IBM High Performance Computing Toolkit

Installation and Usage Guide

Table of Contents

List of Figures

Introduction

The IBM® High Performance Computing Toolkit (IBM HPC Toolkit) is a collection of tools that you can use to analyze the performance of both parallel and serial applications, written in C or FORTRAN, over the AIX® or Linux® operating systems on IBM Power Systems™ Servers. The **Xprof** GUI also supports C++ applications. These tools perform the following functions:

- Provide access to hardware performance counters for performing low level analysis of an application, including analyzing cache utilization and floating point performance.
- Profile and trace an MPI application for analyzing MPI communication patterns and performance problems.
- Profile an OpenMP application for analyzing OpenMP performance problems and to help you determine if an OpenMP application properly structures its processing for best performance.
- Profile application I/O for analyzing an application's I/O patterns and you can improve the application's I/O performance.
- Profile an application's execution for identifying hotspots in the application, and for locating relationships between functions in your application to help you better understand the application's performance.

The IBM HPC Toolkit provides two primary interfaces for you to use. The first is peekperf. **Peekperf** is a GUI that you can use to run hardware performance counter analysis, MPI profiling and tracing, OpenMP profiling, and I/O profiling. **Peekperf** allows you to select the parts of your application that are to be instrumented, instrument those parts of the application, run the instrumented application and view the resulting performance data. **Peekperf** allows you to sort and filter the data to help you find the performance problems in the application.

The second interface is **Xprof**, which you can use to view low level profiling data for your application. **Xprof** allows you to view the performance data in **gmon.out** files, generated by compiling your application using the **–pg** compiler flag. You can view the profiling data, identify hotspots in the application, view relationships between functions in the application, zoom in to areas of the application of greater interest, and sort and filter the data to help identify hotspots in the application.

The IBM HPC Toolkit also provides a utility, **hpctInst**, which you can use to instrument the application without using the **peekperf** GUI. You can specify the types of instrumentation you want to use as well as the locations within the application which are to be instrumented. The **hpctInst** utility rewrites the application binary with the instrumentation you selected. Then, you can run that instrumented executable to obtain the same types of performance measurements that you could by using **peekperf**.

Finally, the IBM HPC Toolkit provides commands to get an overview of hardware performance counters for an application, as well as libraries that allow you to control the

performance data obtained using hardware performance counters and by MPI profiling and tracing.

Installing the IBM High Performance Computing Toolkit

Installing the IBM HPC Toolkit on AIX Systems

The IBM High Performance Computing Toolkit can be installed on IBM Power Systems servers which are supported by IBM Parallel Environment (PE).

The IBM HPC Toolkit has the following software requirements:

Function	Software	AIX 5.3	AIX 6.1
All	Operating System	AIX 5.3.9.0 or later	AIX 6.1.2.0 or later
Compiling C applications	IBM VAC compiler	8.0.0.12 or later	8.0.0.12 or later
Compiling FORTRAN applications	IBM xlf Compiler	11.1.0.0 or later	11.1.0.0 or later
Access hardware performance counters	bos.pmapi	5.3.9.0 or later	6.1.2.0 or later
Running the **Xprof** GUI	X11.motif.lib	5.3.0.60 or later	6.1.1.0 or later

You must install and accept the PE license on each node where you intend to install the IBM HPC Toolkit installation images before you can install the IBM HPC Toolkit.

Prerequisite software must be installed, following the documentation for that software's installation, before you install the IBM HPC Toolkit.

There are two installation images for the AIX version, ppe.hpct and ppe.hpct.rte. The ppe.hpct.rte installation image must be installed on all nodes where applications instrumented with the toolkit will run, and also on any node where the ppe.hpct installation image is installed. The ppe.hpct installation image must be installed on any node where you intend to use the GUI tools, instrument an application, or compile an application where you have coded calls to the instrumentation libraries.

The man pages for the AIX version of the IBM HPC Toolkit are packaged in the ppe.man installation image. You must install that installation image if you want online man pages for the IBM HPC Toolkit.

You can install the IBM HPC Toolkit using either the **installp** command or by using SMIT. You can install using either the installation media, or by copying the installation images to a directory accessible to the node and installing from that directory.

AIX Installation using installp

If the PE product license has not been previously accepted, you must have a copy of the ppe.loc.license installp image in the same directory as the IBM HPC Toolkit installp images.

To install the ppe.hpct.rte installation image using installp where the installation images are stored in ~/images:

```
installp –a –I –X –Y –d ~/images/ppe.hpct.rte ppe.hpct.rte
```

To install the ppe.hpct installation image using installp where the installation images are stored in ~/images:

```
installp –a –I –X –Y –d ~/images/ppe.hpct ppe.hpct
```

To install both the ppe.hpct and ppe.hpct.rte images where those images are both located in ~/images:

```
installp –a –I –X –Y –d ~/images ppe.hpct.*
```

AIX Installation using SMIT

To initially install the installation image using SMIT, follow these instructions:

- Insert the distribution medium in the installation device, unless you are installing over a network.
- Enter **smit install_latest**
  This command invokes SMIT, and takes you directly to its window for installing software.
- Press **List**
  A window opens listing the available INPUT devices and directories for software.
- Select the installation device or directory from the list of available INPUT devices. The window listing the available INPUT devices and directories closes and the original SMIT window indicates your selection.
- Press **Do**
  The SMIT window displays the default installation parameters.
- Type the appropriate file name, as shown in the following table:

File names for different types of installations	
If you want to install:	**Type this in the *SOFTWARE to install* field**:
All the IBM HPC Toolkit software	ppe.hpct*
Just ppe.hpct	ppe.hpct
Just ppe.hpct.rte	ppe.hpct.rte

- After choosing the appropriate software, you might also want to change other options on the panel, as needed. For example, the panel also asks whether or not you want to expand the file systems.
- When you are prompted, answer yes to expand the file systems if you want to allow the filesystem to be expanded.
- Type **yes** in the ACCEPT new license agreements field. If the eLicense is not accepted, none of the IBM HPC Toolkit software components will be installed.
- Press **Do**
  The system installs the installation image.

Installing the IBM HPC Toolkit on Linux Systems

The IBM High Performance Computing Toolkit can be installed on IBM Power Systems servers which are supported by IBM Parallel Environment (PE).

The IBM HPC Toolkit has the following software requirements:

Function	Software	Red Hat Linux	SLES Linux
All	Operating system	Red Hat Enterprise Linux version 5 update 2 or later	SLES 10 SP2 or later
Compiling C applications	IBM VAC compiler	9.0.0.0 or later	8.0.0.12 or later
Compiling FORTRAN applications	IBM xlf compiler	11.1.0.0 or later	11.1.0.0 or later
Accessing hardware performance counters	perfctr	Matching version for your Linux kernel	Matching version for your Linux kernel
Using IBM HPC Toolkit GUIs	32-bit libstdc++	4.1.2-14 or later	4.1.2_20070115-0.11 or later
Using IBM HPC Toolkit GUIs	32-bit openmotif	2.3.0.0-3 or later	N/A
Using IBM HPC Toolkit GUIs	32-bit libXp	1.0.0-8.1 or later	N/A
Using IBM HPC Toolkit GUIs	32-bit libXt	1.0.2-3.1 or later	N/A
Using IBM HPC Toolkit GUIs	32-bit libXext	1.0.1-2.1 or later	N/A
Using IBM HPC Toolkit GUIs	32-bit xorg-x11-libs	N/A	6.9.0-50.45 or later
Using IBM HPC Toolkit GUIs	32-bit openmotif-libs	N/A	2.2.4-21.12 or later

You must install and accept the PE license on each node where you intend to install the IBM HPC Toolkit installation images before you can install the IBM HPC Toolkit.

Prerequisite software must be installed, following the documentation for that software's installation, before you install the IBM HPC Toolkit.

The Linux version of the IBM HPC Toolkit has several RPMs which must be installed as described in the following table. The RPM containing the GUIs and instrumentation tools requires the runtime support RPM to be installed. RPMs providing 64-bit runtime and hardware performance counter access must be installed only if you will be analyzing 64-bit applications. If you install the 64-bit RPMs, you must also install the corresponding 32-bit RPM.

Purpose	Where installed	Red Hat RPM	SLES RPM
IBM HPC Toolkit GUIs and instrumentation tools	Compile/link and login nodes only	ppe_hpct_rh500	ppe_hpct_sles1000
IBM HPC Toolkit runtime support	All nodes	ppe_hpct_runtime_rh500	ppe_hpct_runtime_sles1000
IBM HPC Toolkit support runtime for 64-bit applications	All nodes	ppe_hpct_runtime64_rh500	ppe_hpct_runtime64_sles1000
Hardware performance counter access	All nodes	ppe_hpct_hpm_rh500	ppe_hpct_hpm_sles1000
Hardware performance counter access for 64-bit applications	All nodes	ppe_hpct_hpm64_rh500	ppe_hpct_hpm64_sles1000

You can install the RPMs separately or you can install them as a group. If you install RPMs individually, you must install the 32-bit runtime support RPM first, followed by the RPM containing the GUIs and instrumentation tools (on compile, link and login nodes) and the 32-bit RPM providing support for hardware performance counter access. After the 32-bit RPMs have been installed, you can install the 64-bit RPMs.

To install RPMs individually, use the **rpm –i <***RPM_name***>** command, for example

```
rpm –i ppe_hpct_rh500-5.1.0.0.ppc.rpm
```

To install RPMs as a group, copy the RPMs in to a directory, for instance, ~/images then issue the command

```
rpm –i ~/images/*.rpm
```

Enabling Hardware Performance Counter Support

Note: The hardware performance counter tools for Linux rely on a Linux kernel patch and runtime support library (perfctr) that are not included as part of the Red Hat and SLES Linux distributions. If you choose to use the hardware performance counter tools for Linux, you must download the perfctr kernel patch and library, build a customized kernel and boot your system using that kernel. Since the perfctr support is not part of a standard Red Hat or SLES distribution, the IBM HPC Toolkit hardware performance counter tools and the perfctr patch are not supported by IBM

If you choose to install the RPMs containing IBM HPC Toolkit support for hardware performance counters, you must download and install the perfctr kernel patch and runtime support library. You should download the latest copy of the perfctr source code from the perfctr distribution website, http://user.it.uu.se/~mikpe/linux/perfctr/2.6/, unpack the source code tar file and follow the instructions in the **INSTALL** file located in the top level directory of the unpacked tar file.

Using the IBM High Performance Computing Toolkit

Overview of the IBM High Performance Computing Toolkit

The IBM High Performance Computing Toolkit is a set of tools that you can use to obtain performance measurements for your application and to help you understand where there might be performance problems in your application. Tools are provided to obtain performance measurements in the following areas:

- Hardware performance counters, including measurements for cache misses at all levels of cache, number of floating point instructions executed, number of load instructions resulting in TLB misses, and other measurements that that are supported by your hardware. These measurements help the algorithm designer or developer identify and eliminate performance bottlenecks. The hardware performance counter tools allow you to run individual tasks of an MPI application with different groups of hardware counters monitored in each MPI task, so that you can obtain measurements for more than one hardware counter group within a single execution. These tools also allow you to summarize or aggregate hardware performance counter measurements from a set of MPI tasks, using plug-ins provided with the IBM HPC Toolkit or provided by you. On AIX, you can multiplex or time slice multiple hardware counter groups in a single task, allowing you to get hardware performance counter events from multiple groups in the same application process.
- MPI profiling, where you can generate a trace of MPI calls in your application so you can observe communication patterns and match MPI calls to your source code. MPI profiling also obtains performance metrics including time spent in each MPI function and MPI message sizes.
- OpenMP profiling, where you can obtain information about time spent in OpenMP constructs in your program, information about overhead in OpenMP constructs, and information about how workload is balanced across OpenMP threads in your application.
- Application I/O profiling, where you can obtain information about I/O calls made in your application to help you understand application I/O performance and identify possible I/O performance problems in your application.
- Application profiling, where you can identify functions in your application where the most time is being spent, or where the amount of time spent is different from your expectations. This information is presented in a graphical display that helps you better understand the relationships between functions in your application.
- The IBM HPC Toolkit includes the **peekperf** GUI which allow you to instrument the application, run it, and view the performance measurement data for hardware performance counters, MPI profiling, OpenMP profiling, and application I/O profiling, all from within **peekperf**. You can also sort and filter performance data within **peekperf** to help you better understand you application's performance.

You can obtain performance data for more than one type of measurement in a single execution of an application. You should use care in obtaining multiple performance measurements, since in some cases, particularly hardware performance counter

measurements and application profiling, the overhead of obtaining other measurements might appear in these measurements.

The IBM HPC Toolkit has two instrumentation models. The first model is one in which the application executable is rewritten with the instrumentation specified by you. You specify the instrumentation using either the **peekperf** GUI or the **hpctInst** command line. The IBM HPC Toolkit rewrites the application executable, adding the requested calls to the instrumentation libraries in to the executable. This process does not require any modifications to the application source code, and does not require the application to be relinked.

The second instrumentation model can be used when obtaining measurements using the hardware performance counters (HPM) or to control the generation of the MPI trace and MPI profiling information when using MPI profiling. In this model, you insert calls to functions in the instrumentation library in to your application source, then recompile and relink your application.

When you instrument your application, you should choose only one of the instrumentation models. This is because any calls to instrumentation functions that you code in your application might interfere with the instrumentation calls that are inserted by use of the **peekperf** or **hpctInst** tools.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you can use the toolkit. In order to set these environment variables, you should run the setup scripts that are located in the top level directory of your IBM HPC Toolkit installation. On AIX systems, these setup scripts are located in the /usr/lpp/ppe.hpct directory. On Linux, these setup scripts are located in the /opt/ibmhpc/ppe.hpct directory. If you are using sh, bash, ksh, or similar shell command, invoke the env_sh script as **. env_sh**. If you are using csh, invoke the env_csh script as **source env_csh**.

The IBM HPC Toolkit requires your application to be compiled and linked using the **–g** compiler and linker flag. If your application has not been compiled and linked using the **–g** flag, **peekperf** and **hpctInst** will be unable to instrument your application. If you plan to use **Xprof** to view profiling data for your application, your application must also be compiled and linked using the **–pg** compiler and linker flag.

After your application has been instrumented, you must ensure that it, and any data files that it requires, are properly distributed on the system on which the application will run. If your application resides in a globally accessible filesystem, such as a GPFS™ filesystem, you should not have to do anything to distribute the application. If your system is set up so that only local filesystems are used, you must manually copy the application and any data files it requires to each node on which the application will run, using system utilities that are most appropriate for your system.

You also need to make sure that the application environment has been properly set up. This includes making sure that any environment variables that are required to control the instrumentation in your application are properly set.

After you have set up the application environment, you can invoke the application as you would normally. The performance measurements you requested will be obtained while the application runs, and the results will be written when the application completes.

The performance measurement data is written to the current working directory for the application. Before you can view the performance data, you must ensure that all the performance data files are accessible to the node on which you will run the visualization tools. If the current working directory for the application resides in a global filesystem, you should not need to do anything to make the performance data files accessible to the visualization tools. If the current working directory resides in local filesystems on each node on which the application ran, you need to collect all the performance data files that you want to view in to a single directory that is accessible to the visualization tools. You can use any system utilities appropriate for your system to move the performance data files to a directory that is accessible to the visualization tools.

The following sections of this chapter first describe how to use the GUI tools to instrument, run, and analyze the performance of an application. After that, an explanation of how to instrument your application by inserting calls to the instrumentation libraries in your application is provided, followed by information about how to use the command line instrumentation tools.

Note: The hardware performance counter tools for Linux rely on a Linux kernel patch and runtime support library (perfctr) that are not included as part of the Red Hat and SLES Linux distributions. If you choose to use the hardware performance counter tools for Linux, you must download the perfctr kernel patch and library, build a customized kernel and boot your system using that kernel. Since the perfctr support is not part of a standard Red Hat or SLES distribution, the IBM HPC Toolkit hardware performance counter tools and the perfctr patch are not supported by IBM.

Using the Peekperf GUI

The **peekperf** GUI is the user interface for the IBM HPC Toolkit. You use the GUI for instrumenting your application, running your instrumented application, and obtaining performance measurements in the following areas:

- Hardware performance counter measurements
- MPI profiling
- OpenMP profiling
- Application I/O profiling

The **peekperf** GUI maps performance measurements to your source code, and allows you to sort and filter that data.

Peekperf is started by issuing the command **peekperf**. When you issue the **peekperf** command, the **peekperf** GUI main window is displayed. When the main window opens,

you can load an application executable, open visualization data files containing your performance measurements, and open application source files.

You can also invoke **peekperf**, specifying the name of your application, the name of the visualization data files, or both. If you invoke **peekperf** in this manner, **peekperf** automatically opens the files you specified when the main window opens.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you invoke **peekperf**. In order to set these environment variables, run the setup scripts that are located in the top level directory of your installation. On AIX systems, these setup scripts are located in the **/usr/lpp/ppe.hpct** directory. On Linux, these setup scripts are located in the **/opt/ibmhpc/ppe.hpct directory**. If you are using **sh**, **bash**, **ksh**, or similar shell command, invoke the **env_sh** script as **. env_sh**. If you are using **csh**, invoke the **env_csh** script as **source env_csh**.

The following sections explain the individual windows and menus in the **peekperf** GUI. Each type of instrumentation has some unique menus and options in these windows. Those unique menus and options are explained in later sections, which cover use of **peekperf** for each instrumentation type.

Preparing Your Application

You must compile and link your application using the **–g** compiler option so that **peekperf** will have the line number and symbol table information that it needs in order to show you the instrumentation points in your application, and so that the application can be instrumented.

When you compile a 64-bit application on a Linux system, you might need to use the **–emit-stub-syms** compiler option. If you attempt to instrument an application that requires this option to be used, a message is displayed on the console from which you invoked **peekperf** suggesting you set this linker option.

When you attempt to load a Linux 64-bit application, **peekperf** might be unable to locate the main entry point for your application. If **peekperf** cannot locate the main entry point, it will issue a message on the console from which it was invoked, suggesting that you set the **PSIGMA_MAIN** environment variable. If you see this message, you should set PSIGMA_MAIN to the main entry point for your application, or the name of the main program for FORTRAN applications.

Loading an application

When you invoke **peekperf** using the command **peekperf**, the following window is displayed.

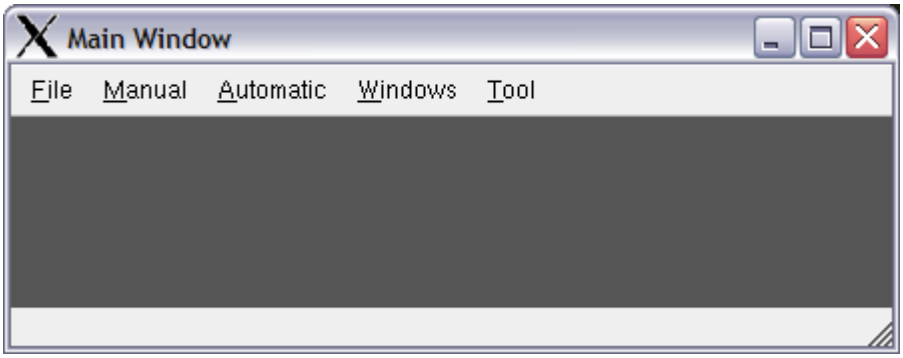


Figure 1: Peekperf main window at startup

Load the application, which must be compiled and linked with the **–g** option, by clicking the **File** menu, then clicking the **Open Binary** selection. A file selector dialog opens. Select the executable file to be opened, then click the **Open** button in that dialog. After you do this, a small dialog window opens in which you can specify additional arguments such as libraries to link with your application when loading it in to **peekperf**. If your application requires shared libraries other than libpmapi, libpthreads or libxlsmp, you must specify them here. If your application requires special link options, those must also be specified here.

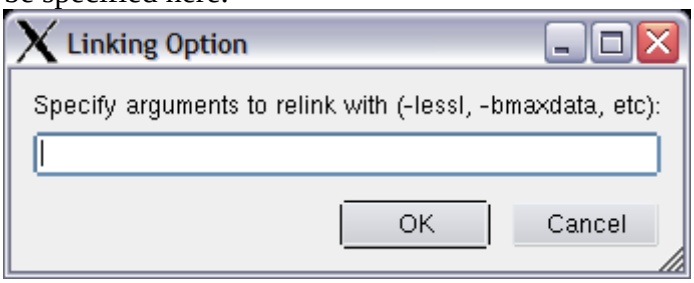


Figure 2: Additional link options (AIX only)

Specify any linker arguments that you need then click the **OK** button. The **peekperf** main window looks similar to the following:

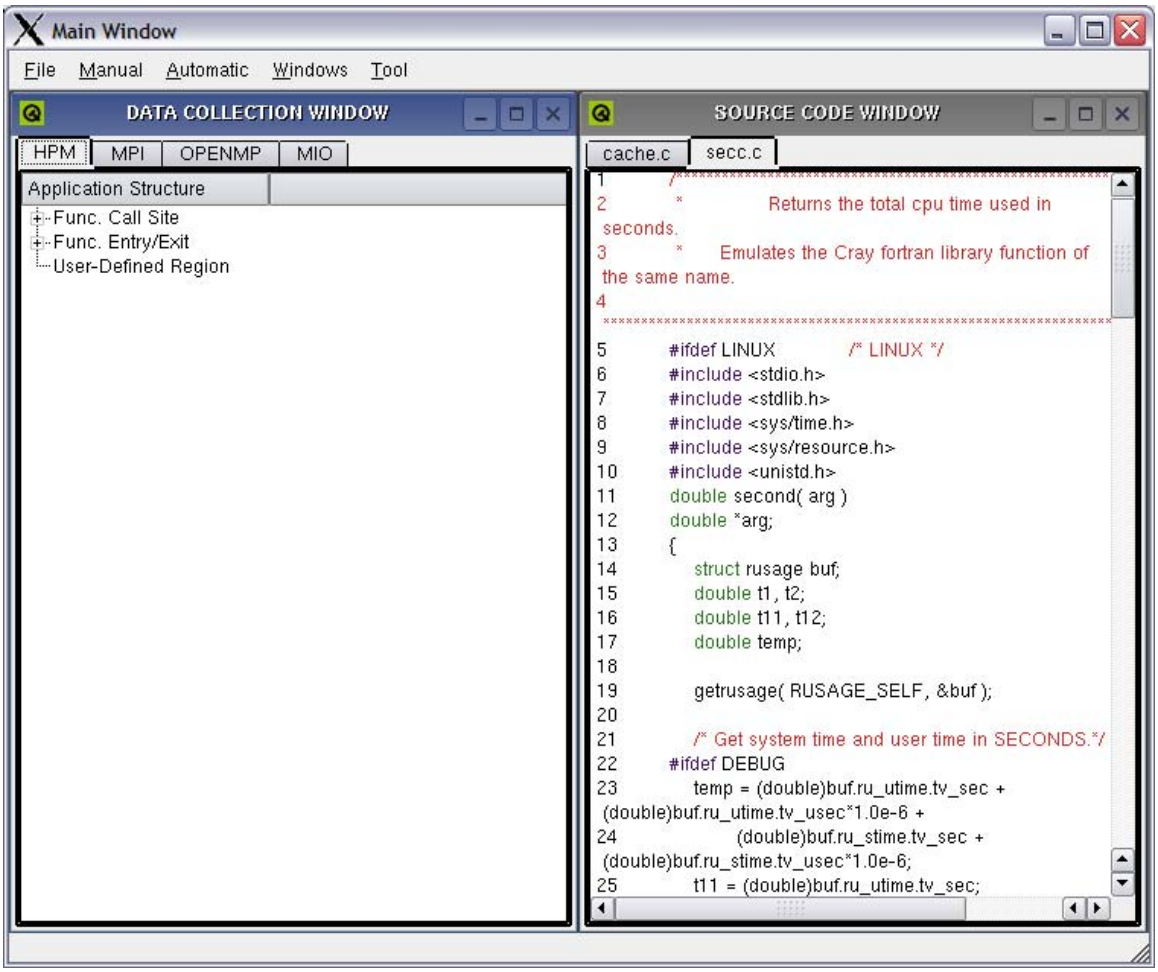


Figure 3: Peekperf Data Collection

The data collection window has four tabs for the different types of instrumentation that are available. If you click a tab, the data collection window shows you a tree view of the application structure that contains all of the places you can insert that type of instrumentation in the application.

You can instrument your application with as many of the four types of instrumentation as you like in a single session. Note that each type of instrumentation has a small overhead associated with it, and that overhead might affect measurements obtained by other types of instrumentation.

You can expand the tree in the data collection window by clicking the + icons at each node or by right-clicking in the white space in the data visualization window and then clicking **Expand the Tree** in the pop-up menu. This gives you a view showing all the instrumentation points in the application. You can collapse nodes in the tree by clicking the '**–**' icons at each node or by clicking **Collapse the Tree** in the same pop-up menu.

Note: The HPM tab in the data collection window is disabled on Linux systems unless the **ppe_hpct_hpm** RPMs are installed on your system.

If **peekperf** can find the source files, it opens a source code window showing the application source code. If there are multiple source files in the application, there are tabs at the top of the source code window for selecting the source file to view. As you select instrumentation in the source code window, the affected regions of source code are highlighted in the source code window.

If **peekperf** cannot find the source code for your application, it displays a pop-up dialog asking you if you want to specify the location of the source files. If you choose to do so, a file selector dialog that you use to select the location of the source files is also diaplayed

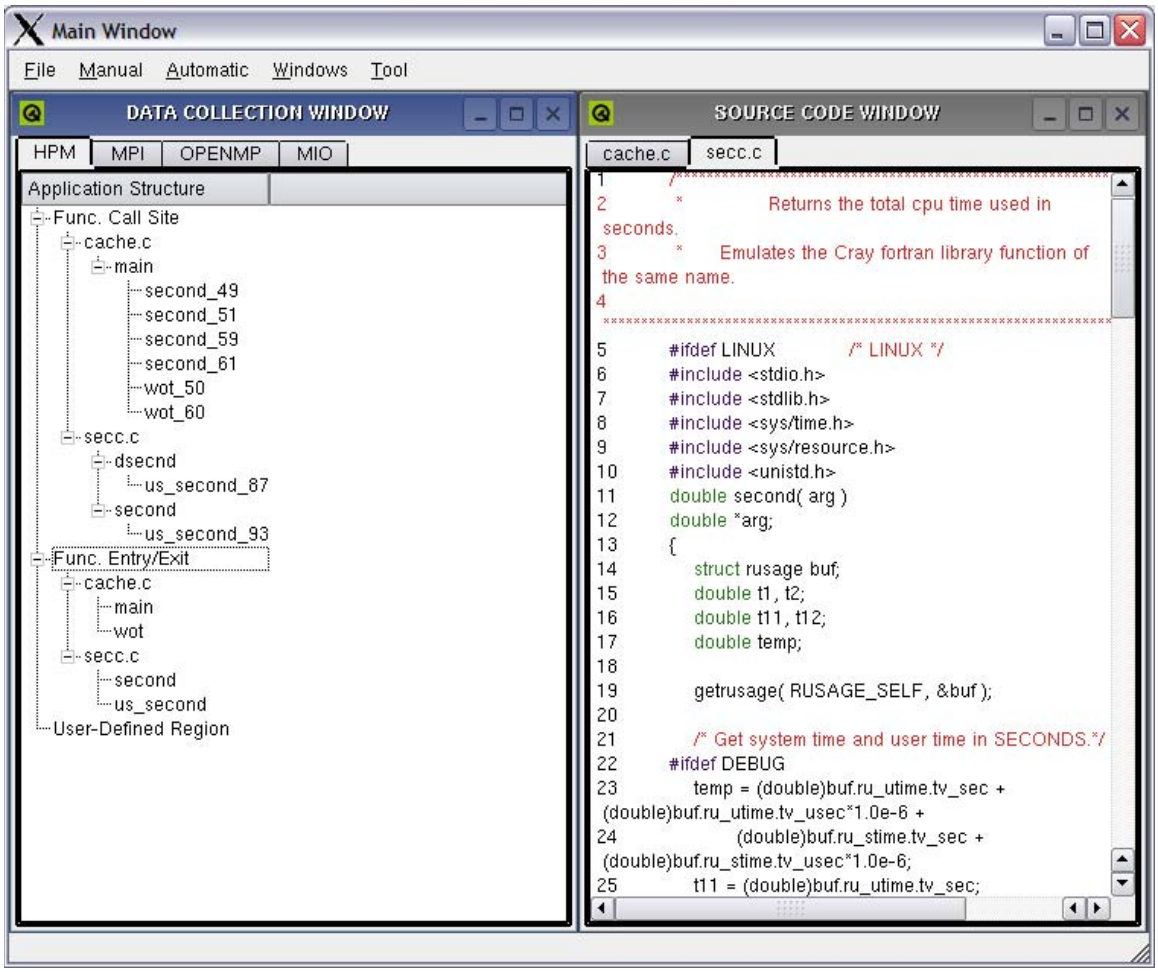
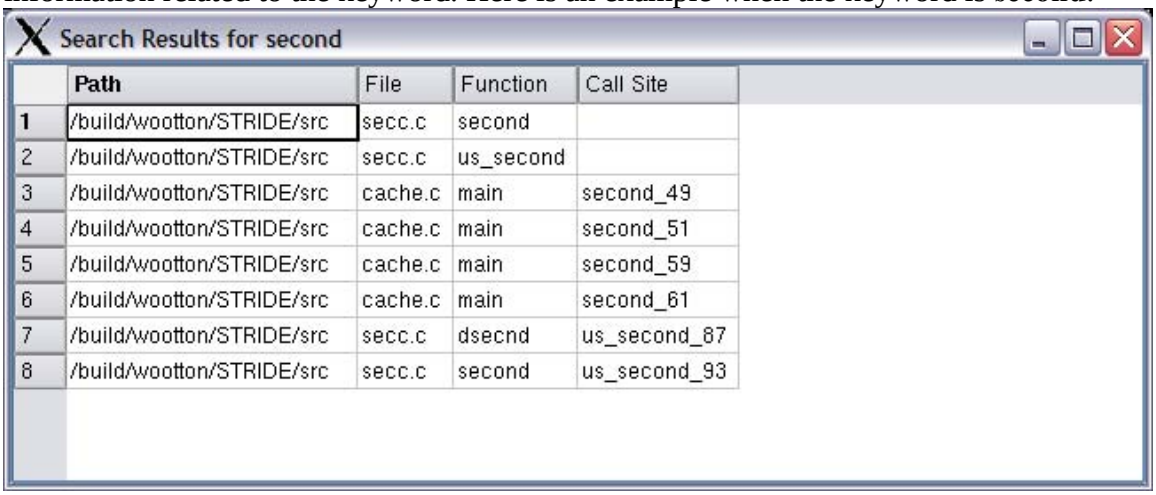


Figure 4: Peekperf data collection window with expanded application structure

The tree in the data collection window panel presents the program structure and will be created based on the type of performance data. For example, the tree in the HPM panel contains two subtrees. The **Func. Entry/Exit** subtree shows all the functions in the application. The call sites for each function call are shown in the **Func. Call Site** subtree.

Peekperf provides some searching capability in this tree. Select the **Search** option in the context menu (that appears when you right-click in white space in the data collection

window) and input the keyword. **Peekperf** searches the entire tree and gives you the information related to the keyword. Here is an example when the keyword is **second**.



	Path	File	Function	Call Site
1	/build/wootton/STRIDE/src	secc.c	second	
2	/build/wootton/STRIDE/src	secc.c	us_second	
3	/build/wootton/STRIDE/src	cache.c	main	second_49
4	/build/wootton/STRIDE/src	cache.c	main	second_51
5	/build/wootton/STRIDE/src	cache.c	main	second_59
6	/build/wootton/STRIDE/src	cache.c	main	second_61
7	/build/wootton/STRIDE/src	secc.c	dsecnd	us_second_87
8	/build/wootton/STRIDE/src	secc.c	second	us_second_93

Figure 5: Peekperf search results window

The points you want to instrument are selected by clicking the left mouse button on the nodes in the tree in the data collection window. For example, when you click the **cache.c** node, the **cache.c** node and all the children of the **cache.c** node are selected and highlighted. If you want to deselect it, click the left mouse button again on the selected node. All the children including this node will be deselected. If you want to clear all your current selected instrumentation, select the **Clear Instrumentation** option from the **Tool** menu. Note that at this point, the instrumented application is not generated yet. You are only selecting the places to put the instrumentation.

Instrumenting the Application

After you browse through each panel and identify the instrumentation you want, where you must have at least one instrumentation point selected, click the **Generate an Instrumented Application** option from the **Automatic** menu. When **peekperf** has created an instrumented application, it will pop up an alert to indicate that the application has been instrumented. It will also tell you the name of the instrumented application.

Running the Application

After you have instrumented your application, you can run your application from the **peekperf** GUI. The instrumented application will run and generate the data files containing measurements from the instrumentation that you selected.

If your application requires environment variables, such as those used by Parallel Environment, and those environment variables were not set before you invoked **peekperf**, you must set them before running the application. You set environment variables by selecting the **Set the Environment Variable** option from the **Automatic** menu. After you select this option, a pop-up dialog appears in which you fill in the name of the environment variable and the value.

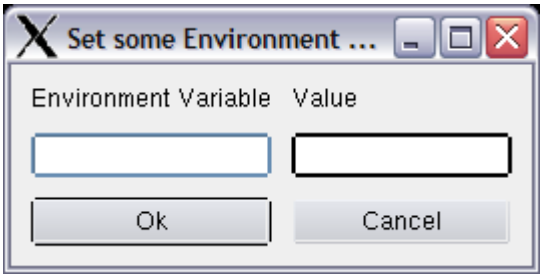


Figure 6:Peekperf environment settings window

After you enter a name and value, click the **OK** button to set the environment variable. The environment variable will remain set for the duration of your **peekperf** session, or until you change it again.

Run the application by selecting **Run an Instrumented Application** from the **Automatic** menu. **Peekperf** displays a set of dialogs requesting information needed to run the application. The first dialog asks if you want to modify a script or other file before running the application. If you click yes, you can edit any file before running your application.

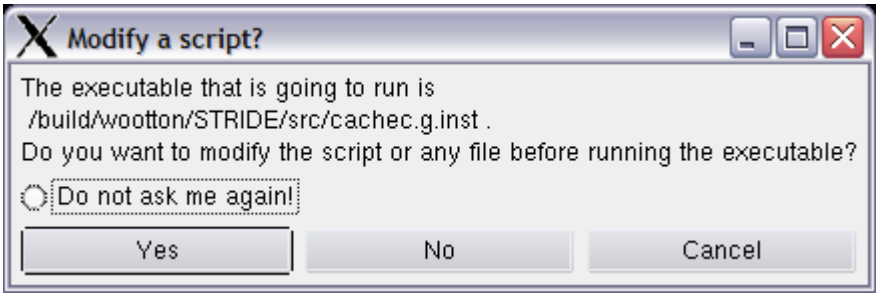


Figure 7: Peekperf run dialog 1 of 3

If you select **Yes**, a file selector dialog is displayed allowing you to select the file that you want to edit. After you select the file, an xterm window running the **vi** editor opens. You can edit the file and save your changes. If you select **No**, **peekperf** proceeds to the next dialog. If you select **Cancel**, the application is not executed.

The next dialog allows you to modify the path of the command you want to run.

Figure 8: Peekperf run dialog 2 of 3

This dialog is initially filled in with the path name of the instrumented application. You can change this path to the path of a script that sets up the environment for your application and invokes it. You can also specify command line arguments for your application. Click **OK** to proceed, or **Cancel** to cancel application execution.

The third dialog asks if you want **peekperf** to attempt to open the instrumentation data files (*.viz) files after your application runs.

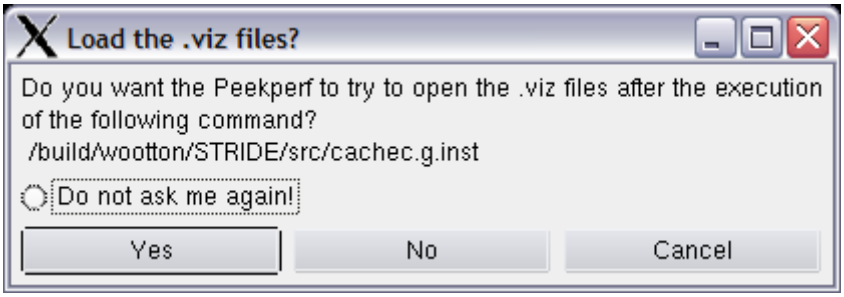


Figure 9: Peekperf run dialog 3 of 3

Select **Yes** if you want **peekperf** to attempt to open the .viz files, **No** if you do not want **peekperf** to try to open the .viz files, or **Cancel** to cancel application execution. After you select **Yes** your application runs to completion.

Note: The first and third dialogs above have a button labeled **Do not ask me again** and the second dialog has a button labeled **Remember the command**. If you select any of these buttons, **peekperf** remembers the settings in that dialog and does not display the dialog again in your **peekperf** session. If you exit and restart **peekperf**, these dialogs appear again the first time you run an application.

Viewing Performance Data

After your application completes, if you specified that **peekperf** should automatically open the performance data files, **peekperf** will attempt to open the .viz files generated by running your instrumented application. If you did not specify that **peekperf** should automatically open the .viz files, you can manually open them by selecting the **Open Performance Data** option from the **File** menu. **Peekperf** displays a file selector dialog in which you can select one or more .viz files to open.

After the .viz files are opened, **peekperf** will display them in a data visualization window.

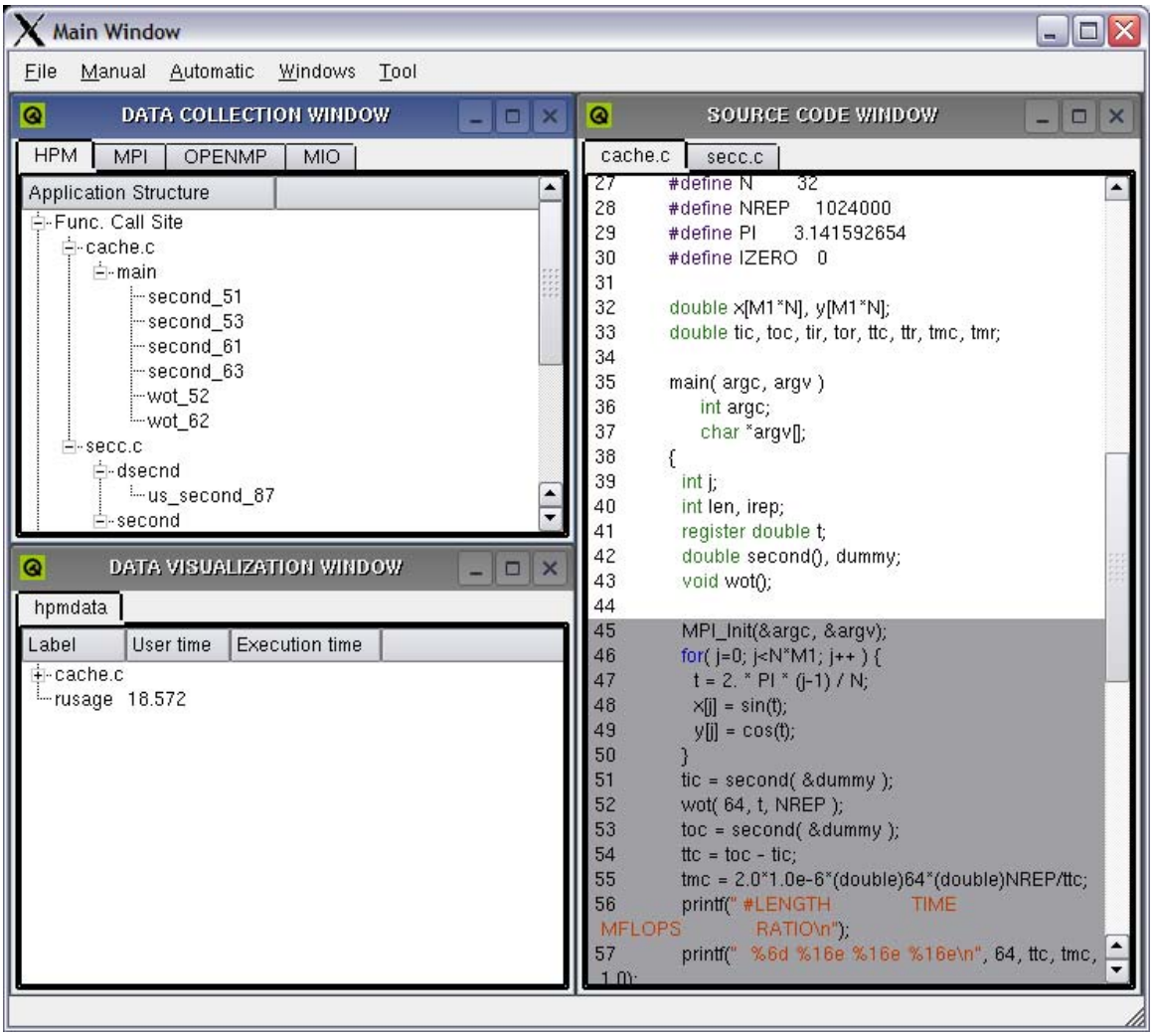


Figure 10: Peekperf data vizualization

The data visualization window has one or more tabs, where each tab contains the data from one set of .viz files. You can switch between sets of performance data by clicking the tabs for each set of data.

The data visualization window displays a tree view of the data collected. You can expand nodes in the tree by clicking the + icons next to the nodes or by selecting the **Expand the Tree** option from the pop-up menu that appears when you right-click in white space in this window. You can collapse nodes by clicking the '-' icon next to the nodes or by selecting **Collapse the Tree** from the same pop-up menu.

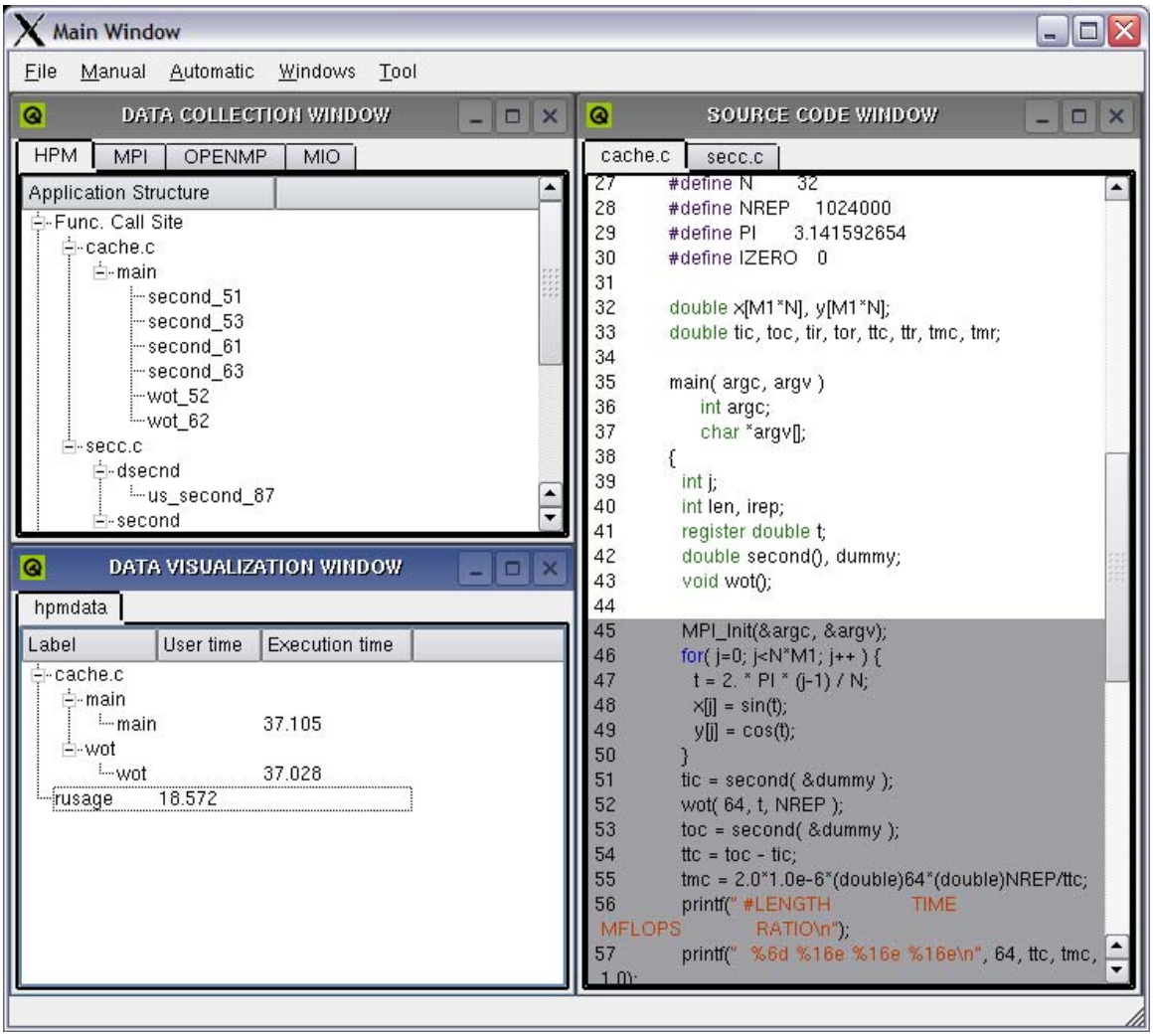


Figure 11: Peekperf expanded data vizualization window

Each leaf node in the tree shows some of the data that was obtained for each instrumented location in the application that was actually executed. You can sort the tree by any column. To do this, the column header for the desired column.

