0.78

Total number of pthread dispatches = 104
Total number of pthread idle dispatches = 0

Processor Summary processor number 1
--------------------------------------------

<table>
<thead>
<tr>
<th>processing</th>
<th>percent</th>
<th>percent</th>
<th>percent</th>
<th>percent</th>
<th>processing category</th>
</tr>
</thead>
<tbody>
<tr>
<td>total time</td>
<td>total time (incl. idle)</td>
<td>busy time</td>
<td>processing category</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(msec)</td>
<td>(incl. idle)</td>
<td>(excl. idle)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4985.81</td>
<td>97.70</td>
<td>97.70</td>
<td>APPLICATION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.09</td>
<td>0.00</td>
<td>0.00</td>
<td>SYSCALL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>HCALL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>KPROC (excluding IDLE and NFS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>NFS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>103.86</td>
<td>2.04</td>
<td>2.04</td>
<td>FLIH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.54</td>
<td>0.25</td>
<td>0.25</td>
<td>SLIH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.97</td>
<td>0.02</td>
<td>0.02</td>
<td>DISPATCH (all procs. incl. IDLE)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>IDLE DISPATCH (only IDLE proc.)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5103.26  100.00  100.00  CPU(s) busy time
0.00  0.00  0.00  IDLE
Avg. Thread Affinity = 0.99

Total number of process dispatches = 516
Total number of idle dispatches = 0

Total Physical CPU time (msec) = 5103.26
Physical CPU percentage = 100.00%
Physical processor affinity = 1.00

Dispatch Histogram for processor (PHYSICAL CPUid : times_dispatched).
Total number of preemptions = 0
Total number of H_CEDE = 0 with preemption = 0
Total number of H_CONFER = 0 with preemption = 0

The following terms are referred to in the example above:

**Total number of process dispatches**
The number of times AIX dispatched any non-IDLE process on the processor.

**Total number of idle dispatches**
The number of IDLE process dispatches.

**Total number of pthread dispatches**
The number of times the libpthreads dispatcher was executed on the processor.

**Total number of pthread idle dispatches**
The number of \texttt{vp_sleep} calls.

**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)**

<table>
<thead>
<tr>
<th>combined (application syscall name)</th>
<th>total (msec)</th>
<th>percent of total processing time</th>
</tr>
</thead>
<tbody>
<tr>
<td>4986.2355 4986.2355 0.0000 24.4214 24.4214 0.0000 cpu(18418 32437)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4985.8051 4985.8051 0.0000 24.4193 24.4193 0.0000 cpu(19128 33557)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4982.0331 4982.0331 0.0000 24.4009 24.4009 0.0000 cpu(18894 28671)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83.8436 2.5062 81.3374 0.4106 0.0123 0.3984 disp+work(20390 28397)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>72.5809 2.7269 69.8540 0.3555 0.0134 0.3421 disp+work(18584 32777)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
---
The output is divided into two main sections:

- The total processing time of the thread in milliseconds (processing total (msec))
- The CPU time that 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 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 the example above, 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 name (PID) (Thread Count) column 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.
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.

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 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.
- **kernel**: The amount of CPU time, expressed in milliseconds, that the thread spent in unidentified kernel mode.
- **operation**: The amount of CPU time, expressed in milliseconds, that the thread spent in traced operations.

**percent of total time**

- **combined**: The amount of CPU time that the thread was running, expressed as percentage of the total processing time.
- **kernel**: The amount of CPU time that the thread spent in unidentified kernel mode, expressed as percentage of the total processing time.
- **operation**: The amount of CPU time that the thread spent in traced operations, 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 traced operations for this type of kernel process.

**Application Pthread Summary by process ID (Pid)**

The Application Pthread Summary, by PID, shows an output of all the multi-threaded processes that were running on the system during trace collection and their CPU consumption, and that have spent time making pthread calls. The process that consumed the most CPU time during the trace collection is at the beginning of the list.

<table>
<thead>
<tr>
<th>name (Pid) (Pthread Count)</th>
<th>processing total (msec)</th>
<th>percent of total application time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>application</td>
<td>other</td>
</tr>
<tr>
<td></td>
<td>1277.6602</td>
<td>1274.9354</td>
</tr>
<tr>
<td></td>
<td>802.6445</td>
<td>801.4162</td>
</tr>
<tr>
<td></td>
<td>. /pth(245964) (52)</td>
<td>. /pth32(245962) (12)</td>
</tr>
</tbody>
</table>

...(lines omitted)...

The output is divided into two main sections:

- The total processing time of the process in milliseconds (processing total (msec))
- The CPU time that the process has consumed, expressed as a percentage of the total application time

The Application Pthread Summary has the following fields:

- **name (Pid) (Pthread Count)**: The name of the process associated with the process ID, and the number of pthreads of this process.
- **processing total (msec)**: The total amount of CPU time, expressed in milliseconds, that the process was running in user mode.
pthread

The amount of CPU time, expressed in milliseconds, that the process spent in traced call to the pthreads library.

other

The amount of CPU time, expressed in milliseconds, that the process spent in non traced user mode.

percent of total application time

application

The amount of CPU time that the process was running in user mode, expressed as percentage of the total application time.

pthread

The amount of CPU time that the process spent in calls to the pthreads library, expressed as percentage of the total application time.

other

The amount of CPU time that the process spent in non traced user mode, expressed as percentage of the total application time.

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.

<table>
<thead>
<tr>
<th>Count</th>
<th>Total Time (msec)</th>
<th>% sys time</th>
<th>Avg Time (msec)</th>
<th>Min Time (msec)</th>
<th>Max Time (msec)</th>
<th>SVC (Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>605</td>
<td>355.4475</td>
<td>1.74%</td>
<td>0.5875</td>
<td>0.0482</td>
<td>4.5626</td>
<td>kwrite(4259c4)</td>
</tr>
<tr>
<td>733</td>
<td>196.3752</td>
<td>0.96%</td>
<td>0.2679</td>
<td>0.0042</td>
<td>2.9948</td>
<td>kread(4259e8)</td>
</tr>
<tr>
<td>3</td>
<td>9.2217</td>
<td>0.05%</td>
<td>3.0739</td>
<td>2.8888</td>
<td>3.3418</td>
<td>execve(1c95d8)</td>
</tr>
<tr>
<td>38</td>
<td>7.6013</td>
<td>0.04%</td>
<td>0.2000</td>
<td>0.0051</td>
<td>1.6137</td>
<td>__loadx(1c9608)</td>
</tr>
<tr>
<td>1244</td>
<td>4.4574</td>
<td>0.02%</td>
<td>0.0036</td>
<td>0.0010</td>
<td>0.0143</td>
<td>__seek(425a60)</td>
</tr>
<tr>
<td>45</td>
<td>4.3917</td>
<td>0.02%</td>
<td>0.0976</td>
<td>0.0248</td>
<td>0.1810</td>
<td>access(507860)</td>
</tr>
<tr>
<td>63</td>
<td>3.3929</td>
<td>0.02%</td>
<td>0.0539</td>
<td>0.0294</td>
<td>0.0719</td>
<td>__select(4e0ee4)</td>
</tr>
<tr>
<td>2</td>
<td>2.6761</td>
<td>0.01%</td>
<td>1.3380</td>
<td>1.3338</td>
<td>1.3423</td>
<td>kfork(1c95c8)</td>
</tr>
<tr>
<td>207</td>
<td>2.3958</td>
<td>0.01%</td>
<td>0.0116</td>
<td>0.0030</td>
<td>0.1135</td>
<td>__poll(4e0ecc)</td>
</tr>
<tr>
<td>228</td>
<td>1.1583</td>
<td>0.01%</td>
<td>0.0051</td>
<td>0.0011</td>
<td>0.2436</td>
<td>__ioctl(4e07ac)</td>
</tr>
<tr>
<td>9</td>
<td>0.8136</td>
<td>0.00%</td>
<td>0.0904</td>
<td>0.0042</td>
<td>0.0988</td>
<td>__smtcheckinit(1b245a8)</td>
</tr>
<tr>
<td>5</td>
<td>0.5437</td>
<td>0.00%</td>
<td>0.1087</td>
<td>0.0696</td>
<td>0.1777</td>
<td>open(4e08d8)</td>
</tr>
<tr>
<td>15</td>
<td>0.3553</td>
<td>0.00%</td>
<td>0.0237</td>
<td>0.0120</td>
<td>0.0322</td>
<td>__smtcheckinit(1b245cc)</td>
</tr>
<tr>
<td>2</td>
<td>0.2692</td>
<td>0.00%</td>
<td>0.1346</td>
<td>0.1339</td>
<td>0.1353</td>
<td>statx(4e0950)</td>
</tr>
<tr>
<td>33</td>
<td>0.2350</td>
<td>0.00%</td>
<td>0.0071</td>
<td>0.0009</td>
<td>0.0210</td>
<td>__sigaction(1c1d4)</td>
</tr>
<tr>
<td>1</td>
<td>0.1999</td>
<td>0.00%</td>
<td>0.1999</td>
<td>0.1999</td>
<td>0.1999</td>
<td>kwaitpid(1cab64)</td>
</tr>
<tr>
<td>102</td>
<td>0.1954</td>
<td>0.00%</td>
<td>0.0019</td>
<td>0.0013</td>
<td>0.0178</td>
<td>klseek(425a48)</td>
</tr>
</tbody>
</table>

...(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 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.

<table>
<thead>
<tr>
<th>Accumulated Time (msec)</th>
<th>SVC (Address)</th>
<th>Procmame (PId Tid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0656</td>
<td>_select(4e0ee4)</td>
<td>sendmail(7844 5001)</td>
</tr>
<tr>
<td>0.0452</td>
<td>_select(4e0ee4)</td>
<td>syslogd(7514 8591)</td>
</tr>
<tr>
<td>0.0712</td>
<td>_select(4e0ee4)</td>
<td>snmpd(5426 9293)</td>
</tr>
<tr>
<td>0.0156</td>
<td>ioctl(4e07ac)</td>
<td>trcstop(47210 18379)</td>
</tr>
<tr>
<td>0.0274</td>
<td>kwaitpid(1cab64)</td>
<td>ksh(20276 44359)</td>
</tr>
<tr>
<td>0.0567</td>
<td>kread(4259e8)</td>
<td>ksh(23342 50873)</td>
</tr>
</tbody>
</table>

...(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.

Procmame (PId Tid) The name of the process associated with the thread that made the system call, its process ID, and the thread ID.

Hypervisor Calls Summary
The Hypervisor Calls Summary provides a list of all the hypervisor 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 hypervisor call.

<table>
<thead>
<tr>
<th>Count</th>
<th>Total Time (msec)</th>
<th>% sys Time</th>
<th>Avg Time (msec)</th>
<th>Min Time (msec)</th>
<th>Max Time (msec)</th>
<th>HCALL (Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.0077</td>
<td>0.00%</td>
<td>0.0019</td>
<td>0.0014</td>
<td>0.0025</td>
<td>H_XIRR(3ada19c)</td>
</tr>
<tr>
<td>4</td>
<td>0.0070</td>
<td>0.00%</td>
<td>0.0017</td>
<td>0.0015</td>
<td>0.0021</td>
<td>H_EOI(3ad6564)</td>
</tr>
</tbody>
</table>

The Hypervisor Calls Summary has the following fields:

Count The number of times that a hypervisor call of a certain type has been called during the monitoring period.

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

% sys Time The total CPU time that the system spent processing the hypervisor calls of this type, expressed as a percentage of the total processing time.

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

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

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

The name of the hypervisor call and the kernel address of its caller.

---

**Pending Hypervisor Calls Summary**

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

```
Pending Hypervisor Calls Summary

Accumulated Time (msec) | HCALL (Address) | Procnme (Pid Tid)
------------------------|----------------|------------------
0.0066 | H_XIRR(3ada19c) | syncd(3916 5981)
```

The Pending Hypervisor Calls Summary has the following fields:

- **Accumulated Time (msec)**: The accumulated CPU time that the system spent processing the pending hypervisor call, expressed in milliseconds.
- **HCALL (address)**: The name of the hypervisor call and the kernel address of its caller.
- **Procnme (Pid Tid)**: The name of the process associated with the thread that made the hypervisor call, its process ID, and the thread ID.

---

**System NFS Calls Summary**

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

```
System NFS Calls Summary

Count | Total Time (msec) | Ave Time (msec) | Min Time (msec) | Max Time (msec) | % Tot Time | % Tot Count | Opcode
------|-------------------|----------------|-----------------|-----------------|------------|-------------|------------------------
253   | 48.4115           | 0.1913         | 0.0952          | 1.0097          | 98.91      | 98.83       | RFS2_READLINK
2     | 0.3959            | 0.1980         | 0.1750          | 0.2209          | 0.81       | 0.78        | RFS2_LOOKUP
1     | 0.1373            | 0.1373         | 0.1373          | 0.1373          | 0.28       | 0.39        | RFS2_NULL

256   | 48.9448           | 0.1912         |                 |                 |            |             | NFS V2 TOTAL

3015  | 4086.9121         | 1.3555         | 0.1035          | 31.6976         | 40.45      | 17.12       | RFS3_READ
145   | 2296.3158         | 15.8367        | 1.1177          | 42.9125         | 22.73      | 0.82        | RFS3_WRITE
10525 | 2263.3336         | 0.2150         | 0.0547          | 2.9737          | 22.40      | 59.77       | RFS3_LOOKUP
373   | 777.2854          | 2.0839         | 0.2839          | 17.5724         | 7.69       | 2.12        | RFS3_READDIRPLUS
2058  | 385.9510          | 0.1875         | 0.0875          | 1.1993          | 3.82       | 11.69       | RFS3_GETATTR
942   | 178.6442          | 0.1896         | 0.0554          | 1.2320          | 1.77       | 5.35        | RFS3_ACCESS
515   | 97.0297           | 0.1884         | 0.0659          | 0.9774          | 0.96       | 2.92        | RFS3_READDIR
25    | 11.3046           | 0.4522         | 0.2364          | 0.9712          | 0.11       | 0.14        | RFS3_READDIR
3     | 2.8648            | 0.9549         | 0.8939          | 0.9936          | 0.03       | 0.02        | RFS3_CREATE
2     | 2.8590            | 0.9530         | 0.5831          | 1.4095          | 0.03       | 0.02        | RFS3_COMMIT
1     | 1.1824            | 0.5912         | 0.2796          | 0.9028          | 0.01       | 0.01        | RFS3_FSSTAT
1     | 0.2773            | 0.2773         | 0.2773          | 0.2773          | 0.00       | 0.01        | RFS3_SETHTR
1     | 0.2366            | 0.2366         | 0.2366          | 0.2366          | 0.00       | 0.01        | RFS3_PATHCONF
1     | 0.1804            | 0.1804         | 0.1804          | 0.1804          | 0.00       | 0.01        | RFS3_NULL

17609 | 10104.3769        | 0.5738         |                 |                 |            |             | NFS V3 TOTAL

105   | 2296.3158         | 15.8367        | 1.1177          | 42.9125         | 22.73      | 0.82        | CLOSE
3025  | 2263.3336         | 0.2150         | 0.0547          | 2.9737          | 22.40      | 59.77       | COMMIT
373   | 777.2854          | 2.0839         | 0.2839          | 17.5724         | 7.69       | 2.12        | CREATE
2058  | 385.9510          | 0.1875         | 0.0875          | 1.1993          | 3.82       | 11.69       | DELEGPURGE
942   | 178.6442          | 0.1896         | 0.0554          | 1.2320          | 1.77       | 5.35        | DELEGRETURN
515   | 97.0297           | 0.1884         | 0.0659          | 0.9774          | 0.96       | 2.92        | GETATTR
25    | 11.3046           | 0.4522         | 0.2364          | 0.9712          | 0.11       | 0.14        | GETFH
3     | 2.8648            | 0.9549         | 0.8939          | 0.9936          | 0.03       | 0.02        | LINK
3     | 2.8590            | 0.9530         | 0.5831          | 1.4095          | 0.03       | 0.02        | LOCK
```
The System NFS Calls Summary has the following fields:

**Count**
The number of times that a certain type of system NFS call (see **Opcode**) has been called during the monitoring period.

**Total Time (msec)**
The total CPU time that the system spent processing system NFS calls of this type, expressed in milliseconds.

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

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

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

**% Tot Time**
The total CPU time that the system spent processing the system NFS calls of this type, expressed as a percentage of the total processing time.

**% Tot Count**
The number of times that a system NFS call of a certain type was made, expressed as a percentage of the total count.

**Opcode**
The name of the system NFS call.

---

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

**Pending NFS Calls Summary**

<table>
<thead>
<tr>
<th>Accumulated Time (msec)</th>
<th>Sequence Number</th>
<th>Proname (Pid Tid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0831</td>
<td>1038711932</td>
<td>nfsvd(1007854 331969)</td>
</tr>
<tr>
<td>0.0833</td>
<td>1038897247</td>
<td>nfsvd(1007854 352459)</td>
</tr>
<tr>
<td>0.0317</td>
<td>1038780652</td>
<td>nfsvd(1007854 413931)</td>
</tr>
<tr>
<td>0.0029</td>
<td></td>
<td>NFS4_ATTRCACHE kbiod(100098 678934)</td>
</tr>
</tbody>
</table>

..(lines omitted)...

The Pending System NFS Calls Summary has the following fields:

**Accumulated Time (msec)**
The accumulated CPU time that the system spent processing the pending system NFS call, expressed in milliseconds.
### Sequence Number
The sequence number represents the transaction identifier (XID) of an NFS operation. It is used to uniquely identify an operation and is used in the RPC call/reply messages. This number is provided instead of the operation name because the name of the operation is unknown until it completes.

### Opcode
The name of pending operation NFS V4.

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

## Pthread Calls Summary
The Pthread Calls Summary provides a list of all the pthread 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 pthread call.

<table>
<thead>
<tr>
<th>Pthread Calls Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>62</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

The Pthread Calls Summary report has the following fields:

- **Count**: The number of times that a pthread call of a certain type has been called during the monitoring period.
- **Total Time (msec)**: The total CPU time that the system spent processing all pthread calls of this type, expressed in milliseconds.
- **% sys time**: The total CPU time that the system spent processing all calls of this type, expressed as a percentage of the total processing time.
- **Avg Time (msec)**: The average CPU time that the system spent processing one pthread call of this type, expressed in milliseconds.
- **Min Time (msec)**: The minimum CPU time the system used to process one pthread call of this type, expressed in milliseconds.
- **Pthread routine**: The name of the routine in the pthread library.

## Pending Pthread Calls Summary
The Pending Pthread Calls Summary provides a list of all the pthread calls that have been executed on the system during the monitoring period but have not completed. The list is sorted by Pid-Ptid.

<table>
<thead>
<tr>
<th>Pending Pthread Calls Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accumulated Time (msec)</td>
</tr>
<tr>
<td>--------------------------</td>
</tr>
<tr>
<td>1990.9400</td>
</tr>
</tbody>
</table>

The Pending Pthread System Calls Summary has the following fields:

- **Accumulated Time (msec)**: The accumulated CPU time that the system spent processing the pending pthread call, expressed in milliseconds.
- **Pthread Routine**: The name of the pthread routine of the libpthreads library.
**Procname (Pid Tid Ptid)**
The name of the process associated with the thread and the pthread which made the pthread call, its process ID, the thread ID and the pthread 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

<table>
<thead>
<tr>
<th>Count</th>
<th>Total Time (msec)</th>
<th>Avg Time (msec)</th>
<th>Min Time (msec)</th>
<th>Max Time (msec)</th>
<th>FLIH Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2183</td>
<td>203.5524</td>
<td>0.0932</td>
<td>0.0041</td>
<td>0.4576</td>
<td>31(DECR_INTR)</td>
</tr>
<tr>
<td>946</td>
<td>102.4195</td>
<td>0.1083</td>
<td>0.0063</td>
<td>0.6590</td>
<td>3(DATA_ACC_PG_FLT)</td>
</tr>
<tr>
<td>12</td>
<td>1.6720</td>
<td>0.1393</td>
<td>0.0828</td>
<td>0.3366</td>
<td>32(QUEUED_INTR)</td>
</tr>
<tr>
<td>1058</td>
<td>183.6655</td>
<td>0.1736</td>
<td>0.0039</td>
<td>0.7001</td>
<td>5(IO_INTR)</td>
</tr>
</tbody>
</table>

### Per CPU FLIH Summary

**CPU Number 0:**

<table>
<thead>
<tr>
<th>Count</th>
<th>Total Time (msec)</th>
<th>Avg Time (msec)</th>
<th>Min Time (msec)</th>
<th>Max Time (msec)</th>
<th>FLIH Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>635</td>
<td>39.8413</td>
<td>0.0627</td>
<td>0.0041</td>
<td>0.4576</td>
<td>31(DECR_INTR)</td>
</tr>
<tr>
<td>936</td>
<td>101.4960</td>
<td>0.1084</td>
<td>0.0063</td>
<td>0.6590</td>
<td>3(DATA_ACC_PG_FLT)</td>
</tr>
<tr>
<td>9</td>
<td>1.3946</td>
<td>0.1550</td>
<td>0.0851</td>
<td>0.3366</td>
<td>32(QUEUED_INTR)</td>
</tr>
<tr>
<td>266</td>
<td>33.4247</td>
<td>0.1257</td>
<td>0.0039</td>
<td>0.4319</td>
<td>5(IO_INTR)</td>
</tr>
</tbody>
</table>

**CPU Number 1:**

<table>
<thead>
<tr>
<th>Count</th>
<th>Total Time (msec)</th>
<th>Avg Time (msec)</th>
<th>Min Time (msec)</th>
<th>Max Time (msec)</th>
<th>FLIH Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.2405</td>
<td>0.0601</td>
<td>0.0517</td>
<td>0.0735</td>
<td>3(DATA_ACC_PG_FLT)</td>
</tr>
<tr>
<td>258</td>
<td>49.2098</td>
<td>0.1907</td>
<td>0.0060</td>
<td>0.5076</td>
<td>5(IO_INTR)</td>
</tr>
<tr>
<td>515</td>
<td>55.3714</td>
<td>0.1075</td>
<td>0.0080</td>
<td>0.3696</td>
<td>31(DECR_INTR)</td>
</tr>
</tbody>
</table>

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 that 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.

<table>
<thead>
<tr>
<th>Count</th>
<th>Total Time (msec)</th>
<th>Avg Time (msec)</th>
<th>Min Time (msec)</th>
<th>Max Time (msec)</th>
<th>Slih Name (Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>7.0434</td>
<td>0.1638</td>
<td>0.0284</td>
<td>0.3763</td>
<td>s_scsiddpin(1a99104)</td>
</tr>
<tr>
<td>1015</td>
<td>42.0601</td>
<td>0.0414</td>
<td>0.0096</td>
<td>0.0913</td>
<td>ssapin(1990490)</td>
</tr>
</tbody>
</table>

**Per CPU Slih Summary**

<table>
<thead>
<tr>
<th>CPU Number 0:</th>
<th>Count</th>
<th>Total Time (msec)</th>
<th>Avg Time (msec)</th>
<th>Min Time (msec)</th>
<th>Max Time (msec)</th>
<th>Slih Name (Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>1.3500</td>
<td>0.1688</td>
<td>0.0209</td>
<td>0.3087</td>
<td>s_scsiddpin(1a99104)</td>
</tr>
<tr>
<td></td>
<td>258</td>
<td>7.9232</td>
<td>0.0307</td>
<td>0.0096</td>
<td>0.0733</td>
<td>ssapin(1990490)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU Number 1:</th>
<th>Count</th>
<th>Total Time (msec)</th>
<th>Avg Time (msec)</th>
<th>Min Time (msec)</th>
<th>Max Time (msec)</th>
<th>Slih Name (Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>1.2685</td>
<td>0.1268</td>
<td>0.0579</td>
<td>0.2818</td>
<td>s_scsiddpin(1a99104)</td>
</tr>
<tr>
<td></td>
<td>248</td>
<td>11.2759</td>
<td>0.0455</td>
<td>0.0138</td>
<td>0.0641</td>
<td>ssapin(1990490)</td>
</tr>
</tbody>
</table>

... (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, the Pending System Calls Summary, the Hypervisor Calls Summary, the Pending Hypervisor Calls Summary, the System NFS Calls Summary, the Pending NFS Calls Summary, and the Pending Pthread Calls Summary.

The additional information in the System Calls Summary, Hypervisor Calls Summary, System NFS Calls Summary, and the Pthread Calls Summary includes the total, average, maximum, and minimum elapsed time that a call was running. The additional information in the Pending System Calls Summary, Pending Hypervisor Calls Summary, Pending NFS Calls Summary, and the Pending Pthread Calls Summary is the accumulated elapsed time for the pending calls. This additional information is present in all the system calls, hypervisor call, NFS call, and pthread call reports: globally, in the process detailed report (-p), the thread detailed report (-t), and the pthread detailed report (-P).

