

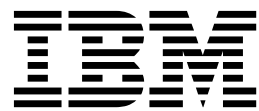
IBM Parallel Environment Developer Edition
High Performance Computing Toolkit
Version 2 Release 2

Installation and Usage Guide



IBM Parallel Environment Developer Edition
High Performance Computing Toolkit
Version 2 Release 2

Installation and Usage Guide



Note

Before using this information and the product it supports, read the information in “Notices” on page 277.

This edition applies to version 2, release 2, modification 0 of the IBM Parallel Environment Developer Edition High Performance Computing Toolkit (HPC Toolkit) (product number 5765-PD2) and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC23-7287-00.

© **Copyright IBM Corporation 2008, 2015.**

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

Contents

Figures	vii	Chapter 5. Generating a log file	21
Tables	ix	Chapter 6. Using the hpctView application	23
About this information	xi	Preparing an application for analysis	24
Who should read this information	xi	Working with the application	25
Conventions and terminology used in this information	xi	Opening the application executable	25
Prerequisite and related information	xii	Using hpctView	26
Parallel Tools Platform component	xiii	Using Eclipse PTP	29
How to send your comments	xiii	Instrumenting the application	29
Summary of changes	xv	Running the instrumented application	30
Limitations and restrictions	xvii	Viewing performance data	38
Part 1. Introduction	1	Chapter 7. Using hardware performance counter profiling.	41
Chapter 1. Introduction to the IBM HPC Toolkit	3	Preparing an application for profiling.	41
The HPC Toolkit	3	Instrumenting the application	41
Collecting performance data from a subset of parallel application tasks	4	Running the instrumented application	43
Performance measurement tools	4	Viewing hardware performance counter data	45
Instrumentation models.	5	Chapter 8. Using GPU hardware counter profiling	47
Setting the user environment	5	Preparing an application for profiling.	47
Part 2. Installation	9	Instrumenting the application	47
Chapter 2. Supported platforms and software levels	11	Running the instrumented application	48
Chapter 3. Installation media contents	13	Viewing GPU hardware performance counter data	49
Chapter 4. Installing the IBM High Performance Computing Toolkit	15	Displaying combined CPU and GPU performance data	50
Installing the IBM HPC Toolkit on Linux systems	15	Chapter 9. Using MPI profiling	53
Installing the IBM HPC Toolkit xCAT kit	16	Preparing an application for profiling.	53
Installing the IBM HPC Toolkit hpctView application	17	Instrumenting the application	53
Installing the IBM HPCT plug-in for Eclipse PTP ..	17	Running the application	55
Part 3. Using the IBM PE Developer Edition graphical performance analysis tools	19	Viewing MPI profiling data	57
		Viewing MPI traces.	58
		Chapter 10. Using I/O profiling	61
		Preparing an application for profiling.	61
		Instrumenting the application	61
		Running the application	63
		Viewing I/O Data	64
		Chapter 11. Using Call Graph analysis	67
		Preparing the application.	67
		Running the application	67
		Viewing profile data	67
		Viewing the Call Graph	68
		Viewing gprof data	69
		Part 4. The hardware performance counter tools	73

Chapter 12. Using the hardware performance counter tools	75	Chapter 17. Commands	113
Using the hpccount command	75	gpm - Lists the available events and metrics ..	114
Using the hpcstat command	76	hpccount - Report hardware performance counter statistics for an application	116
Using the libhpc library	77	hpcrun - Launch a program to collect profiling or trace data.	122
Understanding CPU hardware counter multiplexing	78	hpcstat - Reports a system-wide summary of hardware performance counter statistics	124
Understanding derived metrics	79	hpctInst - Instrument applications to obtain performance data	128
Understanding MFlop issues	79		
Understanding inheritance	80	Chapter 18. Application programming interfaces	131
Understanding inclusive and exclusive event counts	80	gpm_init - Initialize the GPU Performance Monitor runtime environment	133
Understanding parent-child relationships	81	gpm_start - Identify the starting point of an instrumented region of code	136
Handling of overlap issues	81	gpm_stop - Identify the end point of an instrumented region of code	139
Understanding measurement overhead	82	gpm_terminate - Generate GPU Performance Monitoring statistics and trace files and shut down the GPM runtime environment	142
Handling multithreaded program instrumentation issues	82	gpm_Tstart - Identify the starting point of an instrumented region of code	145
Considerations for MPI programs	83	gpm_Tstop - Identify the end point of an instrumented region of code	148
General considerations	83	hpm_error_count, f_hpm_error - Verify a call to a libhpc function	151
Hardware performance counter plug-ins. . . .	83	hpmInit, f_hpmInit - Initialize the Hardware Performance Monitor (HPM) run-time environment	153
Chapter 13. Using GPU hardware counters in HPCT	89	hpmStart, f_hpmstart - Identify the starting point for an instrumented region of code	158
Part 5. The MPI and I/O profiling libraries	93	hpmStartx, f_hpmstartx - Identify the starting point for an instrumented region of code	160
Chapter 14. Using the MPI profiling library	95	hpmStop, f_hpmstop - Identify the end point of an instrumented region of code	163
Compiling and linking with libmpitrace	95	hpmTerminate, f_hpmterminate - Generate HPM statistic files and shut down HPM	165
Controlling traced tasks	96	hpmTstart, f_hpmtstart - Identify the starting point for an instrumented region of code	167
Additional trace controls	96	hpmTstartx, f_hpmtstartx - Identify the starting point for an instrumented region of code	169
Customizing MPI profiling data	97	hpmTstop, f_hpmtstop - Identify the end point of an instrumented region of code	172
Understanding MPI profiling utility functions . .	98	MT_get_allresults - Obtain statistical results . .	174
Performance data file naming	98	MT_get_calleraddress - Obtain the address of the caller of an MPI function	177
Chapter 15. Using the I/O profiling library	99	MT_get_callerinfo - Obtain source code information	178
Preparing your application	99	MT_get_elapsed_time - Obtains elapsed time. . .	180
Setting I/O profiling environment variables . .	99	MT_get_environment - Returns run-time environment information	181
Specifying I/O profiling library module options	101	MT_get_mpi_bytes - Obtain the accumulated number of bytes transferred	182
Running your application	104	MT_get_mpi_counts - Obtain the the number of times a function was called.	183
Performance data file naming	104	MT_get_mpi_name - Returns the name of the specified MPI function	184
Part 6. Using the hpctInst command	105	MT_get_mpi_time - Obtain elapsed time	185
Chapter 16. Instrumenting your application using hpctInst	107	MT_get_time - Get the elapsed time.	186
Instrumenting your application for hardware performance counters	107		
Instrumenting your application for MPI profiling	108		
Instrumenting your application for I/O profiling	109		
Part 7. Command and API reference	111		