If you right-click in white space in the data visualization window, a pop-up menu with several options appears, including the options to expand and collapse the tree. If you select the **Set Filter** option from this menu, a pop-up dialog appears where you can set filtering criteria to reduce the data shown in this window.

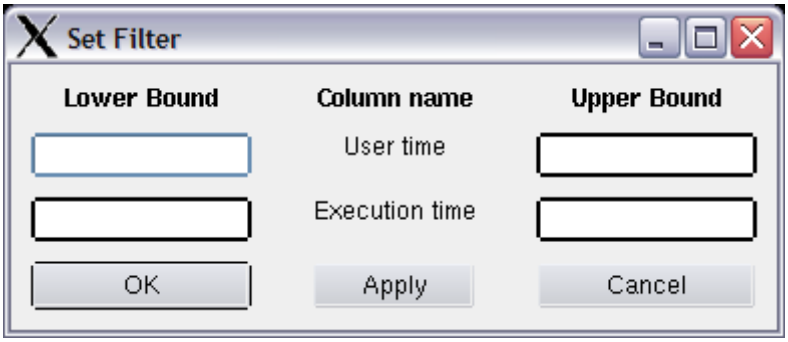


Figure 12: Peekperf data filtering dialog

This dialog has fields for upper and lower bounds for each data column in the tree. If you enter upper and lower bounds for a field then click **OK** or **Apply**. Only rows with a data value within those bounds will be shown. You can also enter only a lower bound or an upper bound. You can enter filter criteria for one or more columns. You can remove the filter by either opening the opening the filter dialog, clearing all the fields and clicking **OK** or by selecting **No Filter** from the pop-up menu in the data visualization window.

If you left-click on a node in the tree where data is shown, **peekperf** highlights the corresponding code region in the source code window. If you right-click on a node in the tree in which data is shown, a metric browser window opens, and displays a table containing additional data for all application tasks that executed code at that instrumentation point. You can have more than one of these tables open at the same time.

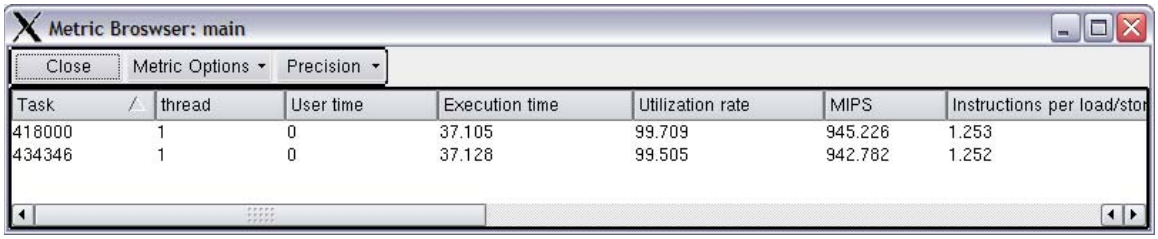


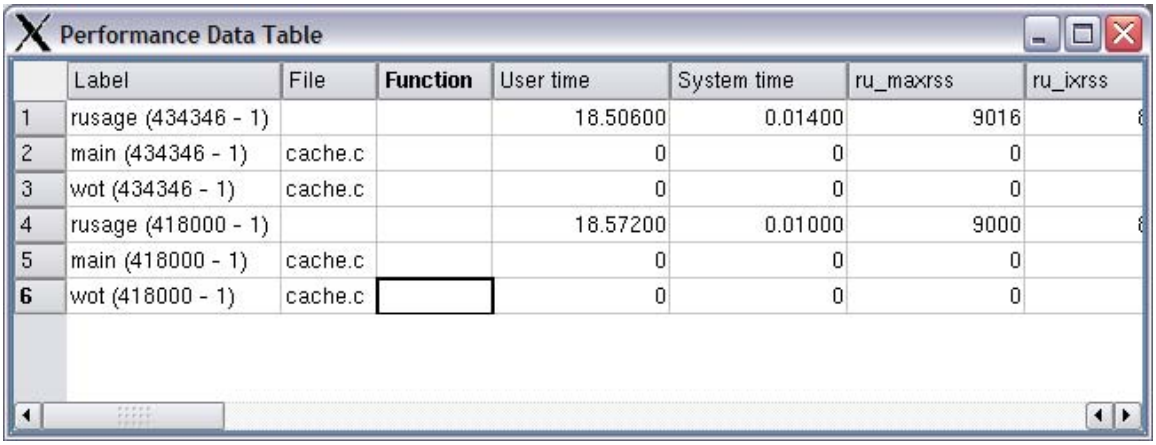
Figure 13: Peekperf metric browser window

You can sort data in this table by any column by clicking the column header for the desired column.

You can select the columns that you want to display in this table by clicking the **Metric Options** menu and then selecting a column from the menu. Selecting a menu option toggles the display state of the selected column, where checked menu items represent columns that are already visible.

You can change the floating point precision of the data displayed in this table by clicking the **Precision** menu and then selecting **Single** or **Double** from the menu.

You can display the data in the data visualization window in tabular form by clicking the **Show as Table** option in the pop-up menu in that window. Selecting that option results in a pop-up window similar to the following window.



	Label	File	Function	User time	System time	ru_maxrss	ru_ixrss
1	rusage (434346 - 1)			18.50600	0.01400	9016	
2	main (434346 - 1)	cache.c		0	0	0	
3	wot (434346 - 1)	cache.c		0	0	0	
4	rusage (418000 - 1)			18.57200	0.01000	9000	
5	main (418000 - 1)	cache.c		0	0	0	
6	wot (418000 - 1)	cache.c		0	0	0	

Figure 14: Peekperf performance data table window

You can sort this table on any column by clicking the column header for the desired column. You can hide columns in the table by right-clicking in the data area of the desired column in the table, then clicking the option in the pop-up menu to hide that column. You can show hidden columns by right-clicking anywhere in the data area of the table and then selecting the option to make the desired column visible again. You can save the table in a CSV format, readable by most spreadsheet programs, by right-clicking in the data area of the table, then selecting the **Save Table to File** option. A file selector dialog appears, in which you can specify the pathname of the file and then click the **Save** button to save the table data.

Viewing MPI Traces

You can view a trace of MPI activity in your application if you instrument your application with MPI instrumentation. After you instrument your application and run it, you can view the trace by selecting the **View Tracer** option from the pop-up menu that is available by right-clicking in the data visualization window. When you do this, the following windows open to display the MPI trace.

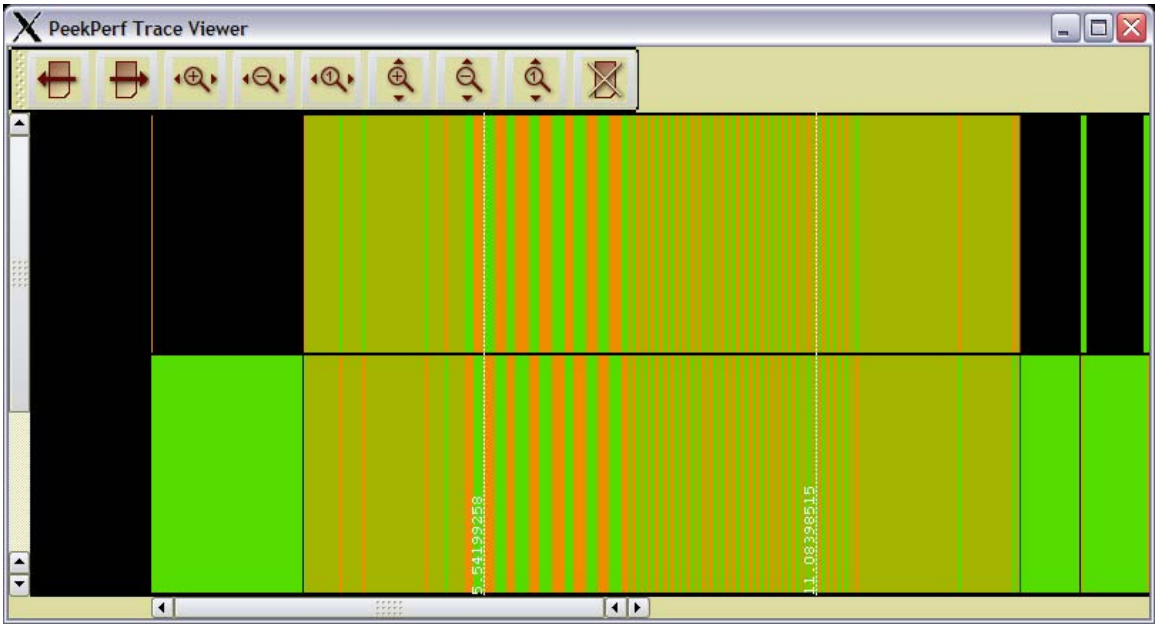


Figure 15: Peekperf trace viewer window

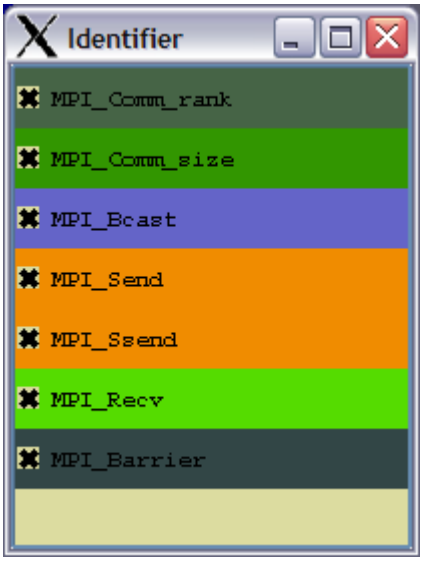


Figure 16: Peekperf trace viewer identifier window

You can obtain information about each MPI call, filter the trace display to show only interesting MPI calls, and zoom in and out of the trace display. The details of how to use the trace display are described in the **Using MPI Profiling in Peekperf** section later in this document.

Viewing I/O Profiling Data

You can view a plot of I/O activity in your application if you instrument it with MIO instrumentation. After you instrument your application and run it, you can view the plot by selecting the **View Tracer** option from the pop-up menu available by right-clicking in the data visualization window. When you do this, the following window is displayed.

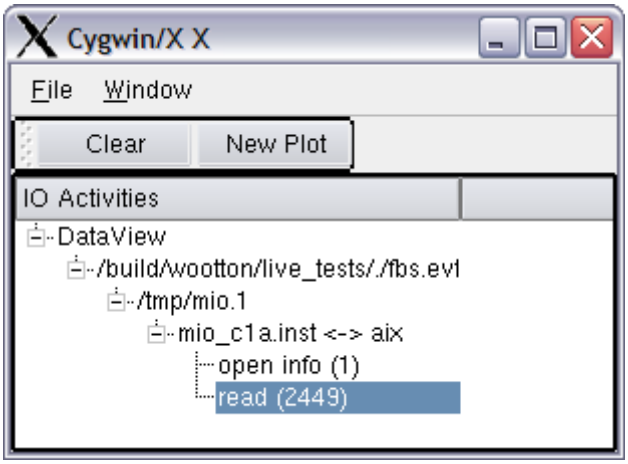


Figure 17: Peekperf I/O profiling selection window

Select the metrics you are interested in, then press the **New Plot** button to display the DataView Plot window.

\

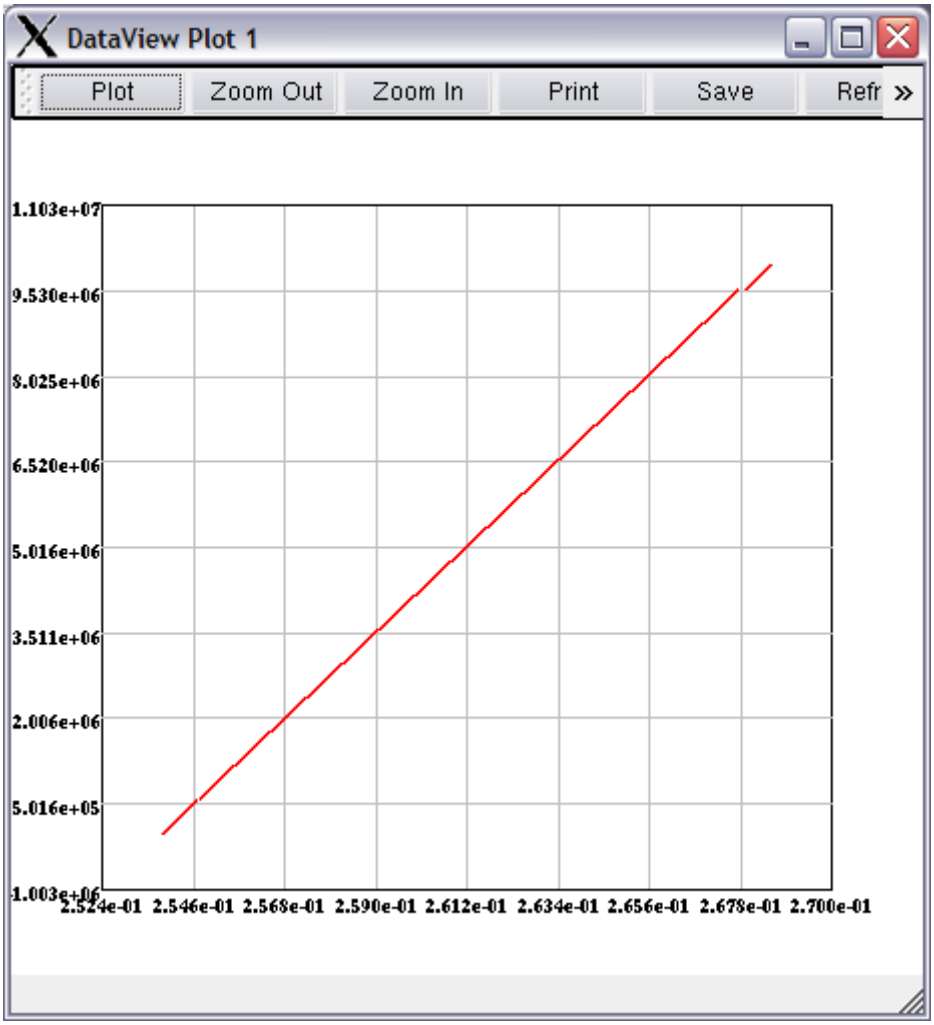


Figure 18: Peekperf data plot window

You can view the I/O activity plot, measure I/O performance at any point in the plot, and tailor the plot with additional information. The details of how to use this plot are described in the **Using I/O Profiling in Peekperf** section later in this document.

Compiling, Editing and Running Programs Using Peekperf

You can edit, compile and run applications within the **peekperf** GUI.

To edit a source file, right-click in the source code window and select the **Edit** option from the pop-up menu. **Peekperf** opens an xterm window, which allows you to edit the source code using the **vi** editor. After you save the changes to the source file, if you left-click in the source code window, **peekperf** detects the changed file and prompt you with a pop up dialog notifying you that the source file might have changed and asking if you want to reload the source file in to the source code window. If you click **yes**, the source code is reloaded. You can also edit any file from within **peekperf** by selecting the **Edit** option from the **Manual** menu. **Peekperf** displays a file selector dialog prompting you for the pathname of the file to edit. After you select the file and click the **Open** button in the file selector dialog, **peekperf** opens an xterm window where the **vi** editor is run. You can edit the source file and save the changed file, if needed.

You can compile an application by selecting the **Compile** command from the **Manual** menu. **Peekperf** displays a pop-up dialog prompting you for the compile command to run. This can be a compiler invocation, a make command, or any other command that you use to compile an application. When you click the **OK** button, **peekperf** runs the command you specified, where the current working directory is the directory where you started **peekperf**. When your compilation command finishes, **peekperf** displays a dialog containing all of the error messages issued by that command.

You can run an application in **peekperf** by selecting the **Run** option from the **Manual** menu. When you do this, **peekperf** displays a file selector dialog from which you can select the application to be executed. After you click the **Open** button, **peekperf** displays the same three pop-up dialogs, as previously described in the section **Running the Application,** and then runs the application.

Miscellaneous Peekperf Features

The following miscellaneous actions can be performed within **peekperf**:

- Open one or more source files by selecting the **Open Sources** option in the **File** menu. When you select this option, **peekperf** displays a file selector dialog in which you can select one or more source files. When you click the **OK** button in the file selector dialog, **peekperf** loads the selected source files in to the source code window, with a separate tab for each source file.
- Open one or more .viz files by selecting the **Open Performance Data** option in the **File** menu. When you select this option, **peekperf** displays a file selector dialog in which you can select one or more .viz files. When you click the **Open**

button in this dialog, **peekperf** loads the selected .viz files in to the data visualization window, creating a new tab in that window.

- Close the current application binary by selecting **Close Binary** from the **File** menu. **Peekperf** unloads the application binary and closes the data collection window. You can select another application to work with by selecting the **Open Binary** option in the file menu.
- Close the source file currently displayed in the source code window by either selecting the **Close the Active Source File** option in the **File** menu or by right-clicking in the source code window and selecting **Close this File** from the pop-up menu that appears.
- Close all source files by selecting **Close all Sources** in the **File** menu. **Peekperf** closes all source files it has loaded and then closes the source code window.
- Close the visualization data file, visible in the data visualization window, by selecting the **Close Active Data** option from the **File** menu.
- Close all visualization data files and the data visualization window by selecting the **Close All Data** option from the **File** menu.

Using Hardware Performance Counters in Peekperf

Preparing Your Application

Your application should not contain any calls to functions listed in the **Hardware Performance Monitoring** section of the **Commands and API Reference** chapter of this document because those calls might interfere with correct operation of the instrumentation you insert using **peekperf**. Your application should not be linked with the hardware performance counter library, (**libhpc.a** for AIX or **libhpc.so** for Linux).

You start **peekperf** and load the application binary as described in the preceding **Using the Peekperf GUI** section.

You must ensure that several environment variables that are required by the IBM HPC Toolkit are properly set before you invoke **peekperf**. In order to set these environment variables, you should run the setup scripts that are located in the top level directory of your IBM HPC Toolkit installation. On AIX systems, these setup scripts are located in the **/usr/lpp/ppe.hpct** directory. On Linux, these setup scripts are located in the **/opt/ibmhpc/ppe.hpct** directory. If you are using **sh**, **bash**, **ksh**, or similar shell command, then invoke the **env_sh** script as **. env_sh**. If you are using **csh**, then you invoke the **env_csh** script as **source env_csh**.

Instrumenting Your Application

After the data collection window opens, you should select the **HPM** tab in that window. This tab shows the possible instrumentation, similar to the following:

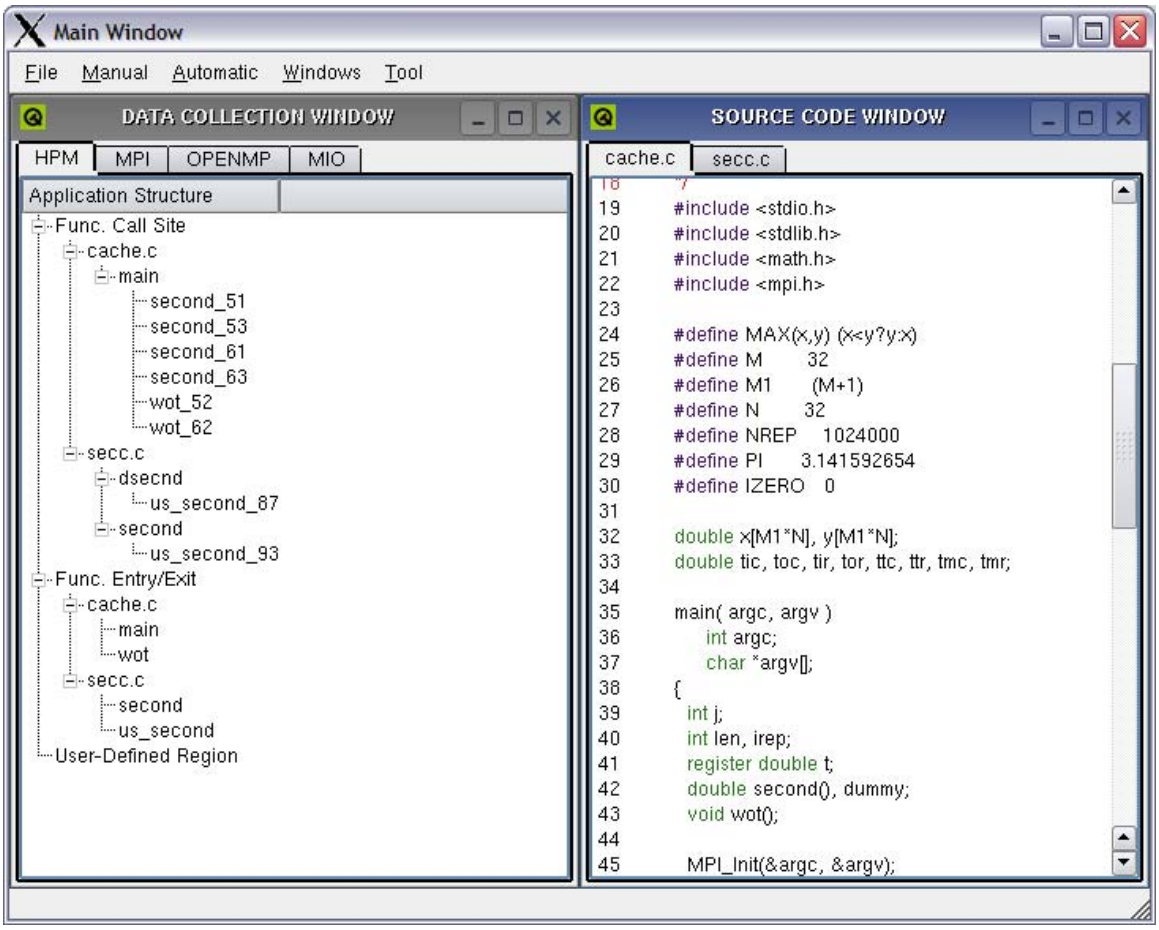


Figure 19: Peekperf HPM data collection window

This image shows the application structure tree fully expanded. There are three classes of instrumentation shown in this tree. The first class is instrumentation of function call sites, or locations in the application where a function call is made. The leaf nodes in this section of the tree are labeled with the name of the function being called, with the line number of the function call appended. The second class of instrumentation is function entry and exit points. The leaf nodes in this section of the tree are labeled with the name of the function. The third class of instrumentation is user-defined region, which is a user-specified region of code, such as a loop, that is instrumented. You can select any region of code to be instrumented, but be careful to ensure that the code at both the beginning and ending of each region is executed in each execution of that region. Leaf nodes in this section of the table are labeled with the source file name with the starting and ending line numbers for the region appended.

If you instrument a function call site, the instrumentation obtains hardware performance counter measurements for that instance of the function being called, and report them independently of any other function call to that same function. If you instrument function entry and exit points, the instrumentation obtains hardware performance counter measurements for that function for every time that function is called, regardless of the

caller. If you instrument a user-defined region, hardware performance measurements are obtained specifically for that region.

User defined regions are specified by highlighting a region of source code and then adding that region to the list of user-defined regions in the data collection window. To add a region to the user-defined region list, select that region by selecting the desired source file in the source code window, left-clicking at the starting line of the region, and dragging while the left mouse button is pressed until all desired source code lines are highlighted. After the desired lines are selected, right-click in the source code window and select the **Add to HPM** option from the pop-up menu**.** You may have as many user-defined regions as you want, and user-defined regions might overlap.

If necessary, you can remove a region of code from the user-defined region list by selecting the same lines of source code as you selected when creating the region, right-clicking, and selecting the **Remove from HPM** option from the pop-up menu.

You can select any combination of function call sites, function entry and exit points and user-defined regions that you want to instrument. You can select instrumentation by individually selecting leaf nodes, or you can select a group of leaf nodes by selecting a higher level node in the tree. You can also deselect instrumentation by clicking again on a highlighted node.

After you have selected the set of instrumentation that you want, you instrument the application by selecting **Generate an Instrumented Application** from the **Automatic** menu.

Running Your Program

Before you can run your instrumented application, you must ensure that any environment variables required by the hardware performance counter instrumentation are correctly set. The description of the **hpmInit**() function call contains a complete list of environment variables that can be used. You must ensure that the **HPM_VIZ_OUTPUT** environment variable is either unset or set to **yes**. You should not set the **HPM_OUTPUT_NAME** environment variable. If your application is an MPI application from which you are collecting performance data for multiple tasks, the **HPM_UNIQUE_FILE_NAME** environment variable should be set to **yes** so that each task's .viz file will be uniquely named. If you are instrumenting a large application, you might need to set the **HPM_NUM_INST_PTS** environment variable to a value larger than the default of 1000 instrumentation points.

You can set these environment variables before you start **peekperf**, or you can set them by selecting the **Set the Environment Variable** option from the **Automatic** menu. Any environment variables you set within **peekperf** remain set for the remainder of the **peekperf** session unless you reset them.

You should also select the hardware counter group before you run your program. You can do this by selecting the **Set the Counter Group** option from the pop-up menu that

appears when you right-click in the white space in the data collection window. A pop-up dialog appears after you do this.

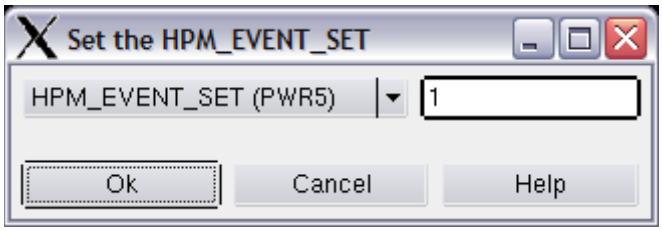


Figure 20: Peekperf hardware counter group selection dialog

Select the specific hardware architecture on which your application is running and then enter the hardware counter group number in the input field. If you click the **Help** button, **peekperf** will pop up another window that contains a list of hardware counter groups, the hardware counters in each group, and a short description of each counter. Click **OK** to select the hardware counter group you want.

After you have set up the environment for your application, you can run it by selecting the **Run an Instrumented Application** from the **Automatic** menu and responding to the dialogs as described previously.

Viewing Hardware Performance Counter Data

Peekperf attempts to open the .viz files after your application has completed execution if you requested **peekperf** to do so when you ran your application. If **peekperf** does not automatically open the .viz files, you can open them manually by selecting **Open Performance Data** from the **File** menu and selecting .viz files from the file selector dialog.

You can see a more detailed view of the data you obtained by right-clicking over a leaf node in the data visualization window, which opens a metrics browser window, or by right-clicking in white space in the data visualization window and selecting **Show as Table** from the pop-up, menu which opens a table view of the performance data.

If you load multiple .viz files in to the data visualization window, you can view each process's performance data separately. If you want to see performance data from a different process, right-click in the white space in the window and select **Show Other Rank** from the pop-up menu. A pop-up dialog appears containing a dropdown list that includes the process ids of other tasks that can be selected.

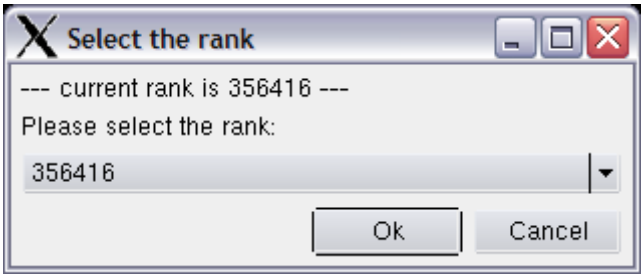


Figure 21: Peekperf task rank selection dialog

Using MPI Profiling in Peekperf

Preparing Your Application

Your application should not call any of the functions described in the **MPI Profiling** section of the **Commands and API Reference** chapter of this document because those calls might interfere with the instrumentation you insert using **peekperf**. Your application should not be linked with the MPI profiling library, libmpitrace.a for AIX or libmpitrace.so for Linux.

You cannot use the MPI profiling library to create a trace for an application that issues MPI function calls from multiple threads in the application.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you invoke **peekperf**. In order to set these environment variables, you should run the setup scripts that are located in the top level directory of your installation. On AIX systems, these setup scripts are located in the **/usr/lpp/ppe.hpct** directory. On Linux, these setup scripts are located in the **/opt/ibmhpc/ppe.hpct directory**. If you are using **sh**, **bash**, **ksh**, or similar shell command, invoke the **env_sh** script as **. env_sh**. If you are using **csh**, invoke the **env_csh** script as **source env_csh**.

You start **peekperf** and load the application binary as described in the preceding **Using the Peekperf GUI** section.

Instrumenting Your Application

After the data collection window opens, you should select the **MPI** tab to show the possible instrumentation points in the application. The data collection window that appears similar to the following:

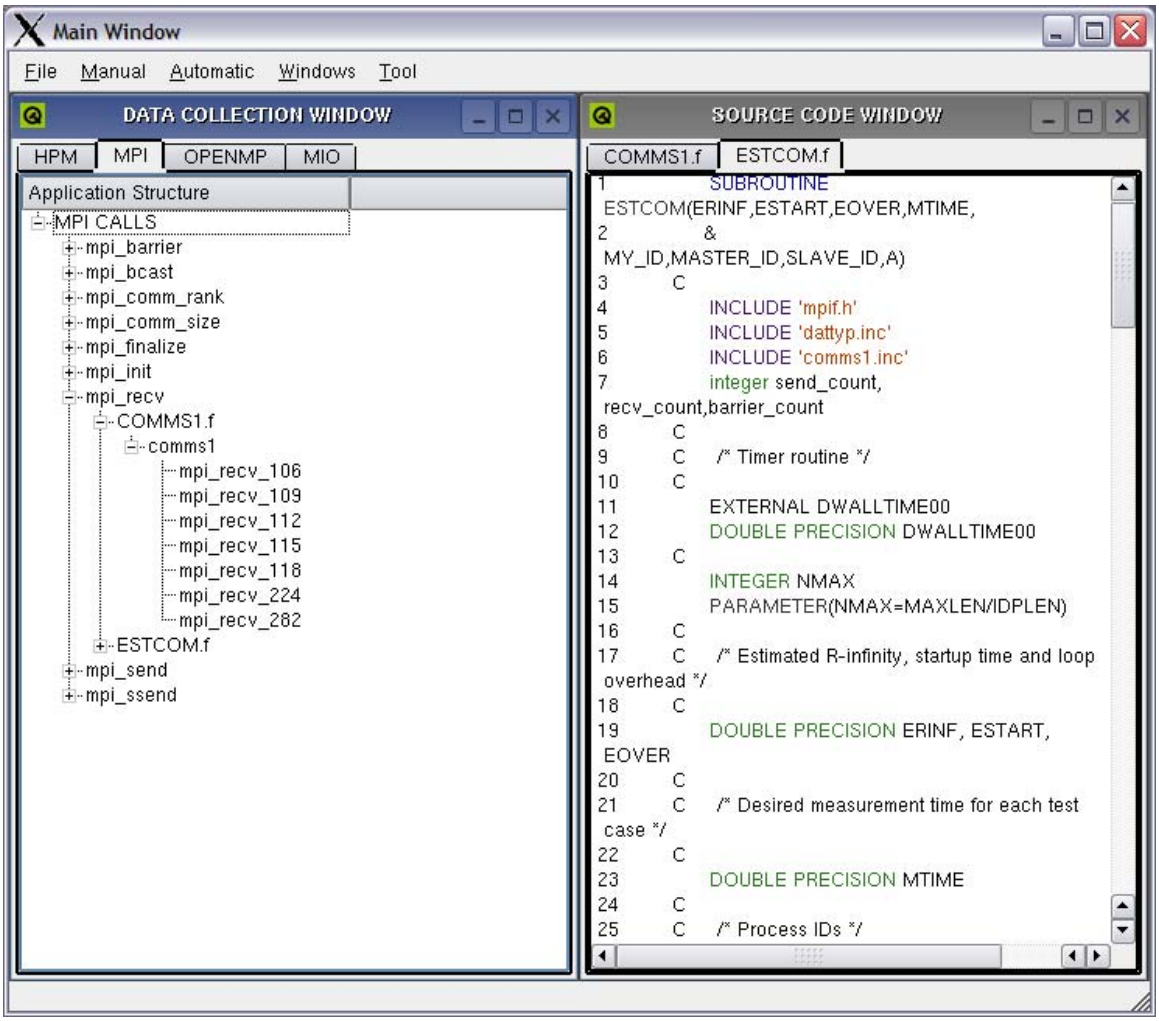


Figure 22: Peekperf MPI profiling data collection window

The data collection window is partially expanded, showing the **MPI_Recv()** calls that can be instrumented. For MPI profiling, **peekperf** labels each instrumentation location with the name of the MPI function at that line in the application and the line number where the MPI function call is located.

You can select as many instrumentation locations in your application as you like. If you select a leaf node, only that MPI function call is instrumented. If you select a node labeled with an application function name or file name, all MPI function calls of the type identified by the enclosing node are instrumented. If you select a node labeled with an MPI function name, all MPI calls for that MPI function in the application are instrumented. You can instrument all MPI function calls in the application by selecting the **MPI CALLS** node. You can deselect an MPI function call by clicking the corresponding highlighted node in the application structure tree.

You can also select MPI function calls to be instrumented by left-clicking over a starting line in the source code window and dragging to the end of the region of interest while

holding the left mouse button down. After you have selected a region of code, you can select all MPI function calls in that region to be instrumented or not instrumented by right-clicking in the source code window and selecting either **Add to MPI** or **Remove from MPI** respectively. When you do this, **peekperf** updates the tree in the application structure window accordingly.

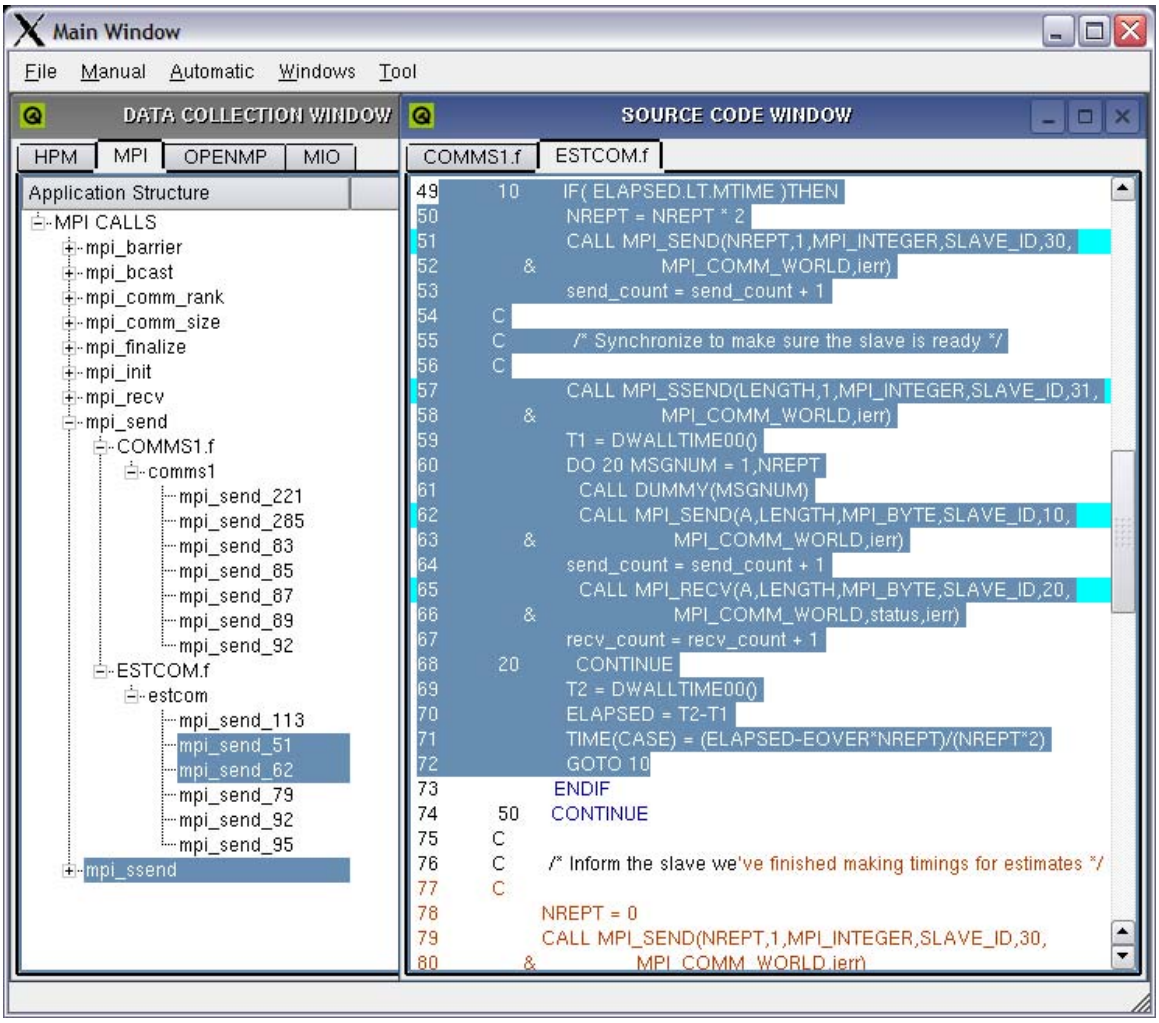


Figure 23: Peekperf MPI profiling data collection with selected instrumentation

After you have selected the set of instrumentation that you want, you instrument the application by selecting **Generate an Instrumented Application** from the **Automatic** menu.

Running Your Application

Before you can run your instrumented application, you must ensure that any environment variables required by the MPI profiling library or by Parallel Environment are correctly set. The environment variables used by the MPI profiling library are listed in the description of the **MT_trace_start**() function in the **MPI Profiling** section of the **Commands and API Reference** chapter of this document.

By default, the MPI profiling library will generate trace files only for the application tasks with the minimum, maximum and median MPI communication time. This is also true for task zero if task zero is not the task with minimum, maximum or median MPI communication time. If you need trace files generated for additional tasks, set the **OUTPUT_ALL_RANKS** environment variable to **yes**. Depending on the number of tasks in your application, you might need to set the **MAX_TRACE_RANK** and **TRACE_ALL_TASKS** environment variables to **yes**. If your application executes many MPI function calls, you might need to set the value of the **MAX_TRACE_EVENTS** to a higher number than the default 30,000 MPI function calls.

After you have set up the environment for your application, run it by selecting the **Run an Instrumented Application** from the **Automatic** menu and respond to the dialogs as described previously.

Viewing MPI Profiling Data

Peekperf attempts to open the .viz files after your application has completed execution, if you requested **peekperf** to do so when you ran your application. If **peekperf** does not automatically open the .viz files, you can open them manually by selecting **Open Performance Data** from the **File** menu and selecting .viz files from the file selector dialog. The following image shows the application structure tree in the data visualization window partially expanded.

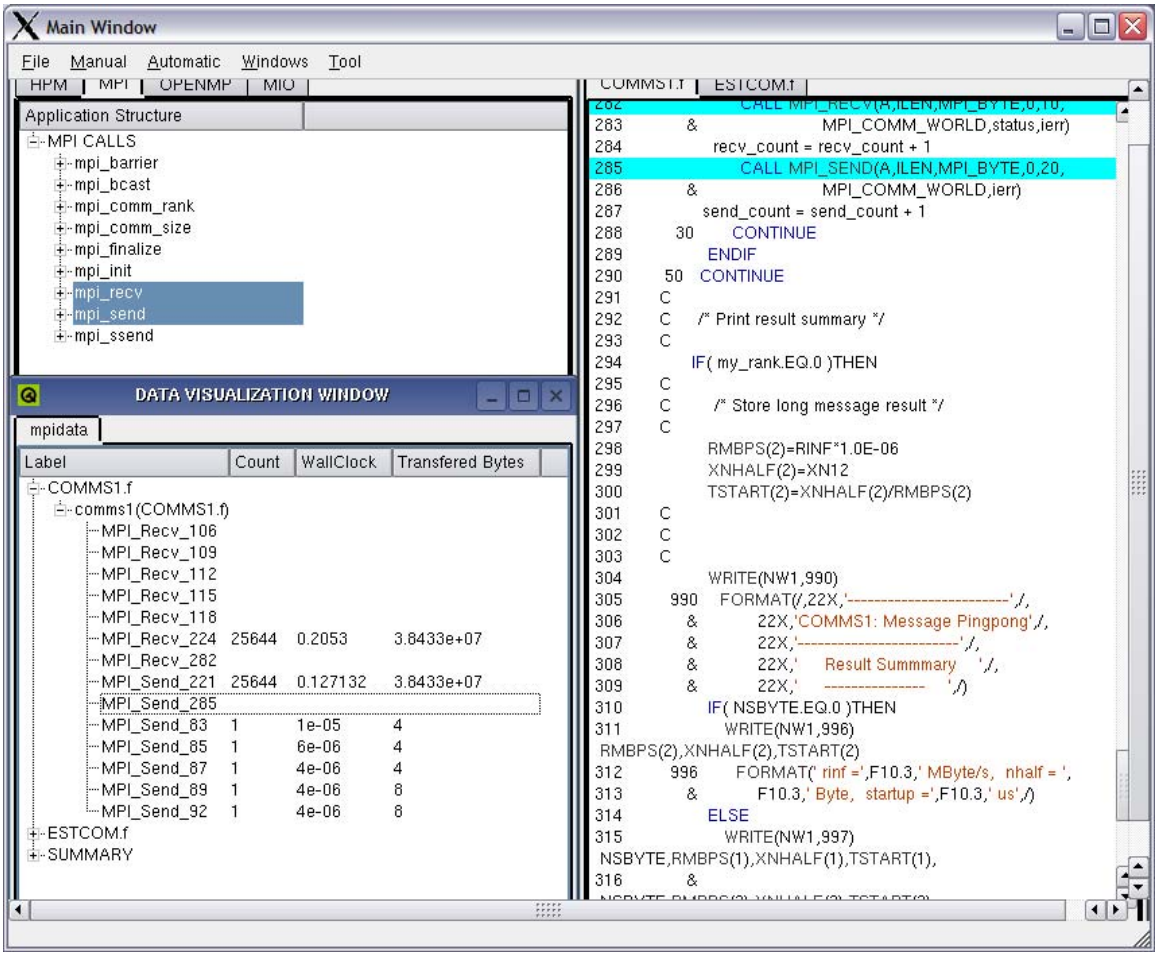


Figure 24: Peekperf MPI profiling data collection window

The data visualization window shows the number of times each function call was executed, the total time spent executing that function call, and the amount of data transferred by that function call. You can see a more detailed view of the data you obtained by right-clicking over a leaf node in the data visualization window, which opens a metrics browser window, or by right-clicking in white space in the data visualization window and selecting **Show as Table** from the pop-up menu, which opens a table view of the performance data.

The data visualization window shows performance data from a single task at one time. If you want to see performance data from a different task, right-click in the white space in the window and select **Show Other Rank** from the pop-up menu. A pop-up dialog appears containing a dropdown list that allows you to select the task indices of other tasks that can be selected.

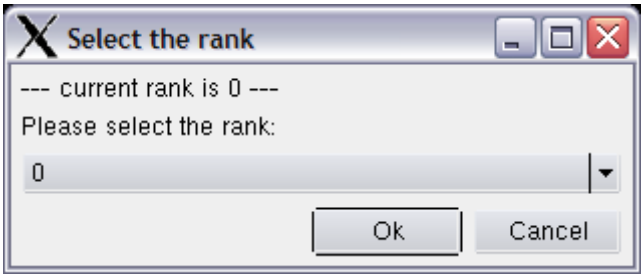


Figure 25: Peekperf MPI task rank selection dialog

Select the desired task and click **OK** to view a different task's performance data.

You can view a trace of MPI activity in your application by right-clicking in white space in the data visualization window and selecting **View Tracer** from the pop-up menu. When you do this, the following two windows appear.

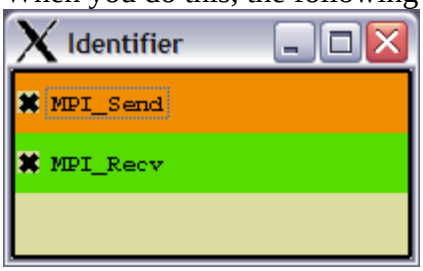


Figure 26 Peekperf MPI trace viewer identifier dialog

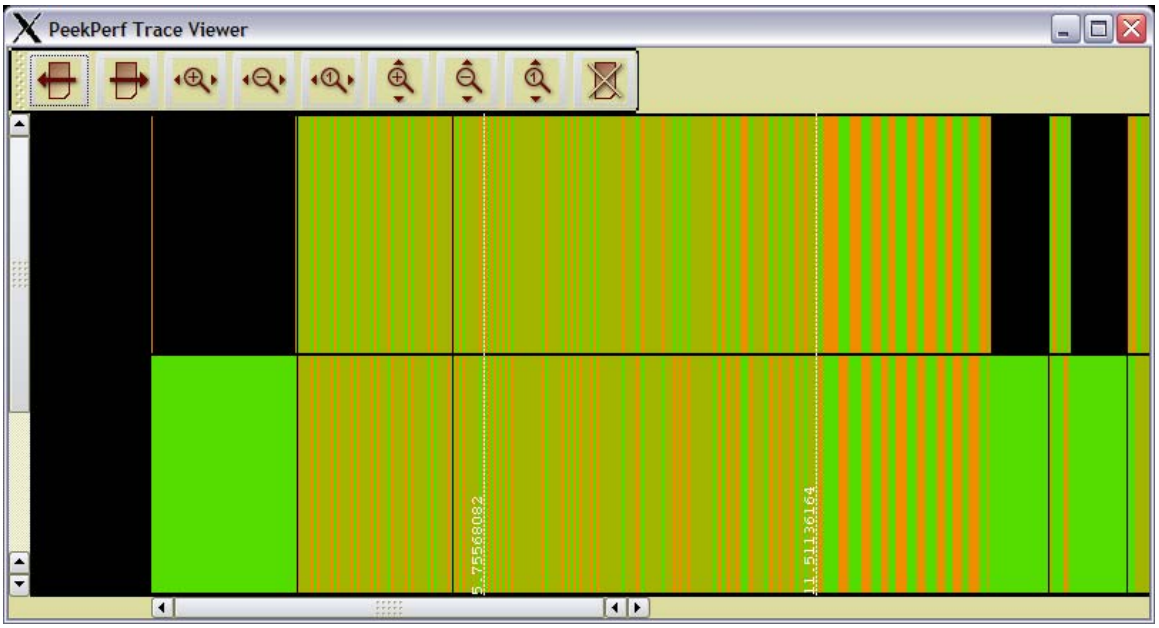


Figure 27: Peekperf trace viewer

You can use the first window, labeled Identifier, to select the types of MPI function calls that are displayed in the PeekPerf Trace Viewer window. Initially, all MPI function calls

that were executed are displayed. You can hide or display trace events for specific MPI functions by clicking on the MPI function labels in the identifier window. Trace events are displayed for MPI functions that have their labels checked while trace events are hidden for MPI functions with unchecked labels.