The following is an example of the additional information reported by using the -e flag:

```bash
# curt -e -i trace.r -m trace.rm -n gensyms.out -o curt.out
# cat curt.out
...
```

### System Calls Summary

<table>
<thead>
<tr>
<th>Count</th>
<th>Total</th>
<th>% sys</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Tot</th>
<th>% sys</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>SVC (Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>605</td>
<td>355.447 1.74</td>
<td>0.5875</td>
<td>0.0482</td>
<td>4.5626</td>
<td>31172.7658</td>
<td>51.5252</td>
<td>0.0482</td>
<td>422.2323</td>
<td>kwrite(4259c4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>733</td>
<td>196.3752 0.96</td>
<td>0.2679</td>
<td>0.0042</td>
<td>2.9948</td>
<td>12967.9407</td>
<td>17.6916</td>
<td>0.0042</td>
<td>265.1204</td>
<td>kread(4259e8)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>9.2217 0.053 3.0739</td>
<td>2.8888</td>
<td>3.3418</td>
<td>57.2051</td>
<td>19.0684</td>
<td>4.5475</td>
<td>40.0557</td>
<td>execve(1c95d8)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>7.6031 0.040</td>
<td>0.2000</td>
<td>0.0561</td>
<td>1.6137</td>
<td>12.5002</td>
<td>0.3290</td>
<td>0.0051</td>
<td>3.3120</td>
<td>_load(1c9608)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1244</td>
<td>4.4574 0.023</td>
<td>0.0036</td>
<td>0.0100</td>
<td>0.0143</td>
<td>4.4574</td>
<td>0.0036</td>
<td>0.0010</td>
<td>0.0143</td>
<td>_seek(425a60)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4.3917 0.025</td>
<td>0.0976</td>
<td>0.0248</td>
<td>0.1810</td>
<td>4.6536</td>
<td>0.1036</td>
<td>0.0248</td>
<td>0.3637</td>
<td>access(507860)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3.3929 0.020</td>
<td>0.0539</td>
<td>0.0294</td>
<td>0.0719</td>
<td>5006.0887</td>
<td>79.4617</td>
<td>0.0294</td>
<td>100.4802</td>
<td>_select(4e4e04)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2.6761 0.019</td>
<td>1.3380</td>
<td>1.3380</td>
<td>1.3423</td>
<td>5350.0262</td>
<td>22.7513</td>
<td>0.0754</td>
<td>37.9281</td>
<td>kfork(1c95c8)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>207</td>
<td>2.3958 0.018</td>
<td>0.0160</td>
<td>0.0000</td>
<td>0.1135</td>
<td>4494.9429</td>
<td>21.7146</td>
<td>0.0630</td>
<td>499.1363</td>
<td>poll(44e0ec)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>228</td>
<td>1.1583 0.015</td>
<td>0.0051</td>
<td>0.0011</td>
<td>0.0236</td>
<td>1.1583</td>
<td>0.0051</td>
<td>0.0181</td>
<td>0.2436</td>
<td>ioctl(4e07ac)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0.0136 0.000</td>
<td>0.0004</td>
<td>0.0082</td>
<td>0.0088</td>
<td>4486.7472</td>
<td>499.8608</td>
<td>0.0082</td>
<td>499.8698</td>
<td>smtcheckinit(1b245a8)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.5437 0.000</td>
<td>0.1087</td>
<td>0.0069</td>
<td>0.1777</td>
<td>0.5437</td>
<td>0.1087</td>
<td>0.0696</td>
<td>0.1777</td>
<td>open(4e080b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0.3553 0.000</td>
<td>0.0237</td>
<td>0.0120</td>
<td>0.0222</td>
<td>0.3553</td>
<td>0.0237</td>
<td>0.0120</td>
<td>0.0222</td>
<td>smtcheckinit(1b245cc)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.2692 0.000</td>
<td>0.1314</td>
<td>0.1339</td>
<td>0.0153</td>
<td>0.2692</td>
<td>0.1346</td>
<td>0.1339</td>
<td>0.1353</td>
<td>statx(4e0960)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>0.2350 0.000</td>
<td>0.0009</td>
<td>0.0020</td>
<td>0.0210</td>
<td>0.2350</td>
<td>0.0071</td>
<td>0.0009</td>
<td>0.0210</td>
<td>_sigaction(1a4a8)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.1999 0.000</td>
<td>0.1999</td>
<td>0.1999</td>
<td>0.1999</td>
<td>5019.0588</td>
<td>5019.0588</td>
<td>5019.0588</td>
<td>5019.0588</td>
<td>kwaitpid(1ca6b4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>0.1954 0.000</td>
<td>0.0019</td>
<td>0.0013</td>
<td>0.0178</td>
<td>0.5427</td>
<td>0.0053</td>
<td>0.0013</td>
<td>0.3658</td>
<td>klseek(425a4b)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...[lines omitted]...
The system call, hypervisor call, NFS call, and pthread call reports in the preceding example have the following fields in addition to the default System Calls Summary, Hypervisor Calls Summary, System NFS Calls Summary, and Pthread Calls Summary:

- **Tot ETime (msec)**: The total amount of time from when each instance of the 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 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 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 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 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, and also includes data on errors returned by system calls as shown by the following:

```
# curt -s -i trace.r -m trace.mm -n gensyms.out -o curt.out
# cat curt.out
```

...(lines omitted)
Errors Returned by System Calls

Errors (errno : count : description) returned for System Call: ioctl(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
Total Hypervisor Call Time (ms): 0.077000

Thread System Call Summary

<table>
<thead>
<tr>
<th>Count</th>
<th>Total Time</th>
<th>Avg Time</th>
<th>Min Time</th>
<th>Max Time</th>
<th>SVC (Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(msec)</td>
<td>(msec)</td>
<td>(msec)</td>
<td>(msec)</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>---------------</td>
</tr>
<tr>
<td>69</td>
<td>34.0819</td>
<td>0.4939</td>
<td>0.1666</td>
<td>1.2762</td>
<td>kwrite(169ff8)</td>
</tr>
<tr>
<td>77</td>
<td>12.0026</td>
<td>0.1559</td>
<td>0.0474</td>
<td>0.2889</td>
<td>kread(16a01c)</td>
</tr>
<tr>
<td>510</td>
<td>4.9743</td>
<td>0.0098</td>
<td>0.0029</td>
<td>0.0467</td>
<td>times(f1e14)</td>
</tr>
<tr>
<td>73</td>
<td>1.2045</td>
<td>0.0165</td>
<td>0.0105</td>
<td>0.0306</td>
<td>select(1d1704)</td>
</tr>
<tr>
<td>68</td>
<td>0.6000</td>
<td>0.0088</td>
<td>0.0023</td>
<td>0.0445</td>
<td>llseek(16a094)</td>
</tr>
<tr>
<td>12</td>
<td>0.1516</td>
<td>0.0126</td>
<td>0.0071</td>
<td>0.0241</td>
<td>getrusage(f1be0)</td>
</tr>
</tbody>
</table>

No Errors Returned by System Calls

Pending System Calls Summary

Accumulated SVC (Address)
Time (msec)

| 0.1420 | kread(16a01c) |

Thread Hypervisor Calls Summary

<table>
<thead>
<tr>
<th>Count</th>
<th>Total Time</th>
<th>% sys Avg Time</th>
<th>% sys Min Time</th>
<th>% sys Max Time</th>
<th>HCALL (Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(msec)</td>
<td>time (msec)</td>
<td>time (msec)</td>
<td>time (msec)</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>---------------</td>
<td>---------------</td>
<td>---------------</td>
<td>----------------</td>
</tr>
<tr>
<td>4</td>
<td>0.0077</td>
<td>0.00%</td>
<td>0.0019</td>
<td>0.0014</td>
<td>H_XIRR(3ada19c)</td>
</tr>
</tbody>
</table>

Pending Hypervisor Calls Summary

Accumulated HCALL (Address)
Time (msec)

| 0.0025 | H_XIRR(3ada19c) |
processor affinity: 0.58333

Dispatch Histogram for thread (CPUid : times_dispatched).

<table>
<thead>
<tr>
<th>CPU</th>
<th>Total Count</th>
<th>Total Time (msec)</th>
<th>Avg Time (msec)</th>
<th>Min Time (msec)</th>
<th>Max Time (msec)</th>
<th>% Total Time</th>
<th>Total ETTime (msec)</th>
<th>Avg ETTime (msec)</th>
<th>Min ETTime (msec)</th>
<th>Max ETTime (msec)</th>
<th>% Total ETTime</th>
<th>% Total Count</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>243</td>
<td>279.934</td>
<td>1.1521</td>
<td>0.0523</td>
<td>0.0131</td>
<td>10.00</td>
<td>279.934</td>
<td>1.1521</td>
<td>0.0523</td>
<td>0.0131</td>
<td>10.00</td>
<td>279.934</td>
<td>100.00</td>
</tr>
</tbody>
</table>

... (lines omitted)...

If the thread belongs to an NFS kernel process, the report will include information on NFS operations instead of System calls:

Report for Thread Id: 1966273 (hex 1e0000) Pid: 1007584 (hex f6000)
Process Name: nfsd

---

Total Kernel Time (ms): 3.198998
Total Operation Time (ms): 28.539927
Total Hypervisor Call Time (ms): 0.000000
---

NFS Calls Summary

Pending NFS Calls Summary
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 Summary**  A system call summary for the thread; this has the same fields as the global System Calls 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.

**Thread Hypervisor Calls Summary**  The hypervisor call summary for the thread; this has the same fields as the global Hypervisor Calls Summary. It also includes elapsed time if the -e flag is specified.

**Pending Hypervisor Calls Summary**  If the thread was executing a hypervisor call at the end of the trace, a pending hypervisor call summary will be printed. This has the Accumulated Time and Hypervisor Call fields. It also includes elapsed time if the -e flag is specified.

**Thread NFS Calls Summary**  An NFS call summary for the thread. This has the same fields as the global System NFS Call Summary. It also includes elapsed time if the -e flag is specified.

**Pending NFS Calls Summary**  If the thread was executing an NFS call at the end of the trace, a pending NFS call summary will be printed. This has the Accumulated Time and Sequence Number or, in the case of NFS V4, Opcode, 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 rediscaches).

**total number of rediscaches due to interrupts being disabled**  The number of rediscaches 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 undispached 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**

The report generated with the -p flag includes the data shown in the default report and also includes a detailed report for each process that includes the Process ID and name, a count and list of the thread IDs, and the count and list of the pthread IDs belonging to the process. The total application time, the system...
call time, and the application time details for all the threads of the process are given. Lastly, it includes summary reports of all the completed and pending system calls, and pthread calls for the threads of the process.

The following example shows the report generated for the router process (PID 129190):

Process Details for Pid: 129190

Process Name: router

7 Tids for this Pid: 245889 245631 244599 82843 78701 75347 28941
9 Ptids for this Pid: 2057 1800 1543 1286 1029 772 515 258 1

Total Application Time (ms): 124.023749
Total System Call Time (ms): 8.948695
Total Hypervisor Time (ms): 0.000000

Application time details:
- Total Pthread Call Time (ms): 1.228271
- Total Pthread Dispatch Time (ms): 2.760476
- Total Pthread Idle Dispatch Time (ms): 0.110307
- Total Other Time (ms): 798.545446
- Total number of pthread dispatches: 53
- Total number of pthread idle dispatches: 3

Process System Calls Summary
---------------------------------.

<table>
<thead>
<tr>
<th>Count</th>
<th>Total Time (msec)</th>
<th>% sys time</th>
<th>Avg Time (msec)</th>
<th>Min Time (msec)</th>
<th>Max Time (msec)</th>
<th>SVC (Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>93</td>
<td>3.6829</td>
<td>0.05%</td>
<td>0.0396</td>
<td>0.0060</td>
<td>0.3077</td>
<td>kread(19731c)</td>
</tr>
<tr>
<td>23</td>
<td>2.2395</td>
<td>0.03%</td>
<td>0.0974</td>
<td>0.0090</td>
<td>0.4537</td>
<td>kwrite(1972f8)</td>
</tr>
<tr>
<td>30</td>
<td>0.8885</td>
<td>0.01%</td>
<td>0.0029</td>
<td>0.0073</td>
<td>0.0460</td>
<td>select(208c5c)</td>
</tr>
<tr>
<td>1</td>
<td>0.5933</td>
<td>0.01%</td>
<td>0.5933</td>
<td>0.5933</td>
<td>0.5933</td>
<td>fsync(1972a4)</td>
</tr>
<tr>
<td>106</td>
<td>0.4902</td>
<td>0.01%</td>
<td>0.0046</td>
<td>0.0035</td>
<td>0.0105</td>
<td>kiseg(1973c)</td>
</tr>
<tr>
<td>13</td>
<td>0.3285</td>
<td>0.00%</td>
<td>0.0025</td>
<td>0.0130</td>
<td>0.0387</td>
<td>semct1(2089e0)</td>
</tr>
<tr>
<td>6</td>
<td>0.2513</td>
<td>0.00%</td>
<td>0.0041</td>
<td>0.0038</td>
<td>0.0650</td>
<td>semop(2089c8)</td>
</tr>
<tr>
<td>3</td>
<td>0.1223</td>
<td>0.00%</td>
<td>0.0040</td>
<td>0.0127</td>
<td>0.0730</td>
<td>stat(2086d4)</td>
</tr>
<tr>
<td>1</td>
<td>0.0793</td>
<td>0.00%</td>
<td>0.0793</td>
<td>0.0793</td>
<td>0.0793</td>
<td>send(1le)</td>
</tr>
<tr>
<td>9</td>
<td>0.0679</td>
<td>0.00%</td>
<td>0.0075</td>
<td>0.0053</td>
<td>0.0147</td>
<td>fstat(2086c8)</td>
</tr>
<tr>
<td>4</td>
<td>0.0524</td>
<td>0.00%</td>
<td>0.0131</td>
<td>0.0023</td>
<td>0.0348</td>
<td>kcont(22aa14)</td>
</tr>
<tr>
<td>5</td>
<td>0.0448</td>
<td>0.00%</td>
<td>0.0090</td>
<td>0.0086</td>
<td>0.0096</td>
<td>yield(11ddec)</td>
</tr>
<tr>
<td>3</td>
<td>0.0444</td>
<td>0.00%</td>
<td>0.0148</td>
<td>0.0049</td>
<td>0.0219</td>
<td>recv(1le)</td>
</tr>
<tr>
<td>1</td>
<td>0.0355</td>
<td>0.00%</td>
<td>0.0355</td>
<td>0.0355</td>
<td>0.0355</td>
<td>open(208674)</td>
</tr>
<tr>
<td>1</td>
<td>0.0281</td>
<td>0.00%</td>
<td>0.0281</td>
<td>0.0281</td>
<td>0.0281</td>
<td>close(19728c)</td>
</tr>
</tbody>
</table>

Pending System Calls Summary
---------------------------------.

<table>
<thead>
<tr>
<th>Accumulated Time (msec)</th>
<th>SVC (Address)</th>
<th>Tid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0452</td>
<td>select(208c5c)</td>
<td>245889</td>
</tr>
<tr>
<td>0.0425</td>
<td>select(208c5c)</td>
<td>78701</td>
</tr>
<tr>
<td>0.0285</td>
<td>select(208c5c)</td>
<td>82843</td>
</tr>
<tr>
<td>0.0284</td>
<td>select(208c5c)</td>
<td>245613</td>
</tr>
<tr>
<td>0.0274</td>
<td>select(208c5c)</td>
<td>244599</td>
</tr>
<tr>
<td>0.0179</td>
<td>select(208c5c)</td>
<td>75347</td>
</tr>
</tbody>
</table>

...(omitted lines)...

Pthread Calls Summary
---------------------------------.

88  AIX Version 6.1 Performance Tools Guide and Reference
If the process is an NFS kernel process, the report will include information on NFS operations instead of System and Pthread calls:

Process Details for Pid: 1007854
Process Name: nfsd
Tids: 1
Pid: 1007854
...

Process NFS Calls Summary

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Count</th>
<th>Total ETime (msec)</th>
<th>Avg ETime (msec)</th>
<th>Min ETime (msec)</th>
<th>Max ETime (msec)</th>
<th>% Tot</th>
<th>% Tot Count</th>
<th>% Tot Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFS4_ATTRCACHE</td>
<td>21</td>
<td>0.1195</td>
<td>0.0077</td>
<td>0.0021</td>
<td>0.0779</td>
<td>0.0177</td>
<td>0.0017</td>
<td>0.0104</td>
</tr>
<tr>
<td>RFS3_ACCESS</td>
<td>201544</td>
<td>2019547</td>
<td>0.0140</td>
<td>0.0026</td>
<td>0.0122</td>
<td>0.0017</td>
<td>0.0017</td>
<td>0.0104</td>
</tr>
<tr>
<td>RFS3_CREATE</td>
<td>200037</td>
<td>2002645</td>
<td>0.1070</td>
<td>0.0340</td>
<td>0.0025</td>
<td>0.0025</td>
<td>0.0017</td>
<td>0.0104</td>
</tr>
<tr>
<td>NFS4_SERVER</td>
<td>199489</td>
<td>199499</td>
<td>0.0953</td>
<td>0.0120</td>
<td>0.0120</td>
<td>0.0017</td>
<td>0.0017</td>
<td>0.0104</td>
</tr>
<tr>
<td>V3</td>
<td>19241</td>
<td>1928692</td>
<td>0.0890</td>
<td>0.0425</td>
<td>0.0140</td>
<td>0.0140</td>
<td>0.0017</td>
<td>0.0104</td>
</tr>
<tr>
<td>V4</td>
<td>19241</td>
<td>1928692</td>
<td>0.0425</td>
<td>0.0210</td>
<td>0.0077</td>
<td>0.0077</td>
<td>0.0017</td>
<td>0.0104</td>
</tr>
<tr>
<td>CLIENT</td>
<td>19241</td>
<td>1928692</td>
<td>0.0135</td>
<td>0.0026</td>
<td>0.0122</td>
<td>0.0017</td>
<td>0.0017</td>
<td>0.0104</td>
</tr>
</tbody>
</table>

Chapter 3. CPU Utilization Reporting Tool (curt)
The information in the process detailed report includes the following:

**Total Application Time (ms)**  
The amount of time, expressed in milliseconds, that the process spent in application mode.

**Total System Call Time (ms)**  
The amount of time, expressed in milliseconds, that the process spent in system call mode.

The information in the application time details report includes the following:

**Total Pthread Call Time**  
The amount of time, expressed in milliseconds, that the process spent in traced pthread library calls.

**Total Pthread Dispatch Time**  
The amount of time, expressed in milliseconds, that the process spent in libpthreads dispatch code.

**Total Pthread Idle Dispatch Time**  
The amount of time, expressed in milliseconds, that the process spent in libpthreads **vp_sleep** code.

**Total Other Time**  
The amount of time, expressed in milliseconds, that the process spent in non-traced user mode code.

**Total number of pthread dispatches**  
The total number of times a pthread belonging to the process was dispatched by the libpthreads dispatcher.

**Total number of pthread idle dispatches**  
The total number of times a thread belonging to the process was in the libpthreads **vp_sleep** code.

The summary information in the report includes the following:

**Process System Calls Summary**  
A system call summary for the process; this has the same fields as the global System Call Summary. It also includes elapsed time information if the **-e** flag is specified and error information if the **-s** flag is specified.

**Pending System Calls Summary**  
If the process 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 information if the **-e** flag is specified.

**Process Hypervisor Calls Summary**  
A summary of the hypervisor calls for the process; this has the same fields as the global Hypervisor Calls Summary. It also includes elapsed time information if the **-e** flag is specified.

**Pending Hypervisor Calls Summary**  
If the process was executing a hypervisor call at the end of the trace, a pending hypervisor call summary will be printed. This has the Accumulated Time and Hypervisor Call fields. It also includes elapsed time information if the **-e** flag is specified.

**Process NFS Calls Summary**  
An NFS call summary for the process. This has the same fields as the global System NFS Call Summary. It also includes elapsed time information if the **-e** flag is specified.

**Pending NFS Calls Summary**  
If the process was executing an NFS call at the end of the trace, a pending NFS call summary will be printed. This has the **Accumulated Time** and **Sequence Number** or, in the case of NFS V4, **Opcode**, fields. It also includes elapsed time information if the **-e** flag is specified.

**Pthread Calls Summary**  
A summary of the pthread calls for the process. This has the same fields as the global pthread Calls Summary. It also includes elapsed time information if the **-e** flag is specified.

**Pending Pthread Calls Summary**  
If the process was executing pthread library calls at the end of the trace, a pending pthread call summary will be printed. This has the Accumulated Time and Pthread Routine fields. It also includes elapsed time information if the **-e** flag is specified.

**Reports Generated with the -P Flag**
The report generated with the **-P** flag includes the data shown in the default report and also includes a detailed report on pthread status that includes the following:
The amount of time the pthread was in application and system call mode
The application time details
The system calls and pthread calls that the pthread made
The system calls and pthread calls that were pending at the end of the trace
The processor affinity
The number of times the pthread was dispatched
To which CPU(s) the thread was dispatched
The thread affinity
The number of times that the pthread was dispatched
To which kernel thread(s) the pthread was dispatched

The report also includes dispatch wait time and details of interrupts.

The following is an example of a report generated with the -P flag:

Report for Pthread Id: 1 (hex 1) Pid: 245962 (hex 3c0ca)
Process Name: ./pth32
---------------------
Total Application Time (ms): 3.919091
Total System Call Time (ms): 8.303156
Total Hypervisor Call Time (ms): 0.000000

Application time details:
Total Pthread Call Time (ms): 1.139372
Total Pthread Dispatch Time (ms): 0.115822
Total Pthread Idle Dispatch Time (ms): 0.036630
Total Other Time (ms): 2.627266

Pthread System Calls Summary
------------------------------------------
<table>
<thead>
<tr>
<th>Count</th>
<th>Total Time (msec)</th>
<th>Avg Time (msec)</th>
<th>Min Time (msec)</th>
<th>Max Time (msec)</th>
<th>SVC (Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.3988</td>
<td>3.3988</td>
<td>3.3988</td>
<td>3.3988</td>
<td>_exit(409e50)</td>
</tr>
<tr>
<td>61</td>
<td>0.8138</td>
<td>0.0113</td>
<td>0.0089</td>
<td>0.0254</td>
<td>kread(5ffdf78)</td>
</tr>
<tr>
<td>11</td>
<td>0.4616</td>
<td>0.0420</td>
<td>0.0262</td>
<td>0.0835</td>
<td>thread_create(407360)</td>
</tr>
<tr>
<td>22</td>
<td>0.2570</td>
<td>0.0117</td>
<td>0.0062</td>
<td>0.0373</td>
<td>mprotect(6d5bd8)</td>
</tr>
<tr>
<td>12</td>
<td>0.2126</td>
<td>0.0177</td>
<td>0.0100</td>
<td>0.0324</td>
<td>thread_setstate(40a660)</td>
</tr>
<tr>
<td>115</td>
<td>0.1875</td>
<td>0.0016</td>
<td>0.0012</td>
<td>0.0037</td>
<td>klseek(5ffe38)</td>
</tr>
<tr>
<td>12</td>
<td>0.1061</td>
<td>0.0088</td>
<td>0.0032</td>
<td>0.0134</td>
<td>sbbrk(6d4f90)</td>
</tr>
<tr>
<td>23</td>
<td>0.0803</td>
<td>0.0035</td>
<td>0.0018</td>
<td>0.0072</td>
<td>trcgent(4078d8)</td>
</tr>
</tbody>
</table>

...(lines omitted)...

Pending System Calls Summary
-------------------------------
| Accumulated SVC (Address) |
|---------------------------|-------------------------|
| 0.0141                    | thread_tsleep(40a4f8)   |

Pthread Calls Summary
----------------------
<table>
<thead>
<tr>
<th>Count</th>
<th>Total Time (msec)</th>
<th>% sys time</th>
<th>Avg Time (msec)</th>
<th>Min Time (msec)</th>
<th>Max Time (msec)</th>
<th>Pthread Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0.9545</td>
<td>0.01%</td>
<td>0.0868</td>
<td>0.0457</td>
<td>0.1833</td>
<td>pthread_create</td>
</tr>
<tr>
<td>8</td>
<td>0.0725</td>
<td>0.00%</td>
<td>0.0091</td>
<td>0.0064</td>
<td>0.0295</td>
<td>pthread_join</td>
</tr>
<tr>
<td>1</td>
<td>0.0553</td>
<td>0.00%</td>
<td>0.0553</td>
<td>0.0553</td>
<td>0.0553</td>
<td>pthread_detach</td>
</tr>
</tbody>
</table>
The information in the pthreads summary report includes the following:

**Pthread ID**
The Pthread ID of the thread.

**Process ID**
The Process ID that the pthread belongs to.

**Process Name**
The process name, if known, that the pthread belongs to.

**Total Application Time (ms)**
The amount of time, expressed in milliseconds, that the pthread spent in application mode.

**Total System Call Time (ms)**
The amount of time, expressed in milliseconds, that the pthread spent in system call mode.

The information in the application time details report includes the following:

**Total Pthread Call Time**
The amount of time, expressed in milliseconds, that the pthread spent in traced pthread library calls.

**Total Pthread Dispatch Time**
The amount of time, expressed in milliseconds, that the pthread spent in libpthreads dispatch code.
### Total Pthread Idle Dispatch Time
The amount of time, expressed in milliseconds, that the pthread spent in libpthread's `vp_sleep` code.

### Total Other Time
The amount of time, expressed in milliseconds, that the pthread spent in non-traced user mode code.

### Total number of pthread dispatches
The total number of times a pthread belonging to the process was dispatched by the libpthread dispatcher.

### Total number of pthread idle dispatches
The total number of times a thread belonging to the process was in the libpthread's `vp_sleep` code.

The summary information in the report includes the following:

#### Pthread System Calls Summary
A system call summary for the pthread; this has the same fields as the global System Call Summary. It also includes elapsed time information if the `-e` flag is specified and error information if the `-s` flag is specified.

#### Pending System Calls Summary
If the pthread 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 information if the `-e` flag is specified.

#### Pthread Hypervisor Calls Summary
A summary of the hypervisor calls for the pthread. This has the same fields as the global hypervisor calls summary. It also includes elapsed time information if the `-e` flag is specified.

#### Pending Hypervisor Calls Summary
If the pthread was executing a hypervisor call at the end of the trace, a pending hypervisor calls summary will be printed. This has the Accumulated Time and Hypervisor Call fields. It also includes elapsed time information if the `-e` flag is specified.

#### Pthread Calls Summary
A summary of the pthread library calls for the pthread. This has the same fields as the global pthread Calls Summary. It also includes elapsed time information if the `-e` flag is specified.

#### Pending Pthread Calls Summary
If the pthread was executing a pthread library call at the end of the trace, a pending pthread call summary will be printed. This has the Accumulated Time and Pthread Routine fields. It also includes elapsed time information if the `-e` flag is specified.

The pthreads summary report also includes the following information:

- **processor affinity**: Probability that for any dispatch of the pthread, the pthread was dispatched to the same processor on which it last executed.
- **Processor Dispatch Histogram for pthread**: The number of times that the pthread was dispatched to each CPU in the system.
- **avg. dispatch wait time**: The average elapsed time for the pthread from being undispatched and its next dispatch.
- **Thread affinity**: The probability that for any dispatch of the pthread, the pthread was dispatched to the same kernel thread on which it last executed.
- **Thread Dispatch Histogram for pthread**: The number of times that the pthread was dispatched to each kernel thread in the process.
- **total number of pthread dispatches**: The total number of times the pthread was dispatched by the libpthread dispatcher.
- **Data on Interrupts that occurred while Pthread was Running**: The number of times each type of FLIH occurred while the pthread was executing.
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 [-h] [topic]

splat [-j]

Flags

- `-i inputfile` Specifies the AIX trace log file input.
- `-n namefile` Specifies the file containing output of the gensyms command.
- `-o outputfile` Specifies an output file (default is stdout).
- `-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.
- `-p` Specifies the use of the PURR register to calculate CPU times.

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 gensyms command.
- `outputfile` File to write reports to.
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** or **gensyms** 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:

<table>
<thead>
<tr>
<th>Hook ID</th>
<th>Event name</th>
<th>Event explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>106</td>
<td>HKWD_KERN_DISPATCH</td>
<td>The thread is dispatched from the run queue to a processor.</td>
</tr>
<tr>
<td>10C</td>
<td>HKWD_KERN_IDLE</td>
<td>The idle process is been dispatched.</td>
</tr>
<tr>
<td>10E</td>
<td>HKWD_KERN_RELOCK</td>
<td>One thread is suspended while another is dispatched; the ownership of a <strong>RunQ</strong> lock is transferred from the first to the second.</td>
</tr>
<tr>
<td>Hook ID</td>
<td>Event name</td>
<td>Event explanation</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>112</td>
<td>HKWD_KERN_LOCK</td>
<td>The thread attempts to secure a kernel lock; the sub-hook shows what happened.</td>
</tr>
<tr>
<td>113</td>
<td>HKWD_KERN_UNLOCK</td>
<td>A kernel lock is released.</td>
</tr>
<tr>
<td>134</td>
<td>HKWD_SYSC_EXECVE</td>
<td>An exec supervisor call (SVC) has been issued by a (forked) process.</td>
</tr>
<tr>
<td>139</td>
<td>HKWD_SYSC_FORK</td>
<td>A fork SVC has been issued by a process.</td>
</tr>
<tr>
<td>419</td>
<td>HKWD_CPU_PREEMPT</td>
<td>A process has been preempted.</td>
</tr>
<tr>
<td>465</td>
<td>HKWD_SYSC_CRTHREAD</td>
<td>A thread_create SVC has been issued by a process.</td>
</tr>
<tr>
<td>46D</td>
<td>HKWD_KERN_WAITLOCK</td>
<td>The thread is enqueued to wait on a kernel lock.</td>
</tr>
<tr>
<td>46E</td>
<td>HKWD_KERN_WAKEUPLOCK</td>
<td>A thread has been awakened.</td>
</tr>
<tr>
<td>606</td>
<td>HKWD_PTHREAD_COND</td>
<td>Operations on a Condition Variable.</td>
</tr>
<tr>
<td>607</td>
<td>HKWD_PTHREAD_MUTEX</td>
<td>Operations on a Mutex.</td>
</tr>
<tr>
<td>608</td>
<td>HKWD_PTHREAD_RWLOCK</td>
<td>Operations on a Read/Write Lock.</td>
</tr>
<tr>
<td>609</td>
<td>HKWD_PTHREAD_GENERAL</td>
<td>Operations on a PThread.</td>
</tr>
</tbody>
</table>

**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` flags 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:
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 might want to re-generate the trace using larger buffers and re-run splat.