MT_get_tracebufferinfo - Obtain information about MPI trace buffer usage	187
MT_output_text - Generate performance statistics	188
MT_output_trace - Control whether an MPI trace file is created	189
MT_trace_event - Control whether an MPI trace event is generated	190
MT_trace_start, mt_trace_start - Start or resume the collection of trace events	192
MT_trace_stop, mt_trace_stop - Suspend the collection of trace events	194

Part 8. Appendixes 195

Appendix A. Performance data file naming	197
File naming conventions.	197
File naming examples	198

Appendix B. Derived metrics, events, and groups supported on POWER8 architecture	203
Derived metrics defined for POWER8 architecture	203

Events and groups supported on POWER8 architecture	204
--	-----

Appendix C. HPC Toolkit environment variables 271

Accessibility features for IBM PE Developer Edition	275
Accessibility features	275
Keyboard navigation	275
IBM and accessibility.	275

Notices	277
Programming interface information	279
Trademarks	279
Terms and conditions for product documentation	279
IBM Online Privacy Statement.	280
Privacy policy considerations	281
Trademarks	281

Index 283

Figures

1. hpctView Welcome page	24	18. GPM subtab containing GPU metrics settings	48
2. hpctView main window	25	19. Trace View displaying GPU performance data from an MPI program	50
3. Remote file browser.	27	20. MPI instrumentation tab	54
4. New connection dialog.	28	21. Subtab containing MPI trace settings	56
5. Initial instrumentation view	29	22. Performance Data view with MPI profiling data	57
6. Expanded instrumentation view.	30	23. MPI trace view	58
7. Profile Configurations dialog.	31	24. MIO instrumentation view	62
8. Resources tab	32	25. MIO subtab of the Performance Analysis tab	63
9. Application tab	33	26. Performance Data view displaying the I/O profiling data..	64
10. Arguments tab	34	27. MIO Data view showing I/O trace data	65
11. Environment tab	35	28. MIO Data view showing a graph of the trace data	66
12. Performance Analysis tab	36	29. Call Graph view	68
13. Performance Data view after loading performance files.	38	30. The gmon view	70
14. Performance Data view showing application data	39		
15. Performance Data view in tabular mode	40		
16. HPM instrumentation tab	42		
17. HPM subtab containing hardware counter settings	44		

Tables

1.	Conventions	xi	11.	Plug-ins shipped with the tool kit	84
2.	Platform-specific file names for the IBM PE Developer Edition HPC Toolkit	13	12.	MPI profiling utility functions	98
3.	Linux software requirements for the IBM HPC Toolkit	15	13.	MIO analysis modules	101
4.	Platform specific Linux package file names for the IBM HPC Toolkit	16	14.	MIO module options	101
5.	Profile Configurations dialog Common environment variables	37	15.	MIO pf module options	101
6.	HPM profile configuration dialog environment variables	44	16.	MIO trace module options	102
7.	GPM profile configuration dialog environment variable	49	17.	MIO recov module options	103
8.	MPI trace profile configuration environment variables	56	18.	Commands	113
9.	MPI trace timeline navigation	59	19.	APIs	131
10.	MIO profile configuration environment variables	64	20.	Controlling the collection of MPI trace events for any function with an identifier	190
			21.	File naming conventions for various tools and their types	197
			22.	Derived metrics defined for POWER8 architecture	203
			23.	HPC Toolkit environment variables	271

About this information

This information applies only to IBM® Parallel Environment (PE) Developer Edition Version 2.2.