The second window is the MPI trace viewer window. This window displays a timeline-based view of your application's execution, where the Y axis is the application task rank and the X axis is elapsed time. Each MPI function call is represented by a block drawn in the color matching the MPI function label in the identifier window.

You can use the icons at the top of the PeekPerf Trace Viewer window to navigate through the trace. From left to right, the icons have the following purposes:

- Scroll the trace data left.
- Scroll the trace data right.
- Zoom the trace display in to display a more detailed level of the X axis.
- Zoom the trace display out to display a less detailed level of the X axis.
- Reset the X axis to its initial, non zoomed, view.
- Enlarge the trace display along the Y axis.
- Shrink the trace display along the Y axis.
- Reset the Y axis to its initial, nonzoomed view.
- Close the trace display windows.
-
- You can use the keyboard to navigate through the trace using the following keys
-
- Left arrow
  - Scroll trace data left
- Right arrow
  - Scroll trace data right
- Up arrow
  - Scroll trace data up
- Down arrow
  - Scroll trace data down
- PageUp
  - Scroll trace data left rapidly
- PageDown
  - Scroll trace data right rapidly
- 'z' or 'y'
  - Zoom trace display in to display a more detailed view of the X axis
- 'x'
  - Zoom trace display out to display a less detailed view of the X axis
- 'a'
  - Enlarge the trace display along the Y axis
- 's'
  - Shrink the trace display along the Y axis

You can also scroll the trace data horizontally and vertically using the scrollbars at the left and bottom of the trace window. You can zoom in to display a more detailed view of the trace by picking a starting point in the trace, left-clicking, and dragging the mouse to the end of the region of interest while holding the left mouse button down. When you release the mouse button, **peekperf** zooms in to display the selected region.

The following window shows a region of the trace that has been zoomed in to where there are alternating **MPI_Send()** and **MPI_Recv()** calls.

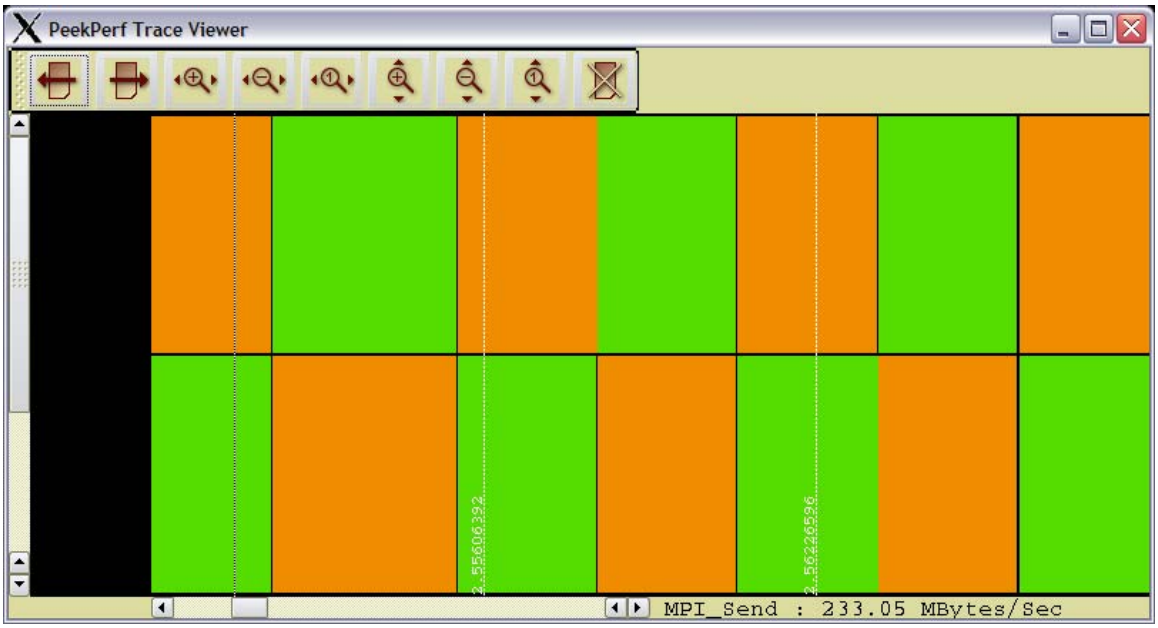


Figure 28: Peekperf trace viewer detailed view

If you left-click over a rectangle for a specific MPI call, **peekperf** displays the name of the MPI function and the data rate for a communications function call in the lower right hand corner of the trace window. **Peekperf** also positions the source code window to the specific MPI function call in the source code and highlights that MPI function call.

If you right-click over a rectangle for a specific MPI function call, **peekperf** will display a pop-up dialog box that displays additional detail for the MPI function call, as long as the right mouse button is pressed.

Using OpenMP Profiling in Peekperf

Preparing Your Application

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you invoke **peekperf**. In order to set these environment variables, you should run the setup scripts that are located in the top level directory of your IBM HPC Toolkit installation. On AIX systems, these setup scripts are located in the **/usr/lpp/ppe.hpct** directory. On Linux, these setup scripts are located in the **/opt/ibmhpc/ppe.hpct directory**. If you are using **sh**, **bash**, **ksh**, or similar shell command, invoke the **env_sh** script as **. env_sh**. If you are using **csh**, invoke the **env_csh** script as **source env_csh**.

You start **peekperf** and load the application binary as described in the preceding **Using the Peekperf GUI** section. There are no additional compiler or linker options that are required to use OpenMP profiling.

Instrumenting Your Application

After the data collection window opens, select the **OPENMP** tab in that window. This tab shows you the instrumentation locations, similar to the following example in which the application structure tree has been fully expanded.

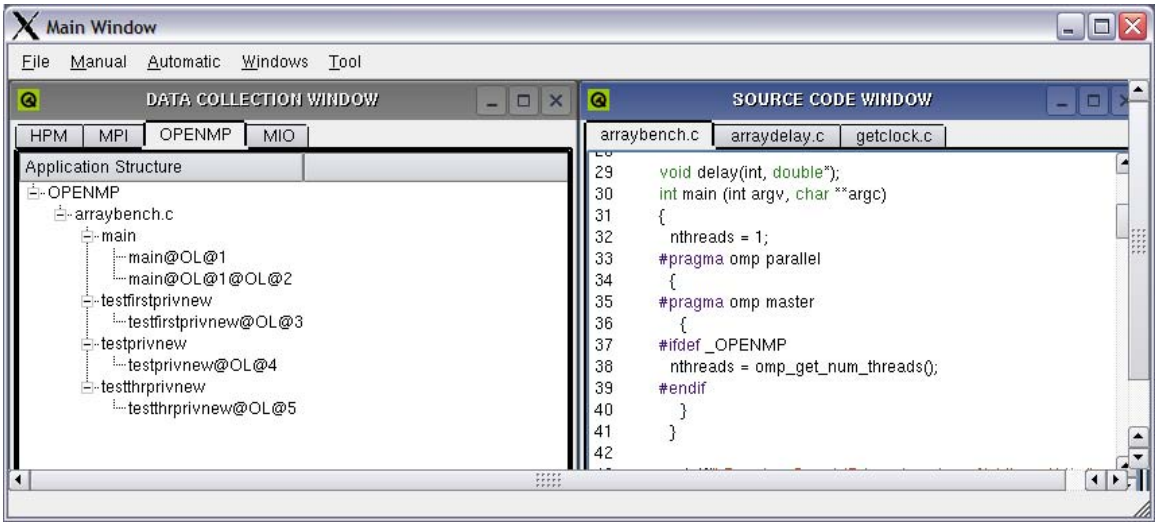


Figure 29: Peekperf OpenMP data collection window

This window shows you the locations in your application that can be instrumented for OpenMP profiling. Leaf nodes in this tree represent individual OpenMP regions in your application. Leaf nodes are labeled with the function name generated by the compiler for that OpenMP region. You can select instrumentation at any level in the tree. If you select a leaf node, only that location will be instrumented. If you select a nonleaf node, all leaf nodes that are children of the selected node will be instrumented.

Note: The IBM HPC Toolkit only supports OpenMP profiling instrumentation for OpenMP regions that are not nested within other OpenMP constructs at runtime. If you set up instrumentation so that nested parallel constructs are instrumented, results will be unpredictable.

You can control the level of instrumentation for each class of OpenMP constructs by setting the **POMP_LOOP**, **POMP_PARALLEL**, and **POMP_USER** environment variables before you instrument the application. Refer to the **hpctInst** command, in the **Commands and API Reference** section of this document, for the description of each environment variable's settings. You set these environment variables by selecting the **Set the Environment Variable** option from the **Automatic** menu, and filling in the name of each environment variable and its value in the pop-up dialog that appears.

After you have selected the set of instrumentation that you want, you instrument the application by selecting **Generate an Instrumented Application** from the **Automatic** menu.

Running Your Application

You can run your instrumented application by selecting the **Run an Instrumented Application** option from the **Automatic** menu and responding to the pop-up dialogs that were described in the **Using the Peekperf GUI** section earlier in this document. There are no special steps that must be followed to run an application with OpenMP instrumentation.

Viewing OpenMP Profiling Performance Data

After your application completes, the **peekperf** GUI displays your performance measurements in the data visualization window, if you requested **peekperf** to open the .viz files on application completion. The main GUI window looks similar to the following:

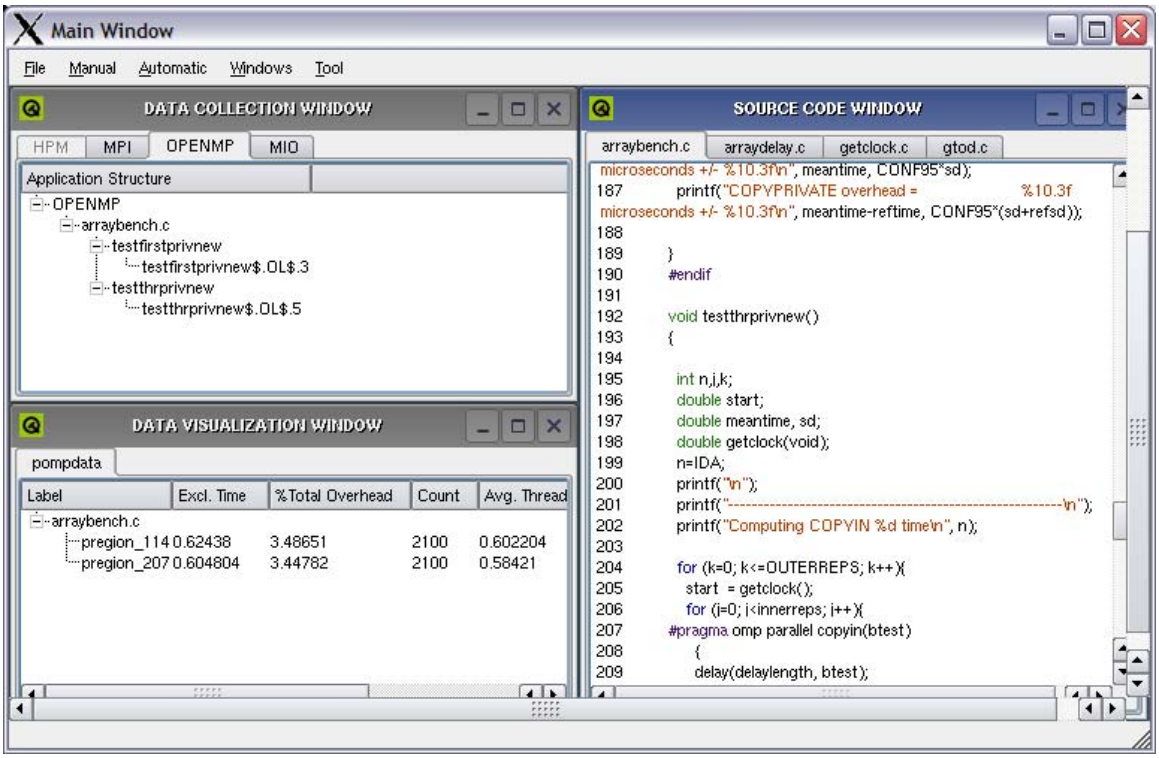


Figure 30: Peekperf OpenMP data visualization window

The data is displayed in a tree format in which source files are the major nodes in the tree and the individual instrumentation points are the leaf nodes. Individual leaf nodes are labeled with the type of OpenMP instrumentation and the starting line number of the instrumented OpenMP construct.

You can view detailed data for a leaf node by right-clicking over a leaf node. A metric browser window contains data for each process and thread that executed the OpenMP construct. You can view all of your performance measurements in a tabular form by selecting the **Show as Table** option from the pop-up menu that appears when you right-click on white space within the data visualization window.
The data displayed in the data visualization window is a summary view of the performance data, displaying the maximum value from all threads for the following statistics:

- Count: the number of times the event was executed
- Exclusive time (Excl. Time): The total time (Does not include time inside of other events)
- Inclusive time (Incl. Time): Wall clock time for the event (including other events).
- Percentage of total overhead (% Overhead): 100 * (thread time – computation time) / thread time
- Percentage of imbalance (% imbalance): 100 * (comp time – MIN(comp time(i))) / MIN(comp time(i))
- Average thread time (Avg. Comp Time): SUM(comp time(i)) / Number of threads

The metrics browser window shows the following statistics for each OpenMP thread:

- Task (Task): MPI Task ID
- Thread (thread): OpenMP thread ID
- Time in master: (Time in Master): Time in the master thread (Wall Clock Time)
- Thread time: (TT: Thread Time): Wall clock time for the execution of each thread
- Computation time: (CT: Comp. Time) Time per thread in the body of the OpenMP construct
- Percentage of imbalance (% imbalance): 100 * (comp time – MIN(comp time(i))) / MIN(comp time(i))
- Total overhead: (TO: TT – CT): thread time – comp time
- Percentage of the total overhead due to barrier (%TO (Barrier)): 100 * barrier time / incl time.The barrier time is measured as the time between the end of the thread execution and end of the last thread.
- Percentage of the total overhead due to the runtime library (%TO (RTL)): 100 * (RTL time – barrier time) / incl time

The performance data table window shows the following statistics for each process and thread in the application:

- Count: The number of times the event was executed.
- Exclusive time (Excl. Time): The total time (Does not include time inside of other events)
- Inclusive time (Incl. Time): Wall clock time for the event (including other events).
- Percentage of total overhead (% Overhead): 100 * (thread time – computation time) / thread time
- Thread time: (TT: Thread Time): Wall clock time for the execution of each thread

- Computation time: (CT: Comp. Time) Time per thread in the body of the OpenMP construct
- Percentage of imbalance (% imbalance): 100 * (comp time – MIN(comp time(i))) / MIN(comp time(i))
- Average thread time (Avg. Comp Time): SUM(comp time(i)) / Number of threads
- Total overhead: (TO: TT – CT): thread time – comp time
- Percentage of the total overhead due to barrier (%TO (Barrier)): 100 * barrier time / incl time.The barrier time is measured as the time between the end of the thread execution and end of the last thread.
- Percentage of the total overhead due to the runtime library (%TO (RTL)): 100 * (RTL time – barrier time) / incl time

Using I/O Profiling in Peekperf

Preparing Your Application

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you invoke **peekperf**. In order to set these environment variables, run the setup scripts that are located in the top level directory of your IBM HPC Toolkit installation. On AIX systems, these setup scripts are located in the **/usr/lpp/ppe.hpct** directory. On Linux, these setup scripts are located in the **/opt/ibmhpc/ppe.hpct directory**. If you are using **sh**, **bash**, **ksh**, or similar shell command, invoke the **env_sh** script as **. env_sh**. If you are using **csh**, invoke the **env_csh** script as **source env_csh**.

You start **peekperf** and load the application binary as described in the preceding **Using the Peekperf GUI** section. There are no additional compiler or linker options that are required to use I/O profiling.

Instrumenting Your Application

After the data collection window opens, select the **MIO** tab. This tab shows the possible instrumentation, similar to the following:

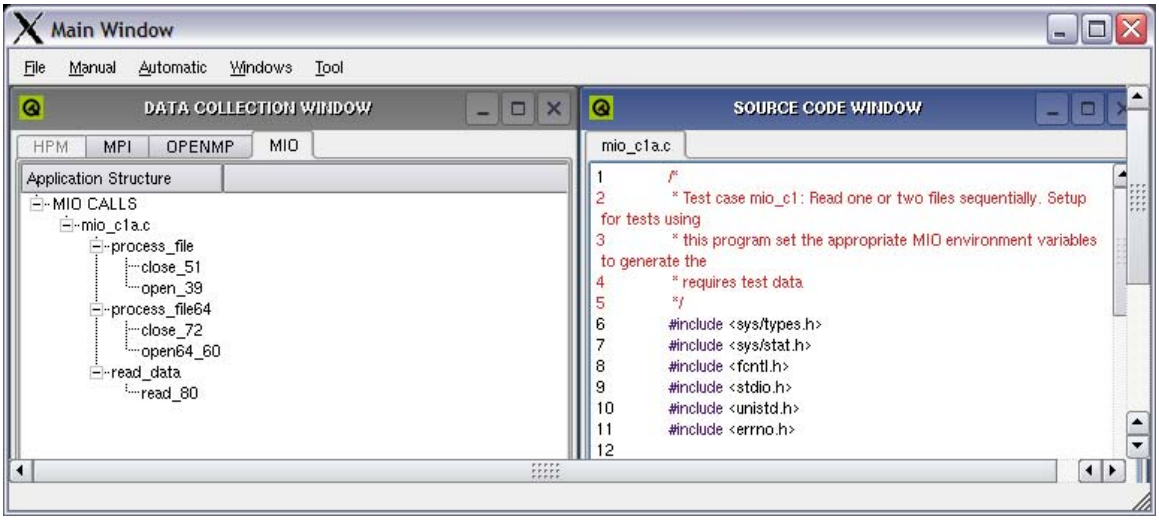


Figure 31: Peekperf I/O profiling data collection window

This image shows the application structure tree fully expanded. The leaf nodes are labeled with the name of the system call at that location and the line number in the source file. If you select leaf nodes, instrumentation is placed only at those specific instrumentation points. If you select a nonleaf node, instrumentation is placed at all leaf nodes that are child nodes of the selected nonleaf node.

In order for I/O profiling to work correctly, you must instrument at least the **open** and **close** system calls that open and close any file for which you want to obtain performance measurements.

After you have selected the set of instrumentation that you want, you instrument the application by selecting **Generate an Instrumented Application** from the **Automatic** menu.

Running Your Application

I/O profiling works by intercepting I/O system calls for any files that you want to obtain performance measurements for. In order to obtain the performance measurement data, the IBM HPC Toolkit uses the **MIO_STATS** and **MIO_FILES** environment variables. See the **I/O Profiling Environment Variables** section for a description of the MIO-related environment variables and their settings.

As a minimum, you must set the MIO_FILES environment variable to

- specify a file name matching pattern
- use the MIO trace module
- specify the **xml** and **events** options to the trace module.

Setting MIO_FILES to ***[trace/xml/events={./mio.evt}]** sets up MIO to collect the performance data that **peekperf** requires. Setting MIO_FILES to this value

- tells MIO to apply the options to all files opened by the application

- to generate the performance data in the XML file format required by **peekperf**
- that the I/O trace file will be written to **./mio.evt**.

After you have set the appropriate environment variables, you can run your instrumented application by selecting the **Run an Instrumented Application** from the **Automatic** menu and responding to the pop-up dialogs that were described in the **Using the Peekperf GUI** section earlier in this document

Viewing I/O Profiling Data

After your application completes, **peekperf** attempts to display the I/O profiling data that was collected when the application was run (if you requested the data be displayed). **Peekperf** displays the data in a tree format, in which the top level node is the file that the application read or wrote and the leaf nodes are the I/O function calls your application issued for that file. The following image shows the data visualization window with this tree fully expanded.

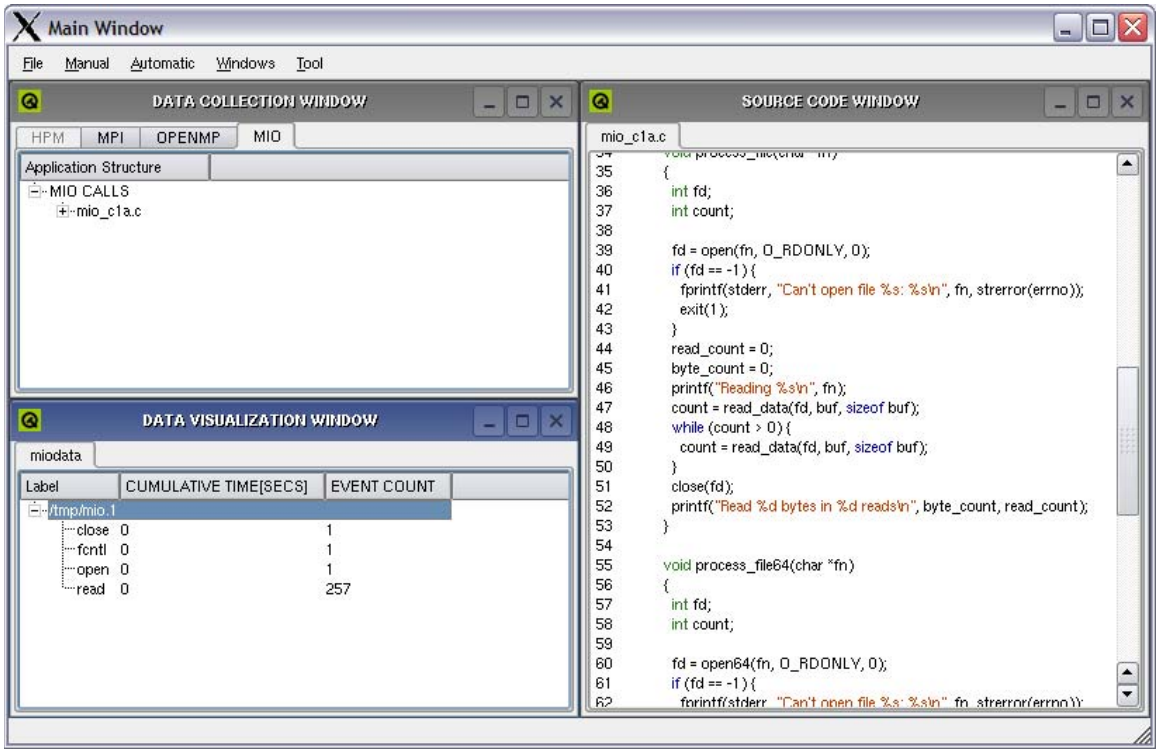


Figure 32: Peekperf I/O profiling data visualization window

Each row shows the time spent in an I/O function call and the number of times that function call was executed.

You can view detailed data for a leaf node by right-clicking over it. A metric browser window contains data for each process that executed that I/O function. You can view all of your performance measurements in a tabular form by selecting the **Show as Table**

option from the pop-up menu that appears when you right-click on white space within the data visualization window.

You can view a plot of your I/O measurements by right-clicking in white space in the data visualization window and then selecting **View Tracer** from the pop-up menu that appears. **Peekperf** displays a window that looks similar to the following:

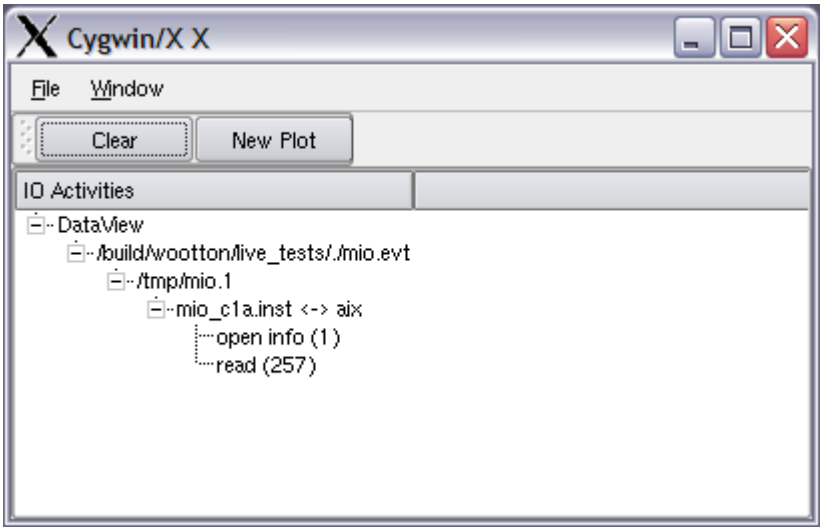


Figure 33: Peekperf I/O profiling data selection window

The window contains a tree view of the MIO performance data files. The top level nodes represent individual performance data files. The next level nodes represent individual files that the application accessed. The next level nodes represent the application program, and the leaf nodes represent the I/O function calls executed in the application. You can select nodes at any level of the tree to include the data from those nodes in the plot window.

The **File** menu has several options:

- **Read Event** opens a file selector dialog in which you can select a file containing I/O profiling data. **Peekperf** adds the selected I/O profiling data file to the IO Activities tree.
- **New Plot** opens a new I/O profiling graph, displaying the I/O profiling data from the selected nodes in the IO Activities tree.
- **Add to Current Plot** adds the I/O profiling data from the selected nodes in the IO Activities tree to the data plotted in the currently active I/O profiling graph.
- **Remove from Current Plot** removes the I/O profiling data for the selected nodes in the IO Activities tree from the currently active I/O profiling graph.
- **Edit Table** opens a table view of the I/O profiling data for the selected nodes in the IO Activities tree.

The **Window** menu displays a list of open I/O profiling graph and data view tables. You can select a window that you want to view from this menu.

The **Clear** button just below the menu bar clears the selection state for all nodes in the IO Activities tree. The **New Plot** button just below the menu bar opens a new I/O profiling graph using data for the nodes selected in the IO Activities tree. At least one node must be selected in the IO Activities tree in order to display a new plot.

When you select a node from the IO Activities tree and then select **New Plot** from the file menu or click the **New Plot** button, an I/O profiling graph window, similar to the following example, showing data from reading a file sequentially, opens.

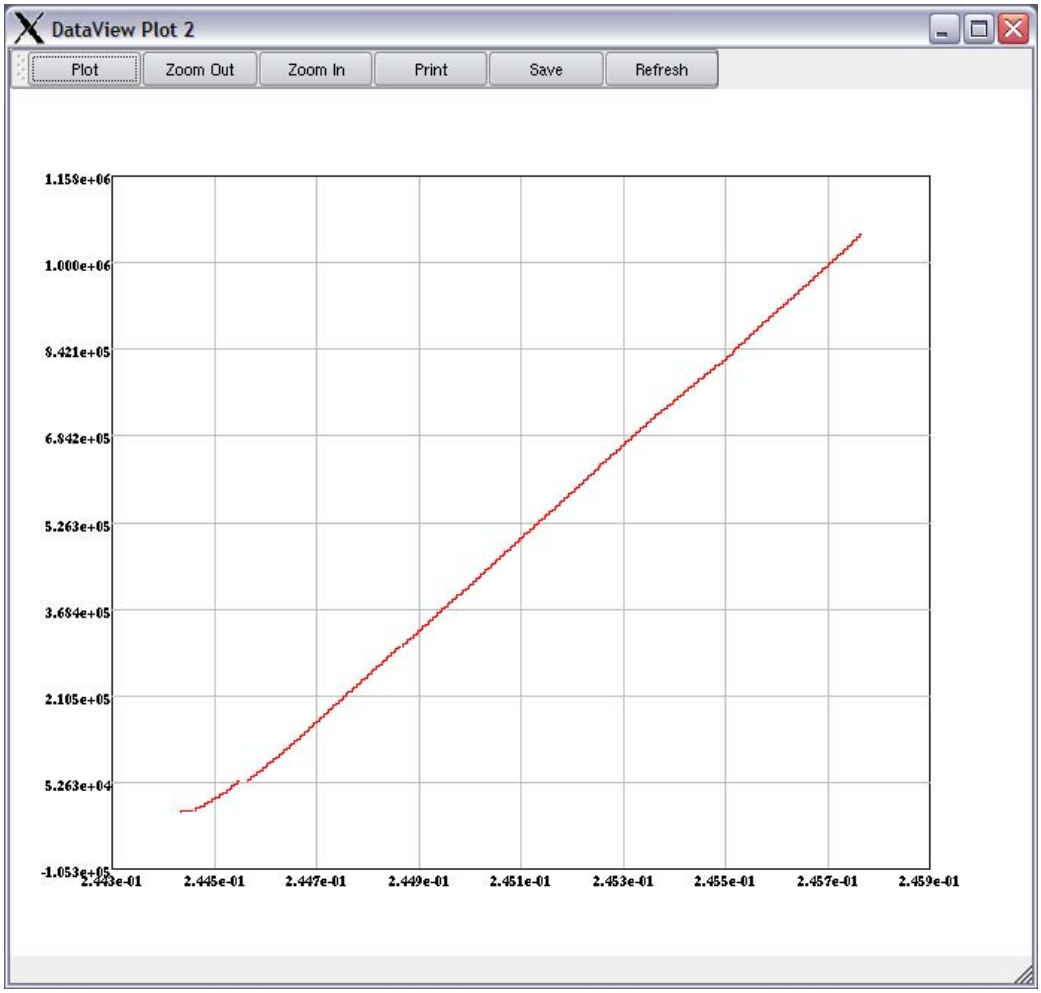


Figure 34: Peekperf I/O profiling data plot window

When the graph is initially displayed, the Y axis represents the file position, in bytes. The X axis of the graph always represents time in seconds.

You can zoom in to an area of interest in the graph by left-clicking at one corner of the desired area and dragging the mouse while holding the left button to draw a box around the area of interest and releasing the left mouse button. When you release the left mouse button, **peekperf** redraws the graph, showing the area of interest. You can then zoom in

and out of the graph by clicking the **Zoom In** and **Zoom Out** buttons at the top of the graph window. As you drag the mouse, **peekperf** displays the X and Y coordinates of the lower left corner of the box and the upper right corner of the box and the slope of the line between those two corners as text in the status bar area at the bottom of the window.

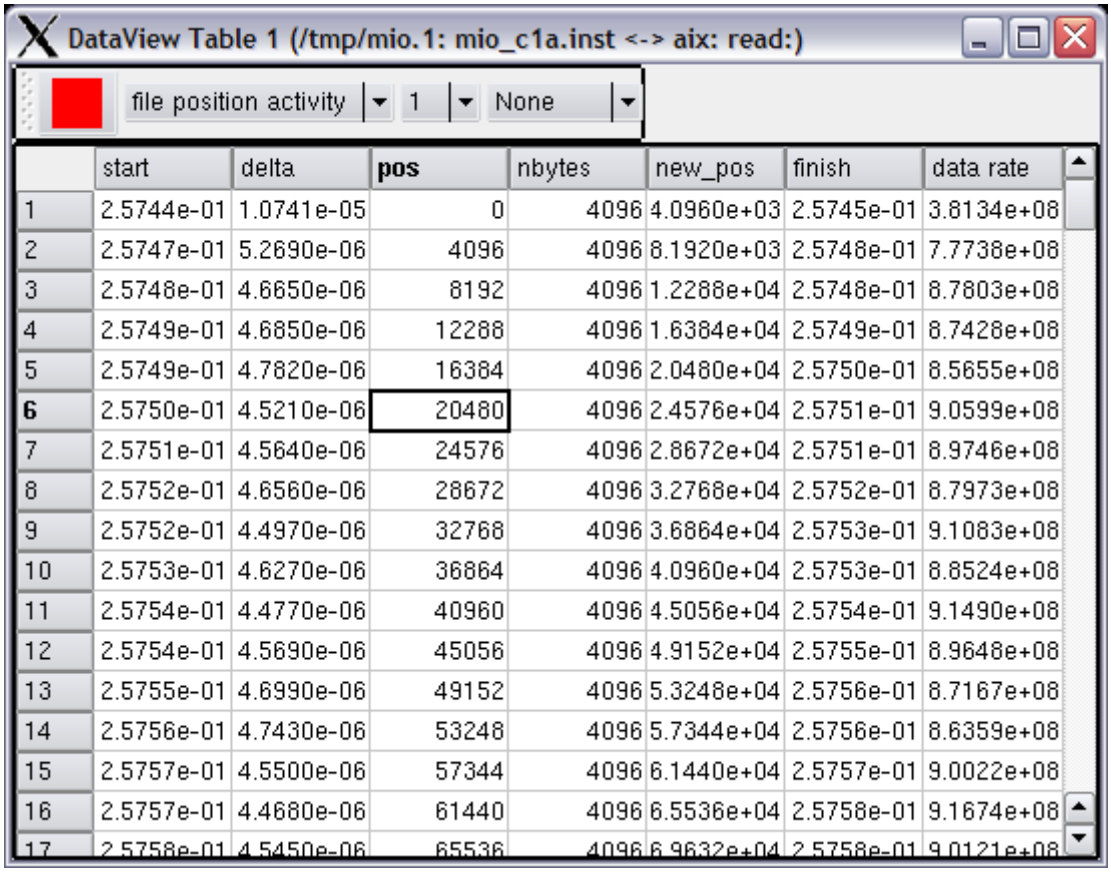
You can determine the I/O data transfer rate at any area in the plot by right-clicking over the desired starting point in the plot and holding down the right mouse button, while tracing over the section of the plot of interest. The coordinates of the starting and ending points of the selection region and the data transfer rate (slope) are displayed in the status area at the bottom of the plot window.

You can save the current plot to a jpeg file by clicking the **Save** button at the top of the plot window. A file selector dialog appears, which allows you to select the pathname of the file to which the screen image will be written to.

You can print the current plot, either to a printer or to a PDF file, by clicking the **Print** button at the top of the plot window. A printer setup dialog appears, where you can specify how the file should be printed.

You can select the printer from the list of available printers in the drop down box at the top of the dialog. If you select the **Print to file** checkbox, you can specify the path to the file in which the plot will be written as a PDF file.

You can view the I/O profiling data in tabular form and modify the characteristics of the current plot by selecting **Edit Table** from the **File** menu in the window that appeared when you selected the **View Tracer** option from the pop-up menu in the data visualization window. A window appears, similar to the following example, showing the I/O profiling data.



	start	delta	pos	nbytes	new_pos	finish	data rate
1	2.5744e-01	1.0741e-05	0	4096	4.0960e+03	2.5745e-01	3.8134e+08
2	2.5747e-01	5.2690e-06	4096	4096	8.1920e+03	2.5748e-01	7.7738e+08
3	2.5748e-01	4.6650e-06	8192	4096	1.2288e+04	2.5748e-01	8.7803e+08
4	2.5749e-01	4.6850e-06	12288	4096	1.6384e+04	2.5749e-01	8.7428e+08
5	2.5749e-01	4.7820e-06	16384	4096	2.0480e+04	2.5750e-01	8.5655e+08
6	2.5750e-01	4.5210e-06	20480	4096	2.4576e+04	2.5751e-01	9.0599e+08
7	2.5751e-01	4.5640e-06	24576	4096	2.8672e+04	2.5751e-01	8.9746e+08
8	2.5752e-01	4.6560e-06	28672	4096	3.2768e+04	2.5752e-01	8.7973e+08
9	2.5752e-01	4.4970e-06	32768	4096	3.6864e+04	2.5753e-01	9.1083e+08
10	2.5753e-01	4.6270e-06	36864	4096	4.0960e+04	2.5753e-01	8.8524e+08
11	2.5754e-01	4.4770e-06	40960	4096	4.5056e+04	2.5754e-01	9.1490e+08
12	2.5754e-01	4.5690e-06	45056	4096	4.9152e+04	2.5755e-01	8.9648e+08
13	2.5755e-01	4.6990e-06	49152	4096	5.3248e+04	2.5756e-01	8.7167e+08
14	2.5756e-01	4.7430e-06	53248	4096	5.7344e+04	2.5756e-01	8.6359e+08
15	2.5757e-01	4.5500e-06	57344	4096	6.1440e+04	2.5757e-01	9.0022e+08
16	2.5757e-01	4.4680e-06	61440	4096	6.5536e+04	2.5758e-01	9.1674e+08
17	2.5758e-01	4.5450e-06	65536	4096	6.9632e+04	2.5758e-01	9.0121e+08

Figure 35: Peekperf I/O profiling data table window

There are four widgets at the top of the table window that you can use to modify the characteristics of the current plot. You can change the values in these widgets as desired, then click the **Plot** button in the current plot window to apply the selections you made to the current plot.

The colored square at the upper left specifies the color to use when drawing the plot. If you click this square, a color selector dialog appears, which allows you to select the color you want to be used in drawing the plot.

The second widget from the left, labeled **file position activity**, selects the metric to be used for the Y and X axis of the plot and also affects the format of the plot. If you select **file position activity**, the Y axis represents **file position** and the X axis represents **time**. If you select **data delivery rate**, the Y axis represents the **data transfer rate** and the X axis represents **time**. If you select **rate vs pos,** the Y axis represents **data transfer rate** and the X axis represents the **start position** in the file

The third widget from the left specifies the pixel width for the graph that is drawn when the file position metric has been selected from the second widget from the left.

The rightmost widget specifies the metric that will have its numeric value displayed next to each data point. You can select any column displayed in the table, or **none** to plot each point with no accompanying data value.

XWindows Performance Profiler (Xprof)

The XWindows Performance Profiler (**Xprof**) 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. **Xprof** 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 **Xprof** topics are covered in this chapter:

- Before You Begin
- Starting the **Xprof** GUI
- Customizing **Xprof** 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 Xprof

Xprof 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 **Xprof** to analyze the resulting profiling information.

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

Requirements and Limitations

To use **Xprof**, your application must be compiled with the **-pg** flag. For more information, see the Compiling Applications to be Profiled *make link* below.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you invoke **Xprof**. In order to set these environment variables, you should run the setup scripts that are located in the top level directory of your installation. On AIX systems, these setup scripts are located in the /usr/lpp/ppe.hpct directory. On Linux, these setup scripts are located in the /opt/ibmhpc/ppe.hpct directory. If you are using sh, bash, ksh, or similar shell command, you should invoke the env_sh script as **. env_sh**. If you are using csh, you should invoke the env_csh script as **source env_csh**.

Note: Beginning with AIX 5.3, you can generate a new format of the thread-level profiling **gmon.out** files. **Xprof** 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, **Xprof** 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 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 FORTRAN application on a server, then try to analyze the profiled data on a workstation, the levels of FORTRAN runtime libraries must match and must be placed in a location on the workstation that **Xprof** recognizes. Otherwise, **Xprof** produces unpredictable results.

Because **Xprof** collects data by sampling, functions that run for a short amount of time might not show any CPU use. **Xprof** does not give you information about the specific threads in a multithreaded program.

Xprof presents the data as a summary of the activities of all the threads.

Comparing Xprof and the gprof Command

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

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

Compiling Applications to be Profiled

To use **Xprof**, 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 in to 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 in to **Xprof**. When you specify more than one **gmon.out** file, **Xprof** shows you the sum of the profile information contained in each file.

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

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

Starting the Xprof GUI

To start **Xprof**, enter the **Xprof** 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 **Xprof** command, or you can enter the **Xprof** 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 **Xprof**.