Some versions of the AIX kernel or PThread library might be incompletely instrumented, so the traces will be missing events. The splat command might 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 gensyms command. The gensyms command also captures the contents of the /usr/include/sys/lockname.h file, which declares classes of kernel locks.

The gensyms 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 -p -sa -da -S100 -i trace.cooked -n gensyms -o splat.out
Trace Cmd: trace -C all -aj 600,603,606,607,608,609 -T 20000000 -L 200000000 -o CONDVAR.raw
Trace Host: darkwing (0054451E4C00) AIX 5.2
Trace Date: Thu Sep 27 11:26:16 2002
PURR was used to calculate CPU times.

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

------------------------  ------------------------
 start                     stop
From the example above, you can see that 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.
- A sentence specifying whether the PURR register was used to calculate CPU times.
- 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.

<table>
<thead>
<tr>
<th></th>
<th>Total Addresses</th>
<th>Acquisitions (or Passes)</th>
<th>Acq. or Passes per Second</th>
<th>Total System Spin Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX (all) Locks:</td>
<td>523</td>
<td>1323045</td>
<td>72175.7768</td>
<td>0.003986</td>
</tr>
<tr>
<td>RunQ:</td>
<td>2</td>
<td>487178</td>
<td>26576.9121</td>
<td>0.000000</td>
</tr>
<tr>
<td>Simple:</td>
<td>480</td>
<td>824898</td>
<td>45000.4754</td>
<td>0.003986</td>
</tr>
<tr>
<td>Transformed:</td>
<td>22</td>
<td>234</td>
<td>352.3452</td>
<td></td>
</tr>
<tr>
<td>Krlock:</td>
<td>50</td>
<td>234</td>
<td>32.6548</td>
<td>0.000458</td>
</tr>
<tr>
<td>Complex:</td>
<td>41</td>
<td>10969</td>
<td>598.3894</td>
<td>0.000000</td>
</tr>
<tr>
<td>PThread CondVar:</td>
<td>7</td>
<td>160623</td>
<td>8762.4305</td>
<td>0.000000</td>
</tr>
<tr>
<td>Mutex:</td>
<td>128</td>
<td>1927771</td>
<td>105165.2585</td>
<td>10.280745 *</td>
</tr>
<tr>
<td>RWLock:</td>
<td>0</td>
<td>0</td>
<td>0.0000</td>
<td>0.000000</td>
</tr>
</tbody>
</table>

*(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 might 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. The Transformed value represents the number of different simple locks responsible for the allocation (and liberation) of at least one Krlock. In this case, two simple locks will be different if they are not created at the same time or they do not have the same address.

**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; the `splat` command produces a separate lock detail report for each instance because the usage might be different. The Transformed value represents the number of different simple locks responsible for the allocation (and liberation) of at least one Krlock. In this case, simple locks created at different times but with the same address increment the counter only once.

**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. The Transformed value represents the number of acquisitions made by a thread holding the corresponding Krlock.

**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. The Transformed value represents the acquisition rate for the acquisitions made by threads holding the corresponding krlock.

**% Total System spin Time**
The cumulative time spent spinning on each synchronizer type, divided by the cumulative 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 asterisk (*). For simple locks, the spin time of the Krlocks is included.

### 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:

<table>
<thead>
<tr>
<th>Lock Names, Class, or Address</th>
<th>T Acquisitions</th>
<th>Wait Transitions</th>
<th>Locks or Passes</th>
<th>Percent Holdtime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>or Passes</td>
<td>or Spins</td>
<td>Locks</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|                  | 000000002FF22870 | 368638          | N/A             | 0.0000           | 100.000          | 9622.964         | 99.9865          | 99.9865          |
|                  | 000000000F00C3704 | 160625           | 0               | 0.0000           | 14.2831          | 8762.540         | 99.7702          | 99.7702          |
|                  | 000000000F00C3750 | 160625           | 175              | 0.1088           | 14.2831          | 8762.540         | 42.9371          | 42.9371          |
|                  | 000000000F00C3800 | 160623           | 624              | 0.0000           | 100.000          | 1271.728         | N/A              | N/A              |
|                  | 000000000F00C3750 | 37               | 0               | 0.0000           | 0.0033           | 2.018            | 0.0037           | 0.0037           |
|                  | 000000000F00C3800 | 30               | 0               | 0.0000           | 0.0027           | 1.637            | 0.0698           | 0.0698           |

(... lines omitted ...)
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**: An enabled simple kernel lock
- **D**: A disabled 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 that the lock (or condition-variable) was spun on during the analysis interval.

**Wait or Transform**

The number of times that a thread was driven into a wait state for that lock or condition-variable during the analysis interval. When Krlocks are enabled, a simple lock never enters the wait state and this value represents the number of Krlocks that the simple lock has allocated, which is the transform count of simple locks.

**%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.

**Percent Holdtime**

**Real CPU**

The percentage of the cumulative processor 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.

**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.

**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.

**Disabled 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` or `gensyms` output file permits, 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.

```
[AIX SIMPLE Lock] ADDRESS: 0000000200000060 KEX: unknown
```

### Lock Activity (mSecs) - Interrupts Disabled

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Acquisitions</th>
<th>Miss Rate</th>
<th>Spin Rate</th>
<th>Transf. Rate</th>
<th>Busy Rate</th>
<th>Percent Held of Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>.dispatch</td>
<td>3177</td>
<td>0.63</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>.setrq</td>
<td>3160</td>
<td>0.19</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>.steal_threads</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>.dispatch</td>
<td>535</td>
<td>2.19</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>.priq_reque</td>
<td>7</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>.setnowq</td>
<td>4</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

The `simple_lock_report` command prints a report of lock activity. The report contains statistics such as the number of lock acquisitions, the rate of acquisition, the busy time, and the percentage of time locks were held. The report is sorted by the class of the lock and includes fields such as `Type`, `Rate`, `Count`, `CPU Elapsed`, `Spin Wait`, `Depth`, and `Held`. The report is useful for analyzing the performance of locks in AIX systems.

The `simple_lock_report` command can be used with various options to filter the output. The `--class` option allows specifying a particular class of locks to report on. The `--sort` option can be used to sort the output by different criteria, such as the rate or the CPU time.

The `simple_lock_report` command is part of the AIX Performance Tools suite, which provides tools for monitoring and diagnosing system performance. The tool is designed to help system administrators identify performance bottlenecks and optimize the performance of AIX systems.

The `simple_lock_report` command is particularly useful for analyzing the performance of locks in AIX systems. The report provides detailed information about the activity of locks, including the number of lock acquisitions, the rate of acquisition, the busy time, and the percentage of time locks were held. The report is sorted by the class of the lock and includes fields such as `Type`, `Rate`, `Count`, `CPU Elapsed`, `Spin Wait`, `Depth`, and `Held`. The report is useful for analyzing the performance of locks in AIX systems.
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.

Total Acquisitions
The number of times that the lock was acquired in the analysis interval. This includes successful `simple_lock_try` calls.

**Acq. holding krlock**
The number of acquisitions made by threads holding a Krlock.

**Transform count**
The number of Krlocks that have been used (allocated and freed) by the simple lock.

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

**Krlocks SpinQ**
The minimum, maximum, and average number of threads spinning on a Krlock allocated by the simple lock, across the analysis interval.

**PROD**
The associated Krlocks `prod` calls count.

**CONFER SELF**
The confer to self calls count for the simple lock and the associated Krlocks.

**CONFER TARGET**
The confer to target calls count for the simple lock and the associated Krlocks

**CONFER ALL**
The confer to all calls count for the simple lock and the associated Krlocks.

**HANDOFF**
The associated Krlocks `handoff` calls count.

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.

**LOCK with KRLOCK**
The thread has successfully acquired the lock, while holding the associated Krlock, and is currently executing.

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

**KRLOCK LOCK**
The thread has successfully acquired the associated Krlock and is currently executing.

**KRLOCK SPIN**
The thread is executing and unsuccessfully attempting to acquire the associated Krlock.

**TRANSFORM**
The thread has successfully allocated a Krlock that it associates itself to and is executing.
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 might report that the number of times that the lock state was entered might 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 might 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.

### Enabled Simple Lock Details

The following example is an enabled simple lock detail report:

```
[Proc simple lock]  CLASS:  PROC_INT_CLASS.00000004
ADDRESS:  0000000000000000

<table>
<thead>
<tr>
<th>Name</th>
<th>Acquisitions</th>
<th>Miss</th>
<th>Spin</th>
<th>Wait</th>
<th>Busy</th>
<th>Percent Held of Total Time</th>
<th>Return Address</th>
<th>Start Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>.dispatch</td>
<td>3177</td>
<td>0.63</td>
<td>0.37</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.setrq</td>
<td>3160</td>
<td>0.31</td>
<td>0.19</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.steal_threads</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.steal_threads</td>
<td>6</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.dispatch</td>
<td>3152</td>
<td>0.31</td>
<td>0.15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.slept</td>
<td>3140</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.steal_threads</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.steal_threads</td>
<td>6</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.dispatch</td>
<td>3152</td>
<td>0.31</td>
<td>0.15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.slept</td>
<td>3140</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.steal_threads</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.steal_threads</td>
<td>6</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

(... lines omitted ...)

104 AIX Version 6.1 Performance Tools Guide and Reference
The SIMPLE lock report fields are as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
<td>If the simple lock was used with interrupts, this field is enabled. Otherwise, this field is disabled.</td>
</tr>
<tr>
<td><strong>Total Acquisitions</strong></td>
<td>The number of times that the lock was acquired in the analysis interval. This includes successful <code>simple_lock_try</code> calls.</td>
</tr>
<tr>
<td><strong>Miss Rate</strong></td>
<td>The percentage of attempts that failed to acquire the lock.</td>
</tr>
<tr>
<td><strong>Spin Count</strong></td>
<td>The number of unsuccessful attempts to acquire the lock.</td>
</tr>
<tr>
<td><strong>Wait Count</strong></td>
<td>The number of times that a thread was forced into a suspended wait state, waiting for the lock to come available.</td>
</tr>
<tr>
<td><strong>Busy Count</strong></td>
<td>The number of <code>simple_lock_try</code> calls that returned busy.</td>
</tr>
<tr>
<td><strong>Seconds Held</strong></td>
<td>This field contains the following sub-fields:</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>The total number of processor seconds that the lock was held by an executing thread.</td>
</tr>
<tr>
<td><strong>Elapsed</strong></td>
<td>The total number of elapsed seconds that the lock was held by any thread, whether running or suspended.</td>
</tr>
<tr>
<td><strong>Percent Held</strong></td>
<td>This field contains the following sub-fields:</td>
</tr>
<tr>
<td><strong>Real CPU</strong></td>
<td>The percentage of the cumulative processor time that the lock was held by an executing thread.</td>
</tr>
<tr>
<td><strong>Real Elapsed</strong></td>
<td>The percentage of the elapsed real time that the lock was held by any thread at all, either running or suspended.</td>
</tr>
<tr>
<td><strong>Comb(ined) Spin</strong></td>
<td>The percentage of the cumulative processor time that running threads spent spinning while trying to acquire this lock.</td>
</tr>
<tr>
<td><strong>Real Wait</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>SpinQ</strong></td>
<td>The minimum, maximum, and average number of threads spinning on the lock, whether executing or suspended, across the analysis interval.</td>
</tr>
<tr>
<td><strong>WaitQ</strong></td>
<td>The minimum, maximum, and average number of threads waiting on the lock, across the analysis interval.</td>
</tr>
</tbody>
</table>

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 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:

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(no lock reference)</strong></td>
<td>The thread is running, does not hold this lock, and is not attempting to acquire this lock.</td>
</tr>
<tr>
<td><strong>LOCK</strong></td>
<td>The thread has successfully acquired the lock and is currently executing.</td>
</tr>
<tr>
<td><strong>SPIN</strong></td>
<td>The thread is executing and unsuccessfully attempting to acquire the lock.</td>
</tr>
<tr>
<td><strong>UNDISP</strong></td>
<td>The thread has become undispached while unsuccessfully attempting to acquire the lock.</td>
</tr>
<tr>
<td><strong>WAIT</strong></td>
<td>The thread has been suspended until the lock comes available. It does not necessarily acquire the lock at that time, but instead returns to a SPIN state.</td>
</tr>
<tr>
<td><strong>PREEMPT</strong></td>
<td>The thread is holding this lock and has become undispached.</td>
</tr>
</tbody>
</table>
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 might 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 might 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.

**Function Detail**
The function detail report is obtained by using the `-df` or `-da` options of `splat`.

The columns are defined as follows:

<table>
<thead>
<tr>
<th>Function Name</th>
<th>The name of the function that acquired or attempted to acquire this lock, if it could be resolved.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquisitions</td>
<td>The number of times that the function was able to acquire this lock. For complex lock and read/write, there is a distinction between acquisition for writing, <em>Acquisition Write</em>, and for reading, <em>Acquisition Read</em>.</td>
</tr>
<tr>
<td>Miss Rate</td>
<td>The percentage of acquisition attempts that failed.</td>
</tr>
<tr>
<td>Spin Count</td>
<td>The number of unsuccessful attempts by the function to acquire this lock. For complex lock and read/write, there is a distinction between spin count for writing, <em>Spin Count Write</em>, and for reading, <em>Spin Count Read</em>.</td>
</tr>
<tr>
<td>Transf. Count</td>
<td>The number of times that a simple lock has allocated a Krlock, while a thread was trying to acquire the simple lock.</td>
</tr>
<tr>
<td>Busy Count</td>
<td>The number of times simple_lock_try calls returned busy.</td>
</tr>
<tr>
<td>Percent Held of Total Time</td>
<td>Contains the following sub-fields:</td>
</tr>
<tr>
<td></td>
<td><strong>CPU</strong> 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.</td>
</tr>
<tr>
<td></td>
<td><strong>Elapse(d)</strong> 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.</td>
</tr>
<tr>
<td></td>
<td><strong>Spin</strong> 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.</td>
</tr>
<tr>
<td></td>
<td><strong>Wait</strong> 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.</td>
</tr>
<tr>
<td>Return Address</td>
<td>The return address to this calling function, in hexadecimal.</td>
</tr>
<tr>
<td>Start Address</td>
<td>The start address to this calling function, in hexadecimal.</td>
</tr>
<tr>
<td>Offset</td>
<td>The offset from the function start address to the return address, in hexadecimal.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThreadID</td>
<td>The thread identifier.</td>
</tr>
<tr>
<td>Acquisitions</td>
<td>The number of times that this thread acquired the lock.</td>
</tr>
<tr>
<td>Miss Rate</td>
<td>The percentage of acquisition attempts by the thread that failed to secure the lock.</td>
</tr>
<tr>
<td>Spin Count</td>
<td>The number of unsuccessful attempts by this thread to secure the lock.</td>
</tr>
<tr>
<td>Transf. Count</td>
<td>The number of times that a simple lock has allocated a Krlock, while a thread was trying to acquire the simple lock.</td>
</tr>
<tr>
<td>Wait Count</td>
<td>The number of times that this thread was forced to wait until the lock came available.</td>
</tr>
<tr>
<td>Busy Count</td>
<td>The number of <code>simple_lock_try()</code> calls that returned busy.</td>
</tr>
<tr>
<td>Percent Held of Total Time</td>
<td>Consists of the following sub-fields:</td>
</tr>
<tr>
<td></td>
<td>CPU The percentage of the elapsed real time that this thread executed while holding the lock.</td>
</tr>
<tr>
<td></td>
<td>Elapse(d) The percentage of the elapsed real time that this thread held the lock while running or suspended.</td>
</tr>
<tr>
<td></td>
<td>Spin The percentage of elapsed real time that this thread executed while spinning on the lock.</td>
</tr>
<tr>
<td></td>
<td>Wait The percentage of elapsed real time that this thread spent waiting on the lock.</td>
</tr>
<tr>
<td>Process ID</td>
<td>The Process identifier (only for simple and complex lock report).</td>
</tr>
<tr>
<td>Process Name</td>
<td>Name of the process using the lock (only for simple and complex lock report).</td>
</tr>
</tbody>
</table>

**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 might 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 permits 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 might require that the thread wait until other threads release their read-locks. The downgrade operation does not.

A thread might 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 permitted to downgrade it because the downgrade is intended to apply only to 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 down the time 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, the `splat` command 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 `gensyms` 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:

**Parent Thread**  
Phandle id of the parent pthread.

**creation time**  
Elapsed time in seconds after the first event recorded in trace (if available).

**deletion time**  
Elapsed time in seconds after the first event recorded in trace (if available).

**PID**  
Process identifier.

**Process Name**  
Name of the process using the lock.

**Call-chain**  
Stack of called methods (if available).

**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.

- **Elapsed(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 learn 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.

Mutex Pthread Detail
If the -dt or -da options are used, the splat command reports the pthread detail as described below:

PThreadID
The PThread identifier.

Acquisitions
The number of times that this pthread acquired the mutex.

Miss Rate
The percentage of acquisition attempts by the pthread that failed to secure the mutex.

Spin Count
The number of unsuccessful attempts by this pthread to secure the mutex.

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

Busy Count
The number of trylock calls 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 pthread executed while holding the mutex.

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

Spin
The percentage of elapsed real time that this pthread executed while spinning on the mutex.

Wait
The percentage of elapsed real time that this pthread spent waiting on the mutex.

Mutex Function Detail
If the -df or -da options are used, the splat command reports the function detail as described below:

PThreadID
The PThread identifier.

Acquisitions
The number of times that this function acquired the mutex.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PThread</td>
<td>The number of threads that are using this lock.</td>
</tr>
<tr>
<td>creation</td>
<td>Elapsed time in seconds after the first event recorded in trace (if available).</td>
</tr>
<tr>
<td>deletion</td>
<td>Elapsed time in seconds after the first event recorded in trace (if available).</td>
</tr>
<tr>
<td>Parent Thread</td>
<td>Pthread id of the parent thread.</td>
</tr>
<tr>
<td>creation time</td>
<td>Elapsed time in seconds after the first event recorded in trace (if available).</td>
</tr>
<tr>
<td>deletion time</td>
<td>Elapsed time in seconds after the first event recorded in trace (if available).</td>
</tr>
</tbody>
</table>

**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.

```plaintext
[PThread RWLock] ADDRESS: 000000002F228E0
Parent Thread: 0000000000000001 creation time: 5.236585 deletion time: 6.090511
PId: 7362 Process Name: /home/testrwlock
Creation call-chain
00000000100000458 .main
00000000100001DC __start
-----------------------------------------------------------------------------
Acquisitions Miss Spin Wait Secs Held CPU Elapsed Real Real Comb Real
15   40.568 785 0 21.037942 12.0346 80.19 99.22 30.45 46.29
-----------------------------------------------------------------------------
Depth  Min Max Avg  Min Max Avg  Min Max Avg
LockQ  0 2 0 0 1 0 0 2 0
SpinQ  0 768 601 0 15 11 0 782 612
WaitQ  0 769 166 0 15 3 0 783 169
-----------------------------------------------------------------------------
PhthreadID Acquisitions Miss Spin Wait Count Wait Count Busy Count Percent Held of Total Time
772    0 207 78.70 0 765 0 796 0 11.58 15.13 29.69 23.21
515    765 0 1.80 14 0 14 0 0 80.10 80.19 49.76 23.08
258    0 178 3.26 0 6 0 5 0 12.56 17.10 10.00 20.02
-----------------------------------------------------------------------------
Function Name Write Read Rate Write Read Write Read Count CPU Elapse Spin Wait Return Address Start Address Offset
__pthread_body 765 385 40.57 14 771 0 0 0 1.55 3.10 1.63 0.00 00000000268944C 000000002684188 0000052CC
-----------------------------------------------------------------------------
```

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

**Parent Thread**

- Elapsed time in seconds after the first event recorded in trace (if available).

**creation time**

- Elapsed time in seconds after the first event recorded in trace (if available).
PID          Process identifier.
Process Name     Name of the process using the lock.
Call-chain       Stack of called methods (if available).
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 pthreads 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 pthread. If the lock is held multiple times by the same pthread, only one hold interval is counted.
                   Elapsed(d) The total number of elapsed seconds that the lock was held by any pthread, whether the pthread 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 pthread.
                   Real Elapsed The percentage of the elapsed real time that the lock was held by any pthread, either running or suspended.
                   Comb(ined) Spin The percentage of the cumulative processor time that running pthreads spent spinning while trying to acquire this lock.
                   Real Wait The percentage of elapsed real time that any pthread 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 pthreads 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 pthreads 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 pthreads 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 pthreads in a timed-wait state for the lock, across the analysis interval. This is broken down by read-acquisitions, write-acquisitions, and total acquisitions.

Note: The pthread and function details for read/write locks are similar to the mutex detail reports, except that they break down the acquisition, spin, and wait counts by whether the lock is to be acquired for reading or writing.

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 pthread 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 pthreads spun while waiting for the condition.

- **Comb Wait**
  
  The total number of elapsed seconds that pthreads spent in a wait state for the condition.

**Depth**

This field contains the following sub-fields:

- **SpinQ**
  
  The minimum, maximum, and average number of pthreads spinning while waiting for the condition, across the analysis interval.

- **WaitQ**
  
  The minimum, maximum, and average number of pthreads waiting for the condition, across the analysis interval.

**Condition-Variable Pthread Detail**

If the `-dt` or `-da` options are used, the `splat` command reports the pthread detail as described below:

**PTThreadID**

The `PThread` identifier.

**Passes**

The number of times that this pthread was notified that the condition passed.

**Fail Rate**

The percentage of times that the pthread checked the condition and did not find it to be true.

**Spin Count**

The number of times that the pthread checked the condition and did not find it to be true.
Wait Count
The number of times that this pthread 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 pthread spun while testing the condition.

Wait
The percentage of elapsed real time that this pthread spent waiting for the condition to hold.

Condition-Variable Function Detail
If the -df or -da options are used, the splat command reports the function detail as described below:

Function Name
The name of the function that passed or attempted to pass this condition.

Passes
The number of times that this function was notified that the condition passed.

Fail Rate
The percentage of times that the function checked the condition and did not find it to be true.

Spin Count
The number of times that the function checked the condition and did not find it to be true.

Wait Count
The number of times that this function 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 function spun while testing the condition.

Wait
The percentage of elapsed real time that this function spent waiting for the condition to hold.

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.
Chapter 5. Hardware Performance Monitor APIs and tools

The bos.pmapi fileset contains libraries and tools that are designed to provide access to some of the counting facilities of the Performance Monitor feature included in select IBM microprocessors. They include the following:

- The pmapi library, which contains a set of low-level application programming interfaces, APIs, includes the following:
  - A set of system-level APIs to permit 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 permit threads to count their own activity.
  - A set of third party kernel-thread-level APIs to permit a debug program to count the activity of target threads.

- The pmcycles command, which returns the processor clock and decremener speeds.

- The pmclist command, which displays information about processors, events, event groups and sets, and derived metrics supported.

- The hpm and hpm_r libraries, which contain a set of high-level APIs that enable the following:
  - Nested instrumentation of sections of code
  - Automatic calculation of derived metrics, and gathering of operating system resource-consumption metrics in addition to the raw hardware counter values

- The hpmstat command, which collects the hardware performance monitor raw and derived metrics concerning total system activity of a machine.

- The hpmcount command, which executes applications and provides the applications’ execution wall clock time, the raw and derived hardware performance monitor metrics and the operating system resource-utilization statistics.

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_initialize API initialization routine, can be verified, unverified, caveat, broken, group-only, thresholdable, or shared (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_initialize routine to force the return of events with status matching the filter. If no filter is passed to the pm_initialize routine, no events will be returned.

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_initialize 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 might not have any meaningful content.

Users can 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.

Thread context

Optional Performance Monitor contexts can also be associated with each 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 redispached), 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 were in use, system-level API calls are not enabled 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 permitted 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 are only permitted when the target process is either being ptraced by the API caller, that is, the caller is already attached to the target process, and the target process is stopped or the target process is stopped on a /proc file system event and the caller has the privilege required to open its control file.

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.

**The pmapi library**

The following rules are common to the Performance Monitor APIs:

- The `pm_initialize` routine must be called before any other API call can be made, and only events returned by a given `pm_initialize` 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 permitted 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. You can not use the pm_init routine with processors newer than POWER4. You must use the pm_initialize routine for newer processors.

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.

The pm_initialize API initialize routine

The pm_initialize routine returns the processor name in a structure of type pm_info2_t defined by its second parameter, its characteristics, the number of counters available, and the list of available events for each counter.

For each event a status is returned, indicating the event status: validated, unvalidated, or validated with caveat. The status also indicates if the event can be used in a group or not, if it is a thresholdable event and if it is a shared event.

Some events on some platforms can be used only within a group. In the case where an event group is specified instead of individual events, the events are defined as grouped only events.
For each returned event, a thresholdable state is also returned. It indicates whether an event can be used with a threshold. If so, specifying a threshold defers counting until it exceeds a number of cycles equal to the threshold multiplied by the selected processor threshold multiplier.

Some processors support two hardware threads per physical processing unit. Each thread implements a set of counters, but some events defined for those processors are shared events. A shared event, is controlled by a signal not specific to a particular thread's activity and sent simultaneously to both sets of hardware counters, one for each thread. Those events are marked by the shared status.

For each returned event, in addition to the testing status, the `pm_initialize` routine returns the identifier to be used in subsequent API calls, as a short name and a long name. The short name is a mnemonic name in the form PM_MNEMONIC. The same events 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 the number of completed instructions, and should exist on all processors.

The Performance Monitoring API enables the specification of event groups instead of individual events. Event groups are predefined sets of events. Rather than to specify individually each event, a single group ID can be specified. The interface to the `pm_initialize` routine returns the list of supported event groups in a structure of type `pm_groups_info_t` whose address is returned in the third parameter.

On some platforms, the use of event groups is required because all events are marked as group-only. Each event group that is 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 the testing status of at least one event is caveat, then the group testing status is at best caveat, and if the status of at least one event is 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 whose status match the filter are returned.

If the proctype parameter is not set to PM_CURRENT, the Performance Monitor APIs library is not initialized and the subroutine only returns information about the specified processor in its parameters, `pm_info2_t` and `pm_groups_info_t`, taking into account the filter. If the proctype parameter is set to PM_CURRENT, in addition to returning the information described, the Performance Monitor APIs library is initialized and ready to accept other calls.

### Basic pmapi library calls

Each of the sections below describes a system-wide API call that has variations for first- 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.

By default, the time spent during interrupts handling is counted. It is possible to override this default behavior by modifying the counting mode.
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/pm_tstart
Starts Performance Monitor counting. pm_tstart returns a timestamp associated with the time the Performance Monitoring counters started counting. This is a timebase value that can be converted to time using time_base_to_time.

pm_stop/pm_tstop
Stops Performance Monitor counting. pm_tstop returns a timestamp associated with the time the Performance Monitoring counters stopped counting. This is a timebase value that can be converted to time using time_base_to_time.

pm_get_data/pm_get_tdata/pm_get_Tdata
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.”

pm_get_tdata is similar to 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.

pm_get_Tdata is also similar to pm_get_data but includes accumulated times corresponding to the interval during which the hardware Performance Monitoring counters were active. The interval is measured in real time, PURR and SPURR (on processors supporting those) values, and returned in timebase units convertible to time using time_base_to_time.

The pm_get_data_cpu, pm_get_tdata_cpu and pm_get_Tdata_cpu interfaces return the Performance Monitor counting data for a single processor. The specified processor number represents a contiguous number going from 0 to _system_configuration.ncpus. This number can represent a different processor from call to call if dynamic reconfiguration operations have occurred.

The pm_get_data_lcpu, pm_get_tdata_lcpu and pm_get_Tdata_lcpu interfaces return the Performance Monitor counting data for a single logical processor. The logical processor numbering is not contiguous, and the call to these interfaces returns an error if the specified logical processor has not been on line since the last call to pm_set_program. A logical processor number always designates the same processor even if dynamic reconfiguration operations have occurred. To get data for all processors, these interfaces must be called in a loop from 0 to _system_configuration.max_ncpus.

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 \texttt{pm\_get\_data\_mygroup} and \texttt{pm\_get\_data\_pgroup} interfaces in a
 \texttt{pm\_groupinfo\_t} structure.

## Counter multiplexing mode

You can set the counting for more events than available hardware counters using counter multiplexing.
This mode is meant to be used to analyze workloads with homogenous performance characteristics.
This avoids the requirement to run the workload multiple times to collect more events than available
hardware counters. In this mode, the pmapi periodically changes the setting of the counting and accumulates values
and counting time for multiple sets of events. The time each event set is counted before switching to the
next set can be in the range of 10 ms to 30 s. The default value is 100 ms.

The values returned include the number of times all sets of events have been counted, and for each set,
the accumulated counter values and the accumulated time the set was counted. The accumulated time is
measured up to three different ways: using Time Base, and when available, using the PURR time and one
the SPURR time. These times are stored in a timebase format that can be converted to time by using the
\texttt{time\_base\_to\_time} function. These times are meant to be used to normalize the results across the
complete measurement interval.

Several basic pmapi calls have the following multiplexing mode variations indicated by the \_mx suffix:

\begin{description}
\item[\texttt{pm\_set\_program\_mx}] Sets the counting configuration. It differs from the \texttt{pm\_set\_program} function in that it accepts a
set of groups (or event lists) to be counted, and the time each set must be counted before
switching to the next set.
\item[\texttt{pm\_get\_program\_mx}] Retrieves the current Performance Monitor settings. It differs from the \texttt{pm\_get\_program} function
in that it returns a set of groups (or event lists).
\item[\texttt{pm\_get\_data\_mx}] Returns the Performance Monitor counting data. It returns a set of counting data, one per group
(or event list) configured. The returned data includes in addition to the accumulated counter
values, the number of times all the configured sets have been counted, and for each set, the
accumulated time it was counted.
\item[\texttt{pm\_get\_tdata\_mx}] Same as \texttt{pm\_get\_data\_mx}, but includes a timestamp indicating the last time that the hardware
Performance Monitor counters were read.\item[\texttt{pm\_get\_data\_cpu\_mx}/\texttt{pm\_get\_tdata\_cpu\_mx}] Same as \texttt{pm\_get\_data\_mx} or \texttt{pm\_get\_tdata\_mx}, but returns the Performance Monitor counting
data for a single processor. The specified processor number must be in the range 0 to
\texttt{\_system\_configuration.ncpus}. This number might represent different processors from call to call
if dynamic reconfiguration operations have occurred.
\item[\texttt{pm\_get\_data\_lcpu\_mx}/\texttt{pm\_get\_tdata\_lcpu\_mx}] Same as \texttt{pm\_get\_data\_cpu\_mx} or \texttt{pm\_get\_tdata\_cpu\_mx}, but returns the Performance Monitor counting
data for a single logical processor. The logical processor numbering is not contiguous, and
the call to these interfaces return an error if the specified logical processor has not been
online since the last call to \texttt{pm\_set\_program\_mx}. A logical processor number always designates
the same processor even if dynamic reconfiguration operations have occurred. To get data for all
processors, these interfaces must be called in a loop from 0 to
\texttt{\_system\_configuration.max\_ncpus}.
\end{description}
Counter multi-mode

Counter multi-mode is similar to multiplexing mode. The counting mode in multiplexing mode is common to all the event sets. The multi-mode allows you to associate a counting mode with each event set, but as the counting mode differs for an event set to another one, the results of the counting cannot be normalized on the complete measurement interval.

Several basic pmapi calls have the following multi-mode variations indicated by the _mm suffix:

**pm_set_program_mm**
Sets the counting configuration. It differs from the pm_set_program_mx function in that it accepts a set of groups and associated counting mode to be counted.

**pm_get_program_mm**
Retrieves the current Performance Monitor settings. It differs from the pm_get_program_mx function in that it accepts a set of groups and associated counting mode.

**Examples of pmapi library usage**
The following examples demonstrate the use of Performance Monitor APIs in pseudo-code. Functional sample code is available in the /usr/samples/pmapi directory.

**Simple single-threaded program example**