Attention:

IBM PE Developer Edition 2.2 only runs on IBM Power Systems™ servers with IBM POWER8® technology in Little Endian (LE) mode running RHEL 7.2.

For information about running Power Systems servers with IBM POWER8 technology in Little Endian (LE) mode running the Ubuntu Server 14.04.01 for IBM Power®, refer to IBM PE Developer Edition 2.1 (http://www-01.ibm.com/support/knowledgecenter/SSFK5S_2.1.0/pedev.v2r1_welcome.html?cp=SSFK5S_2.1.0%2F0-0) in IBM Knowledge Center).

Disclaimer:

The functions or features found herein may not be available on all operating systems or platforms and do not indicate the availability of these functions or features within the IBM product or future versions of the IBM product. The development, release, and timing of any future features or functionality is at IBM's sole discretion. IBM's plans, directions, and intent are subject to change or withdrawal without notice at IBM's sole discretion. The information mentioned is not a commitment, promise, or legal obligation to deliver any material, code or functionality. The information may not be incorporated into any contract and it should not be relied on in making a purchasing decision.

Who should read this information

This information is for parallel application developers looking to improve productivity throughout the edit, compile, debug, and run development cycle. A set of integrated performance analysis tools is provided to help the application developer tune a serial or parallel application.

Conventions and terminology used in this information

Table 1 shows the conventions used in this information:

Table 1. Conventions

Convention	Usage
bold	Bold words or characters represent system elements that you must use literally, such as commands, flags, path names, directories, file names, values, and selected menu options.
<u>bold underlined</u>	<u>bold underlined</u> keywords are defaults. These take effect if you do not specify a different keyword.
constant width	Examples and information that the system displays appear in constant-width typeface.
<i>italic</i>	<i>Italic</i> words or characters represent variable values that you must supply. <i>Italics</i> are also used for information unit titles, for the first use of a glossary term, and for general emphasis in text.

Table 1. Conventions (continued)

Convention	Usage
<key>	Angle brackets (less-than and greater-than) enclose the name of a key on the keyboard. For example, <Enter> refers to the key on your terminal or workstation that is labeled with the word <i>Enter</i> .
\	In command examples, a backslash indicates that the command or coding example continues on the next line. For example: mkcondition -r IBM.FileSystem -e "PercentTotUsed > 90" \ -E "PercentTotUsed < 85" -m d "FileSystem space used"
{item}	Braces enclose a list from which you must choose an item in format and syntax descriptions.
[item]	Brackets enclose optional items in format and syntax descriptions.
<Ctrl-x>	The notation <Ctrl-x> indicates a control character sequence. For example, <Ctrl-c> means that you hold down the control key while pressing <c>.
item...	Ellipses indicate that you can repeat the preceding item one or more times.
	<ul style="list-style-type: none"> In <i>syntax</i> statements, vertical lines separate a list of choices. In other words, a vertical line means <i>Or</i>. In the margin of the document, vertical lines indicate technical changes to the information.

Prerequisite and related information

IBM Parallel Environment Runtime Edition

The IBM Parallel Environment Runtime Edition library consists of:

- *IBM Parallel Environment Runtime Edition: Installation*, SC23-7282
- *IBM Parallel Environment Runtime Edition: Messages*, SC23-7284
- *IBM Parallel Environment Runtime Edition: MPI Programming Guide*, SC23-7285
- *IBM Parallel Environment Runtime Edition: NRT API Programming Guide*, SC23-7286
- *IBM Parallel Environment Runtime Edition: Operation and Use*, SC23-7283
- *IBM Parallel Environment Runtime Edition: PAMI Programming Guide*, SA23-1453

To access the most recent IBM PE Runtime Edition or IBM PE Developer Edition documentation in PDF and HTML format, refer to the IBM Knowledge Center (www.ibm.com/support/knowledgecenter), on the web.

The current IBM PE Runtime Edition or IBM PE Developer Edition documentation is also available in PDF format from the IBM Publications Center (www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss).

To easily locate a book in the IBM Publications Center, supply the book's publication number. The publication number for each of the Parallel Environment books is listed after the book title in the preceding lists.

Parallel Tools Platform component

For information about how to use the Eclipse client Parallel Tools Platform (PTP) component, go to Eclipse Documentation (<http://help.eclipse.org/mars/index.jsp>) and search for these documents:

- *Parallel Development User Guide*
- *PTP Developer's Guide*

These documents are available both on the web and as online Help that can be accessed through the Eclipse user interface.

How to send your comments

Your feedback is important in helping us to produce accurate, high-quality information. If you have any comments about this information unit, send your comments by e-mail to:

mhvrcfs@us.ibm.com

Include the book title and order number, and, if applicable, the specific location of the information you have comments on (for example, a page number or a table number).