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

```
Xprof 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 **Xprof** command-line flags.

Specifying Xprof Command-line Flags

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

You can specify the flags in Table 2 from the command line or from the **Xprof** GUI (see Specifying Command Line Options (from the GUI) for more information).

Table 2. Xprof command-line flags

Use this flag:	To:	For example:
-a	Add alternative paths to search for source code and library files, or changes the current path search order. When using this flag, you can use the "at" symbol (**@**) to represent the default file path, in order to specify that other paths be searched before the default path.	To set an alternative file search path so that **Xprof** searches **pathA**, the default path, then **pathB**, type: `Xprof -a pathA:@:pathB`
-b	Suppress the printing of the field descriptions for the **Flat Profile**, **Call Graph Profile**, and **Function Index** reports when they are written to a file with the **Save As** option of the **File** menu.	Type: `Xprof -b a.out gmon.out`
-c	Load the specified configuration file. If this flag is used on the command line, the configuration file name specified with it will appear in the **Configuration File (-c):** text field in **Load Files Dialog** window and in the **Selection** field of the **Load Configuration File Dialog** window. When both the **-c** and **-disp_max** flags are specified on the command line, the **-disp_max** flag is ignored, but the value that was specified with it will appear in the **Initial Display (-disp_max):** field in the **Load Files Dialog** window the next time this window is opened.	To load the configuration file **myfile.cfg**, type: `Xprof a.out gmon.out -c myfile.cfg`
-disp_max	Set the number of function boxes that **Xprof** initially displays in the function call tree. The value supplied with this flag can be any integer between **0** and **5000**. **Xprof** displays the function boxes for the most CPU-intensive functions through the number you specify. For example, if you specify 50, **Xprof** displays the function boxes for the 50 functions in your program	To display the function boxes for the 50 most CPU-intensive functions in the function call tree, type: `Xprof -disp_max 50 a.out gmon.out`

	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 **Xprof** reports.	
-e	Deemphasize the general appearance of the function box for the specified function in the function call tree, and limits the number of entries for this function in the **Call Graph Profile** report. This also applies to the specified function's descendants, as long as they have not been called by nonspecified 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 nonspecified 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 nonspecified 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 nonspecified function in the program.	To deemphasize the appearance of the function boxes for **foo** and **bar** and their qualifying descendants in the function call tree, and limit their entries in the **Call Graph Profile** report, type: `Xprof -e foo -e bar a.out gmon.out`
-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 nonspecified 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	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: `Xprof -E foo -E bar a.out gmon.out`

	0. The same applies to function boxes for descendant functions, as long as they have not been called by nonspecified 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 nonspecified 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.	
-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 nonspecified functions and nondescendant 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 nonspecified or nondescendant 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: `Xprof -f foo -f bar a.out gmon.out`

-F	Change the general appearance and label information of all function boxes in the function call tree *except* for that of the specified function and its descendants. In addition, the number of entries in the Call Graph Profile report for the nonspecified and nondescendant functions is limited, and the CPU data associated with them is changed. The **-F** flag overrides the **-E** flag. In the function call tree, the function box for the specified function is made unavailable, and its size and shape also changes so that it appears as a square of the smallest 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 nonspecified or nondescendant 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: `Xprof -F foo -F bar a.out gmon.out`
-h|-?	Display the **Xprof** command's usage statement.	`Xprof -h`
-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 **Xprof** GUI. See Setting the File Search Sequence for more information.	To specify **/lib/profiled/libc.a:shr.o** as an alternative path name for your shared libraries, type: `Xprof -L /lib/profiled/libc.a:shr.o`
-s	Produce the **gmon.sum** profile data file (if multiple **gmon.out** files are specified when **Xprof** 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**, in to a file called **gmon.sum**, type: `Xprof -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: `Xprof -z a.out gmon.out`

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

Loading Files from the Xprof GUI

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

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

When you enter the **Xprof** command alone, the **Xprof** 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. All that is visible is a menu bar at the top with dropdowns for **File**, **View**, **Filter**, **Report** and **Utility**.

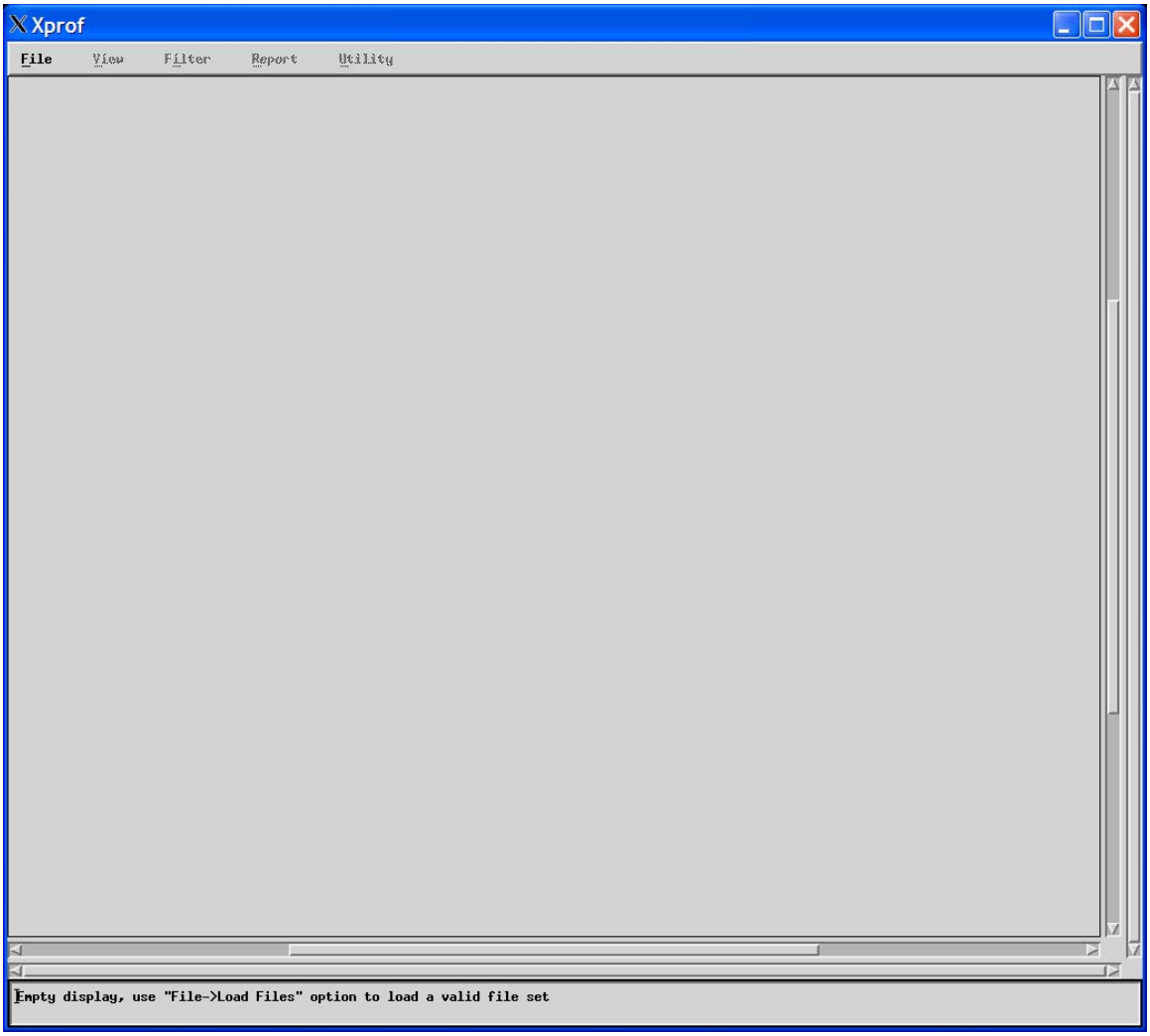


Figure 36: The Xprof main window

From the **Xprof** 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 is split in to 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 following three figures.

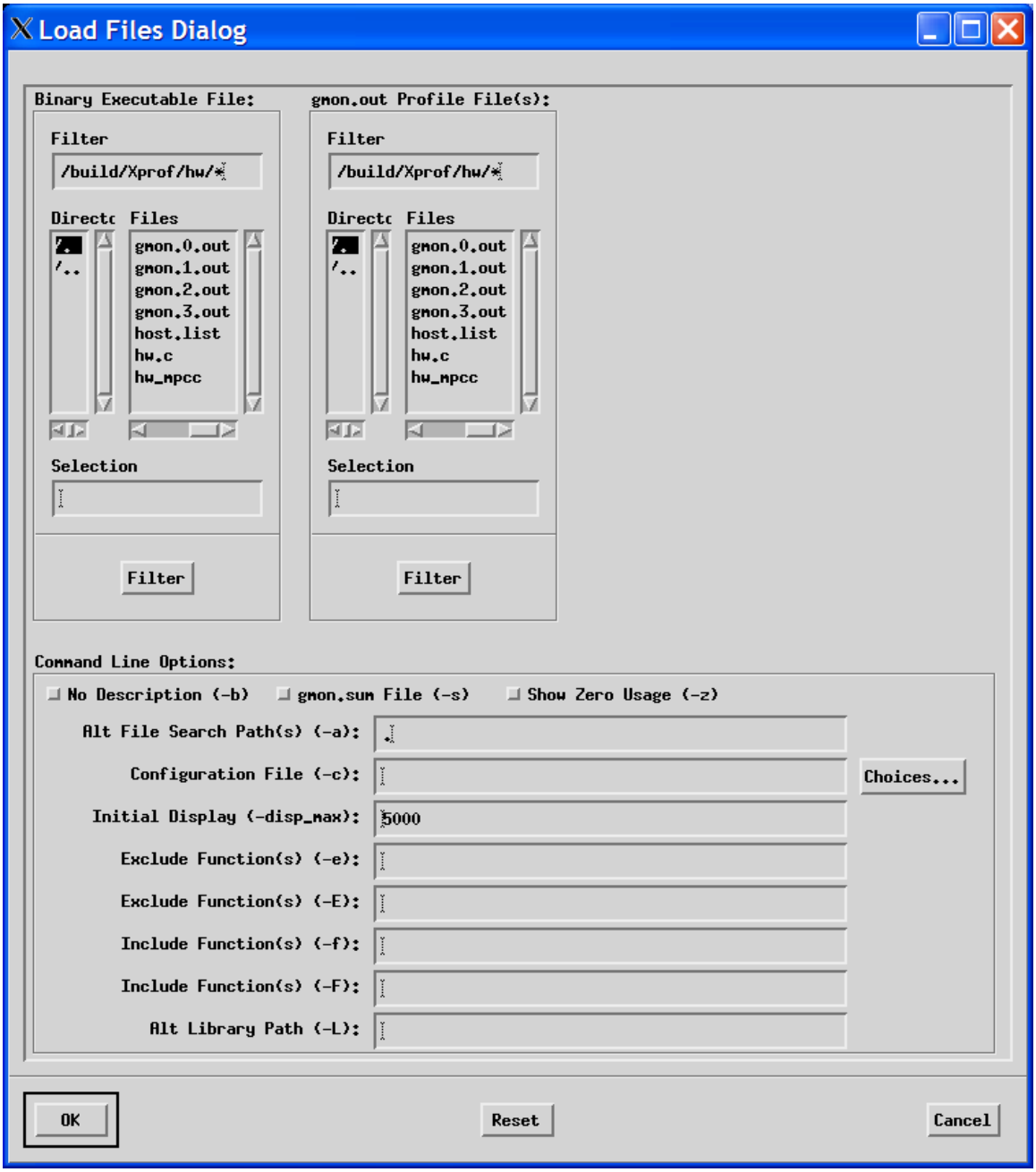


Figure 37. The Load Files Dialog window.

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

Specifying the Binary Executable File

You specify the binary executable file from the **Binary Executable File:** area 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.

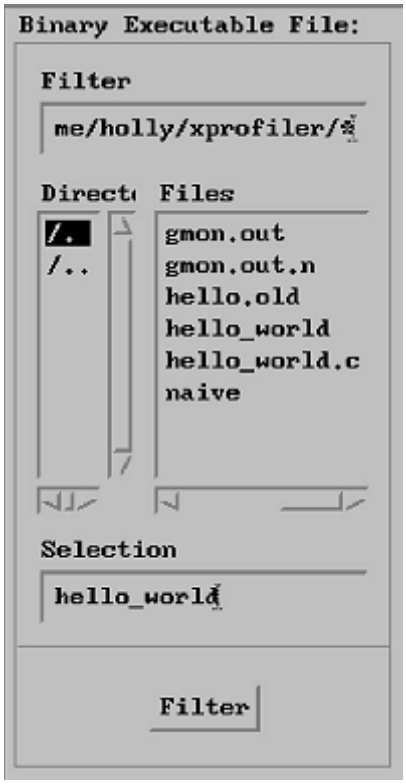


Figure 38. The Binary Executable File dialog.

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

Specifying 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. 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.

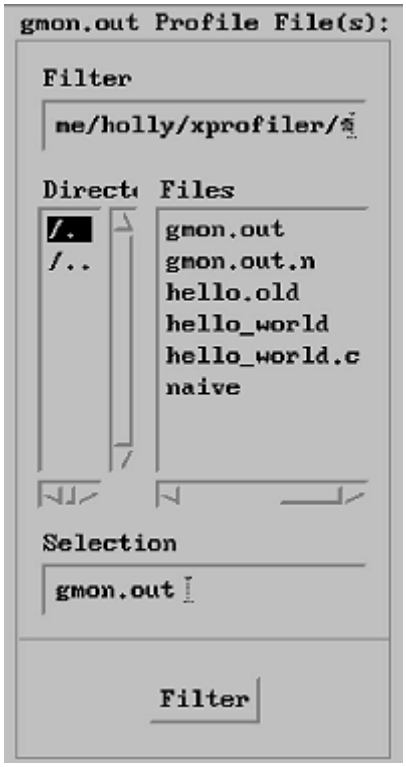


Figure 39. The gmon.out Profile Data File(s) area.

When you start **Xprof** using the **Xprof** command, you are not required to indicate the name of the profile data file. If you do not specify a profile data file, **Xprof** 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. **Xprof** then uses this file as input, even if it is not related to the binary executable file you specify. Because this will cause **Xprof** to display incorrect data, it is important that you enter the correct file in to 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.

Specifying Command Line Options (from the GUI)

Specify command-line flags (see **Xprof** command-line flags) from the **Command Line Options** area of the **Load Files Dialog** window. There are three check boxes side by side at the top:

- No description (-b)
- gmon.sum File (-s)
- Show Zero Usage (-z)

Below that, there are eight boxes corresponding to **Xprof** 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)
- Alt Library Path (-L)

There is also a **Choices** button next to the Configuration File (-c) box which will bring up a dialog that allows you to choose a configuration file.

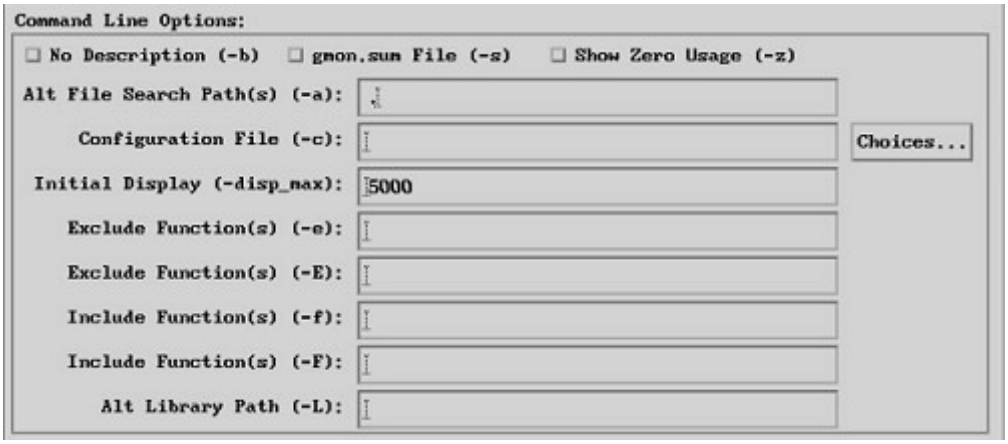


Figure 40. The Command Line Options area.

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. **Xprof** loads your application and displays its performance data.

Setting the File Search Sequence

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

Setting Default Paths

For library files, **Xprof** 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.

Setting 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 **Xprof** 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 **Xprof** session (until you exit this **Xprof** session or load a new set of data).

Understanding the Xprof Display

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

Xprof 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.

The Xprof Main Window

The **Xprof** main window contains a graphical representation of the functions and calls within your application, as well as their interrelationships. The window provides five menus.

After an example application has been loaded, the **Xprof** main window shows one cluster box displaying a function call tree, with an arc pointing down to another cluster box displaying a function call tree.

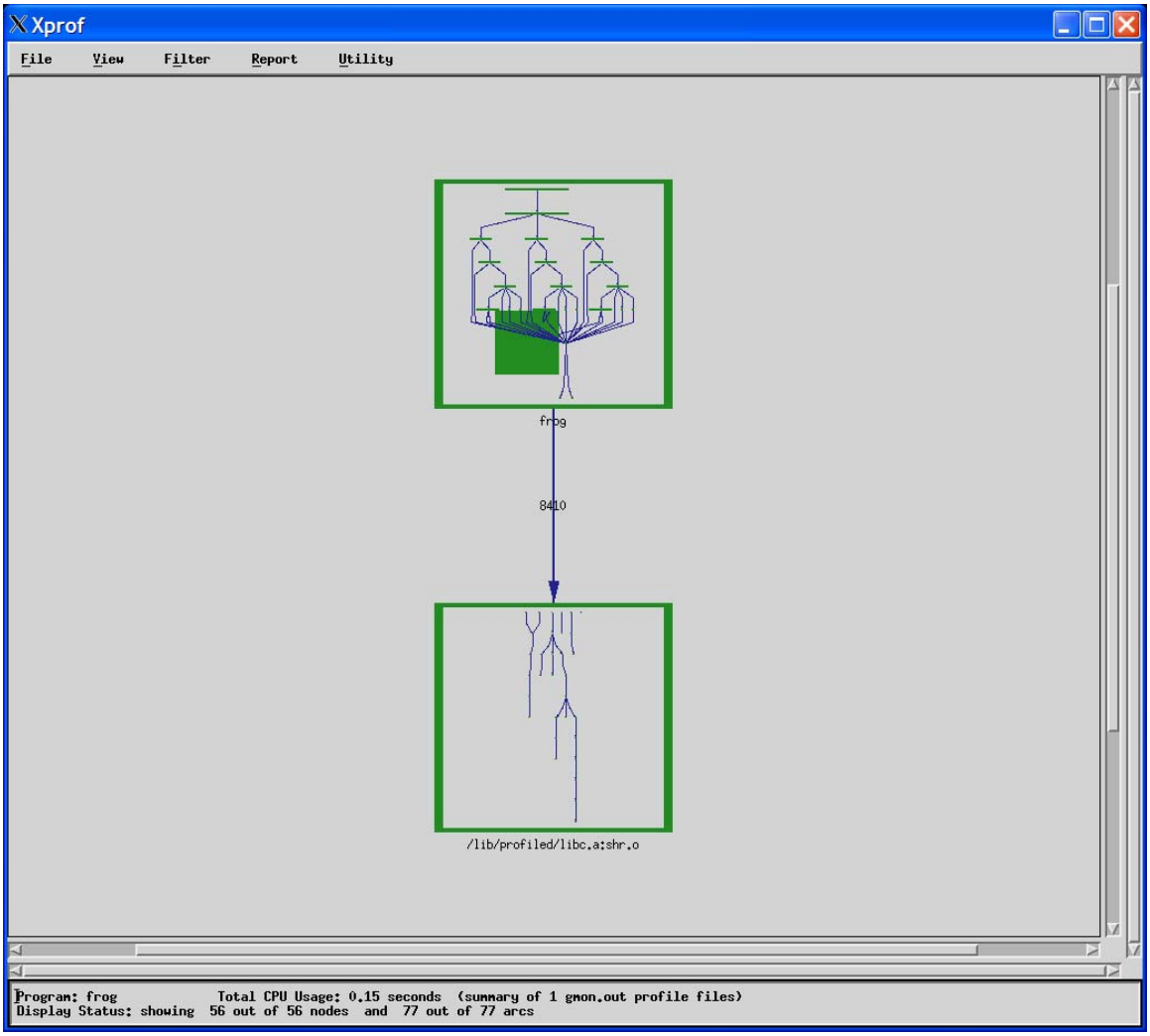


Figure 41. The Xprof main window with application loaded.

In the main window, **Xprof** 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 **Xprof** 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.

Xprof's Main Menus

The **Xprof** 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 **Xprof** will use. It also lets you control how your files are accessed and saved.

The View menu

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

The Filter menu

The **Filter** menu lets you add, remove, and change specific parts of the function call tree. By controlling what **Xprof** 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.

Xprof'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 **Xprof** 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 **Xprof** 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 **Xprof**. 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. 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 **Xprof** 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 **Xprof** 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.

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.

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

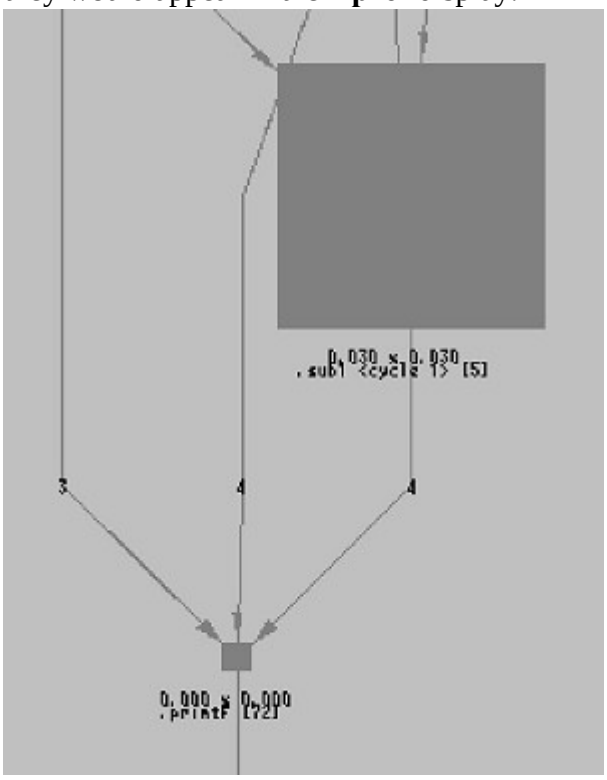


Figure 42. Function boxes and arcs in the Xprof display.

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

Xprof lets you collect the function boxes and call arcs that belong to each of your shared libraries in to 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

Xprof 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 **Xprof** 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.

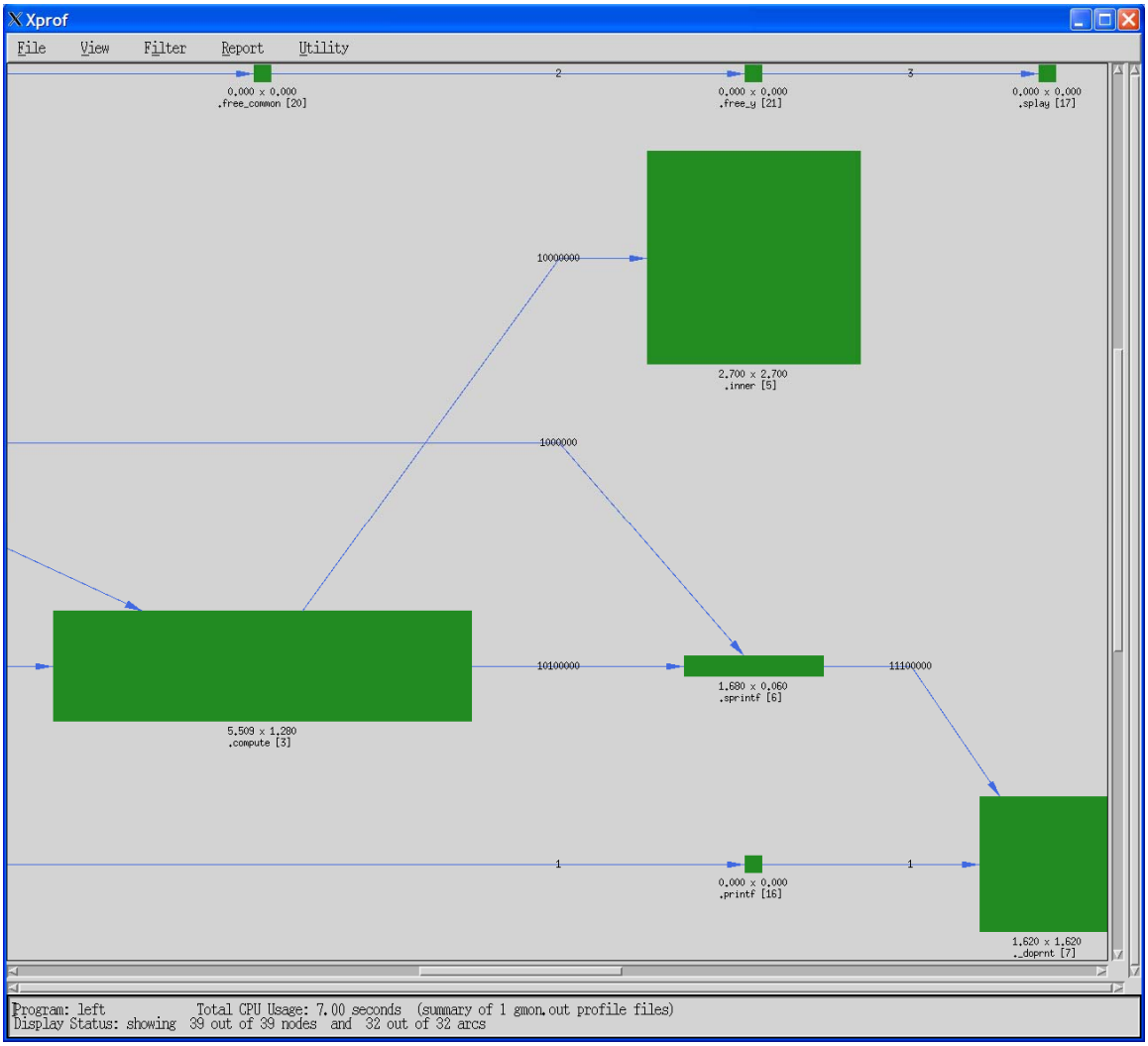


Figure 43. The Xprof main window with Left-to-right format selected.

Controlling the Representation of the Function Call Tree

You can choose to have **Xprof** represent the function call tree in either summary mode or average mode. These choices are only available when processing more than one **gmon.out** file.

When you select the **Summary Mode** option of the **View** menu, the size and shape of each function box is determined by the total CPU time of multiple **gmon.out** files used

on that function alone, and the total time used by the function and its descendant functions. The height of each function node represents the total CPU time used on the function itself. The width of each node represents the total CPU time used on the function and its descendant functions. When the display is in summary mode, the **Summary Mode** option is unavailable and the **Average Mode** option is activated.

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

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

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

Filtering what You See

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

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

Restoring the Status of the Function Call Tree

Xprof 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 **Undo** option which is available from the **Filter** menu, the function call tree is returned to its appearance just prior to the performance of the add or remove operation.

Whenever you invoke the **Undo** option, the function call tree loses its zoom focus and zooms all the way out to reveal the entire function call tree in the main display. When you start **Xprof**, the **Undo** option is unavailable. It is activated only after an add or

remove operation involving the function call tree takes place. After you undo an operation, the option is made unavailable again until the next add or remove operation takes place.

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

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

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

Displaying the Entire Function Call Tree

When you first open **Xprof**, 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 choosel 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. **Xprof** 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 **Xprof** 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 **Xprof** 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 **Filter** 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 **Xprof** window.

To add only specific function boxes back in to 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(s)** field. If you enter more than one function name, you must separate them with a blank space between each function name string. Regular expressions are also supported.

   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 appear in the **Xprof** display with the function call tree.

Filtering by Function Characteristics

The **Filter** menu of **Xprof** 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:

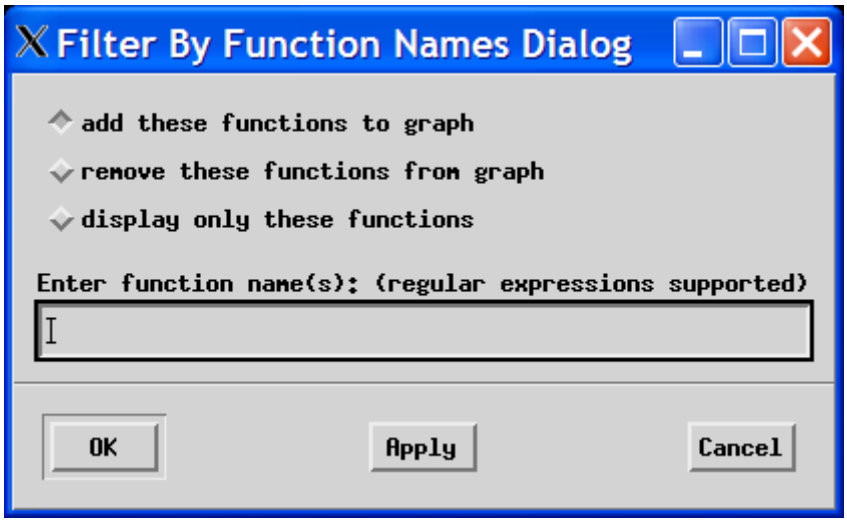


Figure 44. The Filter By Function Names Dialog window.

   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**
   - **Enter function names()** field for specifying a set of function names

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(s)** 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(s)** 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:

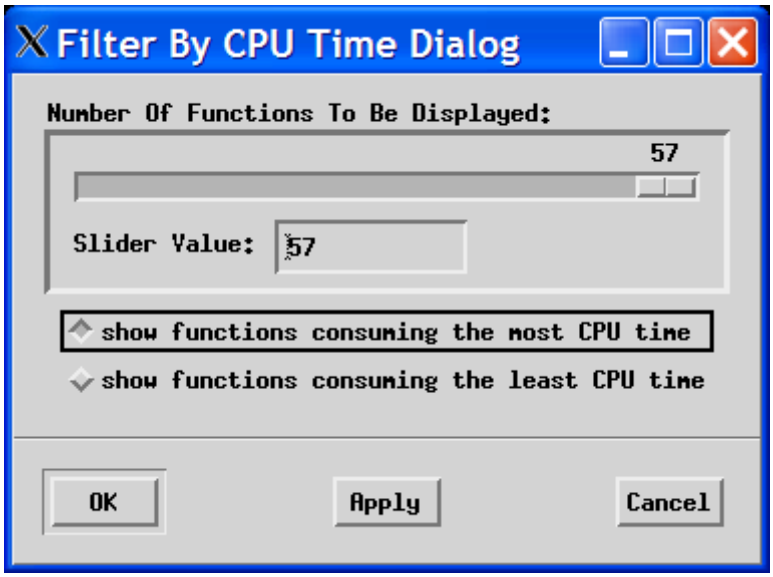


Figure 45. The Filter By CPU Time Dialog window.

The **Filter By CPU Time** Dialog window includes the following options:

- a slider bar to select the number of functions to display
- a value field to select the number of functions to display
- 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. You can move the slider in the **Number Of Functions To Be Displayed** bar until the desired number appears, or you can enter the number in the **Slider Value** field. The slider and **Slider Value** field are synchronized so when the slider is updated, the text field value is updated also. If you enter a value in the text field, the slider is updated to that value when you click **Apply** or **OK.** For example, to display the function boxes for the 10 functions in your application that consumed the most CPU, you would select the **show functions consuming the most CPU** button, and specify 10 with the slider or enter the value 10 in the text field.

4. Click **Apply** to show the changes to the function call tree without closing the dialog. Click **OK** to show the changes and close the dialog.

To filter by call counts, do the following:

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

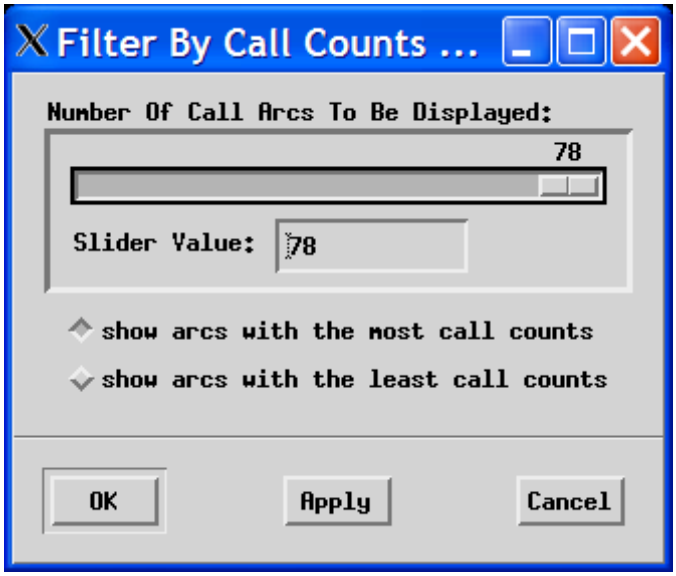


Figure 46. The Filter By Call Counts Dialog window.

The **Filter By Call Counts Dialog** window includes the following options:

- a slider bar to select the number of functions to display
- a value field to select the number of functions to display
- show arcs with the most call counts
- show arcs with the least call counts

2. Select the option you want (**show arcs with the most call counts** is the default).

3. Select the number of call arcs to which you want it applied. 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. **Xprof** 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. **Xprof** 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. **Xprof** 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.

Xprof 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 **Xprof** window, by default, the function boxes of your executable file, and the libraries associated with it, are clustered. Because **Xprof** 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 **Xprof** 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 **Xprof** 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 in to a single area, known as a library cluster.

The following figure shows an application with its function boxes unclustered.

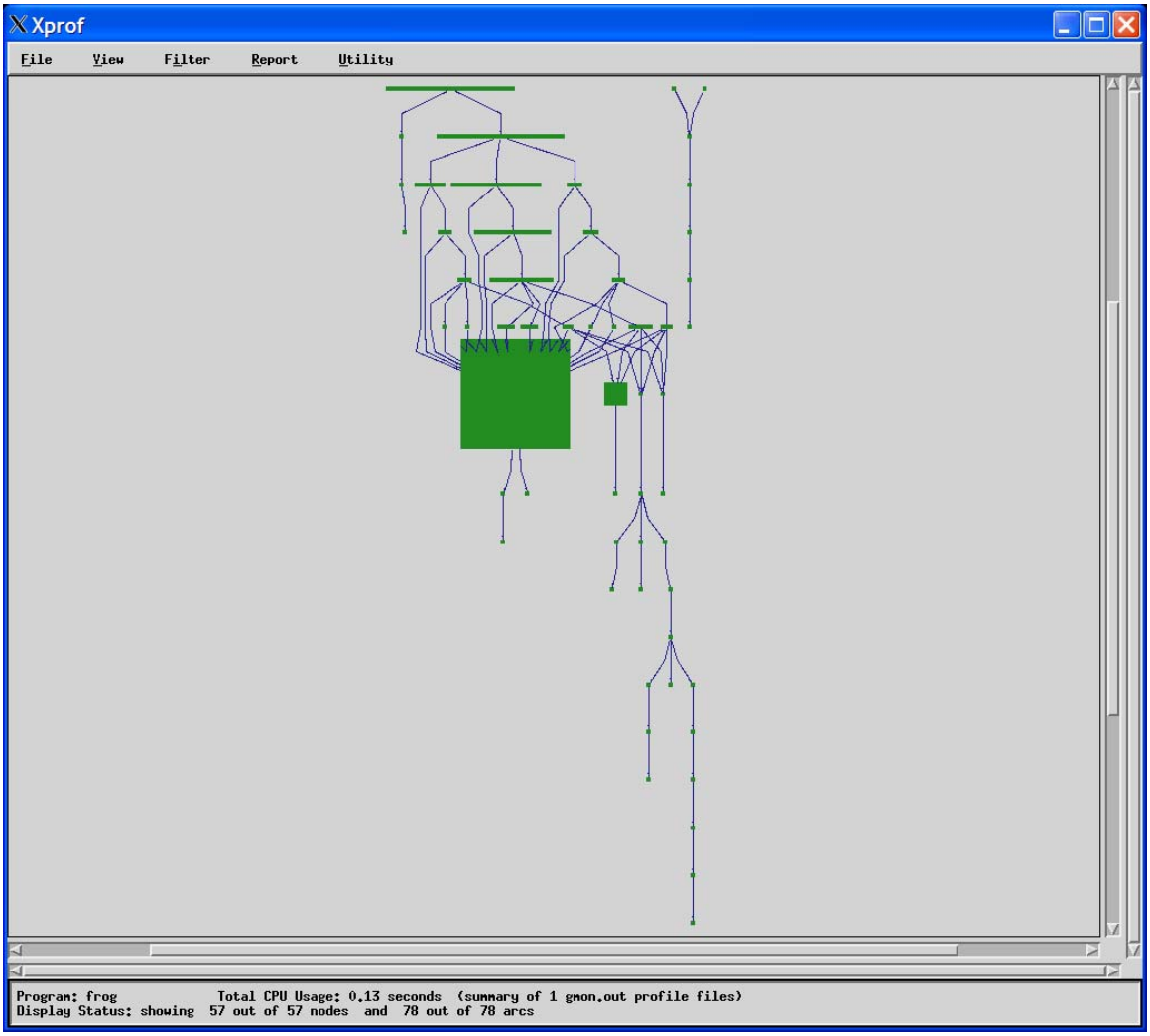


Figure 47. The Xprof main window with functions unclustered.

Clustering Functions

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

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

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

2. Select the **Collapse Cluster Node** option. The cluster box and its contents now appear as a small solid green box. In the following figure, the **/lib/profiled/libc.a:shr.o** library is collapsed.

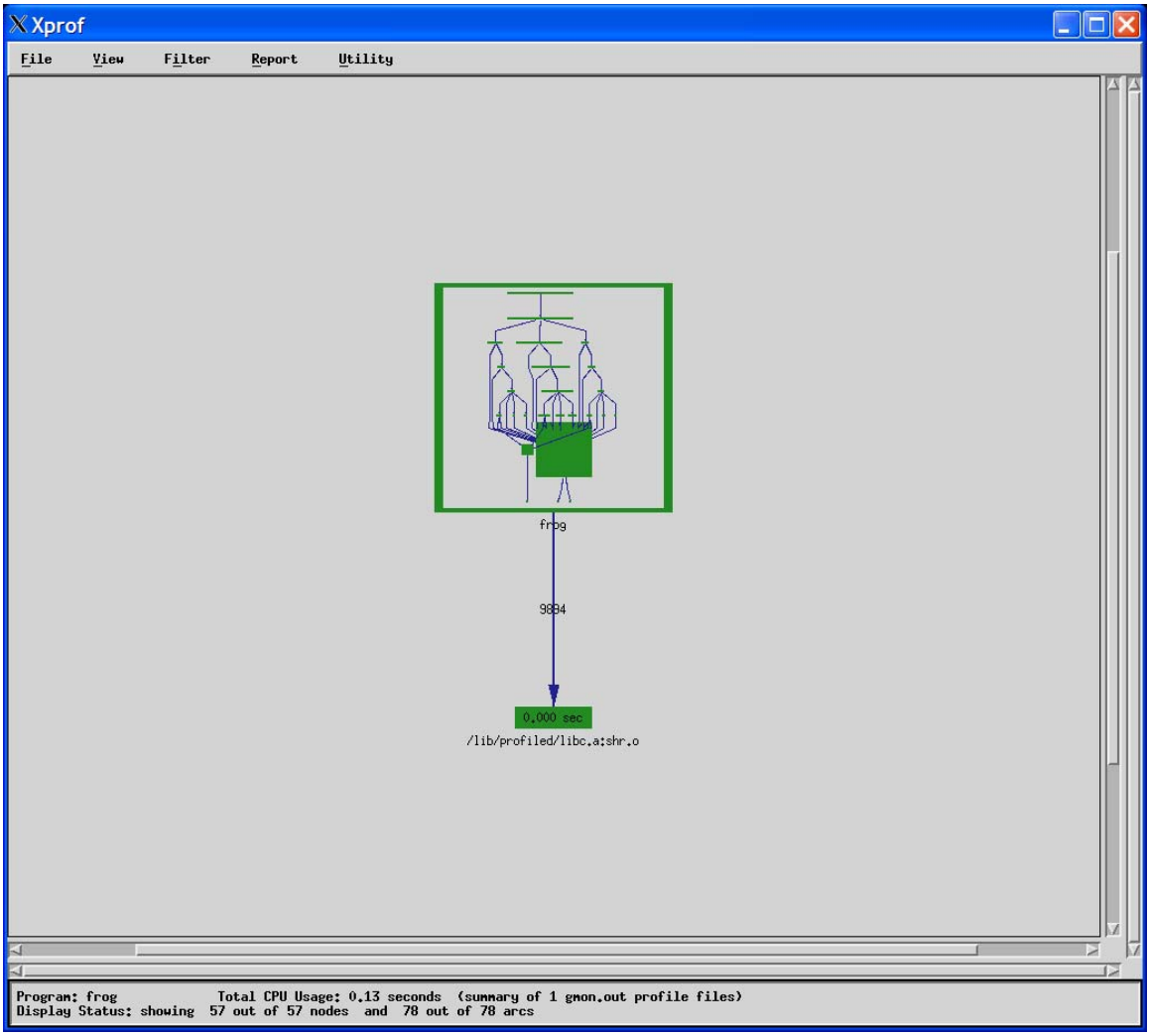


Figure 48. The Xprof main window with one library cluster box collapsed.

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

1. Place your mouse cursor over the collapsed cluster box and press the right mouse button. The **Cluster Node** menu appears.

2. Select the **Expand Cluster Node** option. The cluster box and its contents appear again.

Unclustering Functions

If the functions within your application are clustered, you can use an option of the **Filter** menu to uncluster them. To do this, select the **Filter** menu, and then the **Uncluster Functions** option. The cluster boxes disappear and the functions boxes of each library expand to fill the **Xprof** 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 an application with its cluster box removed. Its shared library remains within its cluster box.

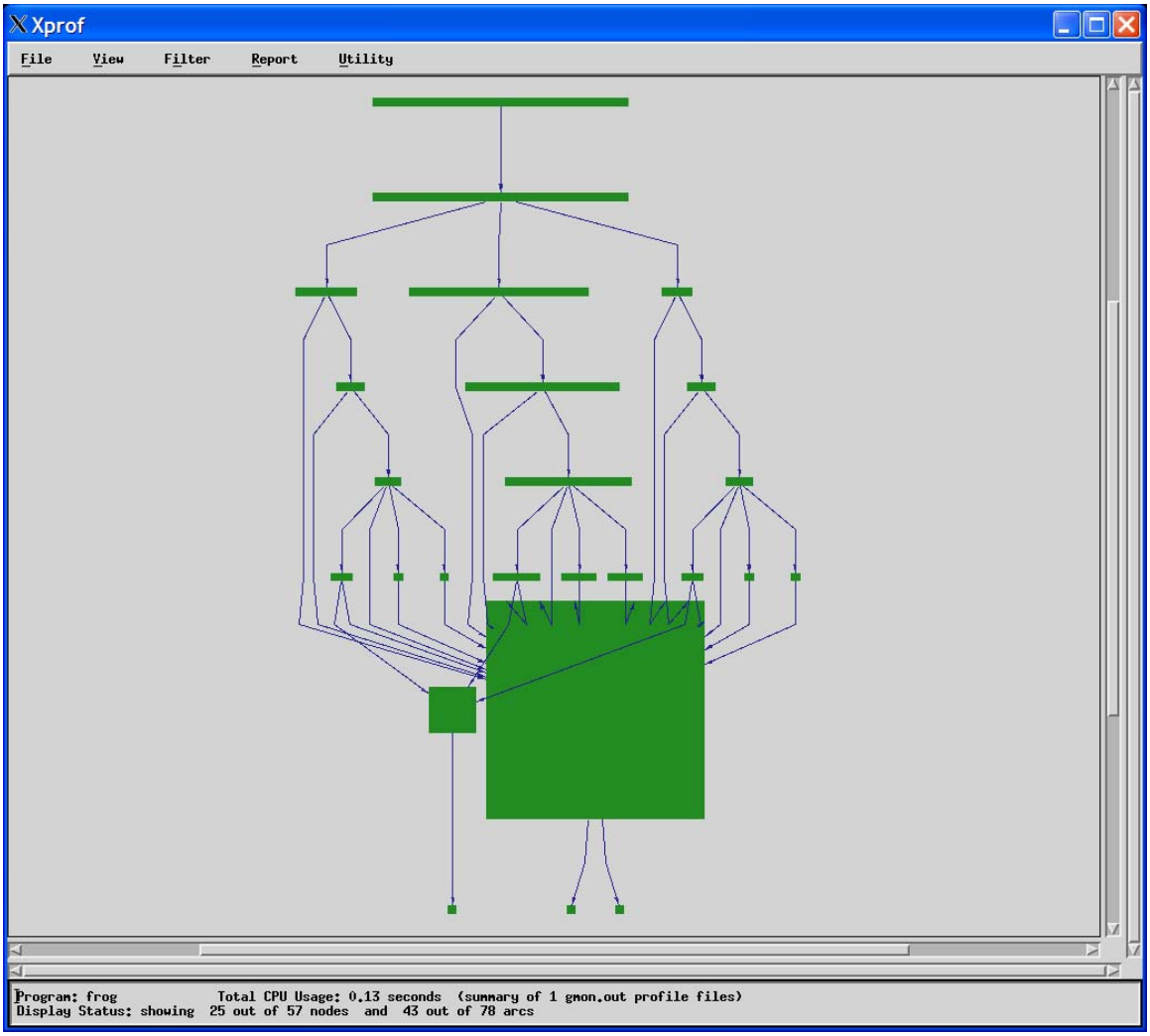


Figure 49. The Xprof main window with one library cluster box removed.

Locating Specific Objects in the Function Call Tree

If you are interested in one or more specific functions in a complex program, you 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). Regular expressions are supported.

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

Xprof 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.

Understanding Basic Function Data

Below each function box in the function call tree is a label that contains basic performance data, similar to the following:

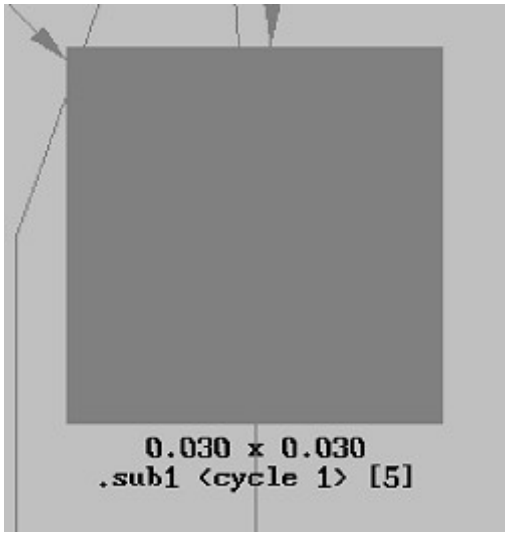


Figure 50. An example of a function box label.

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

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

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

- The amount of CPU time (in seconds) this function spent only on itself (the number on the right of the x).

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

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

For more information about summary mode and average mode, see Controlling the Representation of the Function Call Tree.

Because labels are not always visible in the **Xprof** 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.

Understanding 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, in the screen capture below, there are three arcs pointing to a function box. Each arc has a call arc label that indicates the number of calls that were made between the two functions, and in this example the arc labels are 3, 4, and 4.

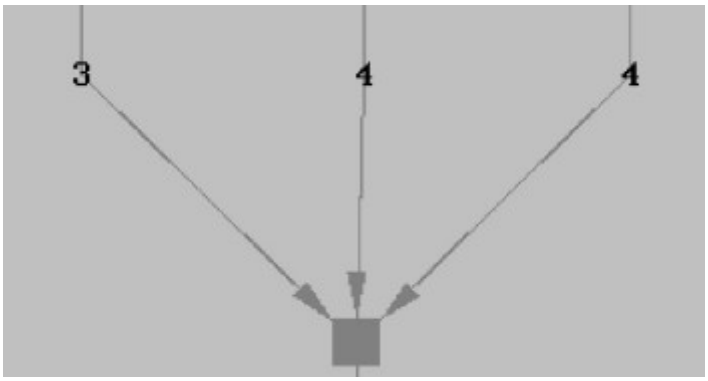


Figure 51. An example of a call arc label.

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

Basic Cluster Data

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

Understanding 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 **Xprof** 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

Using the Function Menu Statistics Report Option

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

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

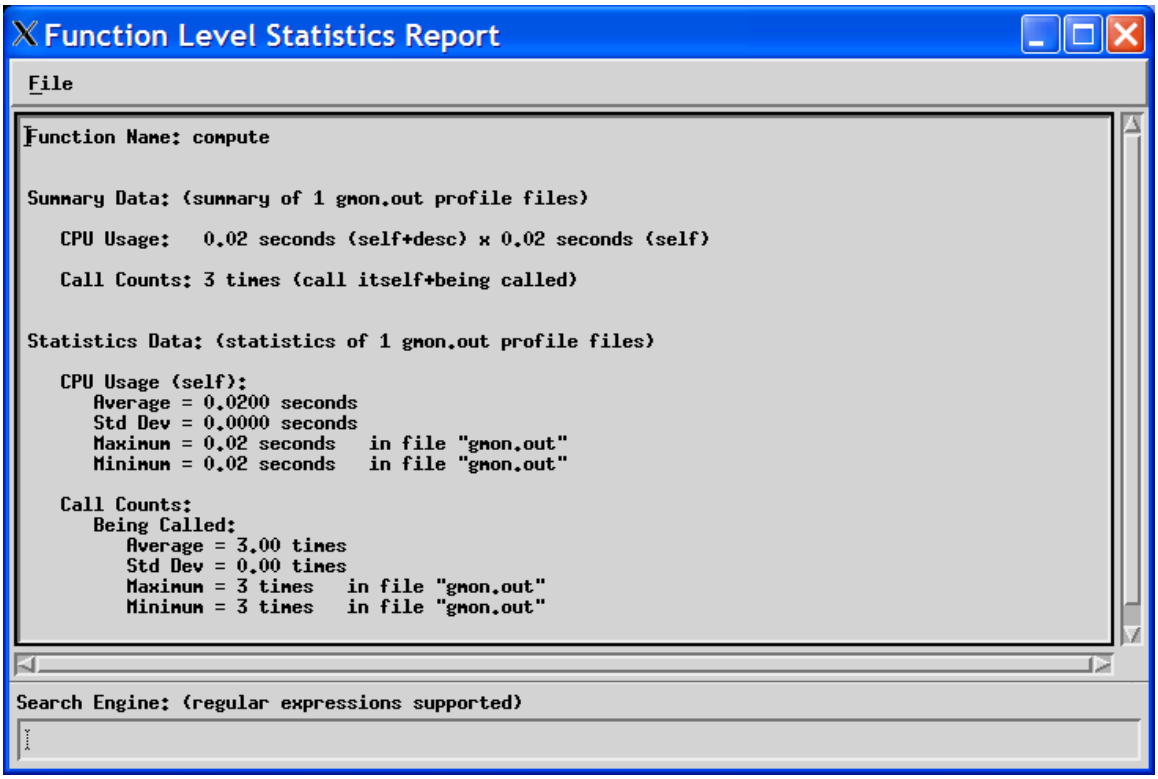


Figure 52. The Function Level Statistics Report window.

The **Function Level Statistics Report** window provides the following information:

Function Name

The name of the function you selected.

Summary Data

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

The CPU Usage field indicates:

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

The Call Counts field indicates:

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

Statistics Data

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

The CPU Usage field indicates:

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

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

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

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

The Call Counts field indicates:

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

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

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

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

Getting Detailed Data from Reports

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

You can access the **Xprof** 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.

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 **Xprof** 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 **Xprof** reports are explained as follows.

Understanding the Flat Profile Report

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

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

The Flat Profile report looks similar to the following:

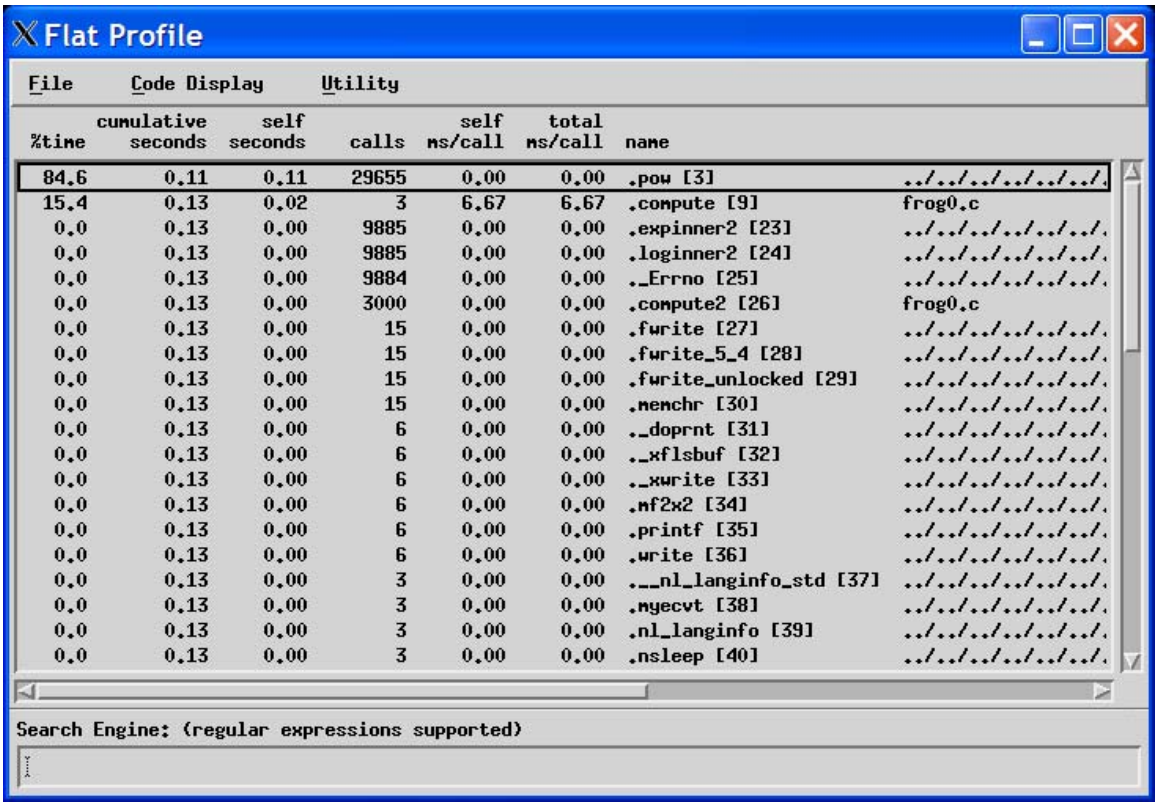


Figure 53. The Flat Profile window.

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. **Xprof** 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 **Xprof**. It also appears below the corresponding function in the function call tree.

Understanding the Call Graph Profile Report

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

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

The Call Graph Profile report looks similar to the following:

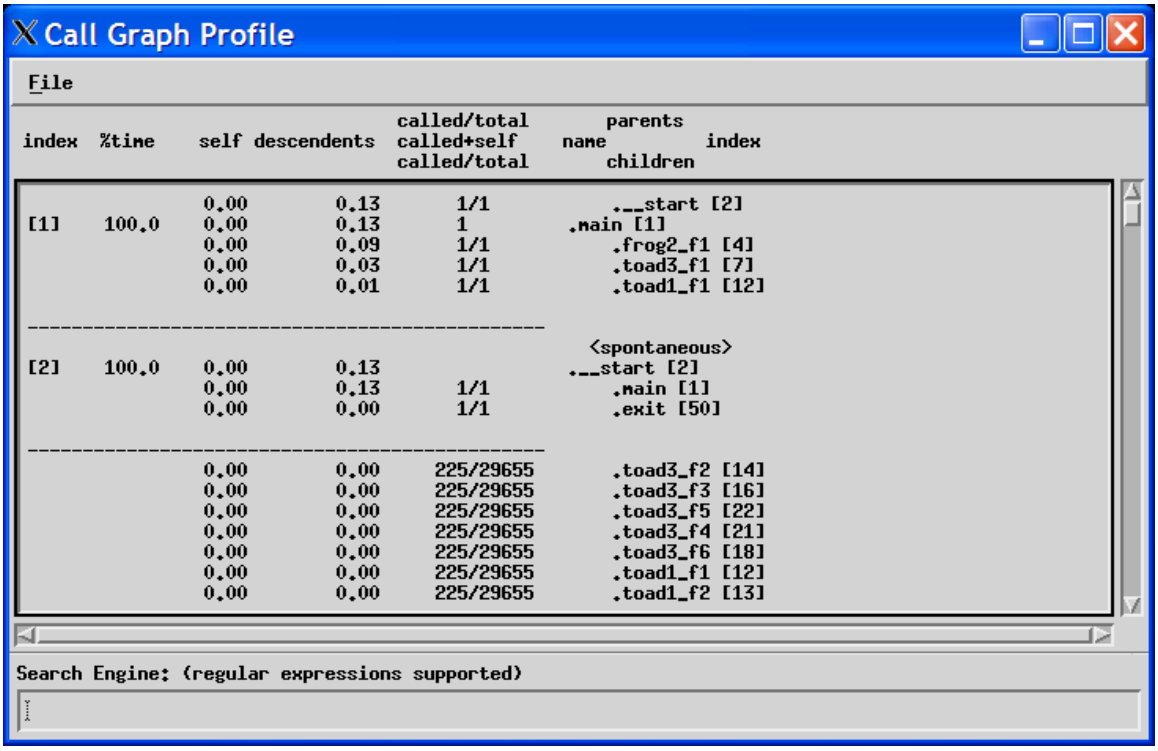


Figure 54. The Call Graph Profile window.

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 **Xprof** 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.

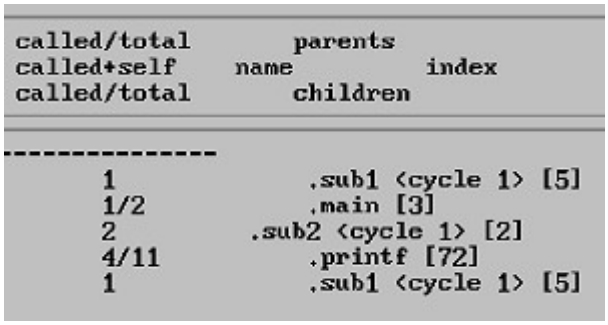


Figure 55. The called/total, call/self, called/total field of the Call Graph Profile window.

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

- called+self

The number of times the function of interest called itself, recursively.

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

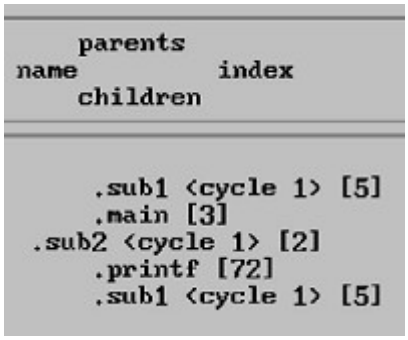


Figure 56. The name/index/parents/children field of the Call Graph Profile window.

- 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, **Xprof** 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.

Understanding the 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 **Xprof**. 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 for more information.

The Function Index report looks similar to the following:

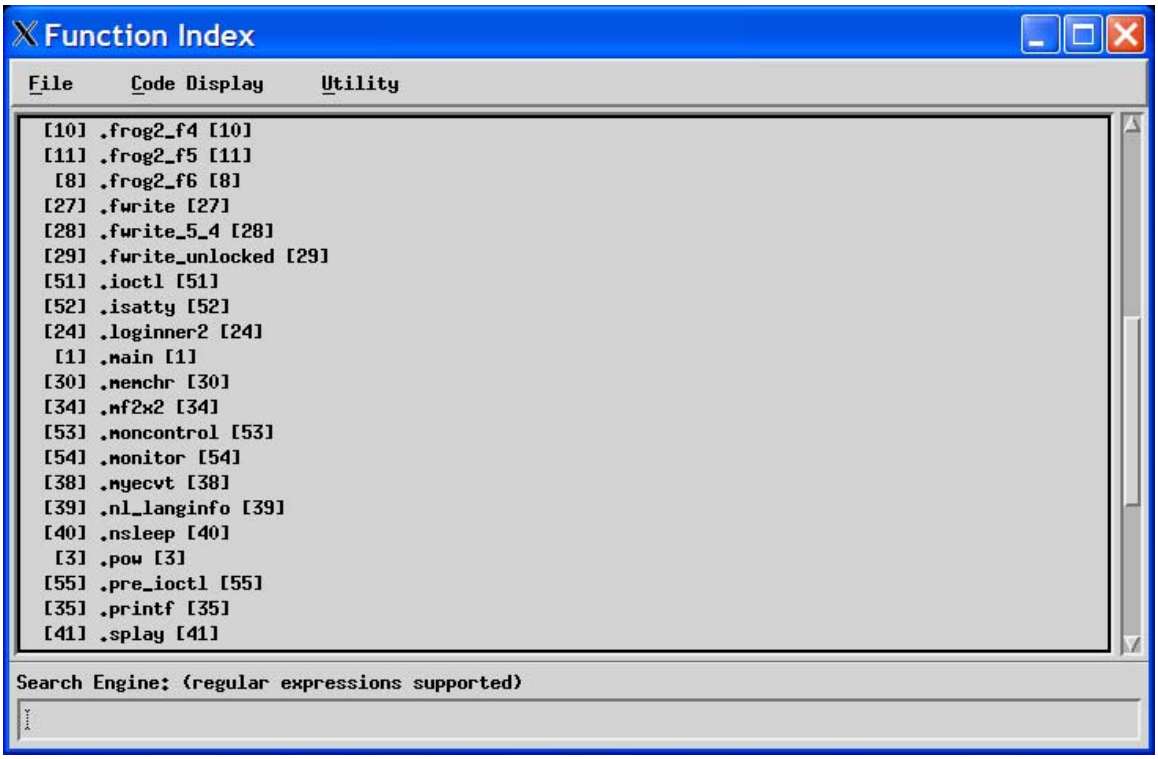


Figure 57. The Function Index window.

Understanding the Function Call Summary Report

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

The Function Call Summary report looks similar to the following:

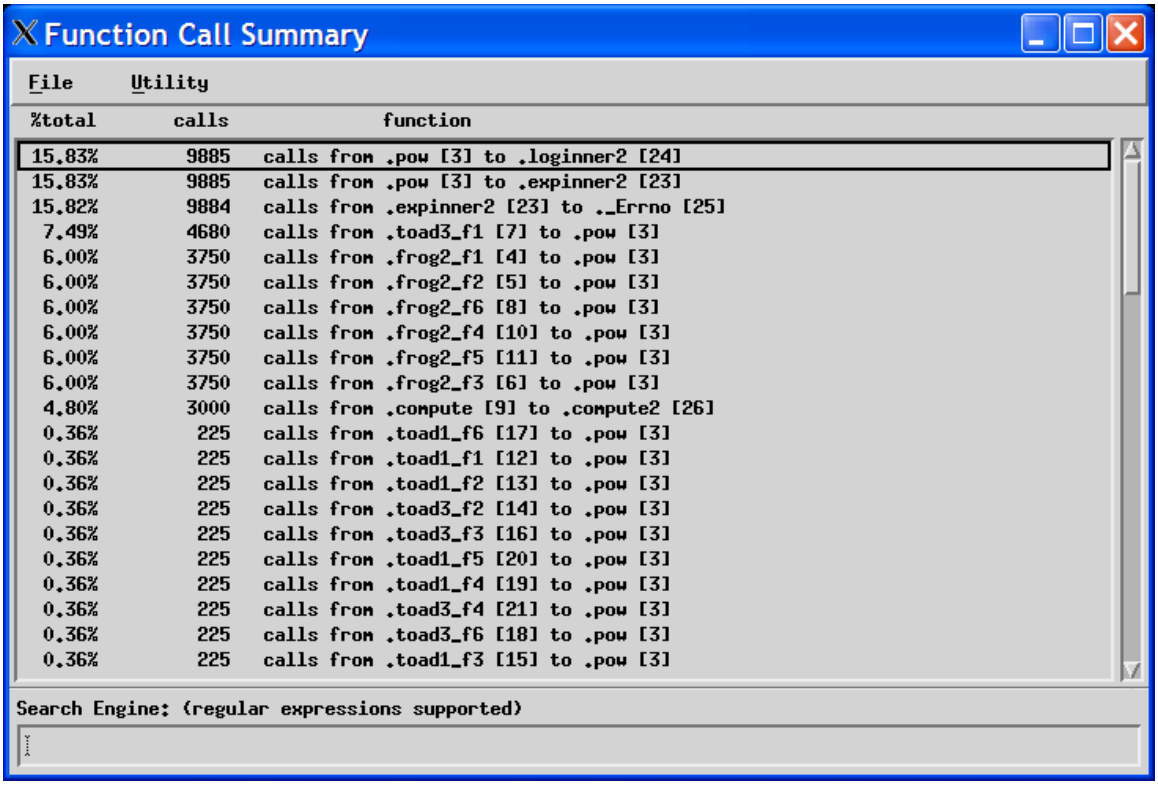


Figure 58. The Function Call Summary window.

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

Understanding the Library Statistics Report

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

The Library Statistics report looks similar to the following:

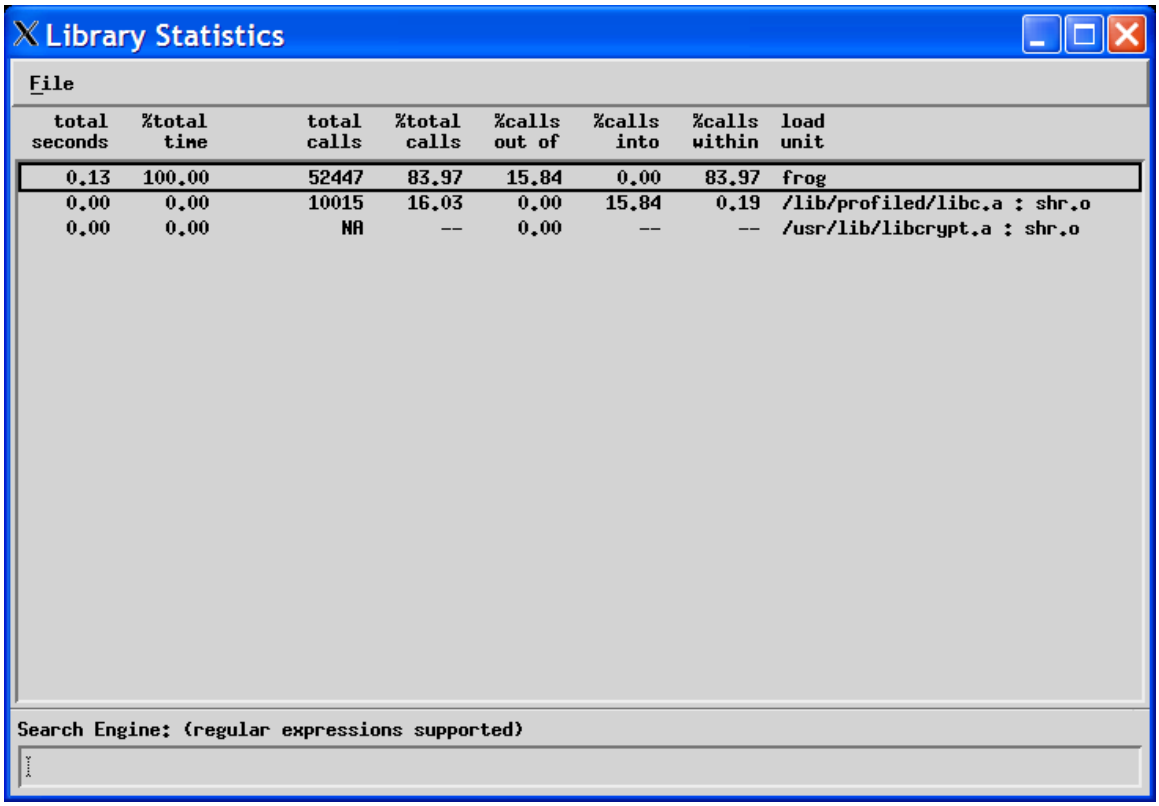


Figure 59. The Library Statistics window.

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 in to 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

Xprof 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 **Xprof** GUI.

Saving a single report

To save a single report, go to the **Report** menu on the **Xprof** 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 **Xprof** 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 Profile Reports Dialog window appears.

To save the reports, do the following:

1. Specify the file in to 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 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 window and the Save Flat Profile window appears, you cannot use any other option of the Flat Profile report window.

The File Selection field of the Save File Dialog windows 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 **Xprof**. 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 **Xprof Save gmon.sum As** option produces the same result as the **Xprof -s** command and the **gprof** -s command. If you run **Xprof** 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 in to which the summarized, profiled data should be placed. By default, **Xprof** puts the data in to 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 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 **Xprof** session or in a different session, you can read this configuration file in using the **Load Configuration** menu option. For more information, see Loading a configuration file.

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 **Forces 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 for more information. The **Load Configuration** menu 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 **Xprof** 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).

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

Xprof 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 **Xprof** 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.

1. To access the Source Code window through the **Function** menu:

2. Click the function box you are interested in with the right mouse button. The **Function** menu appears.

From the **Function** menu, select the **Show Source Code** option. The Source Code window appears.

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

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

2. From the Flat Profile window, select the function you would like to view by clicking on its entry in the window. The entry is highlighted to show that it is selected.

3. Select the **Code Display** menu, and then the **Show Source Code** option. The Source Code window appears, containing the source code for the function you selected.

Using the Source Code window

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

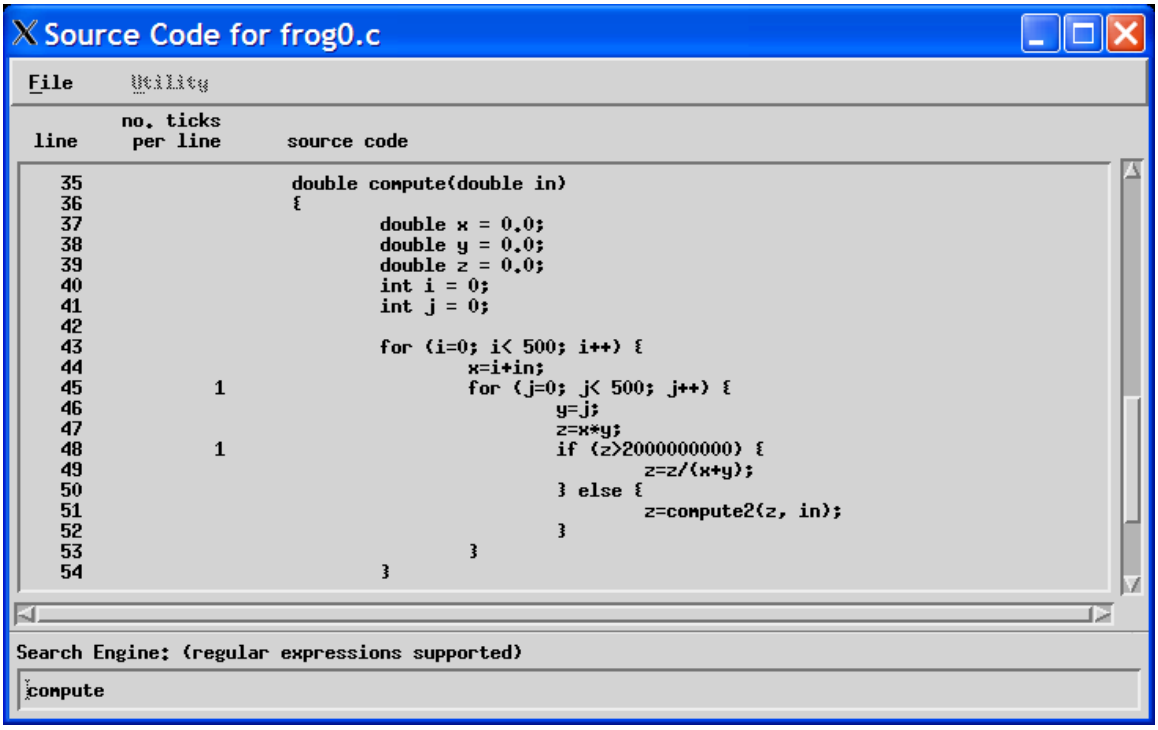


Figure 60. The Source Code window.

The Source Code window contains information in the following fields:

- line
  The source code line number.

- no. ticks per line
  Each tick represents .01 seconds of CPU time used. The value in this field represents the number of ticks used by the corresponding line of code. For example, if the number 3 appeared in this field, for a source statement, this source statement would have used .03 seconds of CPU time. The CPU usage data only appears in this field if you used the **-g** flag when you compiled your application. Otherwise, this field is blank.

- source code
  The application's source code.

The Source Code window contains the following menus:

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

  To close the Source Code window, select the **Close** option from the **File** menu.

- 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 by using the **Report** menu or the **Function** menu. The Disassembler Code window looks similar to the following:

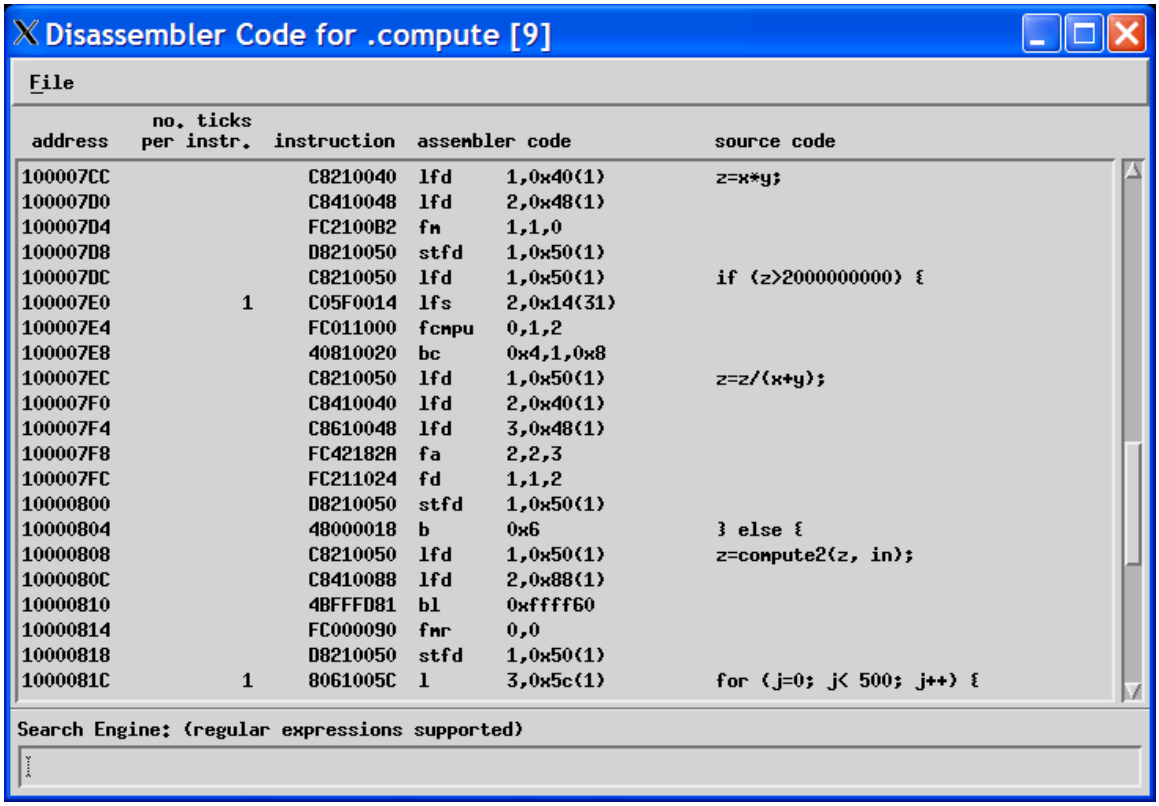


Figure 61. The Disassembler Code window.

The Disassembler Code window contains information in the following fields:

- address
  The address of each instruction in the function you selected (from either the Flat Profile window, the **Function Index** 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 hexadecimal representation of 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 Disassembler Code dialog appears. For information on using the Save Disassembler Code dialog, see Saving the Call Graph Profile, Function Index, and Flat Profile reports to a file.

  To close the Disassembler Code window, select **File** and then **Close**.

Saving Screen Images of Profiled Data

The **File** menu of the **Xprof** GUI includes an option called **Screen Dump** that lets you capture an image of the **Xprof** 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 Options**. The Screen Dump Options Dialog window appears.

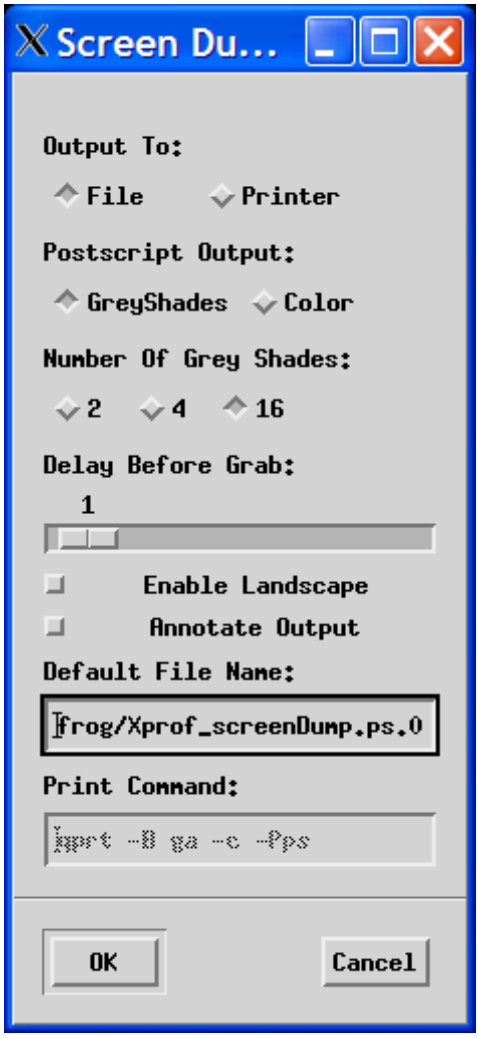


Figure 62. The Screen Dump Options Dialog window.

3. Make the appropriate selections in the fields of the **Screen Dump Options Dialog** window, as follows:
   - Output To:
     This option lets you specify whether you want to save the captured image as a PostScript file or send it directly to a printer.

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

     To send the image directly to a printer, select the **Printer** button. The image is sent to the printer you specify in the **Print Command** field of this dialog window. When you specify the **Printer** 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 **Xprof.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 nonnegative integer), the suffix automatically increases by one every time a new output file is written in the same **Xprof** 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 **Xprof** window, place the cursor in the window and then click the left mouse button.

To capture a portion of the **Xprof** 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 Xprof 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 **Xprof**.

You can customize resources by assigning a value to a resource name in a standard XWindows format. Several resource files are searched according to the following XWindows 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 XWindows should look.

For example, consider the following resource variables for a hypothetical XWindows 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 in to the .Xdefaults file:

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

Setting Xprof Resource Variables

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

Controlling Fonts

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

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

To specify the font used in textual reports:

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

Controlling the Appearance of the Xprof Main Window

To specify the size of the main window:

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

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

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

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

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

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

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

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

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

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

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

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

Function boxes are 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:

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

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

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

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

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

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

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

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

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

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

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

Controlling variables related to the Screen Dump option

To specify whether a screen dump will be sent to a printer or placed in a file:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Use this resource variable:	Specify this default, or a value of your

	choice
Xprofiler*PrintFileName	/tmp/Xprof_screenDump.ps.0

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

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

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

Controlling Variables Related to the View Menu

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

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

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

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

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

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

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

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

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

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

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

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

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

Controlling Variables Related to the Filter Menu

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

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

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

Hardware Performance Counter Tools

The IBM HPC Toolkit provides a command line tool called **hpccount**, and a library called **libhpc**, which access hardware performance counters to help you analyze your application's performance. You can use the **hpccount** command to report hardware performance counter measurements for your entire application. You can obtain measurements from a single hardware counter group. On AIX systems, you can multiplex multiple groups of hardware counters so that you can get an estimate of hardware performance counter events for multiple groups in a single run of your application. The **hpccount** command also can report derived metrics, which are additional measurements computed from hardware performance counter measurements to help you better understand your application's performance.

You can use the **hpcstat** command to obtain overall system statistics for hardware performance counters or for system resource usage. The **hpcstat** command requires root access in order to obtain system-wide statistics.

You can use the **libhpc** library to make more precise measurements of hardware performance counter events by placing calls to regions of code in your application that are of interest. The **libhpc** library provides the same features as the **hpccount** command. In addition, this library supports the use of plugins to aggregate or reduce hardware performance counter measurements made in multiple tasks of an MPI application.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you use the hardware performance counter tools. In order to set these environment variables, you should run the setup scripts that are located in the top

level directory of your IBM HPC Toolkit installation. On AIX systems, these setup scripts are located in the **/usr/lpp/ppe.hpct** directory. On Linux, these setup scripts are located in the **/opt/ibmhpc/ppe.hpct** directory. If you are using **sh**, **bash**, **ksh**, or similar shell command, invoke the **env_sh** script as **. env_sh**. If you are using **csh**, invoke the **env_csh** script as **source env_csh**.

Using the hpccount Command

Hpccount is essentially used the same way as the time command; in the simplest invocation, the user types **hpccount** <*program*>**.** If you are using the **hpccount** command to measure the performance of a parallel program, you should invoke **hpccount** as **poe hpccount** <*program*>. If you invoke **hpccount** as **hpccount poe** <*program*>, you will measure the performance of the **poe** command, and not the performance of your application.

As a result, **hpccount** appends various performance information at the end of the screen (in other words, stdout) output. In particular it prints resource utilization statistics, hardware performance counter information and derived hardware metrics.

The resource usage statistics are directly taken from a call to **getrusage**(). For more information on the resource utilization statistics, refer to the **getrusage** man pages. In particular, the Linux man page for **getrusage**() states that not all fields are meaningful under Linux. The corresponding lines in **hpccount** output have the value ***n/a***.

If you specify the **–l** option, the **hpccount** command displays a list of the hardware performance counter groups that are available on the processor from which you invoked the **hpccount** command. If you specify the **–c** option, the **hpccount** command displays a list of the hardware counters that are available on the processor on which you run the **hpccount** command and the hardware performance counter events that can be counted in that counter.

If you specify the **–g** option, you can specify the hardware performance counter group from which you want to count events from. If you do not specify the **–g** option, the **hpccount** command uses a default hardware counter group as described in the flags section of the **hpccount** command man page. On AIX, if you specify a comma-separated list of hardware performance counter groups, **hpccount** multiplexes the use of the specified hardware performance counter groups in your application process. See the Hardware Counter Multiplexing section for more information.

If you are running a parallel application, you should use the **–u** option so that the output files generated by **hpccount** for each application task are assigned unique file names.

You can obtain derived metrics for your application by using the **–x** option. Derived metrics are additional performance metrics computed from the hardware performance counter measurements you collected. See the Derived Metrics section for more information.

If you plan to view the data obtained by running the **hpccount** command in **peekperf**, you should make sure that the **HPM_VIZ_OUTPUT** environment variable is set to **yes**.

Using the hpcstat Command

The **hpcstat** is a simple system-wide monitor that is based on hardware performance counters. For most of the functionality of **hpcstat**, root privileges are required. The usage is very similar to that of the vmstat command. You can invoke **hpcstat** as follows.

hpcstat

If you specify the **–l** option, the **hpcstat** command displays a list of the hardware performance counter groups that are available on the processor on which you invoked the command. If you specify the **–c** option, the **hpcstat** command will display a list of the hardware counters that are available on the processor on which you run the command and the hardware performance counter events that can be counted in that counter.

If you specify the **–g** option, you can specify the hardware performance counter group from which you want to count events. If you do not specify the **–g** option, the **hpcstat** command will use a default hardware counter group as described in the flags section of the **hpccount** command man page. On AIX, if you specify a comma-separated list of hardware performance counter groups, **hpcstat** multiplexes the use of the specified hardware performance counter groups in your application process. See the Hardware Counter Multiplexing section for more information.

The output of the **hpcstat** command is written to stdout and consists of resource utilization statistics, hardware performance counter information and derived hardware metrics.

The resource usage statistics are directly taken from a call to **getrusage**(). For more information on the resource utilization statistics, refer to the **getrusage** man pages. In particular, on Linux the man page for **getrusage**() states that not all fields are meaningful under Linux. The corresponding lines in hpcstat output have the value ***n/a***.

Using the libhpc Library

The **hpccount** command provides hardware performance counter information and derived hardware metrics for the whole program. If this information is required for only part of the program, instrumentation with the **libhpc** library is required. This library provides a programming interface to start and stop performance counting for an application program.

The **libhpc** library API includes the following function calls:

- **hpmInit** () for initializing the instrumentation library.

- **hpmTerminate** () for generating the reports and visualization data files and shutting down the **libhpc** environment.

- **hpmStart** () for identifying the start of a section of code in which hardware performance counter events will be counted.

- **hpmStop** () for identifying the end of the instrumented section.

The **libhpc** library provides variants of the **hpmStart** () and **hpmStop** () function calls, which you can use in threaded code, and where you need to explicitly identify parent/child relationships between nested instrumentation regions. **libhpc** implements both C and Fortran versions of each function call.

The **hpmStart**() and **hpmStop**() function calls, or their variants, must be executed in pairs, where for each **hpmStart**() function call, a corresponding **hpmStop**() function call must be executed for every time **hpmStart**() is executed.

The part of the application program between the start and stop of performance counting is called an instrumentation section. You assign a unique integer number as the section identifier. You specify this section identifier in the call to the **hpmStart**() function. A simple case of an instrumented program section might look similar to the following:

```
hpmInit( 0, “my program” );
hpmStart( 1, “outer call” );
do_work();
hpmStart( 2, “computing meaning of life” );
do_more_work();
hpmStop( 2 );
hpmStop( 1 );
hpmTerminate( taskID );
```

Calls to **hpmInit**() and **hpmTerminate**() embrace the instrumented part. Every instrumentation section starts with **hpmStart**() and ends with **hpmStop**(). The section identifier is the first parameter to the latter two functions. As shown in the example, **libhpc** supports multiple instrumentation sections and overlapping instrumentation sections. Each instrumented section can also be called multiple times. When **hpmTerminate**() is encountered, the counted values are collected and printed or written to visualization data files.

The example program above provides an example of two properly nested instrumentation sections. For section 1 we can consider the *exclusive* time and *exclusive* counter values. By that we mean the difference of the values for section 1 and section 2. The original values for section 1 would be called *inclusive* for matter of distinction. The terms *inclusive* and *inclusive* for the embracing instrumentation section are chosen to indicate whether counter values and times for the contained sections are included or excluded. For more details see the Inclusive and Exclusive Event Counts section.

Any C source file containing calls to any function contained in the **libhpc** library should include the **libhpc.h** header. Fortran source files containing calls to functions in the **libhpc** library should include the **f_hpc.h** header. Or, if the Fortran source file is compiled using the **–qintsize=8** compiler option, it should include the **f_hpc_i8.h** header file. All of these header files are located in the $(IHPCT_BASE)/include directory.

Fortran source files that include either the **f_hpc.h** or **f_hpc_i8.h** header file should also be processed by the C pre-processor before compilation, for instance by specifying the **–qsuffix=cpp=f** compiler option.

You must link your application with the **libhpc** library, using the **–lhpc** linker option. You must also link your application with the **libm** library, using the **–lm** linker option. On AIX systems, you must also link with the **PMAPI** library, using the **–lpmapi** linker option. On Linux systems, you must also link with the **perfctr** library using the linker option **–lperfctr**. When using the **libhpc** library, compile and link your application as a threaded program (for instance, using the **xlc_r** or **xlf_r** commands), or link with the **pthreads** library using the **-lpthreads** linker option. When linking **libhpc** with your application, you must specify the correct library, using either the **–L$(IHPCT_BASE)/lib** or **–L$(IHPCT_BASE)/lib64** linker option.

Like the **hpccount** command, you can obtain hardware performance counter measurements using multiple hardware performance groups as described in the Hardware Counter Multiplexing section. You can use derived metrics as described in the Derived Metrics section. If you are instrumenting an MPI program, you can use plugins to aggregate performance data from multiple tasks or to filter that data. See the Considerations for MPI Programs section for more detail about these plugins.

Note that **libhpc** collects information and performs summarization during runtime. Thus, there could be a considerable overhead if instrumentation sections are inserted inside inner loops. The **libhpc** library uses the same set of hardware performance counter groups used by **hpccount**.

If an error occurs internally in **libhpc**, the program is not automatically terminated. Instead, the **libhpc** library sets an error indicator and lets the user handle the error. For details see the **hpm_error_count** man page.

Understanding Hardware Counter Multiplexing

The idea behind multiplexing is to run several hardware counter groups concurrently. This is accomplished by running the first group for a short time interval, then switching to the next group for the next short time interval. This is repeated in a round robin fashion for the groups until the event counting is eventually stopped.

HPM (Hardware Performance Monitoring) supports multiplexing only on AIX. The hardware counter groups are specified as a comma separated list. If you are using the **hpccount** or **hpcstat** commands, then you can specify the multiplexed hardware performance counter groups one of two ways, for instance

hpmcount -g 1,2 <*program*>
OR
export HPM_EVENT_SET='1,2'
hpmcount <*program*>

If you are running an application program that has been compiled and linked with **libhpc**, specify the multiplexed hardware counter groups by setting the **HPM_EVENT_SET** environment variable similar before running your application. For example:

export HPM_EVENT_SET='1,2'

On Linux, this leads to the following error message.

HPM ERROR - Multiplexing not supported: too many groups or events specified:

Multiplexing means that none of the specified groups has been run on the whole code, and it is unknown what fraction of the code was measured with which group. It is assumed that the workload is sufficiently uniform that the measured event counts can be (more or less) safely calibrated as if the groups have been run separately on the whole code.

The default time interval for measuring one group on the application is 100 milliseconds. If the total execution time is shorter than this time interval, only the first group in the list is measured. All other counter values from the other hardware counter groups are 0 in this case. This might result in NaNQ values for some derived metrics (see Derived Metrics), if the formula for computing the derived metric requires division by the counter which has a zero value.

The duration of this time interval can be controlled by setting the following environment variable.

export HPM_SLICE_DURATION=<*integer value in ms*>

This value is passed directly to AIX (more precisely **bos.pmapi** component of AIX). AIX requires the value to be between 10 milliseconds and 30 seconds.

The data for each group is printed in a separate section, with separate timing information, and is written separately to the visualization data files used as input to **peekperf**.

For MPI applications, the form of the output depends on the chosen aggregation plug-in as described in the section Considerations for MPI Programs. Without specifying an

aggregator plug-in (in other words, with the default plug-in), the data for each hardware performance counter group is printed in a separate section with separate timing information for each hardware performance counter group. To combine the data from the specified groups in to one big group with more counters, use the local merge aggregator plug-in (**loc_merge.so**), which is described in the section Plug-ins Shipped with the Tookit.

Understanding Derived Metrics

What are Derived Metrics

Some of the hardware counter events are difficult to interpret. Sometimes a combination of events provides better information. Such a recombination of basic events will be called a derived metric. HPM also provides a list of derived metrics, which are defined in section Derived Metrics Description below.

Since each derived metric has its own set of ingredients, not all derived metrics are printed for each group. HPM automatically finds those derived metrics that are computable and prints them. As a convenience to the user, the option **-x** will print the value of the derived metric and its definition. If the HPM_PRINT_FORMULA environment variable is set to **yes,** derived metric formulas are also printed.

Understanding MFlop Issues

The two most popular derived metrics are the MFlop/s rate and the percentage of peak performance. The default group is chosen to make those two derived metrics available without explicitly specifying the **-g** option.

For IBM POWER5™ servers, there is no group that supports enough counters to compute a MFlop/s rate. As a result, an “Algebraic MFlop/s rate” was invented to bridge this gap. This derived metric counts floating point additions, subtractions and multiplictaions (including floating point multiply and add instructions), but misses divides and square roots. If the latter two only occur in negligible numbers (which is desirable for an HPC code anyway), the *Algebraic MFlop/s rate* coincides with the usual definition of MFlop/s. Again, different counter groups can be used to check this hypothesis. Group 137 (which exploits the *Algebraic MFlop/s rate*) is the chosen default group for POWER5.

For PowerPC® 970, POWER4™ and POWER5, the derived metric *percent of peak performance* is based on user time rather than wall clock time, as the result stays correct if multithreaded (e.g. OpenMP) applications are run. For POWER5, this is based on “Algebraic MFlop/s”. This has not been tested on AIX 5.3, particularly when Simultaneous Multithreading (SMT) is active.

On POWER4, PowerPC 970 and POWER6™, weighted MFlop/s are available. These are like ordinary MFlop/s, except that divisions enter the evaluation with a weight different from the other floating point operations. The weight factor is provided by the user through the environment variable HPM_DIV_WEIGHT. If set to **1**, the weighted

MFlop/s coincide with the ordinary MFlop/s. HPM_DIV_WEIGHT can take any positive integer number.

If this environment variable is not set, no weighted MFlop/s metrics are computed.

Understanding Inheritance

On both, AIX and Linux, the *counter virtualization* and the group (in other words, set of events) that is actually monitored is inherited from the process by any of its children. Children in this context mean threads or processes spawned by the parent process. AIX and Linux, however differ in the ability of the parent process to access the counter values of its children.

- On AIX, all counter values of a process group can be collected.
- On Linux, counter values are only available to the parent if the child has exited.

The **hpccount** utility makes use of this inheritance. If **hpccount** is called for a program, the returned counter values are the sum of the counter values of the program and all of the threads and processes spawned by it at the time the values are collected. For Linux, this has to be restricted to the sum of counter values of all children that have finished at the time the values are collected. Even the latter is enough to catch the values of all threads of an OpenMP program.

Assume you are using a small program, named *taskset*, to bind threads to CPUs. When **hpccount** is invoked to run *taskset* as follows

hpccount taskset -g <*num*> <*program_name*>

hpccount would first enable hardware event counting for the application *taskset*. This command then spawns the program<*program name*>, which inherits all hardware counter settings. At the end **hpccount** would print counter values (and derived metrics) based on the sum of events for *taskset* and the called program. Since *taskset* is a very small application, the **hpccount** results would mostly represent the performance of the program <*program name*>, which is what the user intended.

Understanding Inclusive and Exclusive Event Counts

For an example of an application fragment, where the term *exclusive values* applies, see the Using the **libhpc** Library section above. That application fragment provides an example of two properly nested instrumentation sections. For section 1, exclusive time and exclusive counter values are the difference between the values for section 1 and section 2, or excluding the counts of events within the scope of section 2. The original values for section 1 would be called inclusive values since those values also include the count of events that occurred within the scope of section 2. The terms inclusive and exclusive for the enclosing instrumentation section are chosen to indicate whether counter values and times for the contained sections are included or excluded.

The extra computation of exclusive values generates overhead that is not always wanted. Therefore, the computation of exclusive values is only carried out if the environment

variable HPM_EXCLUSIVE_VALUES is set to **yes** or if the HPM_ONLY_EXCLUSIVE parameter is used as described in the Parent-Child Relationships section. The exact definition of *exclusive* is based on parent-child relationships among the instrumented sections. Roughly spoken, the exclusive value for the parent is derived from the inclusive value of the parent reduced by the inclusive value of all children.

Understanding Parent-Child Relationships

The IBM HPC Toolkit provides an automatic search for parents, which is supposed to closely mimic the behavior for strictly nested instrumented regions. For strictly nested instrumented sections, the call to **hpmStart**() or **hpmTstart**() for the parent must occur prior to the corresponding call for the child. In a multithreaded environment, however, this causes problems if the children are executed on different threads. In a kind of race condition, a child might mistake its brother for its father. This generates flawed parent child relationships, which change with every execution of the program. To avoid the race condition safely, the search for a parent region is restricted to calls from the same thread only, because only these exhibit a race condition free call history. The parent region found in this history is the last call of the same kind (in other words, both were started with **hpmStart**() or both were started with **hpmTstart**() or their corresponding Fortran equivalents) that has not posted a matching **hpmStop**() or **hpmTstop**() meanwhile. If no parent is found that matches these rules, the child is declared an orphan. Therefore, automatic parent child relations are never established across different threads.

There might be situations in which the automatic parent child relations prove unsatisfactory. To help this matter, calls are provided in the HPM API to enable you to establish the relations of your choice. These functions are **hpmStartx**() and **hpmTstartx**() and their Fortran equivalents. The first two parameters of this function are the ID of the instrumented section and the ID of the parent instrumented section.

The user has the following choices for the parent ID.

- **HPM _AUTO_PARENT:** This triggers the automatic search and is equivalent to the **hpmStart**() and **hpmTstart**() functions
- **HPM_ONLY_EXCLUSIVE:** This is essentially the same as **HPM_AUTO_PARENT**, but sets the exclusive flag to **true** on this instance only. The environment variable **HPM_EXCLUSIVE_VALUES** sets this flag globally for all instrumented sections.
- **HPM_NO_PARENT:** This suppresses any parent child relations.
- An integer: This must be the ID of an instrumented section with the following restrictions:
  - It has to active when this call to **hpmStartx**() or **hpmTstartx**() is made.

- It has to be of the same kind (in other words, both were started with **hpmStart**() or both were started with **hpmTstart**() or their corresponding Fortran equivalences)

Handling of Overlap Issues

As you can establish almost arbitrary parent child relationships, the definition of the explicit duration or explicit counter values is not obvious.

Each instrumented section can be represented by the corresponding subset of the time line of application execution. Actually this subset is a finite union of intervals with the left or lower boundaries marked by calls to **hpmStart[x]**/**hpmTstart[x]**() and the right or upper boundaries marked by calls to **hpmStop**()/**hpmTstop**(). The duration is the accumulated length of this union of intervals. The counter values are the number of those events that occur within this subset of time.

The exclusive times and values are the times and values when no child has a concurrent instrumented section. Hence, the main step in defining the meaning of exclusive values is defining the subset of the time line with which they are associated. This is done in several steps:

- Represent the parent and every child by the corresponding subset of the time line (henceforth called the parent set and the child sets).
- Take the union of the child sets.
- Reduce the parent set by the portion that is overlapping with this union.
- Take the difference of the parent set with the union of the child sets, using set theoretic terms.

The exclusive duration is the accumulated length of the resulting union of intervals. The exclusive counter values are the number of those events that occur within this subset of time.

Understanding Measurement Overhead

Instrumentation overhead is caught by calls to the wall clock timer at entry and exit of calls to **hpmStart[x]**(), **hpmStop**(), **hpmTstart[x]**(), **hpmTstop**(). The accumulated instrumentation overhead for each instrumented section is printed in the ASCII output (***.hpm**) file.

Based on the magnitude of the overhead, you can decide what to do with this information.

- If the overhead is several orders of magnitude smaller than the total duration of the instrumented section, you can safely ignore the overhead timing.
- If the overhead is the same order of magnitude as the total duration of the instrumented section, the results might be inaccurate since the instrumentation overhead is a large part of the collected event counts.
- If the overhead is within 20% of the measured wall clock time, a warning is printed to the ASCII output file.

To make the use of **libhpc** thread safe, mutexes are set around each call to **hpmStart[x]**(), **hpmStop**(), **hpmTstart[x]**(), **hpmTstop**(), which adds to the measurement overhead. If your application is running on one thread only, the setting of the mutexes can be suppressed by setting the environment variable HPM_USE_PTHREAD_MUTEX to **no**. Results are unpredictable if your program is using multiple threads and you set this environment variable.

Handling Multithreaded Program Instrumentation Issues

When placing instrumentation inside of parallel regions, one should use different ID numbers 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
```

If two threads are using the same ID numbers when calling **hpmTstart**() or **hpmTstop**(), **libhpc** exits with the following error message:

libhpc ERROR - Instance ID on wrong thread

If you place instrumentation calls in parallel loops or parallel regions, use the **hpmTstart**() and **hpmTstop**() function calls. If you use **hpmStart**() and **hpmStop**() function calls, **libhpc** attempts to count hardware performance counter events on all threads in the application (AIX only). Since application threads are not necessarily all executing the same code, the event counts obtained by calls to **hpmStart**() and **hpmStop**() might not be accurate.