```c
#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_set_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**

```c
#include <pmapi.h>
main()
{
   pm_info2_t pminfo;
   pm_prog_t prog;
   pm_groups_info_t pmginfo;

   int filter = PM_VERIFIED; /* get list of verified events */

   pm_initialize(filter, &pminfo, &pmginfo, PM_CURRENT )
```
Get information about an event group processor example

```c
#include <pmapi.h>

main()
{
    pm_events2_t *evp;

    int rc, counter, event;
    pm_info2_t pminfo;
    pm_prog_t prog;
    pm_groups_info_t pmginfo;
    int filter = PM_VERIFIED; /* get list of verified events */

    if ((rc = pm_initialize(filter, &pminfo, &pmginfo, PM_POWER4)) != 0) {
        pm_error("pm_initialize", rc);
        exit(-1);
    }

    printf("Group #%d: %s\n", i, pmginfo.event_groups[i].short_name);
    printf("Group name: %s\n", pmginfo.event_groups[i].long_name);
    printf("Group description: %s\n", pmginfo.event_groups[i].long_name);
    printf("Group members:\n");
    for (counter = 0; counter < pminfo.maxpmcs; counter++) {
        printf("Counter %2d, ", counter+1);
        /* get the event id from the list */
        event = pmginfo.event_groups[i].events[counter];
        if ((event == COUNT_NOTHING) || (pminfo.maxevents[counter] == 0))
            printf("event %2d: No event\n", event);
        else {
            /* find pointer to the event */
            for (j = 0; j < pminfo.maxevents[counter]; j++) {
                evp = pminfo.list_events[counter]+j;
                if (event == evp->event_id) {
                    break;
                }
            }
        }
        printf("event %2d: %s", event, evp->short_name);
        printf(" : %s\n", evp->long_name);
    }
}
```

Debugger program example for initialization program

The following example illustrates how to look at the Performance Monitor data while the program is executing:

from a debugger at breakpoint (1)

```c
pm_initialize(filter);
```

(2) `pm_get_program_pthread(pid, tid, ptid, &prog);`  
... display PM programmation ...

(3) `pm_get_data_pthread(pid, tid, ptid);`  
... display PM data ...
pm_delete_program_pthread(pid, tid, ptid);
prog.events[0] = 2; /* change counter 1 to count event number 2 */
pm_set_program_pthread(pid, tid, ptid, &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.

```c
#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 */
    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.

```c
main()
{
    ... same initialization as in previous example ...

    pm_set_program_mygroup(&prog); /* create counting group */
    pm_start_mygroup();

    pthread_create(&threadid, &attr, doit, NULL)

    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.

Simple thread counting-group with counter-multiplexing 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.

```c
main()
{
    pm_info2_t pminfo;
    pm_groups_info_t pmginfo;
    pm_prog_mx_r prog;
    pm_events_prog_t event_set[2];
    pm_data_mx_t data;
    int filter = PM_VERIFIED; /* get list of verified events */
    pm_initialize(filter, &pminfo, &pmginfo, PM_CURRENT).
    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 */
    prog.events_set = event_set;
    prog.nb_events_prog = 2; /* two event group counted */
    prog.slice_duration = 200; /* slice duration for each event group is 200ms */
    for (i = 0; i < pminfo.maxpmcs; i++) {
        event_set[0][i] = COUNT NOTHING;
        event_set[1][i] = COUNT NOTHING;
    }
    event_set[0][0] = 1; /* count events in group 1 in the first set */
    event_set[1][0] = 3; /* count events in group 3 in the first set */
    pm_set_program_mygroup_mx(&prog); /* create counting group */
    pm_start_mygroup();
    pthread_create(&threadid, &attr, doit, NULL)
    pm_start_mythread();
    ... usefull work ....
    pm_stop_mythread();
    pm_get_data_mythread_mx(&data)
    printf("Main thread results:\n");
    for (i = 0; i < 2; i++)
        printf("Group #\%d: %s\n", i, pmginfo.event_groups[group number].short_name);
    pm_get_data_mygroup(&data)
    printf("Aux thread results:\n");
    for (i = 0; i < 2; i++)
        printf("Group #\%d: %s\n", i, pmginfo.event_groups[group number].short_name);
    printf("Group results:\n");
    pm_get_data_mygroup(&data)
    printf("\n");
}
```
printf("counting time: %d ms\n", data.accu_set[i].accu_time);
printf("counting values: %n");
for (counter = 0; counter < pmginfo.maxpmcs; counter++) {
    printf("event %d: %d\n", counter, data.accu_set[i].accu_data[counter]);
}
}
(1) free(data.accu_set); /* free the memory allocated for the main thread results */
pthread_join(threadid, &status);
... print auxiliary thread results ...
free(data.accu_set); /* free the memory allocated for the thread results */
pm_get_data_mygroup_mx(&data)
... print group results ...
free(data.accu_set); /* free the memory allocated for the group results */
pm_delete_program()
}
(1) Each time data are got in time slice mode, the buffer allocated to return the counters */
must be freed after used.

Simple thread counting-group with counter-multiplexing and multi-mode 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. This example is similar to the previous one except that it uses the multi-mode functionality, and associates a mode with each group counted.
main()
{
    pm_info2_t pminfo;
    pm_groups_info_t pmginfo;
    pm_prog_mm_t prog;
    pm_data_mx_t data;
    pm_prog_t prog_set[2];
    int filter = PM_VERIFIED; /* get list of verified events */
    pm_initialize(filter, &pminfo, &pmginfo, PM_CURRENT);
    prog.prog_set = prog_set;
    prog.nb_set_prog = 2; /* two groups counted */
    prog.slice_duration = 200; /* slice duration for each event group is 200ms */
    prog_set[0].mode.w = 0; /* start with clean mode */
    prog_set[0].mode.b.user = 1; /* grp 0: count only user mode */
    prog_set[0].mode.b.is_group = 1; /* specify event group */
    prog_set[0].mode.b.proctree = 1; /* turns process tree counting on:
            this option is common to all counted groups */
    prog_set[1].mode.w = 0; /* start with clean mode */
    prog_set[1].mode.b.kernel = 1; /* grp 1: count only kernel mode */
    prog_set[1].mode.b.is_group = 1; /* specify event group */
    for (i = 0; i < pmginfo.maxpmcs; i++) {
        prog_set[0].events[i] = COUNT NOTHING;
        prog_set[1].events[i] = COUNT NOTHING;
    }
    prog_set[0].events[0] = 1; /* count events in group 1 in the first set */
    prog_set[1].events[0] = 3; /* count events in group 3 in the first set */
    pm_set_program_mygroup_mm(&prog); /* create counting group */
    pm_start_mygroup();
    pthread_create(&threadid, &attr, doit, NULL);
    pm_start_mythread();
    ... usefull work ....
    pm_stop_mythread();
    pm_get_data_mythread_mx(&data);
    printf("Main thread results:\n");
    for (i = 0; i < 2; i++) {
        group_number = prog_set[i].events[0];
        printf("Group %d: %s\n", group_number, pmginfo.event_groups[group_number].short_name);
        printf("counting time: %d ms\n", data.accu_set[i].accu_time);
        printf("counting values: %n");
        for (counter = 0; counter < pmginfo.maxpmcs; counter++) {
            printf("event %d: %d\n", counter, data.accu_set[i].accu_data[counter]);
        }
    }
    free(data.accu_set); /* free the memory allocated for the main thread results */
    pthread_join(threadid, &status);
    ... print auxiliary thread results ...
    free(data.accu_set); /* free the memory allocated for the thread 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.

```c
main()
{
    ... same initialization as in previous example...
    prog.mode.b.count = 1; /* start counting immediately */
    pm_set_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.

The hpm library and associated tools

The hpm libraries are higher-level instrumentation libraries based on the pmapi library. They support multiple instrumentation sections, nested instrumentation, and each instrumented section can be called multiple times. When nested instrumentation is used, exclusive duration is generated for the outer sections. Average and standard deviation is provided when an instrumented section is activated multiple times.

The libraries support OpenMP and threaded applications, which requires linking with the thread-safe version of the library, libhpm_r. Both 32-bit and 64-bit library modules are provided.

The libraries collect information and hardware Performance Monitor summarization during run-time. So, there could be considerable overhead if instrumentation sections are inserted inside inner loops.
Compiling and linking
The functionality of the libhpm_r library depends upon the corresponding functions in the libpmapi and libm libraries. Therefore, the lpmapi -lm flag must be specified when compiling applications using the hpm libraries.

By default, argument passing from Fortran applications to the hpm libraries is done by reference, or pointer, not by value. Also, there is an extra length argument following character strings. You can modify the default argument passing method by using the %VAL and %REF built-in functions.

Overhead and measurement error issues
It is expected for any software instrumentation to incur some overhead. Since it is not possible to eliminate the overhead, the goal is to minimize it. In the hpm library, most of the overhead is due to time measurement, which tends to be an expensive operation in most systems. A second source of overhead is due to run-time accumulation and storage of performance data. The hpm libraries collect information and perform summarization during run-time. Hence, there could be a considerable amount of overhead if instrumentation sections are inserted inside inner loops.

The hpm library uses hardware counters during the initialization and finalization of the library, retaining the minimum of the two for each counter as an estimate of the cost of one call to the start and stop functions. The estimated overhead is subtracted from the values obtained on each instrumented code section, which ensures that the measurement of error becomes close to zero. However, since this is a statistical approximation, in some situations where estimated overhead is larger than a measured count for the application, the approach fails. When the approach fails, you might get the following error message, which indicates that the estimated overhead was not subtracted from the measured values:

WARNING: Measurement error for <event name> not removed

You can deactivate the procedure that attempts to remove measurement errors by setting the HPM_WITH_MEASUREMENT_ERROR environment variable to TRUE (1).

Common hpm library rules
The following rules are common to the hpm library APIs:
• The hpmInit() or f_hpminit() function must be called before any other function in the API.
• The initialization function can only be called once in an application.
• Performance Monitor contexts, like the event set, event group, or counter/event pairs, cannot be reprogrammed at any time.
• All functions of the API are specified as void and return no value or status.

Overview of the hpm library API calls
The following table lists the hpm library API calls:

<table>
<thead>
<tr>
<th>API Call</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>hpmInit or f_hpminit</td>
<td>Performs initialization for a specified node ID and program name.</td>
</tr>
<tr>
<td>hpmStart or f_hpmstart</td>
<td>Indicates the beginning of an instrumented code segment, which is identified by an instrumentation identifier, InstID.</td>
</tr>
<tr>
<td>hpmStop or f_hpmstop</td>
<td>Indicates the end of an instrumented code segment. For each call to the hpmStart() or f_hpmstart() function, there should be a corresponding call to the hpmStop() or f_hpmstop() function with the matching instrumentation identifier.</td>
</tr>
<tr>
<td>hpmTstart or f_hpmtstart</td>
<td>Performs the same function as the hpmStart() and f_hpmstart() functions, but they are used in threaded applications.</td>
</tr>
<tr>
<td>API Call</td>
<td>Purpose</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>hpmTstop or f_hpmtstop</td>
<td>Performs the same function as the hpmStop() and f_hpmstop() functions, but they are used in threaded applications.</td>
</tr>
<tr>
<td>hpmGetTimeAndCounters or f_hpmgettimeandcounters</td>
<td>Returns the time, in seconds, and the accumulated counts since the call to the hpmlInit() or f_hpminit() initialization function.</td>
</tr>
<tr>
<td>hpmGetCounters or f_hpgetcounter</td>
<td>Returns all the accumulated counts since the call to the hpmlInit() or f_hpminit() initialization function.</td>
</tr>
<tr>
<td>hpmTerminate or f_hpmterminate</td>
<td>Performs termination and generates output. If an application exits without calling the hpmTerminate() or f_hpmterminate() function, no performance information is generated.</td>
</tr>
</tbody>
</table>

**Threaded applications**

The T/tstart and T/tsstop functions respectively start and stop the counters independently on each thread. If two distinct threads use the same instID parameter, the output indicates multiple calls. However, the counts are accumulated.

The instID parameter is always a constant variable or integer. It cannot be an expression because the declarations in the libhpm.h, f_hpm.h, and f_hpm_i8.h header files that contain #define statements are evaluated during the compiler pre-processing phase, which permits the collection of line numbers and source file names.

**Selecting events when using the hpm libraries and tools**

The hpm libraries use the same set of hardware counters and events used by the hpmcount and hpmstat tools. The events are selected by sets. Sets are specially marked event groups for whichever derived metrics are available. For the hpm libraries, you can select the event set to be used by any of the following methods:

- The HPM_EVENT_SET environment variable, which is either explicitly set in the environment or specified in the HPM_flags.env file.
- The content of the libHPMevents file.

For the hpmcount and hpmstat commands, you can specify which event types you want to be monitored and the associated hardware performance counters by any of the following methods:

- Using the -s option
- The HPM_EVENT_SET environment variable, which you can set directly or define in the HPM_flags.env file
- The content of the libHPM_events file

In all cases, the HPM_flags.env file takes precedence over the explicit setting of the HPM_EVENT_SET environment variable and the content of the libHPMevents or libHPM_events file takes precedence over the HPM_EVENT_SET environment variable.

An event group can be specified instead of an event set, using any of the following methods:

- The -g option
- The HPM_EVENT_GROUP environment variable that you can set directly or define in the HPM_flags.env file

In all cases, the HPM_flags.env file takes precedence over the explicit setting of the HPM_EVENT_GROUP environment variable. The HPM_EVENT_GROUP environment variable takes precedence over the explicit setting of the HPM_EVENT_SET environment variable. The HPM_EVENT_GROUP is a comma separated list of group names or group numbers.
A list of derived metric groups to be evaluated can be specified, using any of the following methods:

- The `-m` option
- The `HPM_PMD_GROUP` environment variable that you can set directly or define in the `HPM_flags.env` file

In all cases, the `HPM_flags.env` file takes precedence over the explicit setting of the `HPM_PMD_GROUP` environment variable. The `HPM_PMD_GROUP` is a comma-separated list of derived metric group names.

Each set, group or derived metric group can be qualified by a counting mode. The allowed counting modes are:

- `u`: user mode
- `k`: kernel mode
- `h`: hypervisor mode
- `r`: runlatch mode
- `n`: nointerrupt mode

The counting mode qualifier is separated from the set or group by a colon `:`. For example:

```
HPM_EVENT_GROUP=pm_utilization:uk,pm_completion:u
```

To use the time slice functionality, specify a comma-separated list of sets instead of a single set number. By default, the time slice duration for each set is 100 ms, but this can be modified with the `HPM_MX_DURATION` environment variable. This value must be expressed in ms, and in the range 10 ms to 30000 ms.

**The libHPMevents and libHPM_events files**

The `libHPMevents` and `libHPM_events` files are both supplied by the user and have the same format.

For POWER3 or PowerPC 604 RISC Microprocessor systems, the file contains the counter number and the event name, like in the following example:

```
0  PM_LD_MISS_L2HIT
1  PM_TAG_BURSTRD_L2MISS
2  PM_TAG_ST_MISS_L2
3  PM_FPU0_DENORM
4  PM_LSU_IDLE
5  PM_LQ_FULL
6  PM_FPU_FMA
7  PM_FPU_IDLE
```

For POWER4 and later systems, the file contains the event group name, like in the following example:

```
pm_hpmcount1
```

**The HPM_flags.env file**

The `HPM_flags.env` file contains environment variables that are used to specify the event set and for the computation of derived metrics, like in the following example:

```
HPM_L2_LATENCY 12
HPM_EVENT_SET 5
```

**Output files of the hpm library**

When the `hpmTerminate` function is called, a summary report is written to the `<progName>_<pid>_<taskID>.hpm` file, by default. The `taskID` and `progName` values are the first and second parameters of the `hpmInit()` function, respectively.
You can define the name of the output file with the `HPM_OUTPUT_NAME` environment variable. The hpm libraries always add the `_<taskID>.hpm` suffix to the specified value. You can also include the date and time in the file name using the `HPM_OUTPUT_NAME` environment variable. For example, if you use the following code:

```
MYDATE=$(date +"%m%d:2/2/06M%S")
export HPM_OUTPUT_NAME=myprogram_$MYDATE
```

the output file for task 27 is named `myprogram_yyyymmdd:HHMMSS_0027.hpm`.

You can also generate an XML output file by setting the `HPM_VIZ_OUTPUT=TRUE` environment variable. The generated output files are named either `<progName>_<pid>_<taskID>.viz` or `HPM_OUTPUT_NAME_<taskID>.viz`.

### Output files of the `hpmcount` command

Depending on the environment variables set and the execution environment, the following files are created when you run the `hpmcount` command:

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>file_&lt;myID&gt;_&lt;pid&gt;</code></td>
<td>The value for <code>file</code> is specified with the <code>-o</code> option and the <code>myID</code> value is assigned the value of the <code>MP_CHILD</code> environment variable, which has a default value of 0000.</td>
</tr>
<tr>
<td><code>HPM_LOG_DIR/hpm_log.&lt;pid&gt;</code></td>
<td>When the <code>HPM_LOG_DIR</code> environment variable is set to an existing directory, results are additionally written to the <code>hpm_log.&lt;pid&gt;</code> file.</td>
</tr>
<tr>
<td><code>HPM_LOG_DIR/hpm_log.MP_PARTITION</code></td>
<td>The <code>MP_PARTITION</code> environment variable is provided in POE environments. The <code>hpm_log.MP_PARTITION</code> file contains the aggregate counts.</td>
</tr>
</tbody>
</table>

An XML output can be provided by using the `-x` option.

An alternative time base for the result normalization can be selected using any of the following methods:
- The `-b timelpurrspurr` option
- The `HPM_NORMALIZE` environment variable that you can set directly or define in the `HPM_flags.env` file

### Derived metrics and related environment variables

In relation to the hardware events that are selected to be counted and the hardware platform that is used, the output for the hpm library tools and the `hpmterminate` function includes derived metrics. You can list the globally supported metrics for a given processor with the `pmlist -D -1 [-p Processor_name]` command.

You can supply the following environment variables to specify estimations of memory, cache, and TLB miss latencies for the computation of related derived metrics:
- `HPM_MEM_LATENCY`
- `HPM_L3_LATENCY`
- `HPM_L35_LATENCY`
- `HPM_AVG_L3_LATENCY`
- `HPM_AVG_L2_LATENCY`
- `HPM_L2_LATENCY`
- `HPM_L25_LATENCY`
- `HPM_L275_LATENCY`
• HPM_L1_latency
• HPM_TLB_latency

Precedence is given to variables that are defined in the HPM_flags.env file.

You can use the HPM_DIV_WEIGHT environment variable to compute the weighted flips on systems that are POWER4 and later.

Examples of the hpm tools
The examples in this section demonstrate the usage of the following hpm library commands:
• "The pmlist command"
• "The hpmcount command"
• "The hpmstat command" on page 134

The pmlist command
The following is an example of the pmlist command:
```bash
# pmlist -s

POWER5 supports 6 counters
Number of groups : 144
Number of sets : 8

Threshold multiplier (lower): 1
Threshold multiplier (upper): 32
Threshold multiplier (hyper): 64
Hypervisor counting mode is supported
Runlatch counting mode is supported
```

The following is another example of the pmlist command:
```bash
# pmlist -D -I -p POWER5

Derived metrics supported:
- PMD_UTIL_RATE: Utilization rate
- PMD_MIPS: MIPS
- PMD_INST_PER_CYC: Instructions per cycle
- PMD_HW_FP_PER_CYC: HW floating point instructions per cycle
- PMD_HW_FP_PER_UIMAGE: HW floating point instructions / user time
- PMD_HW_FP_RATE: HW floating point rate
- PMD_FX: Total fixed point operations
- PMD_FX_PER_CYC: Fixed point operations per cycle
- PMD_PP_LO_ST: Floating point load and store operations
- PMD_INST_PER_FP_LO_ST: Instructions per floating point load/store
- PMD_FXC_INST_DISP_CMPL: % instructions dispatched that completed
- PMD_DATA_L2: Total L2 data cache accesses
- PMD_PP_L2_ACCESS: % accesses from L2 per cycle
- PMD_L2_TRAF: L2 traffic
- PMD_L2_BW: L2 bandwidth per processor
- PMD_L2_LO_EST_LAT_AVG: Estimated latency from loads from L2 (Average)
- PMD_UTIL_RATE_RC: Utilization rate (versus run cycles)
- PMD_INST_PER_CYC_RC: Instructions per run cycle
- PMD_LO_ST: Total load and store operations
- PMD_INST_PER_LD_ST: Instructions per load/store
- PMD_PP_LD_MISS: Number of loads per load miss
- PMD_PP_DLDB: Number of loads per DTB miss
- PMD_PP_ST_MISS: Number of stores per store miss
- PMD_PP_TLB: Number of loads per TLB miss
- PMD_PP_TLB_TLB: Number of load/store per TLB miss
- PMD_PP_TLB_LAT: Estimated latency from TLB miss
- PMD_PP_MEM_LRAF: Memory load traffic
- PMD_PP_MEM_BW: Memory bandwidth per processor
- PMD_PP_MEM_LST_LAT: Estimated latency from loads from memory
- PMD_PP_MEM_LD_PER_LDB: Number of loads from local memory per loads from remote memory
- PMD_PP_MEM_LD_RC: % loads from memory per run cycle
```

The hpmcount command
The following is an example of the hpmcount command:
```bash
# hpmcount -m cpi_breakdown ls
bar
foo

Workload context: ls (pid:42234)
Execution time (wall clock time): 0.004222 seconds

### Resource Usage Statistics ###
<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total amount of time in user mode</td>
<td>0.001783</td>
</tr>
<tr>
<td>Total amount of time in system mode</td>
<td>0.000378</td>
</tr>
<tr>
<td>Maximum resident set size</td>
<td>220 Kbytes</td>
</tr>
<tr>
<td>Average shared memory use in text segment</td>
<td>0 Kbytes/sec</td>
</tr>
<tr>
<td>Average unshared memory use in data segment</td>
<td>0 Kbytes/sec</td>
</tr>
<tr>
<td>Number of page faults without I/O activity</td>
<td>63</td>
</tr>
<tr>
<td>Number of page faults with I/O activity</td>
<td>0</td>
</tr>
<tr>
<td>Number of times process was swapped out</td>
<td>0</td>
</tr>
<tr>
<td>Number of times file system performed INPUT</td>
<td>0</td>
</tr>
<tr>
<td>Number of times file system performed OUTPUT</td>
<td>0</td>
</tr>
<tr>
<td>Number of IPC messages sent</td>
<td>0</td>
</tr>
<tr>
<td>Number of IPC messages received</td>
<td>0</td>
</tr>
<tr>
<td>Number of signals delivered</td>
<td>0</td>
</tr>
<tr>
<td>Number of voluntary context switches</td>
<td>0</td>
</tr>
<tr>
<td>Number of involuntary context switches</td>
<td>0</td>
</tr>
</tbody>
</table>