You can also add comments to individual topics in IBM Knowledge Center (www.ibm.com/support/knowledgecenter) by clicking **Add Comment** and filling in the form.

For technical information and to exchange ideas related to high performance computing, go to:

- developerWorks® HPC Central (www.ibm.com/developerworks/wikis/display/hpccentral/HPC+Central)
- developerWorks Forum (www.ibm.com/developerworks/forums)

Summary of changes

The following summarizes changes to the IBM Parallel Environment (PE) Developer Edition Version 2 Release 2 product and library.

Within each information unit in the library, a vertical line in the margin next to text and illustrations indicates technical changes or additions made to the previous edition of the information.

Attention:

IBM PE Developer Edition 2.2 only runs on IBM Power Systems servers with IBM POWER8 technology in Little Endian (LE) mode running RHEL 7.2.

For information about running Power Systems servers with IBM POWER8 technology in Little Endian (LE) mode running the Ubuntu Server 14.04.01 for IBM Power, refer to IBM PE Developer Edition 2.1 (http://www-01.ibm.com/support/knowledgecenter/SSFK5S_2.1.0/pedev.v2r1_welcome.html?cp=SSFK5S_2.1.0%2F0-0) in IBM Knowledge Center).

Disclaimer:

The functions or features found herein may not be available on all operating systems or platforms and do not indicate the availability of these functions or features within the IBM product or future versions of the IBM product. The development, release, and timing of any future features or functionality is at IBM's sole discretion. IBM's plans, directions, and intent are subject to change or withdrawal without notice at IBM's sole discretion. The information mentioned is not a commitment, promise, or legal obligation to deliver any material, code or functionality. The information may not be incorporated into any contract and it should not be relied on in making a purchasing decision.

New information

- Support for graphics processing unit (GPU) hardware counters in the High Performance Computing Toolkit (HPCT) has been added, which includes:
 - A new command called **gpmlist**, which lists the events and metrics that are available for a specific GPU device or the current GPU device.
 - New environment variables:
 - **GPM_EVENT_SET**, which specifies the set of events to profile in a single run
 - **GPM_METRIC_SET**, which specifies the set of metrics to profile in a single run
 - **HPM_ENABLE_GPM**, which instructs the HPM module to drive GPU event and metric profiling
- Support for loading and visualizing OTF2 files generated using HPCT.
- Support for GPU hardware counters in the hpctView application, which changes the visualization of the profile and trace files produced by the HPC Toolkit.

Deleted information

All references to the IBM PE Developer Edition Workbench have been removed because it is not supported in this release.

Limitations and restrictions

The following limitations and restrictions apply:

- On Power Architecture®, the dynamic instrumentation tools (**hpctInst**) will only operate reliably on executables with a text segment size less than 32 MB.
- Using the **hpcrun** command to launch an MPI program compiled with **-hpcprof** is not supported on Linux.
- Instrumentation of executables compiled and linked with the **-pg** flag using the IBM Eclipse HPC Toolkit plug-in or **hpctInst** is not supported.
- Using the **hpcrun** command to launch applications that use dynamic tasking is not supported.
- On Linux running on Power Architecture, instrumenting programs with MPI and HPM at the same time is not supported.
- The only **--tracestore** option that is valid for the **hpcrun** command when collecting I/O trace is **memory**. Although the **--tracestore** option accepts file names for both MPI and I/O tracing, I/O tracing should not be used unless the **memory** option was chosen.
- HPC Toolkit using the **hpcrun** command, which is intended to filter the volume of data collected for large numbers of application tasks, has been tested to 3500 application tasks. When collecting MPI profiling and trace data you might notice performance degradation as the task count increases.

Part 1. Introduction

The topics in this part provide an introduction to the IBM High Performance Computing Toolkit.

Chapter 1. Introduction to the IBM HPC Toolkit

The HPC Toolkit is an integrated set of performance analysis tools that help the application developer tune a serial or parallel application. The HPC Toolkit is a component of IBM Parallel Environment Developer Edition. IBM Parallel Environment Developer Edition also provides a plug-in for the Eclipse-based integrated development environment (IDE) that enables HPC Toolkit to be used to improve parallel application developer productivity throughout the edit, compile, debug, and run development cycle for C, C++ and Fortran applications.

The IBM PE Developer Edition is a companion to IBM PE Runtime Edition.

The HPC Toolkit

The HPC Toolkit is the server (back-end) component that provides the performance analysis tools. These tools perform the following functions:

- Provide access to hardware performance counters for performing low-level analysis of an application, including analyzing cache usage 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 whether 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 HPC Toolkit provides three types of interfaces:

- An application programming interface (API) and shared libraries intended to be used by users to insert performance analysis calls into their applications
- A command line interface (CLI) intended to be used on the login nodes to instrument the user application (**hpctInst**)
- A graphical user interface (GUI) intended to be used on a client machine (desktop or laptop) to instrument and run the user application, and visualize the trace files produced by back-end instrumentation components (hpctView and the HPCT plug-in for Eclipse PTP)

The HPC Toolkit interfaces allow users to do the following:

- Collect performance data from an instrumented application by setting environment variables for the specific analysis tool and then invoking the application from the command line.
- Modify the application to add calls to the hardware performance counter tool, rebuild your application linking with the hardware performance counter library, and then run the application to get hardware performance counter data.
- Link the application directly with the MPI profiling and trace library then run the application to get MPI trace and profiling data.