Considerations for MPI Programs

General Considerations

The **libhpc** library is inherently sequential, looking only at the hardware performance counters of a single process (and its children, as explained in the Inheritance section). When the application is started, each MPI task is doing its own hardware performance counting and these instances are completely ignorant of each other, unless additional action is taken as described in the following subsections. Consequently, each instance is writing its own output. If the environment variable **HPM_OUTPUT_NAME** is used, each instance is using the same file name, which results in writing in to the same file, if a

parallel file system is used. This can be prevented by making the file names unique through the **HPM_UNIQUE_FILE_NAME** environment variable. However, it might be an unwanted side effect to produce one output file for each MPI task.

For this reason, the environment variable **HPM_AGGREGATE** triggers some aggregation before (possibly) restricting the output to a subset of MPI tasks. The environment variable **HPM_AGGREGATE** takes a value, which is the name of a plug-in that defines the aggregation strategy. Each plug-in is a shared object file containing two functions called **distributor** and **aggregator**.

Understanding Distributor Functions

The motivating example for the distributor function is allowing a different hardware counter group on each MPI task. Therefore, the distributor is a subroutine that determines the MPI task id (or MPI rank within **MPI_COMM_WORLD**) from the MPI environment for the current process, and sets or resets environment variables depending on this information. The environment variable can be any environment variable, not just the **HPM_EVENT_SET** environment variable, which specifies the hardware performance counter group.

The **distributor** function is called before any environment variable is evaluated by HPM. The settings of the environment variables done in the distributor take precedence over global environment variable settings.

The aggregator must adapt to the HPM group settings done by the distributor. This is why distributors and aggregators always come in pairs. Each plug-in contains a distributor and aggregator pair.

Understanding Aggregator Functions

The motivating example is the aggregation of the hardware performance counter data across MPI tasks. In the simplest case this could be an average of the corresponding values. Hence this function is called

- after the hardware performance counter data has been gathered,
- before the derived metrics are computed.
- before these data are printed.

In general view, the aggregator takes the raw results and rearranges them for output.

Also, depending on the MPI task rank the aggregator sets (or does not set) a flag to mark the current MPI task for HPM printing.

Plug-ins Shipped with the Tool Kit

The following plug-ins are shipped with the IBM HPC Toolkit. They can be found in **$(IHPCT_BASE)/lib** for 32-bit applications or **$(IHPCT_BASE)/lib64** for 64-bit applications.

- **mirror.so** is the plug-in that is called when no plug-in is requested. The aggregator mirrors the raw hardware performance counter data in a one-to-one fashion to the output function. It also flags each MPI task as a printing task. The corresponding distributor is an empty function. This plug-in does not use MPI and also works in a non-MPI context.
- **loc_merge.so** does a local merge on each MPI task separately. It is identical to the mirror.so plug-in except for those MPI tasks that change the hardware performance counter groups in the course of the measurement (e.g. by multiplexing). The different counter data, which are collected for only part of the measuring interval, are proportionally extended to the whole interval and joined in to one big group that is used for derived metrics computation. This way, more derived metrics can be determined at the risk of computing invalid metrics. The user is responsible for using this plug-in only when it makes sense to use it. It also flags each MPI task as a printing task. The corresponding distributor is an empty function. This plug-in does not use MPI and also works in a -MPI context.
- **single.so** works the same as the **mirror.so** plug-in, but only on MPI task 0. The output on all other tasks is discarded. This plug-in uses MPI functions and cannot be used in a sequential context.
- **average.so** is a plug-in for taking averages across MPI tasks. The distributor reads the environment variable **HPM_EVENT_SET** (which should be a comma separated list of hardware performance counter group numbers) and distributes these group numbers in a round robin fashion to the MPI tasks in the application. The aggregator function creates an MPI communicator of all tasks with equal hardware performance counter group specifications. The communicator groups might be different from the original round robin distribution. This could happen if the counting group has been changed on some of the MPI tasks after the first setting by the distributor function. Next, the aggregator computes the average for each hardware performance counter event across the subgroups formed by this communicator. Finally, it flags the MPI rank 0 in each group as a printing host. This plug-in uses MPI functions and cannot be used in a sequential context.

Why User defined Plug-ins are Useful

This set of plug-ins is a starter kit and many more plug-ins might be desirable. Rather than taking the average of hardware performance counters across a set of MPI tasks, you could compute minimum or maximum values. You could also create a kind of a *history merge.so* by blending in results from previous measurements. You can write your own plug-ins using the interface described in the next section.

The source code for the supplied plug-ins is provided for you to use as examples in developing your own plug-ins. The source files and makefiles for the plug-ins are located in the $IHPCT_BASE/examples/plugins directory.

Understanding the Distributor and Aggregator Interfaces

Each distributor and aggregator is a function returning an integer which equal to zero on success and not equal to zero on error. In most cases the errors occur when calling a system call like **malloc**(), which sets the **errno** variable. If the distributor or aggregator

returns the value of **errno** as the return code, the calling HPM tool can use the **errno** to display a meaningful error message. If returning **errno** is not viable, the function should return a negative value.

The function prototypes are defined in the following file.
$(IHPCT_BASE)/include/hpm_agg.h

This is a very short file with the following contents.

```
#include "hpm_data.h"
int distributor(void);
int aggregator(int num_in, hpm_event_vector in,
               int *num_out, hpm_event_vector *out,
               int *is_print_task);
```

The distributor function has no parameters and is only required to set or reset environment variables (using **setenv**()) if necessary for correct operation of the aggregator function.

The aggregator function takes the current hardware performance counter values on each task as an input vector **in** and returns the aggregated values on the output vector **out** on selected or all MPI tasks. The aggregator is responsible for allocating the memory needed to hold the output vector **out**. The definition of the data types used for in and out are provided in the header file **$(IHPCT_BASE)/include/hpm_data.h**

Finally the aggregator function must set (or reset) the flag, **is_print_task** to mark the current MPI task for HPM printing.

The hpm_event_vector **in** is a vector or list of **num_in** entries of type **hpm_data_item**. This data type is a struct containing members that describe the definition and the results of a single hardware performance counting task.

```
/* NAME                    INDEX */
#define HPM_NTIM           8
#define HPM_TIME_WALLCLOCK 0
#define HPM_TIME_CYCLE     1
#define HPM_TIME_USER      2
#define HPM_TIME_SYSTEM    3
#define HPM_TIME_START     4
#define HPM_TIME_STOP      5
#define HPM_TIME_OVERHEAD  6
#define HPM_TIME_INIT      7
typedef struct {
  int num_data;
  hpm_event_info *data;
  double times[HPM_NTIM];
  int is_mplex_cont;
  int is_rusage;
```

```
  int mpi_task_id;
  int instr_id;
  int count;
  int is_exclusive;
  int xml_element_id;
  char *description;
  char *xml_descr;
} hpm_data_item;
typedef hpm_data_item *hpm_event_vector;
```

- The first element of the in vector contains the data from a call to **getrusage**(). This vector element is the only element with its structure member **is_rusage** set to **true** to distinguish it from ordinary hardware performance counter data
- The count of events from an individual hardware performance counter group on one MPI task is contained in a single element of type hpm_data_item.
- If multiplexing is used, the results span several consecutive elements, each dedicated to one hardware performance counter group that takes part in the multiplex setting. On all but the first element, the member **is_mplex_cont** is set to **true** to indicate that these elements are continuations of the first element belonging to the same multiplex setup.
- If hardware performance counter groups are changed during the measurement, the results for different groups are recorded in different vector elements, but the **is_mplex_cont** flag is not set. This way results obtained using multiplexing can be distinguished from results obtained by an ordinary hardware performance counter group change.
- If several instrumented sections are used, each instrumented code section uses separate elements of type **hpm_data_item** to record the results. Each of these elements will have the member **instr_id** set to the value of the first argument of **hpmStart**() and the logical member **is_exclusive** set to **true** or **false** depending on whether the element hold inclusive or exclusive counter results as described in the section Inclusive and Exclusive Event Counts. Then all these different elements are concatenated in to a single vector.

The output vector is of the same format. Each vector element is used in the derived metrics computation separately (unless **is rusage** is equal to true). Then all vector elements and the corresponding derived metrics are printed in the order given by the vector **out**. The output of each vector element is preceded by the string pointed to by structure member **description** (which might include line feeds, as appropriate). The XML output will be labeled with the text pointed to by **xml_descr**. This way the input vector **in** is providing a complete picture of what has been measured on each MPI task. The output vector **out** is allowing complete control on what is printed on which MPI task in what order.

Getting the plug-ins to work

The sample plug-ins are compiled with the Makefile **$(IHPCT_BASE)/examples/plugins/Makefile** using the command

make ARCH=<*arch*>**.**

The include files for the supported architectures (arch) are located in the subdirectory **make**. Note the following considerations when implementing your own plug-in.

- The Makefile distinguishes sequential (specified in **PLUGIN_SRC**) and parallel plug-ins (specified in **PLUGIN_PAR_SRC**). The latter are compiled and linked with the MPI wrapper script for the compiler and linker. Unlike a static library, generation of a shared object requires linking, not just compilation.
- There are some restrictions to be observed when writing plug- in code.
  - The MPI standard document disallows calling **MPI_Init()** twice in the same process.
  - The distributor function is called by **hpmInit**(). If the distributor function contains MPI calls, the user's application is required to call MPI Init() prior to calling **hpmInit**(). To avoid this restriction, the distributor function must not call any MPI function. The MPI task ID should be extracted by inspecting environment variables, specifically the **MP_CHILD** environment variable, that have been set by the MPI software stack.
  - The aggregator function, however, usually cannot avoid calling MPI functions. Before calling **MPI_Init**(), it has to check whether the instrumented application has already done so, for example, by calling the MPI_**Initialized**() function. If the instrumented application is an MPI application, the aggregator function cannot be called after **MPI_Finalize**(). The aggregator function is called by **hpmTerminate**(). Hence **hpmTerminate**() has to be called between the calls to **MPI Init**() and **MPI Finalize**().
- **libhpc** uses a call to dlopen() to access the plug-in and makes use of its functions.

Specifying Latency Estimates

In addition, users can provide estimations of memory, cache, and TLB miss latencies for the computation of derived metrics, with the following environment variables. Note that not all flags are valid in all systems.

- **HPM_MEM_LATENCY** latency for a memory load.
- **HPM_L3_LATENCY** latency for an L3 load within an MCM.
- **HPM_L35_LATENCY** latency for an L3 load outside of the MCM.
- **HPM_L2_LATENCY** latency for an L2 load from the processor.
- **HPM_L25_LATENCY** latency for an L2 load from the same MCM.
- **HPM_L275_LATENCY** latency for an L2 load from another MCM.
- **HPM_TLB_LATENCY** latency for a TLB miss.

Using the MPI Profiling Library

The MPI profiling library, **libmpitrace**, is a library that you can link with your MPI application to profile the MPI function calls in your application, or to create a trace of those MPI calls. When you link your application with this library, the library intercepts the MPI calls in your application, using the Profiled MPI (PMPI) interface defined by the MPI standard, and obtains the profiling and trace information it needs. This library also

provides a set of functions that you can use to control how profiling and trace data is collected, as well as functions that you can use to customize the trace data.

Although the **libmpitrace** library can be used in a threaded application, it does not correctly record MPI trace events in an application in which MPI function calls are made on multiple threads. You should use libmpitrace only in single threaded applications or applications in which MPI function calls are made only on a single thread.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you use the MPI profiling library. In order to set these environment variables, run the setup scripts that are located in the top level directory of your IBM HPC Toolkit installation. On AIX systems, these setup scripts are located in the **/usr/lpp/ppe.hpct** directory. On Linux, these setup scripts are located in the **/opt/ibmhpc/ppe.hpct** directory. If you are using **sh**, **bash**, **ksh**, or similar shell command, invoke the **env_sh** script as **. env_sh**. If you are using **csh**, invoke the **env_csh** script as **source env_csh**.

When you compile your application, you must use the **–g** compiler flag so that the library can obtain the information it needs to map performance information back to application source code. You might want to consider compiling your application at lower optimization levels since compiler optimizations might affect the accuracy of mapping MPI function calls back to source code and the accuracy of the function call stack for an MPI function call.

Any C source file containing calls to functions in the libmpitrace library should include the **mpt.h** header file, which is located in the **$(IHPCT_BASE)/include** directory. All applications must link the **libmpitrace** library with the application using the **–lmpitrace** linker flag. The **libmpitrace** library is located in **$(IHPCT_BASE)/lib** for 32-bit applications and in **$(IHPCT_BASE)/lib64** for 64-bit applications.

After you link your application with the **libmpitrace** library, you can run your application just as you normally would.

By default, the **libmpitrace** library generates three sets of output files in the current working directory. The first set is named **mpi_profile.*task_rank*** where *task_rank* is the MPI task rank of the task that generated the file. The second set of output files is named **mpi_profile_*task_rank*.viz** which contain the visualization data that can be viewed using **peekperf**. The third output is a file named **single_trace** which is a trace file containing trace data that can be viewed using **peekperf**.

Controlling Profiling and Tracing

Controlling Traced Tasks

The libmpitrace library stores MPI trace events in memory for performance reasons. By default, the number of MPI trace events that are recorded is 30,000 events. Additional MPI trace events beyond this number are discarded. You can override this default by

setting the MAX_TRACE_EVENTS environment variable to the maximum number of MPI trace events to be recorded. Increasing this value means that additional memory will be used to store MPI trace events and that additional memory usage might affect your application program.

By default, for scalability, profiling data files and MPI function call events are generated for a maximum of four tasks in the application:

- task 0
- The task with the minimum MPI communication time
- The task with the maximum MPI communication time
- The task with the median MPI communication time.

If task 0 is the task with minimum, maximum, or median MPI communication time, at most, output files will be generated for only three tasks. If you want output to be generated for all MPI tasks, set the **OUTPUT_ALL_RANKS** environment variable to **yes** before running the application.

By default, the **libmpitrace** library traces MPI tasks 0 through 255 (or less if the application has fewer than 256 MPI tasks). If you need to see MPI traces from all tasks, you must set the **TRACE_ALL_TASKS** environment to **yes** before running the application. If you have an application with more than 256 MPI tasks, but you do not want to see traces from all MPI tasks, you can set the **MAX_TRACE_RANK** to the MPI task index of the highest numbered MPI task that you want traced.

By default, when libmpitrace obtains the calling address for each MPI function that is traced, it gets the address of the MPI function's immediate caller. If MPI functions are called from within another library, or deeply layered within your application, you might want **libmpitrace** to obtain the MPI function caller's address from one or more layers higher in the function call stack. You can specify how many levels to walk back in the function call stack by setting the **TRACEBACK_LEVEL** environment level to the number of levels to walk, where 0 means to obtain the address where the MPI function was actually called.

Additional Trace Controls

You can obtain additional control over MPI trace generation by using function calls in the **libmpitrace** library. You can trace selected sections of your application by bracketing areas of interest with calls to the **MT_trace_start**() and **MT_trace_stop**() functions. In order to use these functions, you must set the **TRACE_ALL_EVENTS** environment variable to **no** before running your application. When you start your application, tracing is initially suspended. When your application invokes the **MT_trace_start**() function, MPI trace event collection is resumed in the task where **MT_trace_start**() was called. Tracing continues until the **MT_stop_trace()** function is called. At that time, MPI trace event collection is suspended in the task that called **MT_trace_stop**(). Tracing can be resumed again by a subsequent call to the **MT_trace_start**() function.

The **MT_trace_start**() and **MT_trace_stop**() functions can be called from C applications. Fortran applications can call the **mt_trace_start**() and **mt_trace_stop**() functions.

You can control which MPI function calls are traced by implementing your own version of the **MT_trace_event**() function. The C function prototype for this function is

int MT_trace_event(int id);

where **id** is an enumeration identifying the specific MPI function that is being executed. You should include the mpi_trace_ids.h header, located in the $(IHPCT_BASE)/include directory, when you implement this function.

Your implementation of **MT_trace_event**() must return **1** if the MPI trace event should be recorded, and must return **0** if the MPI trace event should not be recorded.

You can control which MPI tasks should have MPI trace events recorded by implementing your own version of the **MT_output_trace**() function. The C function prototype for this function is

int MT_output_trace(int task);

where **task** is the MPI task ID of the task calling this function. Your implementation of this function must return **1** if the MPI trace event is to be recorded and return **0** if the MPI trace event is not to be recorded.

Customizing MPI Profiling Data

You can create customized MPI profiling data by implementing your own version of the **MT_output_text**() function. The C function prototype for this function is

int MT_output_text(void);

This function is called for each MPI task when that task calls **Finalize()**. If you implement your own version of the **MT_output_text**() function, you are responsible for generating all profiling data, in whatever format you require. You might use any of the functions described in the MPI Profiling Utility Functions section, in your implementation of the **MT_output_text**() function.

Your implementation of this function should return **1** if it successfully completes and return **-1** if an error occurs in processing.

Understanding MPI Profiling Utility Functions

The **libmpitrace** library provides a set of functions that you can use to obtain information about the execution of your application. You can use these functions when implementing your own versions of **MT_trace_event**(), **MT_output_trace**(), **MT_output_text**(), or anywhere else, including your own application code where they are useful.

There are several functions you can use to obtain information about types of MPI functions called in your application as described in the following table.

Function	Purpose
MT_get_mpi_counts	Determine how many times an MPI function is called.
MT_get_mpi_bytes	Determine total number of bytes transferred by all calls to a specific MPI function.
MT_get_mpi_time	Determine cumulative time spent in all calls to a specific MPI function
MT_get_mpi_name	Obtain the name of an MPI function given the internal ID used by the IBM HPC Toolkit to reference this MPI function
MT_get_time	Determine the elapsed

	time since MPI_Init() was called
MT_get_elapsed_time	Determine the elapsed time between calls to MPI_Init and MPI_Finalize
MT_get_tracebufferinfo	Determine the size and current usage of the internal MPI trace buffer used by the IBM HPC Toolkit
MT_get_calleraddress	Determine the address of the caller of a currently active MPI function
MT_get_callerinfo	Determine source file and line number information for an MPI function call, using

	the address obtained by calling MT_get_calleraddress
MT_get_environment	Obtain information about the MPI execution environment
MT_get_allresults	Obtain statistical information about a specific MPI function call

All of these functions are documented in the MPI Profiling section.

Using the I/O Profiling Library

Preparing Your Application

The IBM HPC Toolkit provides a library that you can use to profile I/O calls in your application.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you use the I/O profiling library. In order to set these environment variables, run the setup scripts that are located in the top level directory of your installation. On AIX systems, these setup scripts are located in the /usr/lpp/ppe.hpct directory. On Linux, these setup scripts are located in the /opt/ibmhpc/ppe.hpct directory. If you are using sh, bash, ksh, or similar shell command, you should invoke the env_sh script as **. env_sh**. If you are using csh, you should invoke the env_csh script as **source env_csh**.

In order to profile your application, you must link your application with the libhpctkio library using the **–L$IHPCT_BASE/lib** and **–lhpctkio** linker options for 32-bit applications or using the **–L$IHPCT_BASE/lib64** and **–lhpctkio** linker options for 64-bit applications.

You must also set the TKIO_ALTLIB environment variable to the pathname of an interface module used by the I/O profiling library before you invoke your application. For 32-bit applications, the TKIO_ALTLIB environment variable should be set to **$IHPCT_BASE/lib/get_hpcmio_ptrs.so**. For 64-bit applications, the TKIO_ALTLIB environment variable should be set to **$IHPCT_BASE/lib64/get_hpcmio_ptrs.so.** Optionally, the I/O profiling library can print messages when the interface module is loaded, and it can abort your application if the interface module cannot be loaded.

In order for the I/O profiling library to display a message when the interface module is loaded, you must append **/print** to the setting of the TKIO_ALTLIB environment variable. In order for the IO profiling library to abort your application if the interface module cannot be loaded, you must append **/abort** to the setting of the TKIO_ALTLIB environment variable. You might specify one, both, or none of these options.

Note that there are no spaces between the interface library pathname and the options. For instance, to load the interface library for a 32-bit application, display a message when the interface library is loaded, and abort the application if the interface library cannot be loaded, you would issue the following command:
export TKIO_ALTLIB="$IHPCT_BASE/lib/get_hpcmio_ptrs.so/print/abort"

Setting I/O Profiling Environment Variables

There are three environment variables that the I/O profiling library uses to determine the files for which I/O profiling is to be performed and the data that will be obtained by profiling. These environment variables should be set, as needed, before you run your application.

The first environment variable is MIO_FILES, which specifies one or more sets of file name and the profiling library options to be applied to that file, where the file name might be a pattern or an actual path name.

The second environment variable is MIO_DEFAULTS, which specifies the I/O profiling options to be applied to any file whose file name does not match any of the file name patterns specified in the MIO_FILES environment variable. If MIO_DEFAULTS is not set, no default actions are performed.

The third environment variable is MIO_STATS, which specifies where the output from the I/O profiling data will be written. If MIO_STATS is set to **stdout** or **stderr**, the data is written to the corresponding file descriptor. If MIO_STATS is set to a file name, the data is written to the file name. If the first character of the file name is a + character, the data is appended to the specified file. If the first character of the file name is not +, any existing file with that name is overwritten. If this environment variable is not set, data is written to a file named MIO_STATS in the application's current directory.

The file name that is specified in the MIO_FILES variable setting might be a simple file name specification, which is used as-is, or it might contain wildcard characters, where the allowed wildcard characters are:

- A single asterisk (*), which matches zero or more characters of a file name.
- A question mark (?), which matches a single character in a file name.
- Two asterisks (**), which match all remaining characters of a file name.

The I/O profiling library contains a set of modules that can be used to profile your application and to tune I/O performance. Each module is associated with a set of options. Options for a module are specified in a list, and are delimited by / characters. If an option requires a string argument, that argument should be enclosed in curly braces ({}), if the argument string contains a / character.

Multiple modules can be specified in the settings for both MIO_DEFAULTS and MIO_FILES. For MIO_FILES, module specifications are delimited by commas (,). For MIO_DEFAULTS, module specifications are delimited by pipe (|) characters.

Multiple file names and file name patterns can be associated with a set of module specifications in the MIO_FILES environment variable. Individual file names and file name patterns are delimited by colon (:) characters. Module specifications associated with a set of file names and file name patterns follow the set of file names and file name patterns, and are enclosed in square brackets ([]).

As an example of the MIO_DEFAULTS environment variable setting, assume that the default options for any file that does not match the file names or patterns specified in the MIO_FILES environment variable are that the trace module is to be used with the **stats** and **mbytes** options and that the pf module is also to be used with the **stats** and **mbytes** options. The setting of the MIO_DEFAULTS environment variable would be
export MIO_DEFAULTS="trace/stats=mio.stats/mbytes,pf/stats=mio.stats/mbytes"

As an example of the MIO_FILES environment variable setting, assume that file **/tmp/testdata** will use the trace module with the events setting and that any files matching the patterns ***.txt** or ***.dat** will use the trace module with the **stats** option and the pf module with the **stats** and **mbytes** options. The setting of MIO_FILES environment variable would be:

```
export MIO_FILES="/tmp/testdata [trace/events={/tmp/events}] \
        *.txt : *.dat [trace/stats={/tmp/stats}|pf/stats={/tmp/stats}/mbytes]"
```

Specifying I/O Profiling Library Module Options

The following modules are available in the I/O profiling library

mio	The interface to the user program
pf	A data prefetching module
trace	A statistics gathering module
recov	Analyzes failed I/O accesses and

	retries in case of failure

The mio module has the following options

mode=	Override the file access mode in the open system call.
nomode	Do not override the file access mode.
direct	Set the O_DIRECT bit in the open system call.
nodirect	Clear the O_DIRECT bit in the open system call.

The default option for the mio module is **nomode.**

The pf module has the following options

norelease	Do not free the global cache pages when the global cache file usage count goes to zero. The **release** and **norelease** options control what happens to a global cache when the file usage count goes to zero. The default behavior is to close and release the global cache. If a global cache is opened and closed multiple times, there could be memory fragmentation issues at some point. Using the norelease option keeps the global cache opened and available, even if the file usage count goes to zero.
release	Free the global cache pages when the global cache file usage count goes to zero.

private	Use a private cache. Only the file that opens the cache might use it.
global=	Use global cache, where the number of global caches is specified as a value between **0** and **255**. The default is **1**, which means that one global cache is used.
asynchronous	Use asynchronous calls to the child module.
synchronous	Use synchronous calls to the child module.
noasynchronous	Alias for synchronous
direct	Use direct I/O.
nodirect	Do not use direct I/O.
bytes	Stats output is reported in units of bytes.
kbytes	Stats is reported in output in units of kbytes.
mbytes	Stats is reported in output in units of mbytes.
gbytes	Stats is reported in output in units of gbytes.
tbytes	Stats is reported in output in units of tbytes.
cache_size=	The total size of the cache (in bytes), between the values of **0** and **1GB**, with a default value of **64 K**
page_size=	The size of each cache page (in bytes), between the value of **4096** bytes and **1GB**, with a

	default value of **4096**.
prefetch=	The number of pages to prefetch, between **1** and **100**, with a default of **1**.
stride=	Stride factor, in pages, between **1** and **1G** pages, with a default value of **1**.
stats=	Output prefetch usage statistics to the specified file. If the file name is specified as **mioout**, or no file name is specified, the statistics file name is determined by the setting of the MIO_STATS environment variable.
nostats	Do not output prefetch usage statistics.
inter	Output intermediate prefetch usage statistics on **kill –USR1.**
nointer	Do not output intermediate prefetch usage statistics.
retain	Retain file data after close for subsequent reopen.
noretain	Do not retain file data after close for subsequent reopen.
listio	Use listio mechanism.
nolistio	Do not use listio mechanism.
tag=	String to prefix stats flow
notag	Do not use prefix stats flow.

The default options for the pf module are:
/nodirect/stats=mioout/bytes/cache_size=64k/page_size=4k/
prefetch=1/asynchronous/global/release/stride=1/nolistio/notag

The trace module has the following options

stats=	Output trace statistics to the specified file name. If the

	file name is specified as **mioout**, or no file name is specified, the statistics file name is determined by the setting of the MIO_STATS environment variable.
nostats	Do not output statistics on close.
events=	Generate a binary events file. The default file name if this option is specified is **trace.events**.
noevents	Do not generate a binary events file.
bytes	Output statistics in units of bytes.
kbytes	Output statistics in units of kilobytes.
mbytes	Output statistics in units of megabytes.
gbytes	Output statistics in units of gigabytes.

tbytes	Output statistics in units of terabytes.
inter	Output intermediate trace usage statistics on **kill –USR1**.
nointer	Do not output intermediate statistics.
xml	Generate statistics file in a format that can be viewed using **peekperf**.

The default options for the trace module are:
/stats=mioout/noevents/nointer/bytes

The recov module has the following options

fullwrite	All writes are expected to be full writes. If there is a write failure due to insufficient space, the recov module retries the write.
partialwrite	All writes are not expected to be full writes. If there is a write failure due to insufficient space, there will be no retry.
stats=	Output recov module statistics to the specified file name. If the

	file name is specified as **mioout**, or no file name is specified, the statistics file name is determined by the setting of the MIO_STATS environment variable.
nostats	Do not output recov statistics on file close.
command=	The system command to be issued on a write error.
open_command=	The system command to be issued on open error resulting from a connection that was refused.
retry=	Number of times to retry, between **0** and **100**, with a default of **1**.

The default options for the recov module are:
partialwrite/retry=1

Running Your Application

The I/O profiling options of most interest when using the IBM HPC Toolkit are the stats option, which specifies the name of the statistics file that contains data about the I/O performance of your application, and the events option which specifies the name of a trace file containing data that can be viewed within **peekperf**.

After you have compiled and linked your application as described in Preparing Your Application and set the MIO_FILES, MIO_DEFAULTS, and MIO_STATS environment variables, as needed, then you can run your application.

After you run your application, you can view trace files generated by the I/O profiling library using **peekperf**.

Instrumenting Your Application Using hpctInst

In addition to modifying your application source code to contain calls to instrumentation functions, you can use the **hpctInst** utility to instrument your application without modifying your application source code. The **hpctInst** utility creates a new copy of your application's executable containing the instrumentation that you specified using command line options to the **hpctInst** utility. You can instrument your application with **hpctInst** to obtain performance measurements for hardware performance counters, MPI profiling, OpenMP profiling and I/O profiling.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you invoke **hpctInst**. To set these environment variables, run the setup scripts that are located in the top level directory of your installation. On AIX systems, these setup scripts are located in the /usr/lpp/ppe.hpct directory. On Linux, these setup scripts are located in the /opt/ibmhpc/ppe.hpct directory. If you are using sh, bash, ksh, or similar shell command, invoke the env_sh script as **. env_sh**. If you are using csh, invoke the env_csh script as **source env_csh**.

If you are going to instrument your application using **hpctInst**, you must compile the application using the **–g** compiler option so that **hpctInst** can find the line number and symbol table information it needs to instrument the application. When you link your application, you should not link it with any libraries from the IBM HPC Toolkit. If you have a 64-bit Linux application, you must link your application using the **–emit-stub-syms** linker option.

After you have instrumented your application, you should set any environment variables that are required by the instrumentation you requested. All of the environment variables described in the sections for hardware performance counters, MPI profiling and I/O profiling can be used, as needed, when running an application instrumented with **hpctInst**. The exception is OpenMP profiling, in which the environment variables are used only to control the insertion of instrumentation when the **hpctInst** utility is run, and must be set before **hpctInst** is run. The OpenMP-specific environment variables are therefore only described in the command reference page for the **hpctInst** command.

Note that if you instrument small, frequently called functions, the instrumentation overhead might be significant, and the accuracy of performance measurements might be affected by this overhead.

Instrumenting Your Application for Hardware Performance Counters

You can instrument your application to obtain hardware performance counter information in the following ways:

- You can instrument the entry and exit points of every function in your application by using the **–dhpm** option. When you do this, you obtain performance data that includes hardware performance counter totals for each function in your

application. You can use this data to identify functions that require tuning for improved performance.

- You can instrument your application to obtain hardware performance counter performance data at specific locations (function call sites) in your application, where a function is called, using the **–dhpm_func_call** option. If you use this option, you will obtain hardware performance counter information for the function called at the specified location. You can use this information to help you identify how a function called from multiple locations in your application performs from each individual location from which it is called.

  You specify the set of function call sites in the file specified as a parameter to the **–dhpm_func_call** option. You can specify locations to be instrumented such that only function calls from specific functions are instrumented, or only function calls within a specified region of source code. The following example shows a file that specifies that calls to function **sum** from function **compute** and calls to function **distribute_data** from source file **main.c** between lines 100 and 200 are instrumented.

  ```
  sum compute
  distribute_data main.c 100 200
  ```

- You can instrument selected regions of your source code by using the **–dhpm_region** option and specifying a file that contains a list of one or more regions of code to be instrumented. Regions of code might overlap. If regions of code overlap, then the considerations described in Handling of Overlap Issues apply. The following example shows a file that specifies that regions of code between lines 1 and 100 of **main.c** and lines 100 to 300 of **report.c** are to be instrumented.

  ```
  main.c 1 100
  report.c 100 300
  ```

Instrumenting Your Application for MPI Profiling

You can instrument your application for MPI profiling in the following ways:

- You can instrument the entire application so that all MPI calls in the application are traced by using the **–dmpi** option.

- You can instrument your application so that only specific MPI functions called from specific functions in your application are instrumented by using the **–dmpi_func_call** option. You specify the set of MPI function call sites in the file specified as a parameter to the **–dmpi_func_call** option. You can specify locations to be instrumented such that MPI function calls only from a specific function in your application are instrumented or only MPI function calls within a specified region of source code are instrumented. The following example shows a

file that specifies that calls to the **MPI_Send()** function from function compute and calls to **MPI_Recv()** from source file **main.c** between lines 100 and 200 are instrumented.

```
MPI_Send compute
MPI_Recv main.c 100 200
```

- You can instrument selected regions of your source code in which all MPI function calls within that region are instrumented by using the **–dmpi_region** option and specifying a file that contains a list of one or more regions of code to be instrumented. The following example shows a file that specifies that regions of code between lines 1 and 100 of **main.c** and lines 100-300 of **report.c** are to be instrumented.

```
main.c 1 100
report.c 100 300
```

Instrumenting Your Application for OpenMP Profiling

You can instrument your application for OpenMP profiling in the following ways:

- Instrument all parallel loops and parallel regions in your application by using the **–dpomp** option.
- Instrument all parallel regions in your application by using the **-dpomp_parallel** option.
- Instrument all parallel loops in your application by using the **-dpomp_loop** option.
- Instrument all user-written functions in your application by using the **–dpomp_user** option, or a selected subset of user-written functions by using the **–dpomp_userfunc** option. If you use the **–dpomp_userfunc** option, you specify the set of user-written functions to be instrumented in a file, which is a simple list of the functions to be instrumented, with one function name per line in that file.

If you want to instrument only selected parallel loops or parallel regions in your application, use **peekperf**. The **peekperf** GUI allows you to select specific OpenMP constructs to instrument.

Instrumenting Your Application for I/O Profiling

You can instrument your entire application for I/O profiling by using the **–dmio** option. You select the specific files that will have performance data obtained for them by setting I/O profiling environment variables as specified in the I/O Profiling Environment Variables section.

If you want to instrument only specific I/O function calls in your application, use peekperf to instrument your application.

Commands and API Reference

Hardware Performance Monitoring

hpccount

Report summary hardware performance counter and resource usage statistics for an application.

Synopsis

hpccount [-o <name>] [-u] [-n] [-x] [-g <group[,group]>] [-a <plug-in>] <program>
hpccount [-h] [-l] [-c]

Flags

-c	Lists the available counters and the hardware counter events that can be counted by each counter.
-g <group[,group]>	A single value that specifies the hardware counter group to be used, or a comma-delimited list of hardware counter groups to be multiplexed (AIX only). If this flag or the HPM_EVENT_SET environment variable is not set, a processor specific default group is used, as follows:

PowerPC 970	23
POWER4	60
POWER5	137
POWER5+™	145
POWER6	127

-h	Displays a usage message.
-l	Lists the available hardware counter groups and the hardware counter events that are counted by each group.
-n	Suppresses **hpccount** output to stdout.
-o <name>	Writes output to file <*name*>

- The file <*name*> can be specified using option **–o** or using the environment variable HPM_OUTPUT_NAME. The option takes precedence if there are conflicting specifications.
- The name <*name*> is expanded in to different file names:
- <*name*>.hpm is the file name for ASCII output, which is a one-to-one copy of the screen output.
- <*name*>.**viz** is the file name for the XML output.
- Which of these output files are generated is governed by additional environment variables. If none of those are set, only the ASCII stdout is generated. If at least one is set, the following rules apply.
- HPM_ASC_OUTPUT, if set to **yes**, triggers the ASCII output

- HPM_VIZ_OUTPUT, if set to **yes**, triggers the XML output.

Unless the **–a** option is chosen, there is one output for each MPI task. To avoid directing all output to the same file, the user is advised to have a different name for each MPI task by using the -u flag below or by directing the file to a nonshared file system

-u Specifies that unique file names will be used for generated ASCII and XML output files according to the following rules:

- A string _<*hostname*>_<*process_id*>_<*date*>_<*time*> is inserted before the last period (.) in the file name.
- If *hostname* is a fully qualified name, the short form of the hostname is used in substituting <hostname>.
- If the application is an MPI program, the library attempts to use the MPI task number in place of <*process_id*>.
- The date is substituted using dd.mm.yyyy format.
- The time is substituted as hh.mm.ss using 24-hour time.

-x Displays formulas for derived metrics as part of the command output.

Description

The **hpccount** command provides comprehensive reports of events that are critical to performance on IBM systems. HPM is able to gather the usual timing information, as well as critical hardware performance metrics, such as the number of misses on all cache levels, the number of floating point instructions executed, and the number of instruction loads that cause TLB misses. These reports help the algorithm designer or programmer identify and eliminate performance bottlenecks.

The **hpccount** command invokes the target program and counts hardware performance counter events generated by the target program. The **hpccount** command reports this information after the target program completes.

Environment Variables

Event Selection Environment Variables

HPM_EVENT_SET A single value that specifies the hardware counter group to be used, or a comma-delimited list of hardware counter groups to be multiplexed (AIX only). If the **–g** flag is not used and HPM_EVENT_SET is not set, a processor-specific default group is used, as follows:

PowerPC 970	23
POWER4	60
POWER5	137
POWER5+	145

	POWER6 127
HPM_SLICE_DURATION	Specifies the interval, in milliseconds, to be used when hardware counter groups are multiplexed. You can specify a value between **10** milliseconds and **30** seconds. The default is **100** milliseconds. This environment variable is supported for AIX only.

Output Control Environment Variables

HPM_ASC_OUTPUT	Set to **yes** to generate an ASCII output file with the name <*programName*>.**hpm**. If neither HPM_ASC_OUTPUT or HPM_VIZ_OUTPUT are set, both an ASCII output file and an XML output file are generated. If HPM_ASC_OUTPUT and HPM_VIZ_OUTPUT are set to **no**, no output is generated.
HPM_OUTPUT_NAME	Specifies the <*name*> portion of the output files <*name*>.**hpm** and <*name*>.**viz**.
HPM_PRINT_FORMULA	Set to **yes** to print the definitions of the derived metrics. Set to **no** to suppress this output. The default is **no**.
HPM_STDOUT	Set to **yes** to write ASCII output to stdout. If HPM_STDOUT is set to **no**, no output is written to stdout. The default is **yes**.
HPM_UNIQUE_FILE_NAME	Set to **yes** in order to generate unique file names for generated ASCII and XML output files. Set to **no** to generate the file name exactly as specified by HPM_OUTPUT_NAME. If HPM_UNIQUE_FILE_NAME is set to **yes**, the following rules apply: • A string _<*hostname*>_<*process_id*>_<*date*>_<*time*> is inserted before the last period (.) in the file name. • If hostname is a fully qualified name, the short form of the hostname is used in substituting <*hostname*>. • If the application is an MPI program, the library attempts to use the MPI task number in place of <*process_id*>. • The date is substituted using dd.mm.yyyy format. The time is substituted as hh.mm.ss using 24-hour time.
HPM_VIZ_OUTPUT	Set to **yes** to generate an XML output file with the name <*programName*>.**xml**.

Latency Environment Variables

HPM_L2_LATENCY	User-specified estimate of latency for an L2 cache load.
HPM_L25_LATENCY	User-specified estimate of latency for an L2 cache load from the same MCM.
HPM_L275_LATENCY	User-specified estimate of latency for an L2 cache load from a different MCM.
HPM_L3_LATENCY	User-specified estimate of latency for an L3 cache load within an MCM.
HPM_L35_LATENCY	User-specified estimate of latency for an L3 cache load outside the MCM.
HPM_MEM_LATENCY	User-specified estimate of latency for a load from memory.
HPM_TLB_LATENCY	User-specified estimate of latency for a TLB miss.

Miscellaneous Environment Variables

HPM_DIV_WEIGHT	User-specified weighting factor, greater than **1**, for computing weighted flops on a POWER4 system.
IHPCT_BASE	Path name of the directory in which IBM HPC Toolkit is installed (/usr/lpp/ppe.hpct for AIX, /opt/ibmhpc/ppe.hpct for Linux).

Files

<name>.hpm	The ASCII output from the **hpccount** invocation. This is a copy of the report displayed at the completion of **hpccount** execution.
<name>.viz	An XML output file containing hardware performance counter data from hpccount execution. This file can be viewed using **peekperf**.

Examples

To list available counter groups for your processor:

```
hpccount –l
```

To report summary hardware counter statistics for an application:

```
hpccount –o stats –g 25 testprog
```

To report summary hardware counter statistics for an MPI program:

```
poe hpccount –o stats –g 25 –u testprog
```

hpcstat

Report a system-wide summary of hardware performance counter statistics and resource usage.

Synopsis

hpcstat [-o <name> [-n] [-x] [-k] [-u] [-I <time>] [-U <time>] [-C <count>]
[-g <group[,<group>]>]
hpcstat [-h] [-l] [-c]

Flags

-c	Lists the available counters and the hardware counter events that can be counted by each counter.
-C <count>	Specifies the number of times **hpcstat** reports statistics. The default is **1**.
-g <group[,<group>]>	A single value that specifies the hardware counter group to be used. If this flag or the HPM_EVENT_SET environment variable is not set, a processor specific default group is used, as follows:

PowerPC 970	23
POWER4	60
POWER5	137
POWER5+	145
POWER6	127

-h	Displays a usage message.
-I<time>	Specifies the interval, in seconds, for reporting statistics.
-k	Specifies that only kernel side events are to be counted.
-l	Lists the available hardware counter groups and the hardware counter events that are counted by each group.
-n	Suppresses **hpcstat** output to stdout.
-o <name>	Writes output to file <*name*>

- The file <*name*> can be specified using option **–o** or using the environment variable HPM_OUTPUT_NAME. The option takes precedence if there are conflicting specifications.
- The name <*name*> is expanded in to different file names:
- <*name*>.**hpm** is the file name for ASCII output which is a one-to-one copy of the screen output.
- <*name*>.**viz** is the file name for the XML output.
- Which of these output files are generated is governed by additional environment variables. If none of those are set, only the ASCII stdout is generated. If at least one is set, the following rules apply.
- HPM_ASC_OUTPUT, if set to **yes**, triggers the

ASCII output.
- HPM_VIZ_OUTPUT, if set to **yes**, triggers the XML output.