### End of Resource Statistics ###

Counting mode: user

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM_IPLUS_PPC_CMPL (One or more PPC instruction completed)</td>
<td>143749096</td>
</tr>
<tr>
<td>PM_GCT_EMPTY_CYC (Cycles GCT empty)</td>
<td>12905400</td>
</tr>
<tr>
<td>PM_GRP_CMPL (Group completed)</td>
<td>144626424</td>
</tr>
<tr>
<td>PM_CYC (Processor cycles)</td>
<td>434717274</td>
</tr>
<tr>
<td>PM_INST_CMPL (Instructions completed)</td>
<td>193121895</td>
</tr>
<tr>
<td>PM_RUN_CYC (Run cycles)</td>
<td>378397903</td>
</tr>
<tr>
<td>PM_GCT_NOSLOT_CYC (Cycles no GCT slot allocated)</td>
<td>87592746</td>
</tr>
<tr>
<td>PM_GCT_NOSLOT_IC_MISS</td>
<td>16066248</td>
</tr>
<tr>
<td>PM_GCT_NOSLOT_SRQ_FULL (No slot in GCT caused by SRQ full)</td>
<td>0</td>
</tr>
<tr>
<td>PM_GCT_NOSLOT_BR_MPRE</td>
<td>27809700</td>
</tr>
<tr>
<td>PM_GRP_MRK (Group marked in IDU)</td>
<td>6041616</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_LSU</td>
<td>117973392</td>
</tr>
<tr>
<td>PMousedownCompl by LSU instruction</td>
<td></td>
</tr>
<tr>
<td>PM_LOADING_CMPL (Internal operations completed)</td>
<td>162398665</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_REJECT (Completion stall caused by reject)</td>
<td>24318036</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_DCACHE_MISS</td>
<td>25055262</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_ERAT_MISS</td>
<td>17332764</td>
</tr>
<tr>
<td>PM_GRP_IC_MISS_BR_REDIR_NOSPEC</td>
<td>2551038</td>
</tr>
<tr>
<td>(Group experienced non-speculative I cache miss or branch redirect)</td>
<td></td>
</tr>
<tr>
<td>PM_CMPLUSTALL_FXU</td>
<td>69575412</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_DDIV</td>
<td>45664068</td>
</tr>
<tr>
<td>PM_FPU_FULL_CYC (Cycles FPU issue queue full)</td>
<td>27660</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_FDIV</td>
<td>319104</td>
</tr>
<tr>
<td>PM月下 downfall caused by FDIV or FORT instruction</td>
<td>500274</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_FPU</td>
<td></td>
</tr>
</tbody>
</table>

Derived metric group: cpi_breakdown

Total cycles: 2.250999

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completion cycles</td>
<td>0.748887</td>
</tr>
<tr>
<td>Completion Table empty (GCT empty)</td>
<td>0.266825</td>
</tr>
<tr>
<td>I-Cache Miss Penalty</td>
<td>0.083192</td>
</tr>
<tr>
<td>Branch Misprediction Penalty</td>
<td>0.144311</td>
</tr>
<tr>
<td>Others GCT stalls</td>
<td>0.039322</td>
</tr>
<tr>
<td>Completion Stall cycles</td>
<td>1.435288</td>
</tr>
<tr>
<td>Stall by LSU instruction</td>
<td>0.610875</td>
</tr>
<tr>
<td>Stall by LSU Reject</td>
<td>0.125921</td>
</tr>
<tr>
<td>Stall by LSU Translation Reject</td>
<td>0.089750</td>
</tr>
<tr>
<td>Stall by LSU Other Reject</td>
<td>0.036170</td>
</tr>
<tr>
<td>Stall by LSU D-cache miss</td>
<td>0.129738</td>
</tr>
</tbody>
</table>
```
Stall by LSU basic latency, LSU Flush penalty: 0.355217
Stall by FXU instruction: 0.360267
Stall by any form of DIV/MTSPR/MFSPR instruction: 0.236452
Stall by FXU basic latency: 0.123815
Stall by FPU instruction: 0.002590
Stall by any form of FDIV/FSQRT instruction: 0.001652
Stall by FPU basic latency: 0.000938
Stall by others: 0.462493

The hpmstat command

The following is an example of the hpmstat command:

```
# hpmstat -s 7
Execution time (wall clock time): 1.003946 seconds
Counting mode: user
PM_TLB_MISS (TLB misses) : 260847
PM_CYC (Processor cycles) : 3013964331
PM_ST_REF_L1 (L1 D cache store references) : 161377371
PM_LD_REF_L1 (L1 D cache load references) : 255317480
PM_INST_CMPL (Instructions completed) : 1027391919
PM_RUN_CYC (Run cycles) : 1495147343
Derived metric group: default
Utilization rate : 181.243 %
Total load and store operations : 416.695 M
Instructions per load/store : 2.466
MIPS : 1023.354
Instructions per cycle : 0.341
```

The following is an example of the hpmstat command with counter multiplexing:

```
# hpmstat -s 1,2 -d
Execution time (wall clock time): 2.129755 seconds
Set: 1
Counting duration: 1.065 seconds
PM_INST_CMPL (Instructions completed) : 244687
PM_FPU1_CMPL (FPU1 produced a result) : 0
PM_ST_CMPL (Store instruction completed) : 31295
PM_LD_CMPL (Loads completed) : 67414
PM_FPU0_CMPL (Floating-point unit produced a result) : 19
PM_CYC (Processor cycles) : 295427
PM_TLB_MISS (TLB misses) : 788
PM_ST_DISP (Store instructions dispatched) : 125440562
PM_LD_DISP (Load instr dispatched) : 258031257
Counting mode: user
PM_TLB_MISS (TLB misses) : 380260
PM_ST_MISS_L1 (L1 D cache store misses) : 79943
PM_LD_MISS_L1 (L1 D cache load misses) : 307338
PM_INST_CMPL (Instructions completed) : 848578245
PM_LSU_IDLE (Cycles LSU is idle) : 229922845
PM_CYC (Processor cycles) : 757442686
PM_ST_DISP (Store instructions dispatched) : 125440562
PM_LD_DISP (Load instr dispatched) : 258031257
Derived metric group: default
Utilization rate : 189.830 %
```
Examples of hpm library usage
The following are examples of hpm library usage:

- "A C programming language example"
- "A Fortran programming language example" on page 136
- "Multithreaded application instrumentation example" on page 136

A C programming language example
The following C program contains two instrumented sections which perform a trivial floating point operation, print the results, and then launch the command interpreter to execute the `ls -R /2>&1` command:

```c
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <libhpm.h>

void do_work()
{
    pid_t p, wpid;
    int i, status;
    float f1 = 9.7641, f2 = 2.441, f3 = 0.0;

    f3 = f1 / f2;
    printf("f3=%f\n", f3);

    p = fork();
    if (p == -1) {
        perror("Mike fork error");
        exit(1);
    }

    if (p == 0) {
        i = execl("/usr/bin/sh", "sh", "-c", "ls -R /2>&1 >/dev/null", 0);
        perror("Mike execl error");
        exit(2);
    }
    else
        wpid = waitpid(p, &status, WUNTRACED | WCONTINUED);

    if (wpid == -1) {
        perror("Mike waitpid error");
        exit(3);
    }

    return;
}
```
main(int argc, char **argv)
{
    int taskID = 999;

    hpInit(taskID, "my_program");
    hpStart(1, "outer call");
    do_work();
    hpStart(2, "inner call");
    do_work();
    hpStop(2);
    hpStop(1);
    hpTerminate(taskID);
}

A Fortran programming language example
The following declaration is required on all source files that have instrumentation calls:
#include "f_hpm.h"

Fortran programs call functions that include the f_ prefix, as you can see in the following example:
call f_hpminit( taskID, "my_program" )
call f_hpmstart( 1, "Do Loop" )
do ...
    call do_work()
    call f_hpmstart( 5, "computing meaning of life" );
    call do_more_work();
    call f_hpmstop( 5 );
end do
call f_hpmstop( 1 )
call f_hpmterminate( taskID )

Multithreaded application instrumentation example
When placing instrumentation inside of parallel regions, you should use a different id for each thread, as shown in the following Fortran example:
!$OMP PARALLEL
!$OMP&PRIVATE (instID)
    instID = 30+omp_get_thread_num()
    call f_hpmtstart( instID, "computing meaning of life" )
!$OMP DO
do ...
    do_work()
end do
    call f_hpmtstop( instID )
!$OMP END PARALLEL

The library accepts the use of the same instID for different threads, but the counters are accumulated for all instances with the same instID.
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 thread–safe, and does not require root authority.

The API supports extensions so binary compatibility is maintained across all releases of AIX. 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 permits 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 filesset, which is installable from the AIX base installation media and requires that the bos.perf.perfstat filesset is installed. The latter 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 filesset. 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 Version 6.1 Files Reference.

API Characteristics

The perfstat library is not WPAR aware and is not supported inside a WPAR. Two types of APIs are available. Global types return global metrics related to a set of components, while 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. In order to do a more selective reset, you can use the perfstat_partial_reset function. For more details, see the “Cached metrics interfaces” on page 163 section.

Most types returned are unsigned long long; that is, unsigned 64-bit data.

Excessive and redundant calls to Perfstat APIs in a short time span can have a performance impact because time-consuming statistics collected by them are not cached.

All of the examples presented in this chapter can be compiled in AIX 5.3 and later using the cc command with -lperfstat.

Global Interfaces

Global interfaces report metrics related to a set of components on a system (such as processors, disks, or 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
- `perfstat_partition_total` Retrieves global partition metrics

The common signature used by all of the global interfaces is as follows:

```c
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.

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 Version 6.1 Files Reference*.

The following code shows an example of how `perfstat_cpu_total` is used:
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <libperfstat.h>
#include <sys/systemcfg.h>

#define XINTFRAC ((double)((_system_configuration.Xint)/(double)(_system_configuration.Xfrac))
#define HTIC2SEC(x) ((double)x * XINTFRAC)/(double)1000000000.0

static int disp_util_header = 1;
static u_longlong_t last_time_base;
static u_longlong_t last_pcpu_user, last_pcpu_sys, last_pcpu_idle, last_pcpu_wait;
static u_longlong_t last_lcpu_user, last_lcpu_sys, last_lcpu_idle, last_lcpu_wait;
static u_longlong_t last_phint = 0, last_vcsw = 0, last_pit = 0;

void display_lpar_util(void);

int main(int argc, char* argv[]) {
    while (1) {
        display_lpar_util();
        sleep(atoi(argv[1]));
    }
    return(0);
}

/* Save the current values for the next iteration */
void save_last_values(perfstat_cpu_total_t* cpustats, perfstat_partition_total_t* lparstats) {
    last_vcsw = lparstats->vol_virt_cswitch + lparstats->invol_virt_cswitch;
    last_time_base = lparstats->timebase_last;
    last_phint = lparstats->phantintrs;
    last_pit = lparstats->pool_idle_time;
    last_pcpu_user = lparstats->puser;
    last_pcpu_sys = lparstats->psys;
    last_pcpu_idle = lparstats->pidle;
    last_pcpu_wait = lparstats->pwait;
    last_lcpu_user = cpustats->user;
    last_lcpu_sys = cpustats->sys;
    last_lcpu_idle = cpustats->idle;
    last_lcpu_wait = cpustats->wait;
}

/* Gather and display lpar usitilization metrics */
void display_lpar_util() {
    u_longlong_t dlt_pcpu_user, dlt_pcpu_sys, dlt_pcpu_idle, dlt_pcpu_wait;
    u_longlong_t dlt_lcpu_user, dlt_lcpu_sys, dlt_lcpu_idle, dlt_lcpu_wait;
    u_longlong_t vcsw, lpctime, pctime;
    u_longlong_t entitled_purr, unused_purr;
    u_longlong_t delta_purr, delta_time_base;
    double phys_proc_consumed, entitlement, percent_ent, delta_sec;
    perfstat_partition_total_t lparstats;
    perfstat_cpu_total_t cpustats;

    /* retrieve the metrics */
    if (!perfstat_partition_total(NULL, &lparstats, sizeof(perfstat_partition_total_t), 1)) {
        perror("perfstat_partition_total");
        exit(-1);
    }

    if (!perfstat_cpu_total(NULL, &cpustats, sizeof(perfstat_cpu_total_t), 1)) {
        perror("perfstat_cpu_total");
        exit(-1);
    }
}
/* Print the header for utilization metrics (only once) */
if (disp_util_header) {
    if (lparstats.type.b.shared_enabled) {
        if (lparstats.type.b.pool_util_authority) {
            fprintf(stdout, "%5s %5s %6s %6s %5s %5s %4s %5s",
                        "%user", "%sys", "%wait", "%idle", "physc", "%entc", "lbusy", "vcsw", "print" );
            fprintf(stdout, "----- ----- ----- ----- ----- ----- ----- ----- -----");
        } else {
            fprintf(stdout, "%5s %5s %6s %6s %5s %5s %4s %5s",
                        "%user", "%sys", "%wait", "%idle", "physc", "%entc", "lbusy", "vcsw", "print" );
            fprintf(stdout, "----- ----- ----- ----- ----- ----- ----- ----- -----");
        }
    } else {
        fprintf(stdout, "%5s %5s %6s %6s",
                        "%user", "%sys", "%wait", "%idle" );
        fprintf(stdout, "----- ----- ----- ----- ----- ----- ----- ----- -----");
    }
    fprintf(stdout, "\n");
}

if (disp_util_header) {
    disp_util_header = 0;
}

if (lparstats.type.b.shared_enabled) {
    entitled_purr = delta_time_base * entitlement;
    if (entitled_purr < delta_purr) {
        /* when above entitlement, use consumption in percentages */
        entitled_purr = delta_purr;
    }
    unused_purr = entitled_purr - delta_purr;
    /* distribute unused purre in wait and idle proportionally to logical wait and idle */
    dlt_pcpu_wait += unused_purr * (double)(dlt_lcpu_wait + dlt_lcpu_idle);
    dlt_pcpu_idle += unused_purr * (double)(dlt_lcpu_wait + dlt_lcpu_idle);
    pcputime = entitled_purr;
}

/* Physical Processor Utilization */
printf("%5.1f \n", (double)dlt_pcpu_user * 100.0 / (double)pcputime);
printf("%5.1f \n", (double)dlt_pcpu_sys * 100.0 / (double)pcputime);

140 AIX Version 6.1 Performance Tools Guide and Reference
printf("%6.1f ", (double)dlt_pcpu_wait * 100.0 / (double)pcputime);
printf("%6.1f ", (double)dlt_pcpu_idle * 100.0 / (double)pcputime);

if (lparstats.type.b.shared_enabled) {
    /* Physical Processor Consumed */
    phys_proc_consumed = (double)delta_purr / (double)delta_time_base;
    printf("%5.2f ", (double)phys_proc_consumed);

    /* Percentage of Entitlement Consumed */
    percent_ent = (double)((phys_proc_consumed / entitlement) * 100);
    printf("%5.1f ", percent_ent);

    /* Logical Processor Utilization */
    printf("%5.1f ", (double)(dlt_lcpu_user+dlt_lcpu_sys) * 100.0 / (double)lcputime);

    if (lparstats.type.b.pool_util_authority) {
        /* Available Pool Processor (app) */
        printf("%5.2f ", (double)(lparstats.pool_idle_time - last_pit) /
               (XINTFRAC*(double)delta_time_base));
    }

    /* Virtual CPU Context Switches per second */
    vcsw = lparstats.vol_virt_cswitch + lparstats.invol_virt_cswitch;
    delta_sec = HTIC2SEC(delta_time_base);
    printf("%4.0f ", (double)(vcsw - last_vcsw) / delta_sec);

    /* Phantom Interrupts per second */
    printf("%5.0f",(double)(lparstats.phantintrs - last_phint) / delta_sec);
}
printf("\n");
save_last_values(&cpustats, &lparstats);

The preceding program emulates lparstat and displays utilization numbers similar to topas (or vmstat, iostat, sar, and mpstat).

perstat_memory_total Interface

The perstat_memory_total function returns a perstat_memory_total_t structure, which is defined in the libperfstat.h file. Selected fields from the perstat_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)
- pgin: Number of pages paged in
- pgout: 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 amount of paging space paged in and out, and amount of system memory) are also returned. For a complete list, see the perstat_memory_total_t section of the libperfstat.h header file in AIX Version 6.1 Files Reference.

The following code shows an example of how perstat_memory_total is used:

#include <stdio.h>
#include <libperfstat.h>
```c
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 : %lu MB\n", minfo.real_total*4096/1024/1024);
    printf("reserved paging space : %lu MB", minfo.pgsp_rsvd);
    printf("virtual memory size : %lu MB", minfo.virt_total*4096/1024/1024);
    printf("number of free pages : %lu\n", minfo.real_free);
    printf("number of pinned pages : %lu\n", minfo.real_pinned);
    printf("number of pages in file cache : %lu\n", minfo.numperm);
    printf("free paging space pages : %lu\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 : %lu\n", minfo.pgspins);
    printf("number of paging space page outs : %lu\n", minfo.pgspouts);
    printf("number of page ins : %lu\n", minfo.pgins);
    printf("number of page outs : %lu\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 and 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 the `libperfstat.h` header file in AIX Version 6.1 Files Reference.

The following code shows an example of how `perfstat_disk_total` is used:

```c
#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

perstat_netinterface_total Interface

The `perstat_netinterface_total` function returns a `perstat_netinterface_total_t` structure, which is defined in the `libperfstat.h` file. Selected fields from the `perstat_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 `perstat_netinterface_total_t` section in the `libperfstat.h` header file in AIX Version 6.1 Files Reference.

The following code shows an example of how `perstat_netinterface_total` is used:

```c
#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("input 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("output 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
```

**perfstat_partition_total Interface**

The `perfstat_partition_total` function returns a `perfstat_partition_total_t` structure, which is defined in the `libperfstat.h` file. Selected fields from the `perfstat_partition_total_t` structure include:

- **type**  
  Partition type

- **online_cpus**  
  Number of virtual CPUs currently allocated to the partition

- **online_memory**  
  Amount of memory currently allocated to the partition

For a complete list, see the `perfstat_partition_total_t` section in the `libperfstat.h` header file in AIX Version 6.1 Files Reference.

The following code shows examples of how to use the `perfstat_partition_total` function.

The first example demonstrates how to emulate the `lpartstat -i` command:

```c
#include <stdio.h>
#include <stdlib.h>
#include <libperfstat.h>

int main(int argc, char* argv[]) {
    perfstat_partition_total_t pinfo;
    int rc;

    rc = perfstat_partition_total(NULL, &pinfo, sizeof(perfstat_partition_total_t), 1);
    if (rc != 1) {
        perror("Error in perfstat_partition_total");
        exit(-1);
    }

    printf("Partition Name: %s\n", pinfo.name);
    printf("Partition Number: %u\n", pinfo.lpar_id);
    printf("Type: %s\n", pinfo.type.b.shared_enabled ? "Shared" : "Dedicated");
    printf("Mode: %s\n", pinfo.type.b.donate_enabled ? "Donating" : pinfo.type.b.capped ? "Capped" : "Uncapped");
    printf("Entitled Capacity: %u\n", pinfo.entitled_proc_capacity);
    printf("Partition Group-ID: %u\n", pinfo.group_id);
    printf("Shared Pool ID: %u\n", pinfo.pool_id);
    printf("Online Virtual CPUs: %u\n", pinfo.online_cpus);
    printf("Maximum Virtual CPUs: %u\n", pinfo.max_cpus);
    printf("Minimum Virtual CPUs: %u\n", pinfo.min_cpus);
    printf("Online Memory: %llu MB\n", pinfo.online_memory);
    printf("Maximum Memory: %llu MB\n", pinfo.max_memory);
    printf("Minimum Memory: %llu MB\n", pinfo.min_memory);
    printf("Variable Capacity Weight: %u\n", pinfo.var_proc_capacity_weight);
    printf("Minimum Capacity: %u\n", pinfo.min_proc_capacity);
    printf("Maximum Capacity: %u\n", pinfo.max_capacity);
    printf("Capacity Increment: %u\n", pinfo.proc_capacity_increment);
    printf("Maximum Physical CPUs in system: %u\n", pinfo.max_phys_cpus_sys);
    return 0;
}
```

AIX Version 6.1 Performance Tools Guide and Reference
printf("Active Physical CPUs in system : %u\n", pinfo.online_phys_cpus_sys);
printf("Active CPUs in Pool : %u", pinfo.phys_cpus_pool);
printf("Unallocated Capacity : %u", pinfo.unalloc_proc_capacity);
printf("Physical CPU Percentage : %4.2f%%", (double)pinfo.entitled_proc_capacity / (double)pinfo.online_cpus);
printf("Unallocated Weight : %u", pinfo.unalloc_var_proc_capacity_weight);
}

The program above produces output similar to the following:

Partition Name : aixlpar
Partition Number : 21
Type : Dedicated
Mode : Donating
Entitled Capacity : 35
Partition Group-ID : 43
Shared Pool ID : 93
Online Virtual CPUs : 8
Minimum Virtual CPUs : 6
Maximum Virtual CPUs : 12
Online Memory : 256 MB
Maximum Memory : 512 MB
Minimum Memory : 123 MB
Variable Capacity Weight : 5
Minimum Capacity : 1.5
Maximum Capacity : 3.5
Capacity Increment : 83
Maximum Physical CPUs in system : 11
Active Physical CPUs in system : 8
Physical CPUs in Pool : 9
Unallocated Capacity : 4.5
Physical CPU Percentage : 84.34
Unallocated Weight : 6

The second example demonstrates how to emulate the lparstat command in default mode:

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <libperfstat.h>
#include <sys/systemcfg.h>
#define XINTFRAC ((double)(_system_configuration.Xint)/(double)(_system_configuration.Xfrac))
#define HTIC2SEC(x) ((double)x * XINTFRAC)/(double)1000000000.0

static int disp_util_header = 1;
static u_longlong_t last_time_base;
static u_longlong_t last_pcpu_user, last_pcpu_sys, last_pcpu_idle, last_pcpu_wait;
static u_longlong_t last_lcpu_user, last_lcpu_sys, last_lcpu_idle, last_lcpu_wait;
static u_longlong_t last_phint = 0, last_vcsw = 0, last_pit = 0;
static u_longlong_t last_idle_donated_purr = 0, last_busy_donated_purr = 0;
static u_longlong_t last_busy_stolen_purr = 0, last_idle_stolen_purr = 0;
int donate_flag=0;

void display_lpar_util(void);

int main(int argc, char* argv[])
{
    while (1) {
        display_lpar_util();
        sleep(atoi(argv[1]));
        return(0);
    }
}

/* Save the current values for the next iteration */
void save_last_values(perfstat_cpu_total_t *cpustats, perfstat_partition_total_t *lparstats)
{
    last_vcsw = lparstats->vol_virt_cswitch + lparstats->invol_virt_cswitch;
    last_time_base = lparstats->timebase_last;
    last_phint = lparstats->phantintrs;
    last_pit = lparstats->pool_idle_time;
    last_pcpu_user = lparstats->puser;
    last_pcpu_sys = lparstats->psys;
    last_pcpu_idle = lparstats->pidle;
    last_pcpu_wait = lparstats->pwait;
    last_lcpu_user = cpustats->user;
    last_lcpu_sys = cpustats->sys;
    last_lcpu_idle = cpustats->idle;
    last_lcpu_wait = cpustats->wait;

    if(donate_flag)
    {
        last_idle_donated_purr = lparstats->idle_donated_purr;
        last_busy_donated_purr = lparstats->busy_donated_purr;
        last_busy_stolen_purr = lparstats->busy_stolen_purr;
        last_idle_stolen_purr = lparstats->idle_stolen_purr;
    }
}

void display_lpar_util()
{
    u_longlong_t dlt_pcpu_user, dlt_pcpu_sys, dlt_pcpu_idle, dlt_pcpu_wait;
    u_longlong_t dlt_lcpu_user, dlt_lcpu_sys, dlt_lcpu_idle, dlt_lcpu_wait;
    u_longlong_t dlt_busy_stolen_purr,dlt_idle_stolen_purr;
    u_longlong_t dlt_idle_donated_purr, dlt_busy_donated_purr;
    u_longlong_t vcsw, lcpumie, pcputime;
    u_longlong_t entitled_purr, unused_purr;
    u_longlong_t delta_purr, delta_time_base;
    double phys_proc_consumed, entitlement, percent_ent, delta_sec;
    perfstat_partition_total_t lparstats;
    perfstat_cpu_total_t cpustats;

    /* retrieve the metrics */
    if (!perfstat_partition_total(NULL, &lparstats, sizeof(perfstat_partition_total_t), 1))
       { perror("perfstat_partition_total"); exit(-1); }
    if (!perfstat_cpu_total(NULL, &cpustats, sizeof(perfstat_cpu_total_t), 1))
       { perror("perfstat_cpu_total"); exit(-1); }