- Link the application with the I/O profiling and trace library to get I/O profiling and trace information.

Collecting performance data from a subset of parallel application tasks

By default, the IBM HPC Toolkit generates and collects performance data for all tasks in the application. If the application has a large number of tasks, the overhead of collecting performance data from all application tasks may be excessive.

The IBM HPC Toolkit provides a command, **hpcrun**, that you can use to collect performance data from a subset of application tasks. The **hpcrun** command allows you to specify a number of application tasks to collect performance data from and the performance metric to use in determining which tasks to collect performance data from.

The **hpcrun** command will collect performance data from the number of application tasks you specify that have the minimum value for the performance metric you specify, the number of application tasks you specify that have the maximum value for the performance metric you specify, the task closest to the average for the metric you selected and application task zero.

You can collect performance data by specifying a metric of **ELAPSED_TIME** to use elapsed wall clock time as the performance metric, or by specifying a metric of **CPU_TIME** to use CPU time as the performance metric.

If you are collecting MPI trace or I/O trace data, you can either collect trace data in memory in each application task or store it to temporary files. If the size of the individual trace files is small, then you should use the in-memory model for better performance. Otherwise, use the temporary file model for trace file generation.

See “hpcrun - Launch a program to collect profiling or trace data” on page 122 for details of specific command line flags and environment variables.

Performance measurement tools

The following tools are provided to help you obtain performance measurements:

- Hardware performance counters, including:
 - Measurements for cache misses at all levels of cache
 - The number of floating point instructions executed
 - The number of load instructions resulting in TLB misses
 - Other measurements 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.

- MPI profiling and trace, where MPI profiling obtains performance metrics including time spent in each MPI function call and MPI trace creates a trace of MPI function calls in your application so you can view MPI communication patterns in your application.
- 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.

Instrumentation models

The IBM HPC Toolkit provides two instrumentation models:

- A model in which the application executable is rewritten with the instrumentation specified by the user. The user specifies the instrumentation using the Eclipse plug-in, the hpctView application, or the **hpctInst** command. The IBM HPC Toolkit rewrites the application executable, adding the requested calls to the instrumentation libraries into the executable. This process does not require any modifications to the application source code, and does not require the application to be relinked.
- A model in which measurements are obtained by using the Hardware Performance Monitor (HPM) or by controlling the generation of the MPI trace and MPI profiling information when using MPI profiling. In this model, users modify their application by inserting calls to functions in the instrumentation library in the application source code, then recompiling and relinking the application.

When you instrument an application, you should choose only one of the instrumentation models. This is because any calls to instrumentation functions that you code into the application might interfere with the instrumentation calls that are inserted by the HPC Toolkit when using the Eclipse plug-in, the hpctView application, or the **hpctInst** command.

For more information about the **hpctInst** utility, see Chapter 16, “Instrumenting your application using hpctInst,” on page 107.

For more information about the hpctView application, see Chapter 6, “Using the hpctView application,” on page 23.

Setting the user environment

Before using the IBM HPC Toolkit, You must ensure that several environment variables required by the IBM HPC Toolkit are properly set. In order to set these environment variables, you must run one of the setup scripts that are located in the top-level directory of the IBM HPC Toolkit installation.

To initialize the IBM HPC Toolkit environment, issue the following command as the shell prompt:

```
. /opt/ibmhpc/ppdev.hpct/env_sh
```

Users using **cs**h issue the following command at the shell prompt:

```
source /opt/ibmhpc/ppdev.hpct/env_csh
```

When using the Eclipse plug-in or the hpctView application, you do not need to set the environment directly. Instead, you set the IBM HPC Toolkit environment using the dialogs available in the hpctView application as described in Part 3, “Using the IBM PE Developer Edition graphical performance analysis tools,” on page 19.

The IBM HPC Toolkit requires the user application to be compiled and linked using the **-g** flag. For Power Linux systems, the user application must be linked with the **-Wl,--hash-style=sysv -emit-stub-syms** flags . If the user application has not been compiled and linked using these flags, the Eclipse plug-in, the hpctView application, and the **hpctInst** command will not be able to instrument the application.

After instrumenting it, users must ensure that the application, and any data files that it requires, are properly distributed on the system where it will run. If the application resides in a globally accessible file system, such as an IBM Spectrum Scale™ file system, you should not have to do anything to distribute the application. If your system is set up so that only local file systems 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.

By default, 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 file system, 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 file systems 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 topics that follow describe how to:

- Install the IBM HPC Toolkit
- Use the GUI tools to instrument, run, and analyze the performance of an application

- Instrument your application by inserting calls to the instrumentation libraries in your application
- Use the command line instrumentation tools

Part 2. Installation

The topics in this part provide information about installing the IBM High Performance Computing Toolkit.