-u	Specifies that only user side events are to be counted.
-U<time>	Specifies the interval, in microseconds, for reporting statistics.
-x	Displays formulas for derived metrics as part of the command output.

Description

The **hpcstat** tool lists a variety of performance information to stdout or to a file. In particular, it prints resource utilization statistics, hardware performance counter information and derived hardware metrics. If the –C flag and either the –I or –U flags are used, **hpcstat** reports hardware performance counter statistics and resource utilization on a periodic basis, similar to the **vmstat** command.

The resource usage statistics are directly taken from a call to **getrusage**(). For more information on the resource utilization statistics, please refer to the **getrusage** man pages.

The **hpcstat** command requires the user to have root privileges. This command is an AIX-only command.

Environment Variables

Event Selection Environment Variables

HPM_EVENT_SET A single value that specifies the hardware counter group to be used
If the **–g** flag is not specified and HPM_EVENT_SET is not set, a processor specific default group is used, as follows:

PowerPC 970	23
POWER4	60
POWER5	137
POWER5+	145
POWER6	127

Output Control Environment Variables

HPM_ASC_OUTPUT Set to **yes** to generate an ASCII output file with the name <*programName*>.**hpm**. If neither HPM_ASC_OUTPUT or HPM_VIZ_OUTPUT are set, both an ASCII output file and an XML output file are generated. If both HPM_ASC_OUTPUT and HPM_VIZ_OUTPUT are set to **no**, then no output is generated.

HPM_OUTPUT_NAME	Specifies the <*name*> portion of the output files <*name*>.**hpm** and <*name*>.**viz**.
HPM_PRINT_FORMULA	Set to **yes** to print the definitions of the derived metrics. Set to **no** to suppress this output. The default is **no**.
HPM_STDOUT	Set to **yes** to write ASCII output to stdout. If HPM_STDOUT is set to **no**, no output is written to stdout. The default is **yes**.
HPM_VIZ_OUTPUT	Set to **yes** to generate an XML output file with the name <*programName*>.**viz**.
HPM_UNIQUE_FILE_NAME	The output file name can be made unique by setting the environment variable HPM_UNIQUE_FILE_NAME=yes. This triggers the following changes. • A string _<*hostname*>_<*process_id*>_<*date*>_<*time*> is inserted before the last period (.) in the file name. • If the host name contains period (.) ("long form"), only the portion preceding the first period (.) is taken. In case a batch queuing system is used, the host name is taken from the execution host, not the submitting host. The date is given as dd.mm.yyyy. The time is given by hh.mm.ss in 24-hour format using the local time zone.

Latency Environment Variables

HPM_L2_LATENCY	User-specified estimate of latency for an L2 cache load.
HPM_L25_LATENCY	User-specified estimate of latency for an L2 cache load from the same MCM.
HPM_L275_LATENCY	User-specified estimate of latency for an L2 cache load from a different MCM.
HPM_L3_LATENCY	User-specified estimate of latency for an L3 cache load within an MCM.
HPM_L35_LATENCY	User-specified estimate of latency for an L3 cache load outside the MCM.
HPM_MEM_LATENCY	User-specified estimate of latency for a load from memory.
HPM_TLB_LATENCY	User-specified estimate of latency for a TLB miss.

Miscellaneous Environment Variables

HPM_DIV_WEIGHT	User-specified weighting factor, greater than **1**, for computing weighted flops on a POWER4 system.
IHPCT_BASE	Path name of the directory in which IBM HPC Toolkit is

installed (/usr/lpp/ppe.hpct for AIX, /opt/ibmhpc/ppe.hpct for Linux).

Files

<name>.hpm The ASCII output from the **hpcstat** invocation. This is a copy of the report displayed at the completion of **hpcstat** execution.

<name>.viz An XML output file containing hardware performance counter data from **hpcstat** execution. This file can be viewed using **peekperf**.

Examples

To list hardware performance counter groups available on your processor:

```
hpcstat –l
```

To report user level hardware performance counter statistics for the system every 30 seconds for five minutes:

```
hpcstat –u –C 10 –I 30 –g 25
```

hpm_error_count, f_error_count

Purpose

Provide a way to verify that a call to a **libhpc** function was successful.

Library

-lhpc
-lpmapi (AIX)
-lperfctr (Linux)

C Synopsis

#include <libhpc.h>

Fortran Synopsis

#include <f_hpc.h>
#include <f_hpc_i8.h>
logical function f_error_count()

Note: Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.

Note: Fortran programs which include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **-qsuffix** option, for instance, **-qsuffix=cpp=f**.

Parameters

None

Description

hpm_error_count is an external variable that HPM library functions set if an error occurs during a call to that function. **f_error_count**() is the equivalent Fortran function that returns a logical value indicating that an error occurred during an HPM library call.

If an HPM library call is successful, **hpm_error_count** is set to zero and **f_error_count**() returns **.false**. If an HPM library call fails, **hpm_error_count** is set to a non zero value and **f_error_count**() returns **.true**.

The **hpm_error_count** variable or **f_error_count**() function should be used at any point where you need to determine if an HPM library function call failed.

Environment Variables

None

Examples

```
#include <libhpc.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    hpmInit(0, “HPMTest”);
    if (hpm_error_count) {
      printf(“hpmInit error\n”);
      exit(1);
    }
    .
    .
    .
    hpmTerminate();
    if (hpm_error_count) {
      printf(“hpmTerminate error\n”);
      exit(1);
    }
}
```

```
program hpmtest
#include “f_hpc.h”
call f_hpminit(0, ‘HPMTest’)
if (f_hpm_error() .eqv. .true.) then
  print *, ‘f_hpminit error’
  stop 1
end if
.
.
.
call f_hpmterminate()
if (f_hpm_error() .eqv. .true.) then
  print *, ‘f_hpmterminate error’
  stop 1
end if
end
```

hpmInit, f_hpminit

Purpose

Initialize the hardware performance counter (HPM) runtime environment.

Library

-lhpc
-lpmapi (AIX)
-lperfctr (Linux)

C Synopsis

#include <libhpc.h>
void hpmInit(int my_ID, const char *progName)

Fortran Synopsis

#include <f_hpc.h>
#include <f_hpc_i8.h>
subroutine f_hpminit(integer my_ID, character progName(*))

Note: Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.

Note: Fortran programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **-qsuffix** option. For instance, **-qsuffix=cpp=f**.

Parameters

my_ID	Unused. Should be set to zero.
progName	Specifies the name of the program. If the HPM_OUTPUT_NAME environment variable is not set, this parameter is used as the name of the visualization files.

Description

This function initializes the runtime environment for obtaining hardware performance counter statistics. It optionally names the output files containing these statistics. It must be the first HPM function call executed in the application.

Environment Variables

Event Selection Environment Variables

HPM_EVENT_SET	A single value that specifies the hardware counter group to be used, or a comma-delimited list of hardware counter groups to be multiplexed (AIX only). If HPM_EVENT_SET is not set, a

processor-specific default group is used, as follows:

PowerPC 970	23
POWER4	60
POWER5	137
POWER5+	145
POWER6	127

HPM_EXCLUSIVE_VALUES — Set to **yes** if exclusive counter values in nested counter regions are to be computed.

HPM_SLICE_DURATION — Specifies the interval, in milliseconds, to be used when hardware counter groups are multiplexed. You can specify a value between **10** milliseconds and **30** seconds. The default is **100** milliseconds. This environment variable is supported for AIX only.

Output Control Environment Variables

HPM_ASC_OUTPUT — Set to **yes** to generate an ASCII output file with the name <*programName*>.**hpm**. If neither HPM_ASC_OUTPUT or HPM_VIZ_OUTPUT are set, both an ASCII output file and an XML output file are generated. If both HPM_ASC_OUTPUT and HPM_VIZ_OUTPUT are set to **no**, no output is generated.

HPM_OUTPUT_NAME — Specifies the <*name*> portion of the output files <*name*>.**hpm** and <*name*>.**viz**. If this environment variable is not set, <*name*> is set to the value of the **progName** parameter.

HPM_PRINT_FORMULA — Set to **yes** to print the definitions of the derived metrics. Set to **no** to suppress this output. The default is **no**.

HPM_STDOUT — Set to **yes** to write ASCII output to stdout. If HPM_STDOUT is set to **no**, no output is written to stdout. The default is **yes**.

HPM_UNIQUE_FILE_NAME — Set to **yes** in order to generate unique file names for generated ASCII and XML output files. Set to **no** to generate the file name exactly as specified by HPM_OUTPUT_NAME. If HPM_UNIQUE_FILE_NAME is set to **yes**, the following rules apply:

- A string _<*hostname*>_<*process_id*>_<*date*>_<*time*> is inserted before the last period (.) in the file name.
- If hostname is a fully-qualified name, the

short form of the hostname is used in substituting <*hostname*>.
- If the application is an MPI program, the library attempts to use the MPI task number in place of <*process_id*>
- The date is substituted using dd.mm.yyyy format. The time is substituted as hh.mm.ss using 24-hour time.

HPM_VIZ_OUTPUT	Set to **yes** to generate an XML output file with the name <*programName*>.**viz**, where <*programName*> is specified by the HPM_OUTPUT_NAME environment variable, or if that is not set, then as specified by the second parameter to the **hpmStart**() call.

Latency Environment Variables

HPM_L2_LATENCY	User-specified estimate of latency for an L2 cache load.
HPM_L25_LATENCY	User-specified estimate of latency for an L2 cache load from the same MCM.
HPM_L275_LATENCY	User-specified estimate of latency for an L2 cache load from a different MCM.
HPM_L3_LATENCY	User-specified estimate of latency for an L3 cache load within an MCM.
HPM_L35_LATENCY	User-specified estimate of latency for an L3 cache load outside the MCM.
HPM_MEM_LATENCY	User-specified estimate of latency for a load from memory.
HPM_TLB_LATENCY	User-specified estimate of latency for a TLB miss.

Plug-in Specific Environment Variables

HPM_EVENT_DISTR	Specifies a comma-delimited list of hardware counter group numbers that are counted by MPI tasks. These group numbers are distributed round-robin to individual MPI tasks, where each task counts only events for the hardware counter group assigned to that task. This environment variable is recognized only when the **average.so** aggregation plug-in is selected.
HPM_PRINT_TASK	Specifies the MPI task that has its results displayed. The default task number is zero. This environment variable is recognized only when the **single.so** aggregation plug-in is selected.

Miscellaneous Environment Variables

HPM_AGGREGATE	Specifies the name of a plug-in that defines the HPM data aggregation strategy. If the plug-in name contains a /, the

	name is treated as an absolute or relative path name. If the name does not contain a /, the plug-in is loaded following the rules for the **dlopen()** function call. The plug-in is a shared object file that implements the **distributor()** and **aggregator()** functions. See the user documentation for details.
HPM_DIV_WEIGHT	User-specified weighting factor, greater than **1**, for computing weighted flops on a POWER4 system.
HPM_NUM_INST_PTS	Sets the maximum number of instrumented sections. The default maximum is **1000**.
IHPCT_BASE	Path name of the directory in which IBM HPC Toolkit is installed (/usr/lpp/ppe.hpct for AIX, /opt/ibmhpc/ppe.hpct for Linux).

Examples

```
#include <libhpc.h>
int main(int argc, char *argv[])
{
    hpmInit(0, “HPMTest”);
    .
    .
    .
    hpmTerminate();
}

program hpmtest
#include “f_hpc.h”
call f_hpminit(0, ‘HPMTest’)
.
.
.
call f_hpmterminate()
end
```

hpmStart, f_hpmstart

Purpose

Identifies the starting point for a region of code in which hardware performance counter events are to be counted.

Library

-lhpc
-lpmapi (AIX)
-lperfctr (Linux)

C Synopsis

#include <libhpc.h>
void hpmStart(int inst_ID, const char *label);

Fortran Synopsis

#include <f_hpc.h>
#include <f_hpc_i8.h>
subroutine f_hpmstart(integer inst_ID, character label(*))

Note: Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.

Note: Fortran programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **-qsuffix** option. For instance, **-qsuffix=cpp=f**.

Parameters

inst_ID	Specifies a unique value identifying the instrumented code region. The value must be less than the value specified by the HPM_NUM_INST_PTS environment variable which has a default value of **1000**.
progName	Specifies the name associated with the instrumented code region. This name is used to identify the code region in the generated performance data.

Description

The **hpmStart**() function identifies the start of a region of code in which hardware performance counter events are to be counted. The end of the region is identified by a call to **hpmStop**() using the same **inst_ID**.

The **hpmStart**() function assigns an identifier and a name to that region. When this function is executed, it records the starting value for the hardware performance counters that are being used. When the corresponding **hpmStop**() function call is

executed, the hardware performance counters are read again and the difference between the current values and the starting values is accumulated.

Regions of code bounded by **hpmStart**() and **hpmStop**() calls can be nested. When regions are nested, **hpmStart**() and **hpmStop**() properly accumulate hardware events so they can be properly accounted for with both inclusive and exclusive reporting.

If **hpmStart**() and **hpmStop**() functions are called in a threaded application, the count of hardware performance counter events is for the entire process rather than for the specific thread on which the calls are made. If you need accurate counts for each thread, use **hpmTstart**() and **hpmTstop**().

Environment Variables

See **hpmInit**().

Examples

```
#include <libhpc.h>
int main(int argc, char *argv[])
{
    int i;
    float x;
    x = 10.0;
    hpmInit(0, “HPMTest”);
    hpmStart(1, “Region 1”);
    for (i = 0; i < 100000; i++) {
       x = x / 1.001;
    }
    hpmStop(1);
    hpmTerminate();
}
```

```
    program hpmtest
    #include “f_hpc.h”
    integer i
    real*4 x
    call f_hpminit(0, ‘HPMTest’)
    x = 10.0;
    call f_hpmstart(1, ‘Region 1’)
    do 10 i = 1, 100000
      x = x / 1.001
10 continue
    call f_hpmstop(1)
    call f_hpmterminate()
    end
```

hpmStartx, f_hpmstartx

Purpose

Identifies the starting point for a region of code in which hardware performance counter events are to be counted, specifying explicit inheritance relationships for nested instrumentation regions.

Library

-lhpc
-lpmapi (AIX)
-lperfctr (Linux)

C Synopsis

#include <libhpc.h>
void hpmStartx(int inst_ID, int parent_ID, const char *label);

Fortran Synopsis

#include <f_hpc.h>
#include <f_hpc_i8.h>
subroutine f_hpmtstart(integer inst_ID, integer parent_ID, character label(*))

Note: Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.

Note: Fortran programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **-qsuffix** option. For instance, **-qsuffix=cpp=f**.

Parameters

inst_ID	Specifies a unique value identifying the instrumented code region. The value must be less than the value specified by the HPM_NUM_INST_PTS environment variable, which has a default value of **1000**.
parent_ID	Specifies the inheritance relationship for nested **hpmStart**() calls. This parameter must have one of the following values: • HPM_AUTO_PARENT • HPM_ONLY_EXCLUSIVE • HPM_NO_PARENT • The **inst_ID** of an active **hpmStart**() or **hpmStartx**() call
progName	Specifies the name associated with the instrumented code region. This name is used to identify the code region in the generated performance data.

Description

The **hpmStartx**() function identifies the start of a region of code in which hardware performance counter events are to be counted, and explicitly specifies the parent relationship for an encompassing region instrumented by **hpmStart**() or **hpmStartx**(). The end of the region is identified by a call to **hpmStop**() using the same **inst_ID**.

The **hpmStartx**() function assigns an identifier and a name to that region. When this function is executed, it records the starting value for the hardware performance counters that are being used. When the corresponding **hpmStop**() function call is executed, the hardware performance counters are read again and the difference between the current values and the starting values is accumulated.

Regions of code bounded by **hpmStartx**() and **hpmStop**() calls can be nested. When regions are nested, **hpmStartx**() and **hpmStop**() properly accumulate hardware events so they can be properly accounted for with both inclusive and exclusive reporting. For reporting of exclusive event counts, the proper parent relationship must be determined. If regions are perfectly nested, such as a set of nested loops, **hpmStart**() is sufficient for determining parent relationships. In more complicated nesting cases, **hpmStartx**() should be used to properly specify those relationships.

Parent relationships are specified by the **parent_ID** parameter which must have one of the following values:

- HPM_AUTO_PARENT: Automatically determine the parent for this **hpmStartx**() call. This is done by searching for the immediately preceding **hpmStart**() or **hpmStartx**() call executed on the current thread in which there has not been a corresponding call made to **hpmStop**().
- HPM_ONLY_EXCLUSIVE: This operates in the same way as if HPM_AUTO_PARENT was specified, and also acts as if the HPM_EXCLUSIVE_VALUES environment variable was set for this call to **hpmStartx**() only. If the HPM_EXCLUSIVE_VALUES environment variable was previously set, this parameter value is equivalent to specifying HPM_AUTO_PARENT.
- HPM_NO_PARENT: Specifies that this **hpmStartx**() call has no parent.
- inst_ID: for a previous **hpmStart**() or **hpmStartx**() call that is currently active, meaning the corresponding call to **hpmStop**() has not been made for this instance of execution.

If **hpmStartx**() and **hpmStop**() functions are called in a threaded application, the count of hardware performance counter events is for the entire process rather than for the specific thread on which the calls were made. If you need accurate counts for each thread, use **hpmTstartx**() and **hpmTstop**().

Environment Variables

See **hpmInit**()

Examples

```
#include <libhpc.h>
int main(int argc, char *argv[])
{
    int i;
    int j;
    float x;
    x = 10.0;
    hpmInit(0, “HPMTest”);
    hpmStartx(1, HPM_NO_PARENT, “Region 1”);
    for (i = 0; i < 100000; i++) {
       hpmStartx(2, 1, “Region 2”);
       for (j = 0; j < 100000; j++) {
         x = x / 1.001;
       }
       hpmStop(2);
       x = x / 1.001;
    }
    hpmStop(1);
    hpmTerminate();
}
```

```
    program hpmtest
    #include “f_hpc.h”
    integer i
    real*4 x
    call f_hpminit(0, ‘HPMTest’)
    x = 10.0;
    call f_hpmstartx(1, HPM_NO_PARENT, ‘Region 1’)
    do 10 i = 1, 100000
      x = x / 1.001
 10 continue
    call f_hpmstop(1)
    call f_hpmterminate()
    end
```

hpmStop, f_hpmstop

Purpose

Identifies the end point of a region of code starting with a call to **hpmStart**() or **hpmStartx**(), in which hardware performance counter events are to be counted. Also accumulates hardware performance counter events for that region.

Library

-lhpc
-lpmapi (AIX)
-lperfctr (Linux)

C Synopsis

#include <libhpc.h>
void hpmStop(int inst_ID);

Fortran Synopsis

#include <f_hpc.h>
#include <f_hpc_i8.h>
subroutine f_hpmstop(integer inst_ID)

Note: Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.

Note: Fortran programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **-qsuffix** option. For instance, **-qsuffix=cpp=f**.

Parameters

inst_ID Specifies a unique value identifying the instrumented code region. This value must match the **inst_ID** specified in the corresponding **hpmStart**() or **hpmStartx**() function call.

Description

The **hpmStop**() function identifies the end of a region of code in which hardware performance counter events are to be monitored. The start of the region is identified by a call to the **hpmStart**() or **hpmStartx**() function using the same **inst_ID**. That function must be called for a specific **inst_ID** before the corresponding call to the **hpmStop**() function.

Environment Variables

None

Examples

```
#include <libhpc.h>
```

```
int main(int argc, char *argv[])
{
    int i;
    float x;
    x = 10.0;
    hpmInit(0, “HPMTest”);
    hpmStart(1, “Region 1”);
    for (i = 0; i < 100000; i++) {
       x = x / 1.001;
    }
    hpmStop(1);
    hpmTerminate();
}
```

```
    program hpmtest
    #include “f_hpc.h”
    integer i
    real*4 x
    call f_hpminit(0, ‘HPMTest’)
    x = 10.0;
    call f_hpmstart(1, ‘Region 1’)
    do 10 i = 1, 100000
      x = x / 1.001
 10 continue
    call f_hpmstop(1)
    call f_hpmterminate()
    end
```

hpmTerminate, f_hpmterminate

Purpose

Generate hardware performance counter (HPM) statistics files and shut down the HPM environment.

Library

-lhpc
-lpmapi (AIX)
-lperfctr (Linux)

C Synopsis

#include <libhpc.h>
void hpmTerminate()

Fortran Synopsis

#include <f_hpc.h>
#include <f_hpc_i8.h>
subroutine f_hpmterminate()

Note: Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.

Note: Fortran programs that include either of these headers might need to be preprocessed by the C preprocessor. This might require the use of the **-qsuffix** option. For instance, **-qsuffix=cpp=f**.

Parameters

None

Description

This function generates output files containing any hardware performance counter statistics obtained during the program's execution and shuts down the HPM runtime environment. This function must be called before the application exits in order to generate statistics. It must be the last HPM function called during program execution.

If the HPM_AGGREGATE environment variable is set, and the instrumented application is an MPI application, **hpmTerminate**() should be called before **MPI_Finalize()** is called, because the plug-in specified by the HPM_AGGREGATE environment variable might call MPI functions as part of its internal processing.

Environment Variables

See **hpmInit**()

Examples

```
#include <libhpc.h>
int main(int argc, char *argv[])
{
    hpmInit(0, “HPMTest”);
    .
    .
    .
    hpmTerminate();
}

program hpmtest
#include “f_hpc.h”
call f_hpminit(0, ‘HPMTest’)
.
.
.
call f_hpmterminate()
end
```

hpmTstart, f_hpmtstart

Purpose

Identifies the starting point for a region of code in which hardware performance counter events are to be counted on a per-thread basis.

Library

-lhpc
-lpmapi (AIX)
-lperfctr (Linux)

C Synopsis

```
#include <libhpc.h>
void hpmTstart(int inst_ID, const char *label);
```

Fortran Synopsis

```
#include <f_hpc.h>
#include <f_hpc_i8.h>
subroutine f_hpmtstart(integer inst_ID, character label(*))
```

Note: Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.

Note: Fortran programs which include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **-qsuffix** option, for instance, **-qsuffix=cpp=f**.

Parameters

inst_ID	Specifies a unique value identifying the instrumented code region. The value must be less than the value specified by the HPM_NUM_INST_PTS environment variable, which has a default value of **1000**.
progName	Specifies the name associated with the instrumented code region. This name is used to identify the code region in the generated performance data.

Description

The **hpmTstart**() function identifies the start of a region of code in which hardware performance counter events are to be counted. The end of the region is identified by a call to **hpmTstop**() using the same **inst_ID**.

The **hpmTstart**() function assigns an identifier and a name to that region. When this function is executed, it records the starting value for the hardware performance counters that are being used. When the corresponding **hpmTstop**() function call is

executed, the hardware performance counters are read again and the difference between the current values and the starting values is accumulated.

Regions of code bounded by **hpmTstart**() and **hpmTstop**() calls can be nested. When regions are nested, **hpmTstart**() and **hpmTstop**() properly accumulate hardware events so they can be properly accounted for with both inclusive and exclusive reporting.

The only difference between the **hpmStart**() and **hpmTstart**() functions is that a call to **hpmStart**() results in reading the hardware performance counters for the entire process while a call to **hpmTstart**() results in reading the hardware performance counters only for the thread from which the call to **hpmTstart**() was made.

Environment Variables

See **hpmInit**().

Examples

```
#include <libhpc.h>
void thread_func();
int main(int argc, char *argv[])
{
    hpmInit(0, “HPMTest”);
    thread_func(); /* assume this function runs on
                      multiple threads */
    hpmTerminate();
}
void thread_func()
{
    int i;
    float x;
    x = 10.0;
    hpmTstart(1, “Region 1”);
    for (i = 0; i < 100000; i++) {
       x = x / 1.001;
    }
    hpmTstop(1);
}
```

```
    program hpmtest
    #include “f_hpc.h”
    real*4 x
    call f_hpminit(0, ‘HPMTest’)
    x = thread_func() ! Assume thread_func runs on
                      ! multiple threads
    call f_hpmterminate()
    end
```

```
      real*4 function thread_func()
      #include “f_hpc.h”
      real*4 x
      integer i
      x = 10.0;
      call f_hpmtstart(1, ‘Region 1’)
      do 10 i = 1, 100000
         x = x / 1.001
   10 continue
      call f_hpmtstop(1)
      thread_func = x
      return
      end
```

hpmTstartx, f_hpmtstartx

Purpose

Identifies the starting point for a region of code in which hardware performance counter events are to be counted on a per-thread basis, specifying explicit inheritance relationships for nested instrumentation regions.

Library

-lhpc
-lpmapi (AIX)
-lperfctr (Linux)

C Synopsis

#include <libhpc.h>
void hpmTstartx(int inst_ID, int parent_ID, const char *label);

Fortran Synopsis

#include <f_hpc.h>
#include <f_hpc_i8.h>
subroutine f_hpmtstartx(integer inst_ID, integer parent_ID, character label(*))

Note: Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.

Note: Fortran programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **-qsuffix** option. For instance, **-qsuffix=cpp=f**.

Parameters

inst_ID	Specifies a unique value identifying the instrumented code region. The value must be less than the value specified by the HPM_NUM_INST_PTS environment variable, which has a default value of **1000**.
parent_ID	Specifies the inheritance relationship for nested **hpmStart**() calls. This parameter must have one of the following values: • HPM_AUTO_PARENT • HPM_ONLY_EXCLUSIVE • HPM_NO_PARENT • The **inst_ID** of an active **hpmTstart**() or **hpmTstartx**() call.
progName	Specifies the name associated with the instrumented code region. This name is used to identify the code region in the generated performance data.

Description

The **hpmTstartx**() function identifies the start of a region of code in which hardware performance counter events are to be counted, on a per-thread basis, and explicitly specifies the parent relationship for an encompassing region instrumented by **hpmTstart**() or **hpmTstartx**(). The end of the region is identified by a call to **hpmTstop**() using the same **inst_ID**.

The **hpmTstartx**() function assigns an identifier and a name to that region. When this function is executed, it records the starting value for the hardware performance counters that are being used. When the corresponding **hpmTstop**() function call is executed, the hardware performance counters are read again and the difference between the current values and the starting values is accumulated.

Regions of code bounded by **hpmTstartx**() and **hpmTstop**() calls can be nested. When regions are nested, **hpmTstartx**() and **hpmTstop**() properly accumulate hardware events so they can be properly accounted for with both inclusive and exclusive reporting. For reporting of exclusive event counts, the proper parent relationship must be determined. If regions are perfectly nested, such as a set of nested loops, **hpmTstart**() is sufficient for determining parent relationships. In more complicated nesting cases, **hpmTstartx**() should be used to properly specify those relationships.

Parent relationships are specified by the **parent_ID** parameter which must have one of the following values:

- HPM_AUTO_PARENT: Automatically determine the parent for this **hpmTstartx**() call. This is done by searching for the immediately preceding **hpmTstart**() or **hpmTstartx**() call executed on the current thread in which there has not been a corresponding call made to **hpmStop**().
- HPM_ONLY_EXCLUSIVE: This operates in the same way as if HPM_AUTO_PARENT was specified, and also acts as if the HPM_EXCLUSIVE_VALUES environment variable was set for this call to **hpmTstartx**() only. If the HPM_EXCLUSIVE_VALUES environment variable was previously set, this parameter value is equivalent to specifying HPM_AUTO_PARENT.
- HPM_NO_PARENT: Specifies that this **hpmTstartx**() call has no parent.
- **inst_ID** for a previous **hpmTstart**() or **hpmTstartx**() call that is currently active, meaning the corresponding call to **hpmTstop**() has not been made for this instance of execution.

Environment Variables

See **hpmInit**()

Example

```
#include <libhpc.h>
void thread_func();
int main(int argc, char *argv[])
```

```
{
    hpmInit(0, “HPMTest”);
    thread_func(); /* assume this function runs on
                      multiple threads */
    hpmTerminate();
}
void thread_func()
{
    int i;
  int j;
    float x;
    x = 10.0;
    hpmTstartx(1, HPM_NO_PARENT, “Region 1”);
    for (i = 0; i < 100000; i++) {
       hpmTstartx(2, 1, “Region 2”);
    for (j = 0; j < 100000; j++) {
      x = x / 1.001;
       }
    hpmTstop(2);
       x = x / 1.001;
    }
    hpmTstop(1);
}

    program hpmtest
    #include “f_hpc.h”
    real*4 x
    call f_hpminit(0, ‘HPMTest’)
    x = thread_func() ! Assume thread_func runs on
                      ! multiple threads
    call f_hpmterminate()
    end

  real*4 function thread_func()
  #include “f_hpc.h”
  real*4 x
     integer i
     integer j
    x = 10.0;
    call f_hpmtstartx(1, HPM_NO_PARENT, ‘Region 1’)
    do 10 i = 1, 100000
      call f_hpmtstartx(2, 1, ‘Region 2’)
      do 20 j = 1, 100000
         x = x / 1.001
20  continue
      call f_hpmtstop(2)
      x = x / 1.001
```

```
10 continue
    call f_hpmtstop(1)
    thread_func = x
  return
    end
```

hpmTstop, f_hpmtstop

Purpose

Identifies the end point of a region of code starting with a call to **hpmTstart**() or **hpmTstartx**() in which hardware performance counter events are to be counted. Also accumulates hardware performance counter events for that region.

Library

-lhpc
-lpmapi (AIX)
-lperfctr (Linux)

C Synopsis

#include <libhpc.h>
void hpmTstop(int inst_ID);

Fortran Synopsis

#include <f_hpc.h>
#include <f_hpc_i8.h>
subroutine f_hpmtstop(integer inst_ID)

Note: Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.

Note: Fortran programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **-qsuffix** option. For instance, **-qsuffix=cpp=f**.

Parameters

inst_ID Specifies a unique value identifying the instrumented code region. This value must match the **inst_ID** specified in the corresponding **hpmTstart**() or **hpmTstartx**() function call.

Description

The **hpmTstop**() function identifies the end of a region of code in which hardware performance counter events are to be monitored. The start of the region is identified by a call to an **hpmTstart**() or **hpmTstartx**() function using the same **inst_ID**. The **hpmTstart**() or **hpmTstartx**() function must be called for a specific **inst_ID** before the corresponding call to the **hpmStop**() function.

Environment Variables

None

Examples

```
#include <libhpc.h>
```

```
int main(int argc, char *argv[])
{
    int i;
    float x;
    x = 10.0;
    hpmInit(0, “HPMTest”);
    hpmTstart(1, “Region 1”);
    for (i = 0; i < 100000; i++) {
       x = x / 1.001;
    }
    hpmTstop(1);
    hpmTerminate();
}
```

```
    program hpmtest
    #include “f_hpc.h”
    integer i
    real*4 x
    call f_hpminit(0, ‘HPMTest’)
    x = 10.0;
    call f_hpmtstart(1, ‘Region 1’)
    do 10 i = 1, 100000
      x = x / 1.001
10 continue
    call f_hpmtstop(1)
    call f_hpmterminate()
    end
```

MPI Profiling

MT_get_allresults

Purpose

Obtain statistical results from performance data for an MPI function.

Library

-lmpitrace

C Synopsis

```
#include <mpt.h>
#include <mpi_trace_ids.h>
int MT_get_allresults(int data_type, int mpi_id, struct MT_summarystruct *data);
```

Parameters

data_type	Specifies the type of data to be returned in the data parameter.
mpi_id	An enumeration specifying the MPI function for which data is obtained.
data	A structure, allocated by the user, containing the statistical data returned by calling this function.

Description

This function computes statistical data from performance data accumulated for an MPI function type or for all MPI functions in an application.

The data_type parameter specifies the statistical measurement that is returned by a call to this function as follows:

COUNTS	The number of times the specified MPI function was called.
BYTES	The total number of bytes of data transferred in calls to the specified MPI function.
COMMUNICATIONTIME	The total time spent in all calls to the specified MPI function.
STACK	The maximum stack address for any call to the specified MPI function.
HEAP	The maximum heap address for any call to the specified MPI function.
ELAPSEDTIME	Either the elapsed time between calls to **MPI_Init()** and **MPI_Finalize()**, or, if that value is zero, the elapsed time since **MPI_Init()** was called.

The **mpi_id** parameter specifies the MPI function for which statistics are to be computed. It must be either an enumeration from the table shown in the **MT_trace_event**() function man page, also found in

$IHPCT_BASE/include/mpi_trace_ids.h header, or ALLMPI_ID to compute statistics for all MPI functions profiled by the MPI trace library in the application.

If the **mpi_ID** parameter is specified as ALLMPI_ID, meaningful results are returned only when the data_type parameter is specified as BYTES or COMMUNICATIONTIME. In all other cases, the returned data is zero.

This function fills in the MT_summarystruct structure allocated by the user, in which the following fields are relevant:

int min_rank	The MPI task rank of the task corresponding to the value in the min_result field.
int max_rank	The MPI task rank of the task corresponding to the value in the max_result field.
int med_rank	The MPI task rank of the task corresponding to the value in the med_result field.
void *min_result	The minimum value from all tasks for the measurement specified by the data_type parameter.
void *max_result	The maximum value from all tasks for the measurement specified by the data_type parameter.
void *med_result	The median value from all tasks for the measurement specified by the data_type parameter.
void *avg_result	The average value from all tasks for the measurement specified by the data_type parameter.
void *sum_result	The sum of the measurements from all tasks for the measurement specified by the data_type parameter.
void *all_result	An array of measurements for all tasks, in MPI task rank order, for the measurement specified by the data_type parameter.
void *sorted_all_result	An array of measurements for all tasks, sorted in data value order, for the measurement specified by the data_type parameter.
int *sorted_rank	An array of MPI task ranks corresponding to the data values in the sorted_all_result array.

The datatype of the min_result, max_result, med_result, avg_result and sum_result fields depends on the value specified for the data_type parameter as follows:

COUNTS	long long
BYTES	double
COMMUNICATIONTIME	double
STACK	double
HEAP	double
ELASPSEDTIME	double

You must cast the above fields to the appropriate data type in your code.

The all_result and sorted_all result arrays are arrays of the same data type as the individual fields described above. You are responsible for freeing these arrays after they are no longer needed.

This function can be useful when you implement your own version of **MT_output_text**().

Returns

This function returns **1** for successful completion. It returns **-1** if an error occurs.

Environment Variables

IHPCT_BASE Path name of the directory in which IBM HPC Toolkit is installed (/usr/lpp/ppe.hpct for AIX, /opt/ibmhpc/ppe.hpct for Linux).

Examples

```
#include <mpi.h>
#include <mpi_trace_ids.h>
#include <stdio.h>
int MT_output_text()
{
  struct MT_summarystruct stats;
  MT_get_allresults(BYTES, SEND_ID, &stats);
  printf("Minimum bytes sent (%11.6f) by task %d\n",
           (double) stats.min_result, stats.min_rank);
  printf("Maximum bytes sent (%11.6f) by task %d\n",
           (double) stats.max_result, stats.max_rank);
  return 0;
}
```

MT_get_calleraddress

Purpose

Obtain the address of the caller of an MPI function.

Library

-lmpitrace

C Synopsis

void *MT_get_calleraddress(int level);

Parameters

level Specifies the number of levels to walk up the call stack to get the caller's address.

Description

This function can be used within your implementation of **MT_trace_event**() to obtain the address of the caller of an MPI function. If this function is called inside your implementation of **MT_trace_event**() and the level parameter is specified as zero, it obtains the address where the MPI function was called. If the level parameter is specified as **1**, this function returns the address where the function that called the current MPI function was called.

Returns

This function returns the caller's address as determined by the level parameter.

Environment Variables

None

Examples

```
#include <mpt.h>
int MT_trace_event(int id)
{
    unsigned long caller_addr;
    caller_addr = MT_getcalleraddress(0);
    return 1;
}
```

MT_get_callerinfo

Purpose

Obtain source code information about the caller of an MPI function.

Library

-lmpitrace

C Synopsis

#include <mpt.h>
int MT_get_callerinfo(unsigned long addr, struct MT_callerstruct src_info);

Parameters

addr	Specifies the address for which source code information is to be obtained.
src_info	Contains the source code information returned by this function for the specified caller address.

Description

This function can be used to obtain the source code information, including the file name and line number corresponding to the address specified by the **addr** parameter.

This function fills in the MT_callerstruct structure passed to it with the following information:

char *filepath	The path name of the directory containing the source file.
char *file name	The file name of the source file.
char *funcname	The name of the function containing the caller address.
int lineno	The source line number corresponding to the address passed in the **addr** parameter.

In order for this function to work correctly, the application should be compiled and linked with the **–g** compiler option so that the required file name and line number information is contained in the executable.

Returns

This function returns zero if the source file information was obtained. It returns **-1** if the source file information could not be obtained.

Environment Variables

None

Examples

```
#include <mpt.h>
#include <stdio.h>
int MT_trace_event(int id)
{
```

```
    struct MT_callerstruct src_info;
    int status;
    unsigned long caller_addr;
    caller_addr = MT_getcalleraddress(0);
    status = MT_getcallerinfo(caller_addr, &src_info);
    if (status == 0) {
      printf("%s was called from %s/%s(%s) line %d\n",
                MT_get_mpi_name(id), src_info.filepath,
                src_info.filename, src_info.funcname,
                src_info.lineno);
    }
    return 1;
}
```

MT_get_elapsed_time

Purpose

Get the elapsed time in seconds between a call to **MPI_Init()** and a call to **MPI_Finalize()**.

Library

-lmpitrace

C Synopsis

#include <mpt.h>
double MT_get_elapsed_time();

Parameters

None

Description

This function returns the elapsed time, in seconds, between a call to **MPI_Init()** and a call to **MPI_Finalize()**.

This function can be useful when you implement your own version of **MT_output_text**().

Returns

This function returns the time, in seconds, between a call to **MPI_Init()** and a call to **MPI_Finalize()**.

Environment Variables

None

Examples

```
#include <stdio.h>
#include <mpt.h>
int MT_output_text()
{
    printf("Time between MPI_Init and MPI_Finalize
is %11.6f seconds\n",
                MT_get_elapsed_time());
    return 0;
}
```

MT_get_environment

Purpose

Returns information about the runtime environment for the application.

Library

-lmpitrace

C Synopsis

#include <mpt.h>
void MT_get_environment(struct MT_envstruct *env);

Parameters

env A pointer to a structure, allocated by the user, which contains information about the application runtime environment.

Description

This function is used to obtain information about the application runtime environment by filling in the MT_envstruct structure allocated by the user. The following fields in the MT_envstruct structure are relevant:

int mpirank	The MPI task rank for this task in the application.
int ntasks	Number of tasks in the MPI application
int nmpi	Maximum index allowed for mpi_id when calling **MT_get_mpi_counts**(), **MT_get_mpi_bytes**(), **MT_get_mpi_time**().

This function can be useful when you implement your own version of **MT_output_text**().

Environment Variables

None

Examples

```
#include <mpt.h>
#include <stdio.h>
int MT_output_text()
{
    MT_envstruct env;
    MT_get_environment(&env);
    printf("MPI task rank is %d\n", env.mpirank);
    return 0;
}
```

MT_get_mpi_bytes

Purpose

Obtain the accumulated number of bytes transferred by a specific MPI function.

Library

-lmpitrace

C Synopsis

#include <mpt.h>
#include <mpi_trace_ids.h>
double MT_get_mpi_bytes(int mpi_ID);

Parameters

mpi_ID An enumeration identifying the MPI function.

Description

The **MT_get_mpi_bytes**() function returns the accumulated number of bytes transferred by this task for all MPI function calls corresponding to the enumeration specified as **mpi_ID**. The **mpi_ID** parameter might be any of the values as specified in the table in the description of the **MT_trace_event**() function, or as specified in the $IHPCT_BASE/include/mpi_trace_ids.h header. However, meaningful results are returned only for MPI functions that either send or receive data.

This function can be useful when you implement your own version of **MT_output_text**().

Returns

This function returns the accumulated number of bytes transferred by this task for the MPI function specified by the **mpi_ID** parameter.

Environment Variables

IHPCT_BASE Path name of the directory in which IBM HPC Toolkit is installed (/usr/lpp/ppe.hpct for AIX, /opt/ibmhpc/ppe.hpct for Linux).

Examples

```
#include <stdio.h>
#include <mpt.h>
#include <mpi_trace_ids.h>
int MT_output_text()
{
    printf("Total bytes sent using MPI_Send: %f11.6
bytes\n",
                MT_get_mpi_bytes(SEND_ID));
    return 0;
```

```
}
```

MT_get_mpi_counts

Purpose

Obtain the number of times the specified MPI function was called.

Library

-lmpitrace

C Synopsis

#include <mpt.h>
#include <mpi_trace_ids.h>
long long MT_get_mpi_counts(int mpi_ID);

Parameters

mpi_ID An enumeration identifying the MPI function.

Description

The **MT_get_mpi_counts**() function returns the number of times the specified MPI function was called in this task. The **mpi_ID** parameter might be any of the values as specified in the table in the description of the **MT_trace_event**() function, or as specified in the $IHPCT_BASE/include/mpi_trace_ids.h header.

This function can be useful when you implement your own version of **MT_output_text**().

Returns

This function returns the number of times the MPI function, specified by the **mpi_ID** parameter, was called in this task.

Environment Variables

IHPCT_BASE Path name of the directory in which IBM HPC Toolkit is installed (/usr/lpp/ppe.hpct for AIX, /opt/ibmhpc/ppe.hpct for Linux).

Examples

```
#include <stdio.h>
#include <mpt.h>
#include <mpi_trace_ids.h>
int MT_output_text()
{
    printf("MPI_Send called %lld times\n",
                MT_get_mpi_counts(SEND_ID));
    return 0;
}
```

MT_get_mpi_name

Purpose

Return the name of the specified MPI function.

Library

-lmpitrace

C Synopsis

#include <mpt.h>
#include <mpi_trace_ids.h>
char *MT_get_mpi_name(int mpi_ID);

Parameters

mpi_ID An enumeration identifying the MPI function.

Description

The **MT_get_mpi_name**() function returns the name of the specified MPI function. The **mpi_ID** parameter might be any of the values as specified in the table in the description of the **MT_trace_event**() function, or as specified in the $IHPCT_BASE/include/mpi_trace_ids.h header.

This function can be useful when you implement your own version of **MT_output_text**().

Returns

This function returns the name of the MPI function specified by the **mpi_ID** parameter.

Environment Variables

IHPCT_BASE Path name of the directory in which IBM HPC Toolkit is installed (/usr/lpp/ppe.hpct for AIX, /opt/ibmhpc/ppe.hpct for Linux).

Examples

```
#include <stdio.h>
#include <mpt.h>
#include <mpi_trace_ids.h>
int MT_output_text()
{
    printf("%s called %lld times\n",
               MT_get_mpi_name(SEND_ID),
               MT_get_mpi_counts(SEND_ID));
    return 0;
}
```

MT_get_mpi_time

Purpose

Obtain the elapsed time, in seconds, spent in the specified MPI function.

Library

-lmpitrace

C Synopsis

#include <mpt.h>
#include <mpi_trace_ids.h>
double MT_get_mpi_time(int mpi_ID);

Parameters

mpi_ID An enumeration identifying the MPI function.

Description

The **MT_get_mpi_time**() function returns the elapsed time spent in the specified MPI function in this task. The **mpi_ID** parameter might be any of the values as specified in the table in the description of the **MT_trace_event**() function, or as specified in the $IHPCT_BASE/include/mpi_trace_ids.h header.

This function can be useful when you implement your own version of **MT_output_text**().

Returns

This function returns the elapsed time spent, in seconds, in this task in the MPI function specified by the **mpi_ID** parameter.

Environment Variables

IHPCT_BASE Path name of the directory in which IBM HPC Toolkit is installed (/usr/lpp/ppe.hpct for AIX, /opt/ibmhpc/ppe.hpct for Linux).

Examples

```
#include <stdio.h>
#include <mpt.h>
#include <mpi_trace_ids.h>
int MT_output_text()
{
    printf("MPI_Send spent %f11.6 seconds elapsed
time\n",
                MT_get_mpi_time(SEND_ID));
    return 0;
}
```

MT_get_time

Purpose

Get the elapsed time, in seconds, since **MPI_Init()** was called.

Library

-lmpitrace

C Synopsis

#include <mpt.h>
double MT_get_time();

Parameters

None

Description

This function returns the time, in seconds, since **MPI_Init()** was called.

This function can be useful when you implement your own version of **MT_output_text**().

Returns

This function returns the time, in seconds, since **MPI_Init()** was called.

Environment Variables

None

Examples

```
#include <stdio.h>
#include <mpt.h>
int MT_output_text()
{
    printf("MPI_Init was called %f11.6 seconds ago.\n",
               MT_get_time());
    return 0;
}
```

MT_get_tracebufferinfo

Purpose

Obtains information about MPI trace buffer usage by the MPI trace library.

Library

-lmpitrace

C Synopsis

#include <mpt.h>
int MT_get_tracebufferinfo(struct MT_tracebufferstruct *info);

Parameters

info A pointer to an MT_tracebufferstruct structure allocated by the user, and which contains the returned results from this function.

Description

This function obtains information about the internal MPI trace buffer used by the MPI trace library. This function fills in an MT_tracebufferstruct allocated by the user, where the following fields in this structure are relevant.

int number_events	Number of MPI trace events currently recorded in this buffer.
double total_buffer	MPI trace buffer size, in megabytes.
double used_buffer	Amount of trace buffer used, in megabytes.
double free_buffer	Remaining free space in buffer, in megabytes.