    /* Print the header for utilization metrics (only once) */
    if (disp_util_header)
    {
        if (lparstats.type.b.shared_enabled)
        {
            fprintf(stderr, "%5s %5s %6s %6s %5s %5s %5s %4s %5s",
                "%user", "%sys", "%idle", "%physc", "%entc", "%busy", "app", "vcsw", "phint");
            fprintf(stderr, "-----", "----", "-----", "-----", "-----", "-----", "-----", "-----", "-----", "-----", "-----");
        }
        else
        {
            fprintf(stderr, "%5s %5s %6s %6s %5s %5s %5s %4s %5s",
                "%user", "%sys", "%idle", "%physc", "%entc", "%busy", "%app", "%vcsw", "%phint");
            fprintf(stderr, "-----", "----", "-----", "-----", "-----", "-----", "-----", "-----", "-----", "-----", "-----");
        }
    }
    else
    {...}
if(lparstats.type.b.donate_enabled)
{
donate_flag=1;
fprintf(stdout, "\%5s \%5s \%6s \%6s", "%user", "%sys", "%wait", "%idle");
if (donate_flag)
    fprintf(stdout, "\%5s \%5s\%5s\%5s", "%physc", "%vcsw");
if (donate_flag)
    fprintf(stdout, "\%5s \%5s\%5s\%5s", "-----", "----", "-----", "-----");
}
fprintf(stdout, "\n");
disp_util_header = 0;
/* first iteration, we only read the data, print the header and save the data */
save_last_values(&cpustats, &lparstats);
return;
}
dlt_pcpu_user = lparstats.puser - last_pcpu_user;
dlt_pcpu_sys = lparstats.psys - last_pcpu_sys;
dlt_pcpu_idle = lparstats.pidle - last_pcpu_idle;
dlt_pcpu_wait = lparstats.pwait - last_pcpu_wait;
delta_purr = pcputime = dlt_pcpu_user + dlt_pcpu_sys + dlt_pcpu_idle + dlt_pcpu_wait;
dlt_lcpu_user = cpustats.user - last_lcpu_user;
dlt_lcpu_sys = cpustats.sys - last_lcpu_sys;
dlt_lcpu_idle = cpustats.idle - last_lcpu_idle;
dlt_lcpu_wait = cpustats.wait - last_lcpu_wait;
lcpu_time = dlt_lcpu_user + dlt_lcpu_sys + dlt_lcpu_idle + dlt_lcpu_wait;
/* Distribute the donated and stolen purr to the existing purr buckets in case if donation is enabled.*/
if(donate_flag)
{
    u_longlong_t r1, r2;
    dlt_idle_donated_purr = lparstats.idle_donated_purr - last_idle_donated_purr;
    dlt_busy_donated_purr = lparstats.busy_donated_purr - last_busy_donated_purr;
    dlt_idle_stolen_purr = lparstats.idle_stolen_purr - last_idle_stolen_purr;
    dlt_busy_stolen_purr = lparstats.busy_stolen_purr - last_busy_stolen_purr;
    if((dlt_lcpu_idle + dlt_lcpu_wait)!=0)
    {
        r1 = dlt_lcpu_idle / (dlt_lcpu_idle + dlt_lcpu_wait);
        r2 = dlt_lcpu_wait / (dlt_lcpu_idle + dlt_lcpu_wait);
    }
    else
        r1=r2=0;
    dlt_pcpu_user += dlt_idle_donated_purr * r1 + dlt_idle_stolen_purr * r1;
    dlt_pcpu_wait += dlt_idle_donated_purr * r2 + dlt_idle_stolen_purr * r2;
    dlt_pcpu_sys += dlt_busy_donated_purr + dlt_busy_stolen_purr;
    delta_purr += dlt_idle_donated_purr + dlt_busy_donated_purr + dlt_idle_stolen_purr + dlt_busy_stolen_purr;
    pcputime=delta_purr;
}
entitlement = (double)lparstats.entitled_proc_capacity / 100.0 ;
delta_time_base = 1parstats.timebase_last - last_time_base;
if (lparstats.type.b.shared_enabled)
{
    entitled_purr = delta_time_base * entitlement;
    if (entitled_purr < delta_purr) {
        /* when above entitlement, use consumption in percentages */
        entitled_purr = delta_purr;
    }
    unused_purr = entitled_purr - delta_purr;
}
If the program above runs in dedicated - donating mode, the program produces output similar to the following:

%user  %sys  %idle  %physc  %vcsw
-----  -----  -----  ------  ------
 0.1  0.3  0.0  99.5  2.00  172
 0.0  0.2  0.0  99.8  1.99  171
 0.0  0.2  0.0  99.8  1.99  170
If the program above runs in shared mode, the program produces output similar to the following:

```
%user  %sys  %wait  %idle  physc  %entc  lbusy  app  vcsw  pint
-----  ----  -----  -----  -----  -----  -----  ---  ----  ----
50.00  5.00  5.00  30.00  2.5  30.00  65.00  1.1  25  10
50.00  5.00  5.00  30.00  2.5  30.00  65.00  1.1  25  10
50.00  5.00  5.00  30.00  2.5  30.00  65.00  1.1  25  10
50.00  5.00  5.00  30.00  2.5  30.00  65.00  1.1  25  10
```

### 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 AIX 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_diskpath`: Retrieves individual disk path 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:

```c
int perfstat_subsystem(perfstat_id_t *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 sometimes 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` header file in AIX Version 6.1 Files Reference.

The following code shows an example of how `perfstat_cpu` is used:

```c
#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 */
    return 0;
}
```
printf("number of perfstat_cpu_t returned : %d\n", retcode);

for (i = 0; i < retcode; i++) {
    printf("Statistics 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)
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 in AIX Version 6.1 Files Reference.

The following code shows an example of how `perfstat_disk` is used:

```c
#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("Statistics for disk : %s\n", statp[i].name);
        printf("\n" );
        printf("description : %s\n", statp[i].description);
        printf("volume group name : %s\n", statp[i].vgroup);
        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 blocks of %llu bytes\n", statp[i].rblks, statp[i].bsize);
        printf("number of blocks written : %llu blocks of %llu bytes\n", statp[i].wblks, statp[i].bsize);
    }
}
```

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 blocks of 512 bytes
number of blocks written : 768176 blocks of 512 bytes

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 blocks of 512 bytes
number of blocks written : 0 blocks of 512 bytes

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 blocks of 2048 bytes
number of blocks written : 0 blocks of 2048 bytes

**perfsat_diskpath Interface**

The `perfsat_diskpath` function returns a set of structures of type `perfsat_diskpath_t`, which is defined in the `libperfsat.h` file. Selected fields from the `perfsat_diskpath_t` structure include:

- **name**: Path name (`<disk_name>_Path<path_id>`)  
- **xfers**: Total transfers via this path (in KB)  
- **adapter**: Name of the adapter linked to the path

Several other disk path-related metrics (such as the number of blocks read from and written via the path) are also returned. For a complete list, see the `perfsat_diskpath_t` section in the `libperfsat.h` header file in AIX Version 6.1 Files Reference.

The following code shows an example of how `perfsat_diskpath` is used:

```c
#include <stdio.h>
#include <stdlib.h>
#include <libperfsat.h>

int main(int argc, char* argv[]) {
    int i, ret, tot;
    perfsat_diskpath_t *statp;
    perfsat_disk_t dstat;
    perfsat_id_t first;
    char *substring;

    /* check how many perfsat_diskpath_t structures are available */
    tot = perfsat_diskpath(NULL, NULL, sizeof(perfsat_diskpath_t), 0);

    /* allocate enough memory for all the structures */
    statp = calloc(tot, sizeof(perfsat_diskpath_t));

    /* set name to first interface */
    strncpy(first.name, FIRST_DISKPATH);

    /* ask to get all the structures available in one call */
    /* return code is number of structures returned */
    ret = perfsat_diskpath(&first, statp, sizeof(perfsat_diskpath_t), tot);

    /* print statistics for each of the disk paths */
    for (i = 0; i < ret; i++) {
        printf("Statistics for disk path : %s\n", statp[i].name);
        printf("----------------------\n");
        printf("number of blocks read : %llu\n", statp[i].rblks);
        printf("number of blocks written : %llu\n", statp[i].wblks);
        printf("adapter name : %s\n", statp[i].adapter);
    }

    /* retrieve paths for last disk if any */
```
if (ret > 0) {
    /* extract the disk name from the last disk path name */
    substring = strstr(statp[ret - 1].name, "_Path");
    if (substring == NULL) {
        return (-1);
    }
    substring[0] = '\0';
    /* set name to the disk name */
    strcpy(first.name, statp[ret-1]);

    /* retrieve info about disk */
    ret = perfstat_disk(&first, &dstat, sizeof(perfstat_disk_t), 1);
    printf("Paths for disk: %s (%d)
", dstat.name, dstat.paths_count);
    printf("----------------------
");

    /* retrieve all paths for this disk */
    ret = perfstat_diskpath(&first, statp, sizeof(perfstat_diskpath_t), dstat.paths_count);

    /* print statistics for each of the paths */
    for (i = 0; i < ret; i++) {
        printf("Statistics for disk path: %s
", statp[i].name);
        printf("----------------------
");
        printf("number of blocks read: %llu\n", statp[i].rblks);
        printf("number of blocks written: %llu\n", statp[i].wblks);
        printf("adapter name: %s\n", statp[i].adapter);
    }
}

The preceding program produces output similar to the following:

Statistics for disk path : hdisk1_Path0
----------------------
number of blocks read  : 253612
number of blocks written: 537132
adapter name           : scsi0

Statistics for disk path : hdisk2_Path0
----------------------
number of blocks read  : 0
number of blocks written: 0
adapter name           : scsi0

Statistics for disk path : hdisk2_Path1
----------------------
number of blocks read  : 26457
number of blocks written: 43658
adapter name           : scsi2

Paths for disk : hdisk2 (2)
----------------------

Statistics for disk path : hdisk2_Path0
----------------------
number of blocks read  : 0
number of blocks written: 0
adapter name           : scsi0

Statistics for disk path : hdisk2_Path1
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 the `libperfstat.h` header file in AIX Version 6.1 Files Reference.

The following code shows an example of how `perfstat_diskadapter` is used:

```c
#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("Statistics for adapter: \n", statp[i].name);
        printf("----------\n");
        printf("description: \n", statp[i].description);
        printf("number of disks connected: \n", statp[i].number);
        printf("total disk size: \n", statp[i].size);
        printf("total disk free space: \n", statp[i].free);
        printf("number of blocks read: \n", statp[i].rblks);
        printf("number of blocks written: \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 header file in AIX Version 6.1 Files Reference.

The following code shows an example of how perfstat_netinterface is used:

```c
#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);
}
```
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** header file in **AIX Version 6.1 Files Reference.**
The following code shows an example of how `perfstat_protocol` is used:

```c
#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("Statistics for protocol : %s\n", pinfo.name);
        printf("-----------------------\n");
        if (!strcmp(pinfo.name, "ip")) {
            printf("number of input packets : %llu\n", pinfo.u.ip.ipackets);
            printf("number of input errors : %llu\n", pinfo.u.ip.ierrors);
            printf("number of output packets : %llu\n", pinfo.u.ip.opackets);
            printf("number of output errors : %llu\n", pinfo.u.ip.oerrors);
        } else if (!strcmp(pinfo.name, "ipv6")) {
            printf("number of input packets : %llu\n", pinfo.u.ipv6.ipackets);
            printf("number of input errors : %llu\n", pinfo.u.ipv6.ierrors);
            printf("number of output packets : %llu\n", pinfo.u.ipv6.opackets);
            printf("number of output errors : %llu\n", pinfo.u.ipv6.oerrors);
        } else if (!strcmp(pinfo.name, "icmp")) {
            printf("number of packets received : %llu\n", pinfo.u.icmp.received);
            printf("number of packets sent : %llu\n", pinfo.u.icmp.sent);
            printf("number of errors : %llu\n", pinfo.u.icmp.errors);
        } else if (!strcmp(pinfo.name, "icmpv6")) {
            printf("number of packets received : %llu\n", pinfo.u.icmpv6.received);
            printf("number of packets sent : %llu\n", pinfo.u.icmpv6.sent);
            printf("number of errors : %llu\n", pinfo.u.icmpv6.errors);
        } else if (!strcmp(pinfo.name, "udp")) {
            printf("number of input packets : %llu\n", pinfo.u.udp.ipackets);
            printf("number of input errors : %llu\n", pinfo.u.udp.ierrors);
            printf("number of output packets : %llu\n", pinfo.u.udp.opackets);
        } else if (!strcmp(pinfo.name, "tcp")) {
            printf("number of input packets : %llu\n", pinfo.u.tcp.ipackets);
            printf("number of input errors : %llu\n", pinfo.u.tcp.ierrors);
            printf("number of output packets : %llu\n", pinfo.u.tcp.opackets);
        } else if (!strcmp(pinfo.name, "rpc")) {
            printf("client statistics:\n");
            printf("number of connection-oriented RPC requests : %llu\n", pinfo.u.rpc.client.stream.calls);
            printf("number of rejected connection-oriented RPCs : %llu\n", pinfo.u.rpc.client.stream.badcalls);
            printf("number of connectionless RPC requests : %llu\n", pinfo.u.rpc.client.dgram.calls);
            printf("number of rejected connectionless RPCs : %llu\n", pinfo.u.rpc.client.dgram.badcalls);
            printf("server statistics:\n");
            printf("number of connection-oriented RPC requests : %llu\n", pinfo.u.rpc.server.stream.calls);
            printf("number of rejected connection-oriented RPCs : %llu\n", pinfo.u.rpc.server.stream.badcalls);
        }
    } while (1);
    return 0;
}
```

158  AIX Version 6.1 Performance Tools Guide and Reference
printf("number of connectionless RPC requests : %llu\n",
     pinfo.u.rpc.server.dgram.calls);
printf("number of rejected connectionless RPCs : %llu\n",
     pinfo.u.rpc.server.dgram.badcalls);
} else if (strcasecmp(pinfo.name,"nfs")) {
    printf("total number of NFS client requests : %llu\n",
     pinfo.u.nfs.client.calls);
    printf("total number of NFS client failed calls : %llu\n",
     pinfo.u.nfs.client.badcalls);
    printf("total number of NFS server requests : %llu\n",
     pinfo.u.nfs.server.calls);
    printf("total number of NFS server failed calls : %llu\n",
     pinfo.u.nfs.server.badcalls);
    printf("total number of NFS version 2 server calls : %llu\n",
     pinfo.u.nfs.server.public_v2);
    printf("total number of NFS version 3 server calls : %llu\n",
     pinfo.u.nfs.server.public_v3);
} else if (strcasecmp(pinfo.name,"nfsv2")) {
    printf("number of NFS V2 client requests : %llu\n",
     pinfo.u.nfsv2.client.calls);
    printf("number of NFS V2 server requests : %llu\n",
     pinfo.u.nfsv2.server.calls);
} else if (strcasecmp(pinfo.name,"nfsv3")) {
    printf("number of NFS V3 client requests : %llu\n",
     pinfo.u.nfsv3.client.calls);
    printf("number of NFS V3 server requests : %llu\n",
     pinfo.u.nfsv3.server.calls);
}

/* make sure we stop after the last protocol */
if (ret = strcmp(protid.name, "")) {
    printf("next 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("number 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` header file in *AIX Version 6.1 Files Reference*.

The following code shows an example of how `perfstat_netbuffer` is used:

```c
#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 %9s %7s %7s %7s
", "By size", "inuse", "calls", "failed",
        "delayed", "free", "hiwat", "freed");
    for (i = 0; i < ret; i++) {
        printf("%-12s %10llu %9llu %6llu %9llu %7llu %7llu
", 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:
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` header file in *AIX Version 6.1 Files Reference*.

The following code shows an example of how `perfstat_pagingspace` is used:

```c
#include <stdio.h>
#include <stdlib.h>
#include <libperfstat.h>

int main(int argc, char argv[]) {
    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("Statistics 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].u.lv_paging.vgname);
        } else {
            printf("hostname: %s\n", pinfo[i].u.nfs_paging.hostname);
            printf("filename: %s\n", pinfo[i].u.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:
Cached metrics interfaces

Cached metrics interfaces are used when the system configuration changes to inform the libperfstat API that it should reset cached metrics, which consist of values that seldom change such as disk size or CPU description.

The following table lists the metrics that are cached:

<table>
<thead>
<tr>
<th>Object</th>
<th>Content</th>
<th>Sample value</th>
</tr>
</thead>
<tbody>
<tr>
<td>perfstat_cpu_total</td>
<td>char cpu_description [IDENTIFIER_LENGTH]</td>
<td>PowerPC_POWER3</td>
</tr>
<tr>
<td></td>
<td>u_longlong_t processorHZ</td>
<td>3750000000</td>
</tr>
<tr>
<td>perfstat_diskadapter</td>
<td>The list of disk adapters</td>
<td>scsi0, scsi1, ide0</td>
</tr>
<tr>
<td></td>
<td>The number of disk adapters</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>u_longlong_t size</td>
<td>17344</td>
</tr>
<tr>
<td></td>
<td>u_longlong_t free</td>
<td>15296</td>
</tr>
<tr>
<td></td>
<td>char description [IDENTIFIER_LENGTH]</td>
<td>Wide/Ultra-3 SCSI I/O Controller</td>
</tr>
<tr>
<td>perfstat_pagingspace</td>
<td>The list of paging spaces</td>
<td>hd6</td>
</tr>
<tr>
<td></td>
<td>The number of paging spaces</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>char automatic</td>
<td>NFS_PAGING</td>
</tr>
<tr>
<td></td>
<td>char type</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>longlong_t lpsize</td>
<td>512</td>
</tr>
<tr>
<td></td>
<td>longlong_t mbsize</td>
<td>pompeii or rootvg</td>
</tr>
<tr>
<td></td>
<td>char hostname [IDENTIFIER_LENGTH]</td>
<td>/var/tmp/nfsswap/swapfile1</td>
</tr>
<tr>
<td></td>
<td>char filename [IDENTIFIER_LENGTH]</td>
<td></td>
</tr>
<tr>
<td>perfstat_disk</td>
<td>char adapter [IDENTIFIER_LENGTH]</td>
<td>scsi0</td>
</tr>
<tr>
<td></td>
<td>char description [IDENTIFIER_LENGTH]</td>
<td>16 Bit LVD SCSI Disk Drive</td>
</tr>
<tr>
<td></td>
<td>char vname [IDENTIFIER_LENGTH]</td>
<td>rootvg</td>
</tr>
<tr>
<td></td>
<td>u_longlong_t size</td>
<td>17344</td>
</tr>
<tr>
<td></td>
<td>u_longlong_t free</td>
<td>15296</td>
</tr>
<tr>
<td>perfstat_diskpath</td>
<td>char adapter [IDENTIFIER_LENGTH]</td>
<td>scsi0</td>
</tr>
<tr>
<td>perfstat_netinterface</td>
<td>char description [IDENTIFIER_LENGTH]</td>
<td>Standard Ethernet Network Interface</td>
</tr>
</tbody>
</table>

You can use the following AIX interfaces to refresh the cached metrics:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Purpose</th>
<th>Definition of interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>perfstat_reset</td>
<td>Resets every cached metric</td>
<td>void perfstat_reset (void);</td>
</tr>
<tr>
<td>perfstat_partial_reset</td>
<td>Resets selected cached metrics or resets the system's minimum and maximum counters for disks</td>
<td>void perfstat_partial_reset (char * name, u_longlong_t resetmask);</td>
</tr>
</tbody>
</table>

The usage of the parameters for all of the interfaces is as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *name</td>
<td>Identifies the name of the component of the cached metric that should be reset from the libperfstat API cache. If the value of the parameter is NULL, this signifies all of the components.</td>
</tr>
</tbody>
</table>
### The perfstat_reset interface

The `perfstat_reset` interface resets every cached metric that is stored by the `libperfstat` API. It also resets the system's minimum and maximum counters related to disks and paths. To be more selective, it is advised to use the `perfstat_partial_reset` interface.

### The perfstat_partial_reset interface

The `perfstat_partial_reset` interface resets the specified cached metrics that are stored by the `libperfstat` API. The `perfstat_partial_reset` interface can also reset the system's minimum and maximum counters related to disks and paths. The following table summarizes the various actions of the `perfstat_partial_reset` interface:

<table>
<thead>
<tr>
<th>The resetmask value</th>
<th>Action taken when the value of name is NULL</th>
<th>Action taken when the value of name is not NULL and a single resetmask value is set</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FLUSH_CPUTOTAL</code></td>
<td>Flushes the speed and description values in every <code>perfstat_cputotal_t</code> structure.</td>
<td>Error. The value of <code>errno</code> is set to <code>EINVAL</code>.</td>
</tr>
<tr>
<td><code>FLUSH_DISK</code></td>
<td>Flushes the description, adapter, size, free, and vgname values in every <code>perfstat_disk_t</code> structure. Flushes the list of disk adapters. Flushes the size, free, and description values in every <code>perfstat_diskadapter_t</code> structure.</td>
<td>Flushes the description, adapter, size, free, and vgname values in the specified <code>perfstat_disk_t</code> structure. Flushes the adapter value in every <code>perfstat_diskadapter_t</code> structure that matches the disk name that is followed by the <em>Path</em> identifier. Flushes the size, free, and description values of each <code>perfstat_diskadapter_t</code> structure that is linked to a path leading to the disk or to the disk itself.</td>
</tr>
<tr>
<td><code>RESET_DISK_MINMAX</code></td>
<td>Resets the following values in every <code>perfstat_diskadapter_t</code> structure: - <code>wq_min_time</code> - <code>wq_max_time</code> - <code>min_rserv</code> - <code>max_rserv</code> - <code>min_wserv</code> - <code>max_wserv</code></td>
<td>Error. The value of <code>errno</code> is set to <code>ENOTSUP</code>.</td>
</tr>
<tr>
<td>The resetmask value</td>
<td>Action taken when the value of name is NULL</td>
<td>Action taken when the value of name is not NULL and a single resetmask value is set</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FLUSH_DISKADAPTER</td>
<td>Flushes the list of disk adapters.</td>
<td>Flushes the list of disk adapters.</td>
</tr>
<tr>
<td></td>
<td>Flushes the size, free, and description values in every perfstat_diskadapter_t structure.</td>
<td>Flushes the size, free, and description values in every perfstat_diskadapter_t structure.</td>
</tr>
<tr>
<td></td>
<td>Flushes the adapter value in every perfstat_diskpath_t structure.</td>
<td>Flushes the adapter value in every perfstat_diskpath_t structure.</td>
</tr>
<tr>
<td></td>
<td>Flushes the description and adapter values in every perfstat_disk_t structure.</td>
<td>Flushes the description and adapter values in every perfstat_disk_t structure.</td>
</tr>
<tr>
<td>FLUSH_DISKPATH</td>
<td>Flushes the adapter value in every perfstat_diskpath_t structure.</td>
<td>Flushes the adapter value in the specified perfstat_diskpath_t structure.</td>
</tr>
<tr>
<td>FLUSH_PAGINGSPACE</td>
<td>Flushes the list of paging spaces.</td>
<td>Flushes the list of paging spaces.</td>
</tr>
<tr>
<td></td>
<td>Flushes the automatic, type, lpsize, mbsize, hostname, filename, and vgnname values in every perfstat_pagingspace_t structure.</td>
<td>Flushes the automatic, type, lpsize, mbsize, hostname, filename, and vgnname values in the specified perfstat_pagingspace_t structure.</td>
</tr>
<tr>
<td>FLUSH_NETINTERFACE</td>
<td>Flushes the description value in every perfstat_netinterface_t structure.</td>
<td>Flushes the description value in the specified perfstat_netinterface_t structure.</td>
</tr>
</tbody>
</table>

You can see how to use the `perfstat_partial_reset` interface in the following example code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <libperfstat.h>

int main(int argc, char *argv[]) {
    int i, retcode;
    perfstat_id_t diskname;
    perfstat_disk_t *statp;
    
    /* set name of the disk */
    strcpy(diskname.name, "hdisk0");
    
    /* we will now reset global system min/max metrics 
       * Be careful as this could interact with other programs. 
       */
    perfstat_partial_reset(NULL, RESET_DISK_MINMAX);
    
    /* min/max values are now reset. 
       * We can now wait for some time before checking the variation range. 
       */
    sleep(60);
    
    /* get disk metrics - min/max counters illustrate variations during the 
       * last 60 seconds unless someone else reset these 
       * values in the meantime. 
       */
    retcode = perfstat_disk(&diskname, statp, sizeof(perfstat_disk_t), 1);
    
    /* At this point, we assume the disk free part changes due to chfs for example */
    
    /* if we get disk metrics here, the free field will be wrong as it was 
       * cached by the libperfstat. 
       */
    
    /* That is why we reset cached metrics */
    perfstat_partial_reset("hdisk0", FLUSH_DISK);
    
    return retcode;
}
```

Chapter 6. Perfstat API Programming 165
Change History of the perfstat API

The following changes and additions have been made to the perfstat APIs:

**Interface Changes**
Beginning with the following filesets:
- `bos.perf.libperfstat 4.3.3.4`
- `bos.perf.libperfstat 5.1.0.50`
- `bos.perf.libperfstat 5.2.0.10`

the `rbiks` and `wblks` fields of `libperfstat` are represented by blocks of 512 bytes in the `perfstat_disk_total_t`, `perfstat_diskadapter_t` and `perfstat_diskpath_t` structures, regardless of the actual block size used by the device for which metrics are being retrieved.

**Interface Additions**
The following interfaces were added in the `bos.perf.libperfstat 5.2.0` fileset:
- `perfstat_netbuffer`
- `perfstat_protocol`
- `perfstat_pagingspace`
- `perfstat_diskadapter`
- `perfstat_reset`

The `perfstat_diskpath` interface was added in the `bos.perf.libperfstat 5.2.0.10` fileset.

The `perfstat_partition_total` interface was added in the `bos.perf.libperfstat 5.3.0.0` fileset.

The `perfstat_partial_reset` interface was added in the `bos.perf.libperfstat 5.3.0.10` fileset.

**Field Additions**
The following additions have been made to the specified fileset levels:

**The bos.perf.libperfstat 5.1.0.15 fileset**
The following fields were added to `perfstat_cpu_total_t`:

```c
    u_longlong_t brdread
    u_longlong_t bwrwrite
    u_longlong_t lrwdread
    u_longlong_t lwritewrite
    u_longlong_t phread
    u_longlong_t phwrite
```

Support for C++ was added in this fileset level.

Note that the version of `libperfstat` for AIX 4.3 is synchronized with this level. No binary or source compatibility is provided between the 4.3.3.4 version and any 5.1 version prior to 5.1.0.15.

**The bos.perf.libperfstat 5.1.0.25 fileset**
The following fields were added to `perfstat_cpu_t`:
The `bos.perf.libperfstat 5.2.0` fileset

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 ""
```
The bos.perf.libperfstat 5.2.0.10 fileset
The following field was added to the perfstat_disk_t interface:
   uint paths_count

The following define was added to libperfstat.h:
   #define FIRST_DISKPATH ""

The bos.perf.libperfstat 5.3.0.0 fileset
The following fields were added to the perfstat_cpu_t interface:

   u_longlong_t puser
   u_longlong_t psyss
   u_longlong_t pidle
   u_longlong_t pwait
   u_longlong_t redisp_sd0
   u_longlong_t redisp_sd1
   u_longlong_t redisp_sd2
   u_longlong_t redisp_sd3
   u_longlong_t redisp_sd4
   u_longlong_t redisp_sd5
   u_longlong_t migration_push
   u_longlong_t migration_S3grq
   u_longlong_t migration_S3pul
   u_longlong_t invol_cswitch
   u_longlong_t vol_cswitch
   u_longlong_t runque
   u_longlong_t bound
   u_longlong_t decriintrs
   u_longlong_t mpcrintrs
   u_longlong_t mpcsintrs
   u_longlong_t devintrs
   u_longlong_t softintrs
   u_longlong_t phantintrs

The following fields were added to the perfstat_cpu_total_t interface:

   u_longlong_t puser
   u_longlong_t psys
   u_longlong_t pidle
   u_longlong_t pwait
   u_longlong_t decriintrs
   u_longlong_t mpcrintrs
   u_longlong_t mpcsintrs
   u_longlong_t devintrs
   u_longlong_t softintrs
   u_longlong_t phantintrs

The bos.perf.libperfstat 5.3.0.10 fileset
The following fields were added to both the perfstat_disk_t and perfstat_diskpath_t interfaces:

   u_longlong_t q_full
   u_longlong_t rserv
   u_longlong_t rtimeout
   u_longlong_t rfailed
   u_longlong_t min_rserv
   u_longlong_t max_rserv
   u_longlong_t wserv
   u_longlong_t wtimeout
   u_longlong_t wfailed
   u_longlong_t min_wserv
   u_longlong_t max_wserv
   u_longlong_t wq_depth
   u_longlong_t wq_sampled
u_longlong_t wq_time
u_longlong_t wq_min_time
u_longlong_t wq_max_time
u_longlong_t q_sampled

In addition, the xrate field in the following data structures has been renamed to _rxfers and contains the number of read transactions when used with selected device drivers or zero:

    perfstat_disk_t
    perfstat_disk_total_t
    perfstat_diskadapter_t
    perfstat_diskpath_t

The following definitions were added to the `libperfstat.h` header file:

    #define FLUSH_CPUTOTAL
    #define FLUSH_DISK
    #define RESET_DISK_MINMAX
    #define FLUSH_DISKADAPTER
    #define FLUSH_DISKPATH
    #define FLUSH_PAGINGSPACE
    #define FLUSH_NETINTERFACE

**The bos.perf.libperfstat 5.3.0.50 fileset**

The following fields were added to `perfstat_partition_total_t`:

    u_longlong_t reserved_pages
    u_longlong_t reservedpagesize

**The bos.perf.libperfstat 5.3.0.60 fileset**

The following fields were added to `perfstat_cpu_t`, `perfstat_cpu_total_t` and `perfstat_partition_total_t`:

    u_longlong_t idle_donated_purr
    u_longlong_t idle_donated_spurr
    u_longlong_t busy_donated_purr
    u_longlong_t busy_donated_spurr
    u_longlong_t idle_stolen_purr
    u_longlong_t idle_stolen_spurr
    u_longlong_t busy_stolen_purr
    u_longlong_t busy_stolen_spurr

The following flags were added to `perfstat_partition_type_t`:

    unsigned donate_capable
    unsigned donate_enabled

**Related Information**

The `libperfstat.h` file.
Chapter 7. Kernel Tuning

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 `tunchange` command is used to change values in a stanza file.
- The `tunsave` command is used to save values to a stanza file.
- The `tunrestore` command is used to apply a file; that is, to change all tunables parameter values to those listed in a file.
- The `tunchk` command must be used to validate a file created manually.
- The `tundefault` command is available to reset tunable parameters to their default values.