Chapter 2. Supported platforms and software levels

The IBM PE Developer Edition is supported on the following software levels:

- The IBM HPC Toolkit back-end (the run-time and the CLI instrumentation tools) is supported on IBM Power Systems servers with IBM POWER8 technology in Little Endian (LE) mode running RHEL 7.2.
- The IBM HPC Toolkit hpctView application is supported on the following platforms:
 - Microsoft Windows 64 bit
 - Mac OS X 10.9 (Mavericks) or later
 - x86_64 Linux

Additional requirements

The following additional requirements are necessary to enable certain features in IBM PE Developer Edition:

- Java™ Runtime version 1.8 or higher is required for the hpctView application. Go to Java (www.java.com) for more information.
- On MacOS, Java JDK 1.8 or higher is required for the hpctView application. Go to Java (www.java.com) for more information.
- The Time/HiRes Perl module must be installed on any login node where jobs will be submitted using IBM PE Developer Edition. The Time/HiRes Perl module can be obtained from CPAN (search.cpan.org/search?query=time-hires&mode=all).

IBM PE Developer Edition supports the following:

- IBM Parallel Runtime Edition 2.3.0.0 or later
- IBM Platform LSF® 9.1.1 or later

Chapter 3. Installation media contents

The IBM PE Developer Edition provides an installation medium for the IBM HPC Toolkit.

IBM PE Developer Edition HPC Toolkit

The IBM PE Developer Edition HPC Toolkit contains install images for the CLI instrumentation commands and runtime and the install image archives for the **hpctView** command.

The install images for all supported platforms are provided on a single DVD (see Table 2). The CLI instrumentation commands and runtime are located in the **hpct** directory, while the install image archives for the **hpctView** commands are located in the **hpctview** directory, as follows:

hpct

hpct/ppedev-hpct-2.2.0-0.ppc64el.rpm
hpct/ppedev-license-2.2.0-0.ppc64el.rpm
hpct/ppedev-runtime-2.2.0-0.ppc64el.rpm
hpct/ppedev-2.2.0-0.tar.bz2

hpctview

hpctview/hpctView-2.2.0-0-linux-gtk-x86_64.tar.gz
hpctview/hpctView-2.2.0-0-macosx-cocoa-x86_64.tar.gz
hpctview/hpctView-2.2.0-0-win64.zip

HPCT plugin for Eclipse PTP

eclipse/ppedev_update-2.2.0-0.zip

Table 2. Platform-specific file names for the IBM PE Developer Edition HPC Toolkit

Purpose	Platform/where installed
HPCT and hpctView application	<p>IBM HPC Toolkit for IBM Power Systems servers with IBM POWER8 technology in LE mode running RHEL 7.2, installed on the compute and login nodes:</p> <ul style="list-style-type: none">• ppedev-hpct-2.2.0-0.ppc64el.rpm• ppedev-license-2.2.0-0.ppc64el.rpm• ppedev-runtime-2.2.0-0.ppc64el.rpm <p>IBM HPC Toolkit xCAT kit for IBM Power Systems servers with IBM POWER8 technology in LE mode running RHEL 7.2:</p> <ul style="list-style-type: none">• ppedev-2.2.0-0.tar.bz2 <p>The following is installed on the user's client machine (desktop or laptop):</p> <p>The hpctView application for Microsoft Windows 64 bit:</p> <ul style="list-style-type: none">• hpctView-2.2.0-0-win64.zip <p>The hpctView application for Mac OS X 64 bit:</p> <ul style="list-style-type: none">• hpctView-2.2.0-0-macosx-cocoa-x86_64.tar.gz <p>The hpctView application for x86_64 Linux:</p> <ul style="list-style-type: none">• hpctView-2.2.0-0-linux-gtk-x86_64.tar.gz <p>The HPCT plugin for PTP, for Windows 64 bit, Mac OS X 64bit and x86_64 Linux:</p> <ul style="list-style-type: none">• ppedev_update-2.2.0-0.zip

The IBM PE Developer Edition HPC Toolkit packaging consists of the following:

- A copy of the IBM PE Developer Edition HPC Toolkit (installation images) for IBM Power Systems servers with IBM POWER8 technology in LE mode running RHEL 7.2
- A copy of the IBM PE Developer Edition HPC Toolkit xCAT kit package
- Copies of the hpctView application install image archives for the supported platforms
- A copy of the IBM PE Developer Edition HPCT plugin for Eclipse PTP

Chapter 4. Installing the IBM High Performance Computing Toolkit

The installation of the IBM PE Developer Edition HPC Toolkit consists of:

1. Installing the runtime and the CLI instrumentation tools on the Linux system (compute and login nodes)
2. Installing the hpctView application on the user's client machine (desktop or laptop)

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) Runtime Edition for Linux.