Returns

This function returns **0** on successful completion and returns a nonzero value on failure.

Environment Variables

None

Examples

```
#include <mpt.h>
#include <stdio.h>
int MT_output_text()
{
    MT_tracebufferstruct info;
    MT_get_tracebufferinfo(&info);
    printf("%d MPI events were recorded\n",
           info.number_events);
    printf("%11.6fMB of %11.6fMB trace buffer used\n",
           info.used_buffer, info.total_buffer);
    return 0;
}
```

MT_output_text

Purpose

Generates the performance statistics for your application when your application calls **MPI_Finalize()**.

Library

-lmpitrace

C Synopsis

#include <mpt.h>
int MT_output_text();

Parameters

None

Description

This function generates performance statistics for your application. The MPI trace library calls this function when **MPI_Finalize()** is called in your application. You can override the default behavior of this function, generating a summary of MPI performance statistics, by implementing your own version of **MT_output_text**() and linking it with your application.

Returns

This function returns **-1** if an error occurs. Otherwise this function returns **1**.

Environment Variables

None

Examples

```
#include <mpt.h>
#include <mpi_trace_ids.h>
#include <stdio.h>
int MT_output_text()
{
    struct MT_summarystruct send_info;
    struct MT_summarystruct recv_info;
    MT_get_allresults(ELAPSEDTIME, SEND_ID, &send_info);
    MT_get_allresults(ELAPSEDTIME, RECV_ID, &recv_info);
    printf("MPI_Send task with min. elapsed time: %d\n",
              send_info.min_rank);
    printf("MPI_Send task with max. elapsed time: %d\n",
              send_info.max_rank);
    printf("MPI_Recv task with min. elapsed time: %d\n",
              recv_info.min_rank);
```

```
    printf("MPI_Recv task with max. elapsed time: %d\n",
              recv_info.max_rank);
    return 0;
}
```

MT_output_trace

Purpose

This function controls whether an MPI trace file is created for a specific task.

Library

-lmpitrace

C Synopsis

#include <mpt.h>
int MT_output_trace(int rank);

Parameters

rank The MPI task rank of this task.

Description

This function controls whether an MPI trace file is generated for a specific MPI task. You can override the default MPI trace library behavior of generating a trace file for all tasks by implementing your own version of this function and linking it with your application. The MPI trace library calls your implementation of this function as part of its processing when your application calls **MPI_Finalize()**.

Returns

This function returns **0** if an MPI trace file is not to be generated for the MPI task from which this function is called. This function returns **1** if an MPI trace file is to be generated for the MPI task from which this function is called.

Environment Variables

None

Examples

```
#include <mpt.h>
int MT_output_trace(int rank)
{
    /* Generate trace files for even rank tasks only */
  if ((rank % 2) == 0) {
    return 1;
  }
  else {
    return 0;
  }
}
```

MT_trace_event

Purpose

Control whether an MPI trace event is generated for a specific MPI function call.

Library

-lmpitrace

C Synopsis

```
#include <mpt.h>
#include <mpi_trace_ids.h>
int MT_trace_event(int mpi_ID)
```

Parameters

mpi_ID An enumeration identifying the MPI function that is about to be traced.

Description

The **MT_trace_event**() function controls whether the MPI trace library should generate a trace event for an MPI function call. The default behavior of the MPI trace library is to generate trace events for all MPI function calls defined in the mpi_trace_ids.h header. You override the default MPI trace library behavior by implementing your own version of this function and linking it with your application. The MPI trace library calls your implementation of this function each time the MPI trace library is about to generate a trace event. You can control collection of MPI trace events for any function with an identifier in the following list.

COMM_SIZE_ID	COMM_RANK_ID	SEND_ID
SSEND_ID	RSEND_ID	BSEND_ID
ISEND_ID	ISSEND_ID	IRSEND_ID
IBSEND_ID	SEND_INIT_ID	SSEND_INIT_ID
RSEND_INIT_ID	BSEND_INIT_ID	RECV_INIT_ID
RECV_ID	IRECV_ID	SENDRECV_ID
SENDRECV_REPLACE_ID	BUFFER_ATTACH_ID	BUFFER_ATTACH_ID
PROBE_ID	IPROBE_ID	TEST_ID
TESTANY_ID	TESTALL_ID	TESTSOME_ID
WAIT_ID	WAITANY_ID	WAITALL_ID
WAITSOME_ID	START_ID	STARTALL_ID
BCAST_ID	BARRIER_ID	GATHER_ID
GATHERV_ID	SCATTER_ID	SCATTERV_ID
SCAN_ID	ALLGATHER_ID	ALLGATHERV_ID
REDUCE_ID	ALLREDUCE_ID	REDUCE_SCATTER_ID
ALLTOALL_ID	ALLTOALLV_ID	

Returns

1 if a trace event should be generated for the MPI function call, **0** if no trace event should be generated.

Environment Variables

See MT_trace_start

Examples

```
#include <mpt.h>
#include <mpi_trace_ids.h>
int MT_trace_event(int id)
{
    /* Trace only MPI_Send and MPI_Recv calls */
    if ((id == RECV_ID) || (id == SEND_ID)) {
      return 1;
    }
    else {
       return 0;
    }
}
```

MT_trace_start, mt_trace_start

Purpose

Start or resume collection of trace events for all MPI calls.

Library

-lmpitrace

C Synopsis

#include <mpt.h>
void MT_trace_start()

Fortran Synopsis

subroutine mt_trace_start()

Parameters

None

Description

MT_trace_start() is used to start collection of MPI trace events or to resume collection of MPI trace events, if collection of these events has been suspended by a call to **MT_trace_stop**(). The environment variable TRACE_ALL_EVENTS must be set to **no** for **MT_trace_start**() to have any effect.

Environment Variables

IHPCT_BASE	Path name of the directory in which IBM HPC Toolkit is installed (/usr/lpp/ppe.hpct for AIX, /opt/ibmhpc/ppe.hpct for Linux).
MAX_TRACE_EVENTS	Specifies the maximum number of trace events that can be collected per task. The default is **30000**.
MAX_TRACE_RANK	Specifies the MPI task rank of the highest rank process that has MPI trace events collected. The default is **256**.
OUTPUT_ALL_RANKS	Set to **yes** to generate trace files for all MPI tasks. The default is to generate trace files only for task **0** and the tasks that have the minimum, maximum and median total MPI communication time. If task **0** is the task with minimum, maximum, or median communication time, only three trace files are generated by default.
TRACE_ALL_EVENTS	Set to **yes** to generate a trace containing trace events for all MPI calls after **MPI_Init()**. If this environment variable is set to **no**, collection of MPI trace events is controlled by **MT_trace_start**() and **MT_trace_stop**(). The default is **yes**.
TRACE_ALL_TASKS	Set to **yes** to generate MPI trace files for all MPI tasks in the application. The default is **no,** which results in

	generating trace files only for MPI tasks **0** through **255**.
TRACEBACK_LEVEL	Specifies the number of levels to walk back in the function call stack when recording the address of an MPI call. This can be used to record profiling information for the caller of an MPI function rather than the MPI function itself, which might be useful if the MPI functions are contained in a library. The default is **0.**

Examples

```
#include <mpt.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MT_trace_start();
      /* MPI communication region of interest */
    MT_trace_stop();
      /* MPI communication region of no interest */
    MPI_Finalize();
}
```

```
program main
include ‘mpif.h’
call mpi_init()
call mt_trace_start()
! MPI communication region of interest
call mt_trace_stop()
! MPI communication region of no interest
call mpi_finalize()
end
```

MT_trace_stop, mt_trace_stop

Purpose

Suspend collection of trace events for all MPI calls.

Library

-lmpitrace

C Synopsis

#include <mpt.h>
void MT_trace_stop()

Fortran Synopsis

subroutine mt_trace_stop()

Parameters

None

Description

MT_trace_stop() is used to suspend collection of MPI trace. The environment variable TRACE_ALL_EVENTS must be set to **no** for **MT_trace_stop**() to have any effect. **MT_trace_start**() might be called after a call to **MT_trace_stop**() to resume collection of MPI trace events.

Environment Variables

See **MT_trace_start**()

Examples

```
#include <mpt.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MT_trace_start();
      /* MPI communication region of interest */
    MT_trace_stop();
      /* MPI communication region of no interest */
    MPI_Finalize();
}

program main
include ‘mpif.h’
call mpi_init()
call mt_trace_start()
! MPI communication region of interest
```

```
call mt_trace_stop()
! MPI communication region of no interest
call mpi_finalize()
end
```

Application Instrumentation

hpctInst

Instrument applications in order to obtain performance data.

Synopsis

hpctInst [-dhpm] [-dhpm_func_call <file name>] [-dhpm_region <file name>]
[-dmpi] [-dmpi_func_call <file name>] [-dmpi_region <file name>]
[-dmio] [-dpomp_parallel {none | EnterExit | BeginEnd }]
[-dpomp_loop {none | EnterExit | Chunks}]
[-dpomp_user {none | BeginEnd}] [-dpomp_userfunc <file name>]
[-dlink <args>] <binary>
hpctInst [-h | --help]

Flags

-dhpm	Instrument all function entry and exit points with HPM instrumentation.
-dhpm_func_call <file name>	Instrument function call sites with HPM instrumentation, as specified by the contents of <*filename*>.
-dhpm_region <file name>	Instrument regions of code with HPM instrumentation, as specified by the contents of <*filename*>.
-dlink <args>	Specify additional libraries to link with the application, for AIX only. If your application uses shared libraries other than libpmapi, libpthreads or libxlsmp, you must specify those additional shared libraries using this option.
-dmpi	Instrument all MPI calls in the application with MPI profiling instrumentation.
-dmpi_func_call <file name>	Instrument MPI calls in functions at locations in the application, as specified by <*filename*>.
-dmpi_region <file name>	Instrument MPI calls in regions of code in the application, as specified by <*filename*>.
-dmio	Instrument all I/O calls in the application for I/O profiling.
-dpomp	Instrument all OpenMP parallel loop and parallel region constructs in the application for OpenMP profiling.
-dpomp_loop {none | EnterExit | Chunks}	Instrument parallel loops for OpenMP profiling.
-dpomp_parallel {none | EnterExit | BeginEnd}	Instrument parallel regions for OpenMP profiling.
-dpomp_user	Instrument user functions for OpenMP profiling.

{none | BeginEnd}	
-dpomp_userfunc <file name>	Instrument the user functions specified in <*filename*> for OpenMP profiling.
-h, --help	Display an **hpctInst** usage message.

Description

The **hpctInst** command is used to rewrite an application with instrumentation as specified by command line flags and environment variables. The instrumented binary is written to a file called <*binary*>.**inst** in the current working directory. After an instrumented application has been created, set the appropriate environment variables for the instrumentation you have requested, then run the application.

If you invoke **hpctInst** with the **–dhpm_func_call** or the **–dmpi_func_call** flag, the format of each line in the file specified by <*filename*> is one of:

called_func [inst_function]
called_func [file_name [start_line [end_line]]]

where:

- **called_func** is the name of the function being called. For the **–dmpi_func_call** option, **called_func** is the name of an MPI function.
- **inst_function** is the name of the only function in which calls to **called_func** are instrumented.
- **file_name** is the name of the only source file in which calls to **called_func** are instrumented.
- **start_line** is the first line number in file name in which calls to **called_func** are instrumented.
- **end_line** is the ending line number in file name in which calls to **called_func** are instrumented.

If **file_name** is specified, and both **start_line** and **end_line** are omitted, all calls to **called_func** in **file_name** are instrumented. The same **called_func** might be specified in one or more lines in this file.

If you invoke **hpctInst** with the **–dhpm_region** or **–dmpi_region** flags, the format of the each line in the file specified by <*file_name*> is:

file_name start_line end_line

where:

- **file_name** is the name of the source file.
- **start_line** is the starting line number in **file_name** that will be instrumented.
- **end_line** is the ending line number in **file_name** that will be instrumented.

The meanings of the OpenMP instrumentation flags, **-dpomp_***, are:

none	No data is collected for this construct

EnterExit	Data is collected for parallel region entry and exit and for loop entry and exit.
BeginEnd	Data is collected at begin and end of a parallel region. EnterExit data is also collected.
Chunks	Data is collected for each parallel execution of the region or loop

If your application does not reside in a global filesystem, copy the instrumented binary to all nodes on which it will run. If you instrument an application for HPM, **hpctInst** creates a file named **.psigma.hpmhandle** in the same directory as the instrumented binary. If you instrument an application for OpenMP profiling, **hpctInst** creates a file named **.psigma.dpomphandle** in the same directory as the instrumented binary. You must ensure these files are accessible in the current working directory on all nodes on which the instrumented application will execute.

The application being instrumented must be compiled and linked with the **–g** option. If you have a 64-bit Linux application, it must also be linked with the **–emit-stub-syms** option.

Environment Variables

IHPCT_BASE	Path name of the directory in which IBM HPC Toolkit is installed (/usr/lpp/ppe.hpct for AIX, /opt/ibmhpc/ppe.hpct for Linux).
LD_LIBRARY_PATH	Must be set to $IHPCT_BASE/lib when instrumenting 32-bit applications on Linux. Must be set to $IHPCT_BASE/lib64 when instrumenting 64-bit applications on Linux.
POMP_LOOP	Specifies the level of OpenMP profiling instrumentation for OpenMP parallel regions. Values can be **none**, **EnterExit**, or **Chunks**. The **–dpomp_loop** flag overrides this environment variable.
POMP_PARALLEL	Specifies the level of OpenMP profiling instrumentation for OpenMP parallel regions. Values can be **none**, **EnterExit**, or **BeginEnd**. The **–dpomp_parallel** flag overrides this environment variable.
POMP_USER	Specifies the level of OpenMP profiling for user functions. Values might be **none** or **EnterExit**. The **–dpomp_**user flag overrides this environment variable.

Examples

To instrument all MPI calls in an application:

```
hpctInst –dmpi testprog
```

To instrument call sites for function **testfunc** from file **main.c** lines 1 through 100 with HPM instrumentation:

```
hpctInst –dhpm_func_call inst_spec testprog
```

where the **inst_spec** file contains the single line

```
testfunc main.c 1 100
```

To instrument all parallel loops and parallel regions in an OpenMP application with **EnterExit** instrumentation:

```
hpctInst –dpomp_loop –dpomp_parallel testprog
```

Performance Data Visualization

dataview

View a trace file generated by the IBM HPC Toolkit Modular I/O (MIO) tool.

Synopsis

dataview [<event_trace> [<event_trace> …]]
dataview [-h | --help]

Flags

-h, --help Display a **dataview** usage message.

Description

The **dataview** GUI can be used to view a trace file generated by the IBM HPC Toolkit I/O Profiling tool. The trace is displayed in a graph format, where in the initial display, the x axis represents elapsed time and the y axis represents file position. You can zoom in and out of this graph display and display I/O transfer rates for selected areas of the graph. You can also display the MIO trace in tabular format and use the tabular view to customize the MIO graph to show additional data.

When you start **dataview**, you can specify an MIO trace file to be viewed, or you can invoke **dataview** without specifying an MIO trace file to open. If you do not specify an MIO trace file, you can do so by selecting the **Read Event** option from the **File** menu in the initial **dataview** window.

When **dataview** starts, and an MIO trace file is loaded, **dataview** displays a window containing a tree view showing the instrumented I/O calls in the application that generated the trace file. You can select one or more of the I/O calls from this tree. If **dataview** was invoked from within **peekperf**, by selecting **View Tracer** from the Data Visualization Window pop-up menu that appears when right-clicking in that window, then when you select an I/O function call from the tree view, **peekperf** positions the source view window to show the I/O call that you selected.

After you select I/O calls, and you select **New Plot** at the top of the tree view, the graph is displayed. You can zoom in and out of the graph using the **Zoom In** and **Zoom Out** buttons at the top of the graph window. You can save the graph as a jpg file by clicking the **Save** button, and print the graph by clicking the **Print** button. You can also zoom in the graph by left-clicking and dragging a rectangle around the area of the graph into which you want to zoom in to.

You can show the slope (data transfer rate) for file I/O by right-clicking over a part of the plot and dragging to draw a line over the plot. The slope of that line is displayed, along with starting and ending coordinates, in the status area at the bottom of the graph. If you trace over a part of the plot while right-clicking, the slope is the data rate in bytes per second for the traced fragment of the graph.

You can display the trace in tabular format by selecting one or more I/O calls from the tree view then clicking **Edit Table** in the **File** menu in the tree view. A table opens for each I/O call, showing the data for each time that I/O call was executed. You can select a column to sort the table by clicking over a column heading. If you right-click over a column, a pop-up menu appears in which you can hide displayed columns, show hidden columns, or save the table to a file in CSV (spreadsheet) format.

You can customize an MIO graph by making selections from the widgets at the top of the data view table. These selections take effect the next time you click **Plot** in a graph window or select **New Plot** from the tree window. There are four widgets that you can use to customize the graph.

The first is the color icon. If you select this icon, a color selector dialog appears, in which you can select the color of the displayed data.

The second widget selects the metric used for the y axis of the graph. You can select file position activity, data delivery rate, or rate vs. position. If you select file position activity, data is displayed in the graph as individual data points. If you select data delivery rate or rate vs. position, data is displayed in the graph as numerical values at the locations at which points would be plotted.

The third widget specifies how many pixels are used to draw each data point.

The fourth widget specifies the metric whose numerical value is to be displayed at each point in the graph.

Environment Variables

IHPCT_BASE Path name of the directory in which IBM HPC Toolkit is installed (/usr/lpp/ppe.hpct for AIX, /opt/ibmhpc/ppe.hpct for Linux).

Examples

peekperf

Provide a GUI interface to instrument an application and to view application performance data.

Synopsis

peekperf [-num max_src_files] [<binary>] [[<vizfile>] …]
peekperf [-h | --help]

Flags

-num	Specifies the maximum number of source files that can be opened without being prompted to select a set of files. The default limit is **15** files.
-h, --help	Display a usage message for **peekperf**.

Description

The **peekperf** GUI is the control center of the HPC Toolkit. It allows you to control the instrumentation, execute the application, and visualize and analyze the collected performance data within the same user interface. The **peekperf** GUI can display the data in the visualization (*.viz) files from the various instrumentation libraries. If more than one visualization file is specified, **peekperf** combines the data from them for display. **Peekperf** also provides filtering and sorting capabilities to help you analyze the data.

Peekperf supports gathering the following types of performance data:

- Hardware performance counters and resource utilization (HPM)
- MPI profiling and trace (MPI)
- OpenMP profiling (OpenMP)
- I/O profiling (MIO)

If you specify the application to be instrumented, using the <*binary*> parameter, **peekperf** parses the specified application executable to identify possible instrumentation points. If you do not specify the application executable on the command line, you can specify the application executable once **peekperf** has started by selecting the **Open Binary** selection from the **File** menu.

If you specify the visualization file(s) on the command line, **peekperf** attempts to load the visualization files as it starts up. You can also load visualization files by selecting the **Open Performance Data** selection from the **File** menu.

The **peekperf** GUI has three main windows, the data collection window, the data visualization window, and the source code window.

The data collection window opens when you open an executable, either by specifying it on the command line, or by selecting **Open Binary** from the **File** menu. The data collection window shows you separate tabs for each type of instrumentation

data you can collect. Each tab shows a window containing a representation of the application structure and the locations within the application that you can instrument for each type of data. You can expand or collapse nodes in this tree, as needed, and select or deselect the desired instrumentation points. If you right-click on a node, a pop-up menu appears in which you can select additional options, including options specific to the type of instrumentation.

The data visualization window shows a summary of the instrumentation data files that have been loaded. Instrumentation data files (visualization files) can be opened by specifying them on the **peekperf** command line, by selecting **Open Performance Data** from the **File** menu, or they will open automatically after an instrumented application completes. You can expand or collapse the tree to view data. You can sort data by clicking on a column heading. You can view additional detail in a metrics browser for a leaf node row in the tree by right-clicking over the leaf node row. If you right-click anywhere in the window other than over a leaf node row, a pop-up menu appears where you can apply filters to eliminate uninteresting rows from the view or to open a table or trace viewer appropriate to the type of visualization data.

The source code window appears when an executable is loaded or by clicking the **Open Sources** option in the **File** menu. The source code window shows a tabbed display of individual source files in the application. As you select instrumentation points in the data collection window, or select rows in the data visualization window, the corresponding lines of code is highlighted in the source code window. You can select regions of code to be instrumented by the HPM tool by clicking and dragging over the region of interest, then right-clicking and selecting **Add to HPM** from the pop-up menu.

It is necessary to compile the application with the **–g** options in order to map the performance data back to source code. This allows you to more easily find bottlenecks and points for optimizations.

Environment Variables

IHPCT_BASE Path name of the directory in which IBM HPC Toolkit is installed (/usr/lpp/ppe.hpct for AIX, /opt/ibmhpc/ppe.hpct for Linux).

Examples

To invoke **peekperf** without loading any performance data or parsing an application executable:
peekperf

To invoke **peekperf**, specifying an application executable to be parsed for instrumentation:
peekperf myapp

To invoke **peekperf** to view performance data:

peekperf myapp.viz

peekview

Display an MPI trace file generated by an application that is instrumented by the IBM HPC Toolkit to obtain MPI profiling data.

Synopsis

peekview [[-b | –bright] | [-d | --dark]] <tracefile> [<tracefile>] …

Flags

-b, --bright	Display the MPI trace with a bright background.
-d, --dark	Display the MPI trace with a dark background. This is the default display option
-h, --help	Display a **peekview** usage message

Description

The **peekview** GUI displays one or more MPI trace files generated using the MPI profiling tool in the IBM HPC Toolkit. Traces are displayed in a timeline format with the x axis representing elapsed time and the y axis representing MPI task index. Each unique MPI function is displayed in a different color so individual MPI function calls can be easily identified.

The **peekview** GUI displays a second window, the identifier window, which you can use to enable or disable the display of MPI events by MPI function type, so only the desired MPI function types are visible.

You can scroll through the trace display by using the scrollbars on the trace window. You can also zoom in and out of the trace by using the zoom icons at the top of the trace window, or by clicking and dragging a rubber band box around a region of interest in the trace window.

If you left-click over a trace event, some information about that event is displayed in the status area at the bottom of the trace window. If you right-click over a trace event, a pop-up message with additional information about that event appears.

If **peekview** is invoked from within **peekperf**, by selecting the **View Tracer** option from the Data Visualization Window pop-up menu that appears when right-clicking in that window, then when you left-click on a trace event in the trace window, **peekperf** scrolls its source code window to display the line of source code corresponding to the MPI function call.

By default, **peekview** displays the trace using a dark background (the **–d** option). If the **–b** option is specified, each MPI event is drawn with a black rectangle around it, making it easier to see short duration MPI events. However, if there are many events in the MPI trace display, the **–b** option could result in a distorted view of the MPI trace.

Environment Variables

IHPCT_BASE Path name of the directory in which IBM HPC Toolkit is installed (/usr/lpp/ppe.hpct for AIX, /opt/ibmhpc/ppe.hpct for Linux).

Examples

Display an MPI trace with a dark background:
peekview –d single_trace

Xprof

Xprof is a GUI-based performance profiling tool that displays the application as a function tree based on the runtime call structure and profiling statistics.

Synopsis

Xprof [program] [-b] [-s] [-z] [-a <path>] [-c <file>] [-L <pathname>]
[[-e <function>]...] [[-E <function>]...] [[-f <function>]...]
[[-F function]...] [-disp_max number_of_functions] [[**gmon.out**]...]
Xprof -h | -help

Flags

-a	To specify an alternate search path or paths for library files and source code files. If more than one path is specified, the paths must be embraced by a comma (,) and each path should be separated by either a colon (:) or a space.
-b	Suppresses 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.
-c	Loads a configuration file that contains information to be used to determine which functions will be displayed when **Xprof** is brought up.
-disp_max	Sets the number of function boxes that **Xprof** initially displays in the function call tree. The value supplied with this flag can be any integer between **0** and **5,000**. **Xprof** displays the function boxes for the most processor-intensive functions through the number you specify.
-e	De-emphasizes the general appearance of the function box or boxes for the specified functions in the function call tree, and limits the number of entries for these functions in the Call Graph Profile report. This also applies to the specified function's descendants, as long as they have not been called by nonspecified functions. In the function call tree, the function boxes for the specified functions appear grayed out. Its 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 nonspecified functions. 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 nonspecified 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 nonspecified function in the program.
-E	Changes the general appearance and label information of the function box or boxes for the specified functions in the function call tree. Also limits the number of entries for these functions in the Call Graph Profile report, and changes the processor data associated with them. These results also apply to the specified function's descendants, as long as they

have not been called by nonspecified functions in the program. In the function call tree, the function box for the specified function appears grayed out, and its size and shape also changes so that it appears as a square of the smallest allowable size. In addition, the processor time shown in the function box label appears as **0** (zero). The same applies to function boxes for descendant functions, as long as they have not been called by nonspecified functions. This option also causes the processor time spent by the specified function to be deducted from the left side processor total 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 nonspecified function as its parent. When this is the case, the time in the self and descendants columns for this entry is set to **0** (zero). In addition, the amount of time that was in the descendant's column for the specified function is subtracted from the time listed under the descendant's 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.

-f De-emphasizes the general appearance of all function boxes in the function call tree, except for that of the specified function(s) and its descendant(s). In addition, the number of entries in the Call Graph Profile report for the nonspecified functions and nondescendant 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(s) and it descendant(s) appear grayed out. The size of these boxes and the content of their labels remain the same. For the specified function(s), and its descendants, the appearance of the function boxes and labels remain the same. In the Call Graph Profile report, an entry for a nonspecified or nondescendant 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.

-h, --help Writes the **Xprof** usage message to STDERR and then exits.

-L Uses an alternate path name for locating shared libraries. If you plan to specify multiple paths, use the Set File Search Paths option of the File menu on the **Xprof** GUI.

-s If multiple **gmon.out** files are specified when **Xprof** is started, produces the **gmon.sum** profile data file. 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.

-z Includes functions that have both zero processor usage and no call counts in the Flat Profile, Call Graph Profile, and Function Index reports. A function does not have a call count if the file that contains its definition was not compiled with the **-pg** option, which is common with system library files.

Description

The **Xprof** command is used to analyze the performance of both serial and parallel applications at the function, source line, and machine code levels. **Xprof** uses data collected after running a program that is compiled and linked with the -pg option. It presents a graphical representation of the application functions in addition to providing textual data in several report windows. These presentation formats are intended to allow easy identification of the functions that are most processor-intensive.

Functions are represented by solid green boxes or nodes in the function call tree. The size and shape of each function node indicates its CPU utilization. The height of each node represents the amount of CPU time it spent on executing itself. The width of each node represents the amount of CPU time it spent executing itself, plus its descendant functions. More detail such as source code mapping or statistical data can be displayed by selecting an individual function.

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. Selecting an arc shows the number of calls, the callee and the caller.

The call graph can be filtered in a variety of ways including by CPU time, call counts and function name. It can also be pruned or expanded based on library, ancestor and descendant relationship.

Limitations

If you compile your application on one processor, and 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.

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

On AIX systems, **Xprof** supports only the old format (prior to 5.3) **gmon.out** file; but it is easy to have the application program generate these. Refer to the **GPROF** environment variable below for details.

Environment Variables

GPROF — Set options for generating the **gmon.out** file. To have an effect, this environment variable must be set before running the user application.

```
GPROF=profile:<profile-type>,
    scale:<scaling-factor>,file:<filetype>,
    filename:<file name>
```

<*profile-type*> describes the type of profiling that needs to be performed. This can be either process or thread. Type *process* indicates that profiling granularity is at process level, *thread* indicates that profiling granularity is at thread level.

<*scaling-factor*> describes how much memory need to be allocated for the Call Graph Profile. By default, the scaling factor is 2 for process level profiling and 8 for thread level profiling. A scaling factor of 2 indicates that a memory of 1/2 of the process's size is allocated for every process or thread. A scaling factor of 8 indicates that a memory of 1/8th of the process's size is allocated for every process or thread. This memory is the buffer area used to store the call graph information.

<*file-type*> describes the type of **gmon.out** file required A value of **multi** indicates that one **gmon.out** file per process is required. A value of **multithread** indicates that one **gmon.out** file per thread is required. If an application is profiled with **-pg** option, and it does a fork, specifying **multi** generates one **gmon.out** file for the parent process and another for the child process. The naming convention for the generated **gmon.out** files are as follows:

- For multi file-type: <*prefix*>**-processname-pid.out**
- For multithread file-type: <*prefix*>**-processname-pid-Pthread**<*threadid*>**.out**
-

<*filename*> describes the prefix that needs to be used for the generated **gmon.out** files. By default the prefix is **gmon**.

IHPCT_BASE Path name of the directory in which IBM HPC Toolkit is installed (/usr/lpp/ppe.hpct for AIX, /opt/ibmhpc/ppe.hpct for Linux).

Examples

To use **Xprof**, you must first compile your program (for example, **foo.c**) with **-pg**:

```
xlc -pg -o foo foo.c
```

When the program foo is executed a **gmon.out** file will be generated. To invoke **Xprof**, enter:

```
Xprof foo gmon.out
```

To use **Xprof** on your MPI program, you must first compile your program (for example, **foo.c**) with **–pg**. Note that to be able to map the functions displayed in the call graph by **Xprof** back to the source code, your program must also be compiled with **–g**.

```
mpcc -g -pg -o foo foo.c
```

When the MPI program foo is executed, by default, one **gmon.out**.<*taskid*> file is generated for each task involved in the parallel execution. To invoke **Xprof**, enter:

```
Xprof foo gmon.out.*
```

When invoking **Xprof** with multiple **gmon.out** files, they can be displayed as a summary or average of their performance statistics.

Appendices

Derived Metrics Description

The derived metrics implemented by **hpccount, hpcstat** and **libhpc** are described as follows

Utilization rate
    100.0 * user_time / wall_clock_time

Total FP load and store operations
    fp_tot_ls = (PM_LSU_LDF + PM_FPU_STF) * 0.000001

MIPS
    PM_INST_CMPL * 0.000001 / wall_clock_time

Instructions per cycle
    (double)PM_INST_CMPL / PM_CYC

Instructions per run cycle
    (double)PM_INST_CMPL / PM_RUN_CYC

Instructions per load/store
    (double)PM_INST_CMPL/(PM_LD_REF_L1 + PM_ST_REF_L1)

Percent of Instructions dispatched that completed
    100.0 * PM_INST_CMPL / PM_INST_DISP

Fixed point operations per cycle
    (double)PM_FXU_FIN / PM_CYC

Fixed point operations per load/stores
    (double)PM_INST_CMPL/(PM_LD_REF_L1 + PM_ST_REF_L1)

Branches mispredicted percentage
    100.0* (PM_BR_MPRED_CR + PM_BR_MPRED_TA) / PM_BR_ISSUED

Number of loads per load miss
    (double)PM_LD_REF_L1 / PM_LD_MISS_L1

Number of stores per store miss
    (double)PM_ST_REF_L1 / PM_ST_MISS_L1

Number of load/stores per L1 miss
    ((double)PM_LD_REF_L1 + (double)PM_ST_REF_L1)
    / ((double)PM_ST_MISS_L1 + (double)PM_LD_MISS_L1)

L1 cache hit rate

```
100.0 * (1.0 - ((double)PM_LD_REF_L1 + (double)PM_ST_REF_L1)
/ ((double)PM_ST_MISS_L1 + (double)PM_LD_MISS_L1))
```

Number of loads per TLB miss

```
(double)PM_LD_REF_L1 / PM_DTLB_MISS
```

Number of loads/stores per TLB miss

```
((double)PM_LD_REF_L1 + (double)PM_ST_REF_L1) / PM_DTLB_MISS
```

Total Loads from L2

```
tot_ld_L2 = sum((double)PM_DATA_FROM_L2*) / (1024 * 1024)
```

L2 load traffic

```
L1_cache_line_size * tot_ld_L2
```

L2 load bandwidth per processor

```
L1_cache_line_size * tot_ld_L2 / wall_clock_time
```

Estimated latency from loads from L2

```
( HPM_L2_LATENCY*(double)PM_DATA_FROM_L2
+ HPM_L25_LATENCY*(double)sum(PM_DATA_FROM_L25*)
+ HPM_L275_LATENCY*(double)sum(PM_DATA_FROM_L275*)) *
cycle_time
```

Percent of loads from L2 per cycle

```
100.0 * tot_ld_L2 / PM_CYC
```

Total Loads from local L2

```
tot_ld_l_L2 = (double)PM_DATA_FROM_L2 / (1024 * 1024)
```

Local L2 load traffic

```
L1_cache_line_size * tot_ld_l_L2
```

Local L2 load bandwidth per processor

```
L1_cache_line_size * tot_ld_l_L2 / wall_clock_time
```

Estimated latency from loads from local L2

```
HPM_L2_LATENCY * (double)PM_DATA_FROM_L2 * cycle_time
```

Percent of loads from local L2 per cycle

```
100.0 * (double)PM_DATA_FROM_L2 / PM_CYC
```

Total Loads from L3

```
tot_ld_L3 = sum((double)PM_DATA_FROM_L3*) / (1024*1024)
```

L3 load traffic
    L2_cache_line_size * tot_ld_L3

L3 load bandwidth per processor
    L2_cache_line_size * tot_ld_L2 / wall_clock_time

Estimated latency from loads from L3
    (HPM_L3_LATENCY*(double)PM_DATA_FROM_L3
    + HPM_L35_LATENCY*(double)PM_DATA_FROM_L35) * cycle_time

Percent of loads from L3 per cycle
    100.0 * (double)sum(PM_DATA_FROM_L3*) / PM_CYC

Total Loads from local L3
    tot_ld_l_L3 = (double)PM_DATA_FROM_L3 / (1024 * 1024)

Local L3 load traffic
    L2_cache_line_size * tot_ld_l_L3

Local L3 load bandwidth per processor
    L2_cache_line_size * tot_ld_l_L3 / wall_clock_time

Estimated latency from loads from local L3
    HPM_L3_LATENCY * (double)PM_DATA_FROM_L3 * cycle_time

Percent of loads from local L3 per cycle
    100.0 * (double)PM_DATA_FROM_L3 / PM_CYC

Total Loads from memory
    tot_ld_mem = (double)PM_DATA_FROM_MEM / (1024 * 1024)

Memory load traffic
    L3_cache_line_size * tot_ld_mem

Memory load bandwidth per processor
    L3_cache_line_size * tot_ld_mem / wall_clock_time

Estimated latency from loads from memory
    HPM_MEM_LATENCY * (double)PM_DATA_FROM_MEM * cycle_time

Percent of loads from memory per cycle
    100.0 * (double)PM_DATA_FROM_MEM / PM_CYC

Total Loads from local memory
    tot_ld_lmem = (double)PM_DATA_FROM_LMEM / (1024 * 1024)

Local memory load traffic
    L3_cache_line_size * tot_ld_lmem

Local memory load bandwidth per processor
    L3_cache_line_size * tot_ld_lmem / wall_clock_time

Estimated latency from loads from local memory
    HPM_MEM_LATENCY * (double)PM_DATA_FROM_LMEM * cycle_time

Percent of loads from local memory per cycle
    100.0 (double)PM_DATA_FROM_LMEM / PM_CYC

Percent of TLB misses per cycle
    100.0 * (double)PM_DTLB_MISS / PM_CYC

Percent of TLB misses per run cycle
    100.0 * (double)PM_DTLB_MISS / PM_RUN_CYC

Estimated latency from TLB misses
    HPM_TLB_LATENCY * (double)PM_DTLB_MISS * cycle_time

HW floating point instructions (flips)
    (flips = (double) PM_FPU_FIN) * 0.000001
    -- or --
    (flips = (double)(PM_FPU0_FIN + PM_FPU1_FIN)) * 0.000001

HW floating point instructions per cycle
    flips / PM_CYC

HW floating point instructions per run cycle
    flips / PM_RUN_CYC

HW floating point instr. rate (HW flips / WCT)
    flips * 0.000001 / wall_clock_time

HW floating point instructions / user time
    flips * 0.000001 / user_time

Total floating point operations
    (flops = (double)(PM_FPU0_FIN + PM_FPU1_FIN
    + PM_FPU_FMA - PM_FPU_STF)) * 0.000001

Flop rate (flops / WCT)
    flops * 0.000001 / wall_clock_time

Flops / user time
flops * 0.000001 / user_time

Algebraic floating point operations
(aflops = (double)(PM_FPU_1FLOP + 2 * PM_FPU_FMA)) * 0.000001

Algebraic flop rate (flops / WCT)
aflops * 0.000001 / wall_clock_time

Algebraic flops / user time
aflops * 0.000001 / user_time

Weighted Floating Point operations
(wflops = flops + (HPM_DIV_WEIGHT-1) * PM_FPU_FDIV) * 0.000001

Weighted flop rate (flops / WCT)
wflops * 0.000001 / wall_clock_time

Weighted flops / user time
wflops * 0.000001 / user_time

FMA percentage
100.0 * 2*PM_FPU_FMA / flops (on POWER4)
100.0 * 2*PM_FPU_FMA / aflops (on POWER5)
100.0 * 2*PM_FPU_FMA / flops (on POWER6)

Computation intensity
flops / fp_tot_ls

Percent of peak performance
100.0 * flops * cycle_time / (4*user_time) (on POWER4)
100.0 * aflops * cycle_time / (4*user_time) (on POWER5)
100.0 * flops * cycle_time / (4*user_time) (on POWER6)

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 grant 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.

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

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

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

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS 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.

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

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

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

For AIX:
IBM Corporation
Department LRAS, Building 003
11400 Burnet Road
Austin, Texas 78758-3498
U.S.A

For Linux:
IBM Corporation
Department LJEB/P905
2455 South Road
Poughkeepsie, NY 12601-5400
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.

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

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or

any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

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

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples 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.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

(C) (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. (C) Copyright IBM Corp. _enter the year or years_. All rights reserved.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol ((R) or (TM)), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

PostScript is either a registered trademark or trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

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

NOTICES AND INFORMATION

IBM Parallel Environment for AIX, V5.1
IBM Parallel Environment for Linux, V5.1

The IBM license agreement and any applicable information on the web download page for IBM products refers You to this file for details concerning notices applicable to code included in the products listed above or otherwise identified as Excluded Components in the License Information document for the above-listed products ("the Program").

Notwithstanding the terms and conditions of any other agreement You may have with IBM or any of its related or affiliated entities (collectively "IBM"), the third party software code identified below are "Excluded Components" and are subject to the terms and conditions of the License Information document accompanying the Program and not the license terms that may be contained in the notices below. The notices are provided for informational purposes.

Please note: This Notices file may identify information or Excluded Components listed in the agreements for the Program that are not used by, or that were not shipped with, the Program as You installed it.

IMPORTANT: IBM does not represent or warrant that the information in this NOTICES file is accurate. Third party websites are independent of IBM and IBM does not represent or warrant that the information on any third party web site referenced in this NOTICES file is accurate. IBM disclaims any and all liability for errors and omissions or for any damages accruing from the use of this NOTICES file or its contents, including without limitation URLs or references to any third party websites.

The following are Excluded Components:
FreeType 2.3.4
libpng 1.2.16
zlib 1.2.3

IBM Parallel Environment V5
---
zlib 1.2.3

/* zlib.h -- interface of the 'zlib' general purpose compression library
version 1.2.3, July 18th, 2005

Jean-loup Gailly jloup@gzip.org
Mark Adler madler@alumni.caltech.edu

*/

****

libpng 1.2.16

distributed according to the same disclaimer and license as libpng-1.0.6
with the following individuals added to the list of Contributing Authors

Simon-Pierre Cadieux
Eric S. Raymond
Gilles Vollant

and with the following additions to the disclaimer:

There is no warranty against interference with your enjoyment of the
library or against infringement. There is no warranty that our
efforts or the library will fulfill any of your particular purposes
or needs. This library is provided with all faults, and the entire
risk of satisfactory quality, performance, accuracy, and effort is with
the user.

libpng versions 0.97, January 1998, through 1.0.6, March 20, 2000, are
Copyright (c) 1998, 1999 Glenn Randers-Pehrson, and are
distributed according to the same disclaimer and license as libpng-0.96,
with the following individuals added to the list of Contributing Authors:

Tom Lane
Glenn Randers-Pehrson
Willem van Schaik

libpng versions 0.89, June 1996, through 0.96, May 1997, are
Copyright (c) 1996, 1997 Andreas Dilger
Distributed according to the same disclaimer and license as libpng-0.88,
with the following individuals added to the list of Contributing Authors:

John Bowler
Kevin Bracey
Sam Bushell
Magnus Holmgren
Greg Roelofs
Tom Tanner

libpng versions 0.5, May 1995, through 0.88, January 1996, are
Copyright (c) 1995, 1996 Guy Eric Schalnat, Group 42, Inc.

For the purposes of this copyright and license, "Contributing Authors"
is defined as the following set of individuals:

Andreas Dilger
Dave Martindale
Guy Eric Schalnat

****

FreeType 2.3.4

The FreeType Project LICENSE

2006-Jan-27

Introduction
==========

The FreeType Project is distributed in several archive packages; some of them may contain, in addition to the FreeType font engine, various tools and contributions which rely on, or relate to, the FreeType Project.

This license applies to all files found in such packages, and which do not fall under their own explicit license. The license affects thus the FreeType font engine, the test programs, documentation and makefiles, at the very least.

This license was inspired by the BSD, Artistic, and IJG (Independent JPEG Group) licenses, which all encourage inclusion and use of free software in commercial and freeware products alike. As a consequence, its main points are that:

- We don't promise that this software works. However, we will be interested in any kind of bug reports. (`as is' distribution)

- You can use this software for whatever you want, in parts or full form, without having to pay us. (`royalty-free' usage)

- You may not pretend that you wrote this software. If you use it, or only parts of it, in a program, you must acknowledge somewhere in your documentation that you have used the FreeType code. (`credits')

We specifically permit and encourage the inclusion of this software, with or without modifications, in commercial products. We disclaim all warranties covering The FreeType Project and assume no liability related to The FreeType Project.

Finally, many people asked us for a preferred form for a credit/disclaimer to use in compliance with this license. We thus encourage you to use the following text:

"""
Portions of this software are copyright (c) <year> The FreeType Project (www.freetype.org). All rights reserved.
"""

Please replace <year> with the value from the FreeType version you actually use.

Legal Terms
==========

0. Definitions

Throughout this license, the terms `package', `FreeType Project', and `FreeType archive' refer to the set of files originally distributed by the authors (David Turner, Robert Wilhelm, and Werner Lemberg) as the `FreeType Project', be they named as alpha, beta or final release.

`You' refers to the licensee, or person using the project, where `using' is a generic term including compiling the project's source code as well as linking it to form a `program' or `executable'. This program is referred to as `a program using the FreeType engine'.

This license applies to all files distributed in the original FreeType Project, including all source code, binaries and documentation, unless otherwise stated in the file in its original, unmodified form as distributed in the original archive. If you are unsure whether or not a particular file is covered by this license, you must contact us to verify this.

The FreeType Project is copyright (C) 1996-2000 by David Turner, Robert Wilhelm, and Werner Lemberg. All rights reserved except as specified below.

1. No Warranty

THE FREETYPE PROJECT IS PROVIDED `AS IS' WITHOUT WARRANTY OF ANY
KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. IN NO EVENT WILL ANY OF THE AUTHORS OR COPYRIGHT HOLDERS
BE LIABLE FOR ANY DAMAGES CAUSED BY THE USE OR THE INABILITY TO
USE, OF THE FREETYPE PROJECT.

2. Redistribution

This license grants a worldwide, royalty-free, perpetual and irrevocable right and license to use, execute, perform, compile, display, copy, create derivative works of, distribute and sublicense the FreeType Project (in both source and object code forms) and derivative works thereof for any purpose; and to authorize others to exercise some or all of the rights granted herein, subject to the following conditions:

- Redistribution of source code must retain this license file (`FTL.TXT') unaltered; any additions, deletions or changes to the original files must be clearly indicated in accompanying documentation. The copyright notices of the unaltered, original files must be preserved in all copies of source files.

- Redistribution in binary form must provide a disclaimer that states that the software is based in part of the work of the FreeType Team, in the distribution documentation. We also encourage you to put an URL to the FreeType web page in your documentation, though this isn't mandatory.

These conditions apply to any software derived from or based on the FreeType Project, not just the unmodified files. If you use our work, you must acknowledge us. However, no fee need be paid to us.

3. Advertising

Neither the FreeType authors and contributors nor you shall use the name of the other for commercial, advertising, or promotional purposes without specific prior written permission.

We suggest, but do not require, that you use one or more of the following phrases to refer to this software in your documentation or advertising materials: `FreeType Project', `FreeType Engine', `FreeType library', or `FreeType Distribution'.

As you have not signed this license, you are not required to accept it. However, as the FreeType Project is copyrighted material, only this license, or another one contracted with the authors, grants you the right to use, distribute, and modify it. Therefore, by using, distributing, or modifying the FreeType Project, you indicate that you understand and accept all the terms of this license.

4. Contacts

There are two mailing lists related to FreeType:

o freetype@nongnu.org

Discusses general use and applications of FreeType, as well as future and wanted additions to the library and distribution. If you are looking for support, start in this list if you haven't found anything to help you in the documentation.

o freetype-devel@nongnu.org

Discusses bugs, as well as engine internals, design issues, specific licenses, porting, etc.

Our home page can be found at

http://www.freetype.org

--- end of FTL.TXT ---