The preceding commands work on both current and reboot values.

All six tuning commands (`no`, `nfs`, `vm`, `ioo`, `ras` and `sched`) 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 that the command manages. A large majority of tunable parameter values are not modifiable when the login session is initiated outside of the global WPAR partition. Attempts to modify such a read only tunable parameter value is refused by the command and a diagnostic message written to standard error output.

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" on page 172
- "Tunables File Directory" on page 172
- "Tunable Parameters Type" on page 173
- "Common Syntax for Tuning Commands" on page 173
- "Tunable File-Manipulation Commands" on page 175
- "Initial setup" on page 179
- "Reboot Tuning Procedure" on page 179
- "Recovery Procedure" on page 179
- "Kernel Tuning Using the SMIT Interface" on page 180
- "Kernel Tuning using the Performance Plug-In for Web-based System Manager" on page 185
- "Files" on page 196
- "Related Information" on page 196

**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 of the information in this section does not apply to compatibility mode. For more information, see AIX 5.2 compatibility mode in Performance management.

When a machine is initially installed with AIX 5.2, it is automatically set to run in AIX 5.2 tuning mode, which is described in this chapter. The tuning 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 six stanzas corresponding to the six tuning commands: schedo, vmo, loo, no, raso, 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 of 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 (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. Restricted tunables modified from their default value are marked, after the value, with an additional comment # RESTRICTED not at 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 (loo, raso, vmo, schedo, no or nfso) with the -p or -r options, which will update the /etc/tunables/nextboot file.
A new file can also be created directly with an editor or copied from another machine. Running `tuncheck [r l -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 have been classified into the following 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
- **Connect**: if changes to the parameter are only effective for future socket connections
- **Deprecated**: if changing this parameter is no longer supported by the current release of AIX

The manual page for each of the six 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.

For parameters of type Bosboot, whenever a change is performed, the tuning commands automatically prompt the user to ask if they want to execute the `bosboot` command. When specifying a restricted tunable for modification in association with the option `-p` or `-r`, you are also prompted to confirm the change. For parameters of type Connect, the tuning commands automatically restart the `inetd` daemon.

The tunables classified as restricted use tunables exist primarily for specialized intervention by the support or development teams and are not recommended for end user modification. For this reason, they are not displayed by default and require the force option on the command line. When modifying a restricted tunable, a warning message is displayed and confirmation required if the change is specified for reboot or permanent.

### Common Syntax for Tuning Commands

The `no`, `nfso`, `vmo`, `ioo`, `raso`, 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] [-F] -a`
- `command -h [tunable]`
- `command [-F] -L [tunable]`
- `command [-F] -x [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. If a tunable is not supported by the running kernel or the current platform, "n/a" 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.

### -F

Forces display of restricted tunable parameters when the options `-a`, `-L`, or `-x` are specified alone on the command line to list all tunables. When `-F` is not specified, restricted tunables are not included in a display unless specifically named in association with a display option.

### -h [tunable]

Displays help about tunable parameter. Otherwise, displays the command usage statement.

### -o tunable[=newvalue]

Displays the value or sets tunable to *newvalue*. If a tunable needs to be changed (the specified value is 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. If a tunable is not supported by the running kernel or the current platform, “n/a” 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.

### -x [tunable]

Lists the characteristics of one or all tunables, one per line, using the following format:

```
tunable,current,default,reboot, min,max,unit,type,{dtunable}
```

where:

- `current` = current value
- `default` = default value
- `reboot` = reboot value
- `min` = minimal value
- `max` = maximum value
- `unit` = tunable unit of measure
- `type` = parameter type: `D` (for Dynamic), `S` (for Static), `R` (for Reboot), `B` (for Bosboot), `M` (for Mount), `I` (for Incremental), `C` (for Connect), and `d` (for Deprecated)
- `dtunable` = space separated list of dependent tunable parameters
-L [tunable]

Lists the characteristics of one or all tunables, one per line, using the following format:

<table>
<thead>
<tr>
<th>NAME</th>
<th>CUR</th>
<th>DEF</th>
<th>BOOT</th>
<th>MIN</th>
<th>MAX</th>
<th>UNIT</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>memory_frames</td>
<td>128K</td>
<td>128K</td>
<td>4KB</td>
<td>pages</td>
<td></td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>maxfree</td>
<td>128</td>
<td>128</td>
<td>16</td>
<td>200K</td>
<td>4KB</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

where:

CUR = current value
DEF = default value
BOOT = reboot value
MIN = minimal value
MAX = maximum value
UNIT = tunable unit of measure
TYPE = parameter type: D (for Dynamic), S (for Static), R (for Reboot), B (for Bosboot), M (for Mount), I (for Incremental), C (for Connect), and d (for Deprecated)

DEPENDENCIES = list of dependent tunable parameters, one per line

Any change (with -o, -d or -D) to a restricted tunable parameter will result in a message being displayed to warn the user that a tunable of the restricted use type has been modified and, if the -r or -p options are also specified on the command line, the user will be prompted for confirmation of the change. In addition, at system reboot, the presence of restricted tunables modified to a value different from their default using a command line specifying the -r or -p options will cause the addition of an error log entry identifying the list of these modified tunables.

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 change (with -o, -d or -D flags) to a parameter of type Connect will result in the inetd daemon being restarted, and a message will display to warn the user that the change is only effective for socket connections.

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 normally manipulate files in the `/etc/tunables` directory, but the files can be located anywhere. Therefore, as long as the file name does not contain a forward slash (/), 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.

**tunchange Command**

The `tunchange` command is used to update one or more tunable stanzas in a file. Its syntax is as follows:

```
tunchange -f filename (-t stanza ( {-o parameter[=value}] | -D ) | -m filename2)
```
where stanza is `schedo`, `vmo`, `ioo`, `raso`, `no`, or `nfso`.

The following is an example of how to update the `pacefork` parameter in the `/etc/tunables/mytunable` directory:
```
tunchange -f mytunable -t schedo -o pacefork=10
```

The following is an example of how to unconditionally update the `pacefork` parameter in the `/etc/tunables/nextboot` directory. This should be done with caution because no warning will be printed if a parameter of type `bosboot` was changed.
```
tunchange -f nextboot -t schedo -o pacefork=10
```

The following is an example of how to clear the `schedo` stanza in the `nextboot` file.
```
tunchange -f nextboot -t schedo -D
```

The following is an example of how to merge the `/home/admin/schedo_conf` file with the current `nextboot` file. If the file to merge contains multiple entries for a parameter, only the first entry will be applied. If both files contain an entry for the same tunable, the entry from the file to merge will replace the current `nextboot` file's value.
```
tunchange -f nextboot -m /home/admin/schedo_conf
```

The `tunchange` command is called by the tuning commands to implement the `-p` and `-r` flags using `-f nextboot`.

**tuncheck Command**

The `tuncheck` command is used to validate a file. Its syntax is as follows:
```
tuncheck [-r|-p] -f filename
```

The following is an example of how to validate the `/etc/tunables/mytunable` file for usage on current values.
```
tuncheck -f mytunable
```

The following is an example of how to validate the `/etc/tunables/nextboot` file or `my_nextboot` file for usage during reboot. Note that the `-r` flag is the only valid option when the file to check is the `nextboot` file.
```
tuncheck -r -f nextboot
```
```
tuncheck -r -f /home/bill/my_nextboot
```

All parameters in the `nextboot` or `my_nextboot` 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 parameters 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 \texttt{-p} is the same as calling it twice; once with no argument, and once with the \texttt{-r} flag. This checks whether a file can be used both immediately, and at reboot time.

\textbf{Note:} Users creating a file with an editor, or copying a file from another machine, must run the \texttt{tuncheck} command to validate their file.

\noindent \textbf{tunrestore Command}

The \texttt{tunrestore} command is used to restore all the parameters from a file. Its syntax is as follows:

\begin{verbatim}
tunrestore \ -R \ | \ [-r] \ -f \ filename
\end{verbatim}

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.

\begin{verbatim}
tunrestore \ -f \ mytunable
\end{verbatim}

\begin{verbatim}
tunrestore \ -f \ /etc/tunables/mytunable
\end{verbatim}

In case of problems, only the changes possible will be made.

For example, the following will change the \texttt{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 \texttt{/etc/tunables/nextboot} file.

\begin{verbatim}
tunrestore \ -r \ -f \ mytunable
\end{verbatim}

If changes to parameters of type \texttt{Bosboot} are detected, the user will be prompted to run the \texttt{bosboot} command.

The following command can only be called from the \texttt{/etc/inittab} file and changes tunable parameters to values from the \texttt{/etc/tunables/nextboot} file.

\begin{verbatim}
tunrestore \ -R
\end{verbatim}

Any problem found or change made is logged in the \texttt{/etc/tunables/lastboot.log} file. A new \texttt{/etc/tunables/lastboot.log} file is always created with the list of current values for all parameters. Any change to restricted tunables from their default values will cause the addition of an error log entry identifying the list of these modified tunables.

If \texttt{filename} does not exist, an error message displays. If the \texttt{nextboot} file does not exist, an error message displays if \texttt{-r} was used. If \texttt{-R} was used, all the tuning parameters of a type other than \texttt{Bosboot} will be set to their default value, and a \texttt{nextboot} file containing only an info stanza will be created. A warning will also be logged in the \texttt{lastboot.log} file.

Except when \texttt{-r} is used, parameters requiring a call to \texttt{bosboot} and a \texttt{reboot} are not changed, but an error message is displayed to indicate they could not be changed. When \texttt{-r} is used, if any parameter of type \texttt{Bosboot} needs to be changed, the user will be prompted to run \texttt{bosboot}. Parameters missing from the file are simply left unchanged, except when \texttt{-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 \texttt{-R}).

\noindent \textbf{tunsave Command}

The \texttt{tunsave} command is used to save current tunable parameter values into a file. Its syntax is as follows:

\begin{verbatim}
tunsave \ [-a|-A]\ -f\ -F\ filename
\end{verbatim}

For example, the following saves all of the current tunable parameter values that are different from their default into the \texttt{/etc/tunables/mytunable} file.
tunsave -f mytunable

If the file already exists, an error message is printed instead. The -F flag must be used to overwrite an existing file.

For example, the following saves all of the current tunable parameter values different from their default into the /etc/tunables/nextboot file.

tunsave -f nextboot

If necessary, the tunsave command will prompt the user to run bosboot.

For example, the following saves all of the current tunable parameters values (including parameters for which default is their value) into the mytunable file.

tunsave -A -f mytunable

This permits 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 parameter values into the /etc/tunables/mytunable file or the mytunable file in the current directory.

tunsave -a -f mytunable

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 permits you to return to a known setup later using the tunrestore command.

**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 example 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 example 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 example 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 beginning of the `/etc/inittab` file:

```bash
tunable:23456789:wait:/usr/bin/tunrestore -R > /dev/console 2>&1
```

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 *Performance management*.

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 using `# DEFAULT VALUE`, 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 in step 2.

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.

All of these error 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 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:

<table>
<thead>
<tr>
<th>Tuning Kernel &amp; Network Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save/Restore All Kernel &amp; Network Parameters</td>
</tr>
<tr>
<td>Tuning Scheduler and Memory Load Control Parameters</td>
</tr>
<tr>
<td>Tuning Virtual Memory Manager Parameters</td>
</tr>
<tr>
<td>Tuning Network Parameters</td>
</tr>
<tr>
<td>Tuning NFS Parameters</td>
</tr>
<tr>
<td>Tuning I/O Parameters</td>
</tr>
<tr>
<td>Tuning RAS Parameters</td>
</tr>
<tr>
<td>Tuning Development Parameters</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Save/Restore All Kernel Tuning Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>View Last Boot Parameters</td>
</tr>
<tr>
<td>View Last Boot Log File</td>
</tr>
<tr>
<td>Save All Current Parameters for Next Boot</td>
</tr>
<tr>
<td>Save All Current Parameters</td>
</tr>
<tr>
<td>Restore All Current Parameters from Last Boot Values</td>
</tr>
<tr>
<td>Restore All Current Parameters from Saved Values</td>
</tr>
<tr>
<td>Reset All Current Parameters To Default Value</td>
</tr>
<tr>
<td>Save All Next Boot Parameters</td>
</tr>
<tr>
<td>Restore All Next Boot Parameters from Last Boot Values</td>
</tr>
<tr>
<td>Restore All Next Boot Parameters from Saved Values</td>
</tr>
<tr>
<td>Reset All Next Boot Parameters To Default Value</td>
</tr>
</tbody>
</table>

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**
   - 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**
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**

   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**

   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**

   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**
Type or a select values for the entry field. Pressing F4 displays a 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

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

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

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 six 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:
Interaction between parameter types and the different SMIT sub-panels

The following table shows the interaction between parameter types and the different SMIT sub-panels:

<table>
<thead>
<tr>
<th>Sub-panel name</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>List All Characteristics of Current Parameters</td>
<td>Lists current, default, reboot, limit values, unit, type and dependencies. This is the output of a tuning command called with the -L option.</td>
</tr>
<tr>
<td>Change / Show Current Parameters</td>
<td>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.</td>
</tr>
<tr>
<td>Change / Show Parameters for Next Boot</td>
<td>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).</td>
</tr>
<tr>
<td>Save Current Parameters for Next Boot</td>
<td>Writes current parameters in the nextboot file, bosboot will be proposed if any parameter of type Bosboot was changed.</td>
</tr>
<tr>
<td>Reset Current Parameters to Default value</td>
<td>Resets current parameters to default values, except those which need a bosboot plus reboot or a reboot (bosboot and reboot type).</td>
</tr>
<tr>
<td>Reset Next Boot Parameters to Default value</td>
<td>Clears values in the nextboot file, and propose bosboot if any parameter of type Bosboot was different from its default value.</td>
</tr>
</tbody>
</table>

Each of the sub-panels behavior is explained in the following sections using examples of the scheduler and memory load control sub-panels:

1. List All Characteristics of Tuning Parameters
   The output of schedo -L is displayed.

2. Change/Show Current Scheduler and Memory Load Control Parameters
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. The following is an example of the Change / Show Scheduler and Memory Load Control Parameters for next boot panel.

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. The following is an example of the Save Current Scheduler and Memory Load Control Parameters for Next Boot panel.

   **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. The following is an example of the 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. The following is an example of the 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 Manager Performance plug-in. The Performance Plug-in is available from the Web-based System Manager main console which looks similar to the following:
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, RAS, 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:
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, raso, 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.
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 30. Web-based System Manager Tuning menu](image)

![Figure 31. Save All Current Parameters for next boot dialog](image)
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.

   ![Save All Current Parameters dialog](image)

   **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` 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 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](image)

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 combo box 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](image)

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 combo box 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 enables all the characteristics of the tunable parameters to be viewed at a glance. The table has two editable columns, Current Value and Next Boot Value. Each cell in these two columns is an editable combobox, with only one predefined value of Default, for the capture of new value for a parameter. Data entered in these columns is validated when pressing ENTER.

![Figure 36. Memory VMM window](image)

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 permitted vary according to parameter types:
- Static parameters do not have an editable cell.
- 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 enabling you to save new values for the parameters listed in the Current Value and Next Boot Value columns of the table. The two options are checked by default. They are:
Selecting Update and apply current values 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 a parameter of type Incremental has a new value smaller than 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 Update next boot values 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.

Selecting both options makes all the desired changes now and for the next reboot.

2. Save Current Parameters for Next Boot

A warning dialog is opened.

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 permits resetting of current or next boot values for all the parameters listed in the table to their default value. Two options are available:

   - 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.
Show Restricted Parameters
Clicking on the **Show Restricted Parameters** check box in the tunable type menu will redisplay the list of tunable parameters to include the restricted use development parameters. The name of a development parameter is followed by the indication "(R)" to distinguish them from others.
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, nfs, no, raso, schedo, tunsave, tunrestore, tuncheck, tundefault, and vmo commands.
The tunables file.
Chapter 8. The procmon tool

This topic provides detailed information about the **procmon** tool and contains the following sections:

- "Overview of the procmon tool"
- "Components of the procmon tool" on page 198
- "Filtering processes" on page 200
- "Performing AIX commands on processes" on page 200

Overview of the procmon tool

You can use the **procmon** tool on systems running AIX 5.3 or later. The **procmon** tool enables you to view and manage the processes running on a system. The **procmon** tool has a graphical interface and displays a table of process metrics that you can sort on the different fields that are provided. The default number of processes listed in the table is 20, but you can change the value in the **Table Properties** panel from the main menu. Only the top processes based on the sorting metric are displayed and the default sorting key is CPU consumption.

The default value of the refresh rate for the table of process metrics is 5 seconds, but you can change the refresh rate by either using the **Table Properties** panel in the main menu or by clicking on the **Refresh** button.

By default, the **procmon** tool displays the following:

- How long a process has been running
- How much CPU resource the processes are using
- Whether processes are being penalized by the system
- How much memory the processes are using
- How much I/O a process is performing
- The priority and nice values of a process
- Who has created a particular process

You can choose other metrics to display from the **Table Properties** panel in the main menu. For more information, see "The process table of the procmon tool" on page 198.

You can filter any of the processes that are displayed. For more information, see "Filtering processes" on page 200.

You can also perform certain AIX performance commands on these processes. For more information, see "Performing AIX commands on processes" on page 200.

The **procmon** tool is a Performance Workbench plugin, so you can only launch the **procmon** tool from within the Performance Workbench framework. You must install the **bos.perf.gtools** fileset by either using the **smitty** tool or the **installp** command. You can then access the Performance Workbench by running the **/usr/bin/perfwb** script.

**Note:** Do not run the **/opt/perfwb/perfwb** binary file.
Components of the procmon tool

The graphical interface of the procmon tool consists of the following components:

- The global statistics area of the procmon tool
- The process table of the procmon tool
- The status line of the Performance Workbench on page 199
- The WPAR table of the procmon tool on page 200

The global statistics area of the procmon tool

The global statistics area is a table that is displayed at the top of the procmon tool window. The global statistics area displays the amount of CPU and memory that is being used by the system. You can refresh the statistics data by either clicking on the Refresh button in the menu bar or by activating the automatic refresh option through the menu bar. To save the statistics information, you can export the table to any of the following file formats:

- XML
- HTML
- CSV

The process table of the procmon tool

The process table is the main component of the procmon tool. The process table displays the various processes that are running on the system, ordered and filtered according to the user configuration. The default value of the number of processes listed in the process table is 20, but you can change this value from the Table Properties panel from the main menu.

The yellow arrow key in the column header indicates the sort key for the process table. The arrow points either up or down, depending on whether the sort order is ascending or descending, respectively. You can change the sort key by clicking on any of the column headers.

You can customize the process table, modify the information on the various processes, and run commands on the displayed processes. By default, the procmon tool displays the following columns:

<table>
<thead>
<tr>
<th>PID</th>
<th>Process identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUPER</td>
<td>Percentage of CPU used per process since the last refresh</td>
</tr>
<tr>
<td>PRM</td>
<td>Percent real memory usage</td>
</tr>
<tr>
<td>ELOGIN</td>
<td>Effective login of the process user</td>
</tr>
<tr>
<td>COMMAND</td>
<td>Short name of the process launched</td>
</tr>
<tr>
<td>WPAR</td>
<td>WPAR of the process</td>
</tr>
</tbody>
</table>

You can choose to display other metrics, like the following:

<table>
<thead>
<tr>
<th>PPID</th>
<th>Parent process identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>NICE</td>
<td>Nice value for the process</td>
</tr>
<tr>
<td>PRI</td>
<td>Priority of the process</td>
</tr>
<tr>
<td>DRSS</td>
<td>Data resident set size</td>
</tr>
<tr>
<td>TRSS</td>
<td>Text resident set size</td>
</tr>
<tr>
<td>STARTTIME</td>
<td>Time when the command started</td>
</tr>
<tr>
<td>EUID</td>
<td>Effective user identifier</td>
</tr>
<tr>
<td>RUID</td>
<td>Real user identifier</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>EGID</td>
<td>Effective group identifier</td>
</tr>
<tr>
<td>RGID</td>
<td>Real group identifier</td>
</tr>
<tr>
<td>THCOUNT</td>
<td>Number of threads used</td>
</tr>
<tr>
<td>CLASSID</td>
<td>Identifier of the class which pertains to the WLM process</td>
</tr>
<tr>
<td>CLASSNAME</td>
<td>Name of the class which pertains to the WLM process</td>
</tr>
<tr>
<td>TOTDISKIO</td>
<td>Disk I/O for that process</td>
</tr>
<tr>
<td>NVCSEW</td>
<td>N voluntary context switches</td>
</tr>
<tr>
<td>NIVCSW</td>
<td>N involuntary context switches</td>
</tr>
<tr>
<td>MINFLT</td>
<td>Minor page faults</td>
</tr>
<tr>
<td>MAJFLT</td>
<td>Major page faults</td>
</tr>
<tr>
<td>INBLK</td>
<td>Input blocks</td>
</tr>
<tr>
<td>OUBLK</td>
<td>Output blocks</td>
</tr>
<tr>
<td>MSGSEND</td>
<td>Messages sent</td>
</tr>
<tr>
<td>MSGRECV</td>
<td>Messages received</td>
</tr>
<tr>
<td>EGROUP</td>
<td>Effective group name</td>
</tr>
<tr>
<td>RGROUP</td>
<td>Real group name</td>
</tr>
</tbody>
</table>

You can use either the table properties or preference to display the metrics you are interested in. If you choose to change the table properties, the new configuration values are set for the current session only. If you change the preferences, the new configuration values are set for the next session of the procmon tool.

There are two types of values listed in the process table:
- Real values
- Delta values

Real values are retrieved from the kernel and displayed in the process table. An example of a real value is the PID, PPID, or TTY.

Delta values are values that are computed from the last-stored measurements. An example of a delta value is the CPU percent for each process, which is computed using the values measured between refreshes.

Below the process table, there is another table that displays the sum of the values for each column of the process table. For example, this table might provide a good idea of the percentage of total CPU used by the top 20 CPU-consuming processes.

You can refresh the data by either clicking on the Refresh button in the menu bar or by activating the automatic refresh option through the menu bar. To save the statistics information, you can export the table to any of the following file formats:
- XML
- HTML
- CSV

**The status line of the Performance Workbench**

The Performance Workbench status line displays the date on which the information was retrieved, as well as the name of the system. The status line is hidden if you activate another view or perspective, but automatically reappears if you refresh the information.
The WPAR table of the procmon tool

A WPAR tabulation displays all the WPAR defined on the system in a table. By default, the procmon tool displays the following columns:

<table>
<thead>
<tr>
<th>Name</th>
<th>WPAR name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hostname</td>
<td>WPAR hostname</td>
</tr>
<tr>
<td>Type</td>
<td>WPAR type, either System or Application</td>
</tr>
<tr>
<td>State</td>
<td>WPAR state—this can have one of the following values: Active, Defined, Transitional, Broken, Paused, Loaded, Error</td>
</tr>
<tr>
<td>Directory</td>
<td>WPAR root directory</td>
</tr>
<tr>
<td>Nb. virtual PIDs</td>
<td>Number of virtual PIDs running in this WPAR</td>
</tr>
</tbody>
</table>

Filtering processes

You can filter processes based on the various criteria that is displayed in the process table. To create a filter, select Table Filters from the menu bar. A new window opens and displays a list of filters.

Performing AIX commands on processes

You can run the following AIX commands on the processes you select in the process table:

- The svmon command
- The renice command
- The kill command
- The following proctools commands:
  - The procf files command
  - The proctree command
  - The procsig command
  - The procstack command
  - The procrun command
  - The procmap command
  - The proclflags command
  - The proccred command
  - The proclidd command

To run any of the above commands on one or more processes, select the processes in the process table and right click your mouse, and select either Commands or Modify and then select the command you want to run. A new window opens, which displays the command output while the command is running. You can interrupt the command by clicking on the STOP button.
Chapter 9. Profiling tools

You can use profiling tools to identify which portions of the program are executed most frequently or where most of the time is spent. Profiling tools are typically used after a basic tool, such as the `vmstat` or `iostat` commands, shows that a CPU bottleneck is causing a performance problem.

Before you begin locating hot spots in your program, you need a fully functional program and realistic data values.

The following is a list of the profiling tools you can use:

- Chapter 2, “X-Windows Performance Profiler (Xprofiler),” on page 3
- “The timing commands”
- “The prof command”
- “The gprof command” on page 203
- “The tprof command” on page 205

The timing commands

Use the timing commands discussed in [Using the time command to measure CPU use] for testing and debugging programs whose performance you are recording and trying to improve.

The output from the `time` command is in minutes and seconds, as follows:

```
real  0m26.72s
user  0m26.53s
sys   0m0.03s
```

The output from the `timex` command is in seconds, as follows:

```
real  26.70
user  26.55
sys   0.02
```

Comparing the `user+sys` CPU time to the real time will give you an idea if your application is CPU-bound or I/O-bound.

**Note:** Be careful when you do this on an SMP system. For more information, see [time and timex Cautions].

The `timex` command is also available through the SMIT command on the Analysis Tools menu, found under Performance and Resource Scheduling. The `-p` and `-s` options of the `timex` command enable data from accounting (`-p`) and the `sar` command (`-s`) to be accessed and reported. The `-o` option reports on blocks read or written.

The prof command

The `prof` command displays a profile of CPU usage for each external symbol, or routine, of a specified program. In detail, it displays the following:

- The percentage of execution time spent between the address of that symbol and the address of the next
- The number of times that function was called
- The average number of milliseconds per call
The **prof** command interprets the profile data collected by the **monitor()** subroutine for the object file (**a.out** by default), reads the symbol table in the object file, and correlates it with the profile file (**mon.out** by default) generated by the **monitor()** subroutine. A usage report is sent to the terminal, or can be redirected to a file.

To use the **prof** command, use the **-p** option to compile a source program in C, FORTRAN, PASCAL, or COBOL. This inserts a special profiling startup function into the object file that calls the **monitor()** subroutine to track function calls. When the program is executed, the **monitor()** subroutine creates a **mon.out** file to track execution time. Therefore, only programs that explicitly exit or return from the main program cause the **mon.out** file to be produced. Also, the **-p** flag causes the compiler to insert a call to the **mcount()** subroutine into the object code generated for each recompiled function of your program. While the program runs, each time a parent calls a child function, the child calls the **mcount()** subroutine to increment a distinct counter for that parent-child pair. This counts the number of calls to a function.