Table 3 shows the IBM HPC Toolkit software requirements:

Table 3. Linux software requirements for the IBM HPC Toolkit

Function	Software	IBM Power Systems servers with POWER8 technology in LE mode running RHEL 7.2
All	Operating system	IBM Power Systems servers with POWER8 technology in LE mode running RHEL 7.2
Compiling C applications	C compiler	IBM xLC 13.1.1 or later
Compile and run MPI or PAMI programs	Parallel Environment Runtime Edition	2.3.0 or later
Compiling Fortran applications	Fortran Compiler	IBM XL Fortran 15.1.1 or later

The installation images for the IBM HPC Toolkit use the **ppedev** prefix. The following install packages are available for the backend:

Package	Description
ppedev-license-2.2.0-0.arch.rpm	Provides the IBM PE Developer Edition license agreement and license acceptance script
ppedev-runtime-2.2.0-0.arch.rpm	Provides the runtime instrumentation functionality
ppedev-runtime-2.2.0-0.arch.dev	Provides the CLI instrumentation tools

In addition the IBM HPC Toolkit provides the **hpctView-2.2.0-0** archives for the supported platforms that provides the hpctView application. It also provides the **ppedev_update-2.2.0-0.zip** update site image for installation into existing Eclipse PTP installations.

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

To install the license:

1. Create a directory, for example:

```
~/images
```

2. Copy the following to that directory:

```
ppedev_license-2.2.0-0.ppc64le.rpm
```

3. Change the directory (**cd**) to that directory.

4. As **root**, invoke the command:

```
rpm -i ppedev-license-2.2.0-0.ppc64le.rpm
```

Note: If **root** privileges are not available, you can use the **sudo** command, as follows:

```
sudo rpm -i ppedev-license-2.2.0-0.ppc64le.rpm
```

5. Set the **IBM_PPEDEV_LICENSE_ACCEPT** environment variable to **no**, for example, export:

```
IBM_PPEDEV_LICENSE_ACCEPT=no
```

6. Invoke the following command to accept the license for installation images:

```
/opt/ibmhpc/ppdev.hpct/lap/accept_ppdev_license.sh
```

7. Accept the license once you have reviewed the license terms.

8. If you have already reviewed the IBM PE Developer Edition license, you can set:

```
IBM_PPEDEV_LICENSE_ACCEPT=yes
```

before invoking:

```
/opt/ibmhpc/ppdev.hpct/lap/accept_ppdev_license.sh
```

in order to accept the license without reviewing it again.

Table 4 shows where to install the software packages provided by IBM PE Developer Edition HPC Toolkit.

Table 4. Platform specific Linux package file names for the IBM HPC Toolkit

Purpose	Where installed	Ubuntu Server 14.04.01 Linux distribution on POWER8 servers in Little Endian (LE) mode
IBM PE Developer Edition license	login and compute nodes	ppedev-license-2.2.0-0.ppc64le.rpm
IBM HPC Toolkit run-time support	login and compute nodes	ppedev-runtime-2.2.0-0.ppc64le.rpm
IBM HPC Toolkit CLI instrumentation tools	login node	ppedev-hpct-2.2.0-0.ppc64le.rpm
IBM HPC Toolkit hpctView application	client machine (desktop or laptop)	hpctView-2.2.0-0-win64.zip hpctView-2.2.0-0-macosx-cocoa-x86_64.tar.gz hpctView-2.2.0-0-linux-gtk-x86_64.tar.gz
IBM HPCT plug-in for Eclipse PTP	client machine (desktop or laptop)	ppedev_update-2.2.0-0.zip

Installing the IBM HPC Toolkit xCAT kit

The IBM PE Developer Edition HPC Toolkit xCAT kit package is shipped with the distribution media. The system administrator must use the xCAT commands to install these kit packages on the boot image. For more information, see xCAT Commands (http://xcat.sourceforge.net/man1/xcat.1.html#xcat_commands).

Installing the IBM HPC Toolkit hpctView application

The installation of the hpctView application package must be done on the user's client machine running one of the supported platforms:

- Microsoft Windows, 64 bit
- Mac OS X
- x86_64 Linux

In order to install the hpctView application, you must simply extract the program files from the archive provided using the corresponding tool for your platform.

For example, for Microsoft Windows, do the following:

1. Copy the file from the installation media to your client machine.
2. Right-click on the name of the file and select the **Extract all** choice for the menu that pops up.
3. Follow the presented dialogs to extract all the files in the directory of choice.

On x86_64 Linux, you can use the **tar** command to extract the files from the install image archive, as follows:

```
tar -xf hpctView-2.2.0-0-linux-gtk-x86_64.tar.gz
```

to extract the archived files (you can use other options for the tar command to extract in specific directory, and so on).

On Mac OS X, you can simply extract the files from the archive by double-clicking on its icon. The **Archive Utility** message window pops up and will extract the hpctView application in the same directory. You can then move the application to the **Application** folder in your Finder.

Note: Regardless of the platform of choice, the installation of the hpctView application does not require **root** or administrative privileges.