**Note:** You cannot use the **prof** command for profiling optimized code.

By default, the displayed report is sorted by decreasing percentage of CPU time. This is the same as when specifying the **-t** option.

The **-c** option sorts by decreasing number of calls and the **-n** option sorts alphabetically by symbol name.

If the **-s** option is used, a summary file **mon.sum** is produced. This is useful when more than one profile file is specified with the **-m** option (the **-m** option specifies files containing monitor data).

The **-z** option includes all symbols, even if there are zero calls and time associated.

Other options are available and explained in the **prof** command in the *AIX Version 6.1 Commands Reference*.

The following example shows the first part of the **prof** command output for a modified version of the Whetstone benchmark (Double Precision) program.

```
# cc -o cwhet -p -lm cwhet.c
# cwhet > cwhet.out
# prof
Name       %Time Seconds Cumsecs #Calls msec/call
.main      32.6  17.63    17.63  1 17630.
._mcount   28.2  15.25    32.88 
.mod8      16.3  8.82     41.70  8990000 0.0010
.mod9      9.9   5.38    47.08  6160000 0.0009
.cos       2.9   1.57    48.65  9200000 0.0008
.exp       2.4   1.32    49.97  9300000 0.0014
.log       2.4   1.31    51.28  9300000 0.0014
.mod3      1.9   1.01    52.29 1400000 0.0072
.sin       1.2   0.63    52.92  640000 0.0010
.sqrt      1.1   0.59    53.51 
.atan      1.1   0.57    54.08  640000 0.0009
.pout      0.0   0.00     54.08  1 0.0
.exit      0.0   0.00     54.08  1 0.
.free      0.0   0.00     54.08  2 0.
.free_y    0.0   0.00     54.08  2 0.
```

In this example, we see many calls to the **mod8()** and **mod9()** routines. As a starting point, examine the source code to see why they are used so much. Another starting point could be to investigate why a routine requires so much time.

**Note:** If the program you want to monitor uses a **fork()** system call, be aware that the parent and the child create the same file (**mon.out**). To avoid this problem, change the current directory of the child process.
The `gprof` command

The `gprof` command produces an execution profile of C, PASCAL, FORTRAN, or COBOL programs. The statistics of called subroutines are included in the profile of the calling program. The `gprof` command is useful in identifying how a program consumes CPU resources. It is roughly a superset of the `prof` command, giving additional information and providing more visibility to active sections of code.

Implementation of the `gprof` command

The source code must be compiled with the `-pg` option. This action links in versions of library routines compiled for profiling and reads the symbol table in the named object file (`a.out` by default), correlating it with the call graph profile file (`gmon.out` by default). This means that the compiler inserts a call to the `mcount()` function into the object code generated for each recompiled function of your program. The `mcount()` function counts each time a parent calls a child function. Also, the `monitor()` function is enabled to estimate the time spent in each routine.

The `gprof` command generates two useful reports:

- The call-graph profile, which shows the routines, in descending order by CPU time, plus their descendants. The profile permits you to understand which parent routines called a particular routine most frequently and which child routines were called by a particular routine most frequently.
- The flat profile of CPU usage, which shows the usage by routine and number of calls, similar to the `prof` output.

Each report section begins with an explanatory part describing the output columns. You can suppress these pages by using the `-b` option.

Use `-s` for summaries and `-z` to display routines with zero usage.

Where the program is executed, statistics are collected in the `gmon.out` file. These statistics include the following:

- The names of the executable program and shared library objects that were loaded
- The virtual memory addresses assigned to each program segment
- The `mcount()` data for each parent-child
- The number of milliseconds accumulated for each program segment

Later, when the `gprof` command is issued, it reads the `a.out` and `gmon.out` files to generate the two reports. The call-graph profile is generated first, followed by the flat profile. It is best to redirect the `gprof` output to a file, because browsing the flat profile first might answer most of your usage questions.

The following example shows the profiling for the `cwhet` benchmark program. This example is also used in "The prof command" on page 201:

```
# cc -o cwhet -pg -lm cwhet.c
# cwhet > cwhet.out
# gprof cwhet > cwhet.gprof
```

The call-graph profile

The call-graph profile is the first part of the `cwhet.gprof` file and looks similar to the following:

```
granularity: each sample hit covers 4 byte(s) Time: 62.85 seconds

<table>
<thead>
<tr>
<th>index</th>
<th>%time</th>
<th>self</th>
<th>descendents</th>
<th>called/total</th>
<th>parents</th>
<th>called+self</th>
<th>name</th>
<th>index</th>
<th>children</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>64.6</td>
<td>19.44</td>
<td>21.18</td>
<td>1/1</td>
<td>.start</td>
<td>.main</td>
<td>.mod</td>
<td>8.89</td>
<td>0.00 899000/899000</td>
</tr>
</tbody>
</table>
```

Chapter 9. Profiling tools
Usually the call graph report begins with a description of each column of the report, but it has been deleted in this example. The column headings vary according to type of function (current, parent of current, or child of current function). The current function is indicated by an index in brackets at the beginning of the line. Functions are listed in decreasing order of CPU time used.

To read this report, look at the first index [1] in the left-hand column. The .main function is the current function. It was started by .__start (the parent function is on top of the current function), and it, in turn, calls .mod8 and .mod9 (the child functions are beneath the current function). All the accumulated time of .main is propagated to .__start. The self and descendents columns of the children of the current function add up to the descendents entry for the current function. The current function can have more than one parent. Execution time is allocated to the parent functions based on the number of times they are called.

**Flat profile**
The flat profile sample is the second part of the cwhet.gprof file and looks similar to the following:

granularity: each sample hit covers 4 byte(s) Total time: 62.85 seconds

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>time</th>
<th>self seconds</th>
<th>calls</th>
<th>self ms/call</th>
<th>total ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.9</td>
<td>19.44</td>
<td>19.44</td>
<td>1</td>
<td>19440.00</td>
<td>40620.00</td>
<td>.main [1]</td>
</tr>
<tr>
<td>30.5</td>
<td>38.61</td>
<td>19.17</td>
<td>1</td>
<td>19440.00</td>
<td>40620.00</td>
<td>.mcount [3]</td>
</tr>
<tr>
<td>14.1</td>
<td>47.50</td>
<td>8.89</td>
<td>899000</td>
<td>0.00</td>
<td>0.00</td>
<td>.mod8 [4]</td>
</tr>
<tr>
<td>9.0</td>
<td>53.14</td>
<td>5.64</td>
<td>616000</td>
<td>0.00</td>
<td>0.00</td>
<td>.mod9 [5]</td>
</tr>
<tr>
<td>2.5</td>
<td>54.72</td>
<td>1.58</td>
<td>930000</td>
<td>0.00</td>
<td>0.00</td>
<td>.exp [6]</td>
</tr>
<tr>
<td>2.4</td>
<td>56.25</td>
<td>1.53</td>
<td>192000</td>
<td>0.00</td>
<td>0.00</td>
<td>.cos [7]</td>
</tr>
<tr>
<td>2.2</td>
<td>57.62</td>
<td>1.37</td>
<td>930000</td>
<td>0.00</td>
<td>0.00</td>
<td>.log [8]</td>
</tr>
<tr>
<td>2.0</td>
<td>58.88</td>
<td>1.26</td>
<td>640000</td>
<td>0.00</td>
<td>0.00</td>
<td>.qincrement [9]</td>
</tr>
<tr>
<td>1.6</td>
<td>59.90</td>
<td>1.02</td>
<td>140000</td>
<td>0.01</td>
<td>0.01</td>
<td>.mod3 [10]</td>
</tr>
<tr>
<td>1.2</td>
<td>60.68</td>
<td>0.78</td>
<td>640000</td>
<td>0.00</td>
<td>0.00</td>
<td>.__stack_pointer [11]</td>
</tr>
<tr>
<td>1.0</td>
<td>61.31</td>
<td>0.63</td>
<td>640000</td>
<td>0.00</td>
<td>0.00</td>
<td>.atan [12]</td>
</tr>
<tr>
<td>0.9</td>
<td>61.89</td>
<td>0.58</td>
<td>640000</td>
<td>0.00</td>
<td>0.00</td>
<td>.qincrement1 [13]</td>
</tr>
<tr>
<td>0.8</td>
<td>62.41</td>
<td>0.52</td>
<td>640000</td>
<td>0.00</td>
<td>0.00</td>
<td>.sin [14]</td>
</tr>
<tr>
<td>0.7</td>
<td>62.85</td>
<td>0.44</td>
<td>640000</td>
<td>0.00</td>
<td>0.00</td>
<td>.sqrt [15]</td>
</tr>
<tr>
<td>0.0</td>
<td>62.85</td>
<td>0.00</td>
<td>180</td>
<td>0.00</td>
<td>0.00</td>
<td>.fwrite [16]</td>
</tr>
<tr>
<td>0.0</td>
<td>62.85</td>
<td>0.00</td>
<td>180</td>
<td>0.00</td>
<td>0.00</td>
<td>.memchr [17]</td>
</tr>
<tr>
<td>0.0</td>
<td>62.85</td>
<td>0.00</td>
<td>90</td>
<td>0.00</td>
<td>0.00</td>
<td>.__flsbuf [18]</td>
</tr>
<tr>
<td>0.0</td>
<td>62.85</td>
<td>0.00</td>
<td>90</td>
<td>0.00</td>
<td>0.00</td>
<td>.flsbuf [19]</td>
</tr>
</tbody>
</table>

The flat profile is much less complex than the call-graph profile and very similar to the output of the prof command. The primary columns of interest are the self seconds and the calls columns. These reflect the CPU seconds spent in each function and the number of times each function was called. The next columns to look at are self ms/call (CPU time used by the body of the function itself) and total ms/call (time in the body of the function plus any descendant functions called).

Normally, the top functions on the list are candidates for optimization, but you should also consider how many calls are made to the function. Sometimes it can be easier to make slight improvements to a frequently called function than to make extensive changes to a piece of code that is called once.
A cross reference index is the last item produced and looks similar to the following:

Index by function name

```
[23] .__ioctl [33] .free_y [28] .nl_langinfo
```

Note: If the program you want to monitor uses a fork() system call, be aware that by default, the parent and the child create the same file, gmon.out. To avoid this problem, use the GPROF environment variable. You can also use the GPROF environment variable to profile multi-threaded applications.

The **tprof** command

The typical program execution is a variable combination of application code, library subroutines, and kernel services. Frequently, programs that have not been tuned expend most of their CPU cycles in certain statements or subroutines. You can determine which particular statements or subroutines to examine with the **tprof** command.

The **tprof** command is a versatile profiler that provides a detailed profile of CPU usage by every process ID and name. It further profiles at the application level, routine level, and even to the source statement level and provides both a global view and a detailed view. In addition, the **tprof** command can profile kernel extensions, stripped executable programs, and stripped libraries. It does subroutine-level profiling for most executable programs on which the **stripnm** command produces a symbols table. The **tprof** command can profile any program produced by any of the following compilers:

- C
- C++
- FORTRAN
- Java™

The **tprof** command only profiles CPU activity. It does not profile other system resources, such as memory or disks.

The **tprof** command can profile Java programs using Java Persistence API (JPA) (-x java -Xrunjpa) to collect Java Just-in-Time (JIT) source line numbers and instructions, if the following parameters are added to -Xrunjpa:

- **source=1**; if IBM® Java Runtime Environment (JRE) 1.5.0 is installed, this parameter enables JIT source line collecting.
- **instructions=1**; enables JIT instructions collecting.

You can use the following types of profiling with the **tprof** command:

- "Time-based profiling" on page 206
- "Event-based profiling" on page 206
Time-based profiling

Time-based profiling is the default profiling mode and it is triggered by the decremener interrupt, which occurs every 10 milliseconds. With time-based profiling, the tprof command cannot determine the address of a routine when interrupts are disabled. While interrupts are disabled, all ticks are charged to the unlock_enable() routines.

Event-based profiling

Event-based profiling is triggered by any one of the software-based events or any Performance Monitor event that occurs on the processor. The primary advantages of event-based profiling over time-based profiling are the following:

- The routine addresses are visible when interrupts are disabled.
- The ability to vary the profiling event
- The ability to vary the sampling frequency

With event-based profiling, ticks that occur while interrupts are disabled are charged to the proper routines. Also, you can select the profiling event and sampling frequency. The profiling event determines the trigger for the interrupt and the sampling frequency determines how often the interrupt occurs. After the specified number of occurrences of the profiling event, an interrupt is generated and the executing instruction is recorded.

Note: Event-based profiling is not supported in manual offline mode.

The default type of profiling event is processor cycles. The various types of software-based events include the following:

- Emulation interrupts (EMULATION)
- Alignment interrupts (ALIGNMENT)
- Instruction Segment Lookaside Buffer misses (ISLBMISS)
- Data Segment Lookaside Buffer misses (DSLBMISS)

The sampling frequency for the software-based events is specified in milliseconds and the supported range is 1 to 500 milliseconds. The default sampling frequency is 10 milliseconds.

The following command generates an interrupt every 5 milliseconds and retrieves the record for the last emulation interrupt:

```
# tprof -E EMULATION -f 5
```

The following command generates an interrupt every 100 milliseconds and records the contents of the Sampled Instruction Address Register, or SIAR:

```
# tprof -E -f 100
```

The other types of profiling events, the Performance Monitor events, include the following:

- Completed instructions
- Cache misses

For a list of all the Performance Monitor events that are supported on the processors of the system, use the pmlist command. The chosen Performance Monitor event must be taken in a group where we can also find the PM_INST_CMPL Performance Monitor event. The sampling frequency for these events is specified in the number of occurrences of the event. The supported range is 10,000 to MAXINT occurrences. The default sampling frequency is 10,000 occurrences.

The following command generates an interrupt after the processor completes 50,000 instructions:

```
# tprof -E PM_INST_CMPL -f 50000
```
Event-based profiling uses the SIAR, which contains the address of an instruction close to the executing instruction. For example, if the profiling event is PM_FPU0_FIN, which means the floating point unit 0 produces a result, the SIAR might not contain that floating point instruction but might contain another instruction close to it. This is more relevant for profiling based on Performance Monitor events. In fact for the proximity reason, on systems based on POWER4 and later, it is recommended that the Performance Monitor profiling event be one of the marked events. Marked events have the PM_MRK prefix.

Certain combinations of profiling event, sampling frequency, and workload might cause interrupts to occur at such a rapid rate that the system spends most of its time in the interrupt handler. The tprof command detects this condition by keeping track of the number of completed instructions between two consecutive interrupts. When the tprof command detects five occurrences of the count falling below the acceptable limit, the trace collection stops. Reports are still generated and an error message is displayed. The default threshold is 1,000 instructions.

**Implementation of the tprof command**

The tprof command uses the system trace facility. Since you can only execute the trace facility one user at a time, you can only execute one tprof command at a time.

You can obtain the raw data for the tprof command through the trace facility. For more information about the trace facility, see [Analyzing Performance with the Trace Facility](#) in the *Performance management*.

When a program is profiled, the trace facility is activated and instructed to collect data from the trace hook with hook ID 234 that records the contents of the Instruction Address Register, or IAR, when a system-clock interrupt occurs (100 times a second per processor). Several other trace hooks are also activated to enable the tprof command to track process and dispatch activity. The trace records are not written to a disk file. They are written to a pipe that is read by a program that builds a table of the unique program addresses that have been encountered and the number of times each one occurred. When the workload being profiled is complete, the table of addresses and their occurrence counts are written to disk. The data-reduction component of the tprof command then correlates the instruction addresses that were encountered with the ranges of addresses occupied by the various programs and reports the distribution of address occurrences, or ticks, across the programs involved in the workload.

The distribution of ticks is roughly proportional to the CPU time spent in each program, which is 10 milliseconds per tick. After the high-use programs are identified, you can take action to restructure the hot spots or minimize their use.

**An example of the tprof command**

You can view the complete details of the tprof command in *AIX Version 6.1 Commands Reference*.

The following example demonstrates how to collect a CPU tick profile of a program using the tprof command. The example was executed on a 4-way SMP system and since it is a fast-running system, the command completed in less than a second. To make this program run longer, the array size, or Asize, was changed to 4096 instead of 1024.

Upon running the following command, the `version1.prof` file is created in the current directory:

```
# tprof -z -u -p version1 -x version1
```

The `version1.prof` file reports how many CPU ticks for each of the programs that were running on the system while the `version1` program was running.

The following is an example of what the `version1.prof` file contains:

<table>
<thead>
<tr>
<th>Process</th>
<th>Freq</th>
<th>Total</th>
<th>Kernel</th>
<th>User</th>
<th>Shared</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait</td>
<td>4</td>
<td>5810</td>
<td>5810</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>./version1</td>
<td>1</td>
<td>1672</td>
<td>35</td>
<td>1637</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
The first section of the report summarizes the results by program, regardless of the process ID, or PID. It shows the number of different processes, or Freq, that ran each program at some point.

The second section of the report displays the number of ticks consumed by, or on behalf of, each process. In the example, the version1 program used 1637 ticks itself and 35 ticks occurred in the kernel on behalf of the version1 process.

The third section breaks down the user ticks associated with the executable program being profiled. It reports the number of ticks used by each function in the executable program and the percentage of the total run's CPU ticks (7504) that each function's ticks represent. Since the system's CPUs were mostly idle, most of the 7504 ticks are idle ticks.

To see what percentage of the busy time this program took, subtract the wait thread's CPU ticks, which are the idle CPU ticks, from the total and then divide the difference from the total number of ticks.

Total number of ticks / (Total - Idle CPU ticks) = % busy time of program
1637 / (7504 - 5810) =
1637 / 1694 = 0.97

So, the percentage of system busy ticks is 97%.

208 AIX Version 6.1 Performance Tools Guide and Reference
The raso tunables

As the root user, you can tune the sampling frequency with the following raso tunables:

- tprof_cyc_mult
- tprof_evt_mult

For example, for events based on processor cycles, setting the tprof_cyc_mult tunable to 50 and specifying the -f flag as 100 is equivalent to specifying a sampling frequency of 100/50 milliseconds.

For other Performance Monitor events, setting the tprof_evt_mult tunable to 100 and specifying the -f flag as 20,000 is equivalent to specifying a sampling frequency of 20,000/100 occurrences.

As the root user, you can tune the instruction threshold with the tprof_inst_threshold tunable of the raso command.

Manual offline processing with the tprof command

You can perform offline processing of trace files with the tprof command, but you must specify filenames with a rootstring name. Also, there are certain suffixes required for the input files that the tprof command uses. For example, the trace binary file must end in .trc. Also, you need to collect the gensyms command output and put it in a file called the rootstring.syms file.

To insure the trace file contains sufficient information to be post-processed by tprof, the trace command line must include the -M and -j tprof flags.

If you name the rootstring file trace1, to collect a trace, you can use the trace command using all of the hooks or at least the following hooks:

```
# trace -af -M -T 1000000 -L 10000000 -o trace1.trc -j tprof
# workload
# trcoff
# gensyms > trace1.syms
# trcstop
# trcstop -r trace1 -k -u -s -z
```

The example above creates a trace1.prof file, which gives you a CPU profile of the system while the trace command was running.
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 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
- Hypervisor
- IBM

Java and all Java-based trademarks and logos are registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

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

A
a.out file 6
about this book v
API calls
  basic
    pm_delete_program 119
    pm_get_data 119
    pm_get_program 119
    pm_get_tdata 119
    pm_set_data 119
    pm_set_program 119
    pm_start 119
    pm_stop 119
    pm_tstart 119
    pm_tstop 119
applications
  compiling for Xprofiler 4

B
binary executable
  specifying from Xprofiler GUI 12

C
Call Graph Profile report 43
  calls between functions, how depicted 24
  clustering functions 33
  clusters, library 25
code
  disassembler
    viewing 52
  source
    viewing 50
command-line flags
  specifying from Xprofiler GUI 14
  Xprofiler 6
commands
  gprof 203
  prof 201
  tprof 205
configuration files
  saving 49
configuration files
  loading 50
controlling how the display is updated 25
counter multiplexing mode 121
  pm_get_data_mx 121
  pm_get_program_mx 121
  pm_get_tdata_mx 121
  pm_set_program_mx 121
CPU Utilization Reporting Tool
  see curt 63
curt (continued)
  Application Summary by Process ID (PID)
    Report 73
  Application Summary by Thread ID (Tid) Report 72
default reports 67
  Event Explanation 64
  Event Name 64
  examples 65
  FILH Summary Report 81
  flags 63
  FLIH types 82
  General Information 68
  Global SLIH Summary Report 82
  Hook ID 64
  Kproc Summary (by Tid) Report 74
  measurement and sampling 64
  parameters
    gensymsfile 63
    inputfile 63
    outputfile 63
    pidnamefile 63
    timestamp 63
    trcnmfile 63
  Pending Pthread Calls Summary Report 80
  Pending System Calls Summary Report 77
  Processor Summary Report 70
  Pthread Calls Summary Report 80
  report overview 65
  sample report
    -e flag 83
    -p flag 87
    -P flag 90
    -s flag 84
    -t flag 85
  syntax 63
  System Calls Summary Report 76
  System Summary Report 68
customizable resources
  Xprofiler 56

D
data
  basic 37
  detailed 41
  getting from reports 41
  performance 37
disassembler code
  viewing 52
disk space requirements 5
display
  Xprofiler 20

E
texts
  examples
    performance monitor APIs 122
features
- X-Windows
  - customizing 56
file
- binary executable
  - specifying from Xprofiler GUI 12
- profile data
  - specifying from Xprofiler GUI 13
files
- loading from Xprofiler GUI 10
filtering, function call tree 27
finding objects in call tree 35
flags
- specifying from Xprofiler GUI 14
  - Xprofiler 6
Flat Profile report 42
function call tree
- clustering 32
- controlling graphic style 25
- controlling orientation of 25
- controlling representation of 26
- displaying 28
- excluding specific objects 28
- filtering 27
- including specific objects 28
- restoring 27
Function Index report 45
functions, how depicted 22

G
gennames utility 98
Global Actions on Tunable Parameters 187
gmon.out file 6
gprof
  - and Xprofiler 4

I
info stanza 172
installp 5
introduction 1
iso 9000  v

K
kernel tuning (continued)
  - introduction 171, 185
  - migration and compatibility 171
  - reboot tuning procedures 179
  - recovery procedure 179
  - SMIT interface 180
  - tunable parameters 171
  - tunables file directory 172
  - tunables parameters
    - type 173
    - Web-based System Manager 185
kernel tuning 171
attributes
  - pre520tune 171
commands 171
  - flags 173
  - tunchange 175
  - tuncheck 176
  - tundefault 178
  - tunrestore 177
  - tunsave 177
commands syntax 173
file manipulation commands 175
initial setup 179

L
lastboot 172
lastboot.log 172
library clusters 25
Library Statistics report 47
limitations
  - Xprofiler 3
  - locating objects in call tree 35

N
nextboot 172

O
objects, locating in call tree 35

P
parameter details 194
performance data, getting 37
performance monitor API
  - accuracy 115
  - common rules 117
  - context and state 116
    - state inheritance 116
    - system level context 116
    - thread context 116
    - thread counting-group and process context 116
  - programming 115
  - security considerations 117
  - thread accumulation 116
  - thread group accumulation 116
performance monitor plug-in 185
perfstat 137
  - characteristics 137
  - component-specific interfaces 149
  - global interfaces 137
  - perfstat_cpu interface 150
  - perfstat_cpu_total Interface 138
  - perfstat_disk interface 151
  - perfstat_disk_total Interface 142
  - perfstat_diskadapter interface 155
  - perfstat_diskpath interface 153
  - perfstat_memory_total Interface 141
  - perfstat_netbuffer interface 161
  - perfstat_netinterface interface 156

214  AIX Version 6.1 Performance Tools Guide and Reference
perfstat (continued)
  perfstat_netinterface_total Interface 143
  perfstat_pagingspace interface 162
  perfstat_partition_total Interface 144
  perfstat_protocol interface 157
perfstat API programming
  see perfstat 137
Plug-In for Web-based System Manager System Tuning 185
  pm_delete_program 117
  pm_error 117
  pm_groups_info_t 118
  pm_info_t 118
  pm_init API initialization 118
  pm_initialize 117
  pm_initialize API initialization 118
  pm_set_program 117
pmapi library 117
procmom tool 197
profile data files
  specifying from Xprofiler GUI 13
profiled data
  saving screen images of 54
profiling 201
programs
  compiling for Xprofiler 4

R
reboot procedure 179
recovery procedure 179
related publications v
release specific features 166
reports
  Call Graph Profile 43
  Flat Profile 42
  Function Index 45
  getting data from 41
  Library Statistics 47
  saving to a file 48
requirements
  Xprofiler 3
resource settings
  Xprofiler 56
resource variables
  Xprofiler 57
resources
  Xprofiler
    customizing 56
resources, customizable
  Xprofiler 56

S
screen images
  saving 54
search file sequence
  setting 19
settings, resource
  Xprofiler 56

simple performance lock analysis tool (splat)
  see splat 95
SMT Interface 180
software requirements 5
source code
  viewing 50
splat 95
  address-to-name resolution 98
AIX kernel lock details 101
  flags 95
condition-variable report 112
event explanation 96
event name 96
execution, trace, and analysis intervals 97
hook ID 96
measurement and sampling 96
mutex function detail 110
mutex pthread detail 110
mutex reports 108
parameters 95
PThread synchronizer reports 108
read/write lock reports 111
reports 98
  execution summary 98
  gross lock summary 99
  per-lock summary 100
simple and runQ lock details 102, 104
trace discontinuities 97

T
text highlighting v
thread counting-group information 120
  consistency flag 120
  member count 120
  process flag 120
timing commands 201
tunable parameters
  global actions 187
tunables 172
tuncheck 172
tundefault 172
tuning tables
  actions 192
  using 191
tunrestore 172
tunsave 172

U
unclustering functions 34

V
variables, resource
  Xprofiler 57
X

X-Windows
  features
    customizing 56
X-Windows Performance Profiler (Xprofiler)
  see Xprofiler 3
Xprofiler 3
  about 3
  and gprof 4
  before you begin 3
  binary executable file
    specifying 12
  command-line flags 6
    specifying from GUI 14
  compiling applications for 4
  controlling fonts 57
  customizable resources 56
  display 20
  file menu
    controlling variables 58
  files and directories created 6
  filter menu
    controlling variables 60
  hidden menus 22
  how installation alters system 6
  installing 5
    using SMIT 5
  limitations 3, 5
  loading files from GUI 10
  main menus 21
  main window 20, 57
  profile data files
    specifying 13
  requirements 3
  resource settings 56
  resource variables 57
  resources
    customizing 56
  screen dump
    controlling variables 58
  setting search file sequence 19
  starting 6
  view menu
    controlling variables 60
Xprofiler installation information 4
Xprofiler preinstallation information 5
Readers’ Comments — We’d Like to Hear from You

AIX Version 6.1
Performance Tools Guide and Reference

Publication No. SC23-5254-00

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

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. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.
Submit your comments using one of these channels:
• Send your comments to the address on the reverse side of this form.
• Send your comments via e-mail to: aix6koub@austin.ibm.com

If you would like a response from IBM, please fill in the following information:

Name

Address

Company or Organization

Phone No.

E-mail address