Installing the IBM HPCT plug-in for Eclipse PTP

The installation of the **hpctView** application package must be done on your client machine that is running one of the supported platforms:

- Microsoft Windows, 64 bit
- Mac OS X
- x86_64 Linux

Before you begin, you must have write access to the directory where your copy of Eclipse PTP is installed.

To install the **hpctView** application, follow these steps:

1. Copy the **ppedev_update-2.2.0-0.zip** update site image to your local machine.
2. Start your copy of Eclipse PTP.
3. Click **Help** in the Eclipse main menu and then click **Install New Software** in the menu that is displayed.
4. Click the **Add** button in the upper right corner of the **Install** dialog.
5. Enter a name in the **Name** field of the **Add Repository** dialog, such as **hpctView Plugins**.

6. Click the **Archive** button in the **Add Repository** dialog. A **file selector** dialog is displayed. Use that dialog to locate and select the **ppedev_update-2.0.0-0.zip** file.
7. Click **OK** in the **Add Repository** dialog.
8. Make sure the entry you just created is displayed in the **Work With** combo box at the top of the **Install** dialog. Entries for the IBM High Performance Computing Toolkit and related software will be displayed in the name box in the **Install** dialog.
9. Click the **Select All** button at the left center of the **Install** dialog.
10. Click the **Next** button in the **Install** dialog. A list of software to be installed will be displayed.
11. Verify the list of software, and if it is correct, click the **Next** button in the **Install** dialog. License information will be displayed.
12. Review the license information. If the license terms are acceptable, click the **I accept the terms of the license agreement** button then click the **Next** button. A **Security Warning** dialog is displayed, warning you that you are installing unsigned content.
13. Click **OK** in this dialog to continue the installation. A **Software Updates** dialog is displayed, prompting you to restart Eclipse.
14. Click **Yes** to restart Eclipse.
15. If this is the first time you have installed the **ppedev_update-2.2.0-0.zip** file in your copy of Eclipse PTP, the terms of the license agreement will be displayed.
16. Accept the terms of the license agreement in order to continue.

Part 3. Using the IBM PE Developer Edition graphical performance analysis tools

The topics in this part provide information about using the IBM PE Developer Edition graphical performance analysis tools for collecting, analyzing and visualizing profile and trace data collected from high performance computing applications.

| The IBM PE Developer Edition provides the hpctView application, which is a
| stand-alone tool that allows developers to instrument, run, collect and analyze data
| from an existing application. For more information, see Chapter 6, “Using the
| hpctView application,” on page 23.

Chapter 5. Generating a log file

All components of the IBM HPC Toolkit support log files that record execution results. From binary instrumentation to runtime execution, the IBM HPC Toolkit allows you to specify the required level of logging and the location of the log files.

The **HPCTLOG** and **HPC_TEMPDIR** environment variables are used to control the IBM HPC Toolkit execution:

- The **HPCTLOG** environment variable sets the required level of logging. **HPC_TEMPDIR** sets the directory where the log files are generated. By default, the log files are generated in the **/temp** directory.
- The **HPCTLOG** environment variable has the following values:

Value	Description
1	Logs errors.
2	Logs warnings.
3	Logs information.
4	Logs function entry and exit.
5	Logs debug information.

The levels of logging are cumulative, so if you set the logging level to **4**, HPCT will also log categories with a lower value (in this case, categories **1**, **2**, and **3**).

The names of the log files have the following format:

<user name>.<component name><process id>.log

where:

user name

Is the name of the user.

component name

Is the name of the component generating the log, for example:

hpm For the Hardware Performance Monitor (HPM) component.

gpm For the GPU Performance Monitoring (GPM) component.

hpctInst

For the **hpctInst** command.

process id

Is the ID of the process (pid).

Remember: Log files are an important tool to use to determine the cause of unexpected runtime results. However, enabling logging will result in performance degradation because each process produces its own log file. For this reason, it is not recommended for use with large scale jobs.

Chapter 6. Using the hpctView application

The hpctView application is part of the IBM Parallel Environment Developer Edition and is the main user interface for the IBM HPC Toolkit performance tools. You can use hpctView to profile your application and obtain performance measurements in the following areas:

- Hardware performance counter measurements (see Chapter 7, “Using hardware performance counter profiling,” on page 41)
- GPU hardware counters (see Chapter 8, “Using GPU hardware counter profiling,” on page 47)
- MPI profiling (see Chapter 9, “Using MPI profiling,” on page 53)
- Application I/O profiling (see Chapter 10, “Using I/O profiling,” on page 61)

To profile your application and obtain performance measurements, you use the hpctView application to open the existing executable, then specify the points at which you want to gather profiling or trace information. Once these points have been selected, hpctView will generate a new, instrumented, executable that can then be run to generate the profile or trace data. After execution finishes, hpctView will automatically load the performance data and allow you to visualize and analyze the results. hpctView is also able to load and display the *gprof data* (including call graph and histogram information) that is generated by compiling with the **-pg** option.

When you first launch hpctView, you will see the Welcome page shown in Figure 1 on page 24. Follow the links to obtain more information and explore the product features. When you are ready to begin using hpctView, click on the **hpctView** tab.

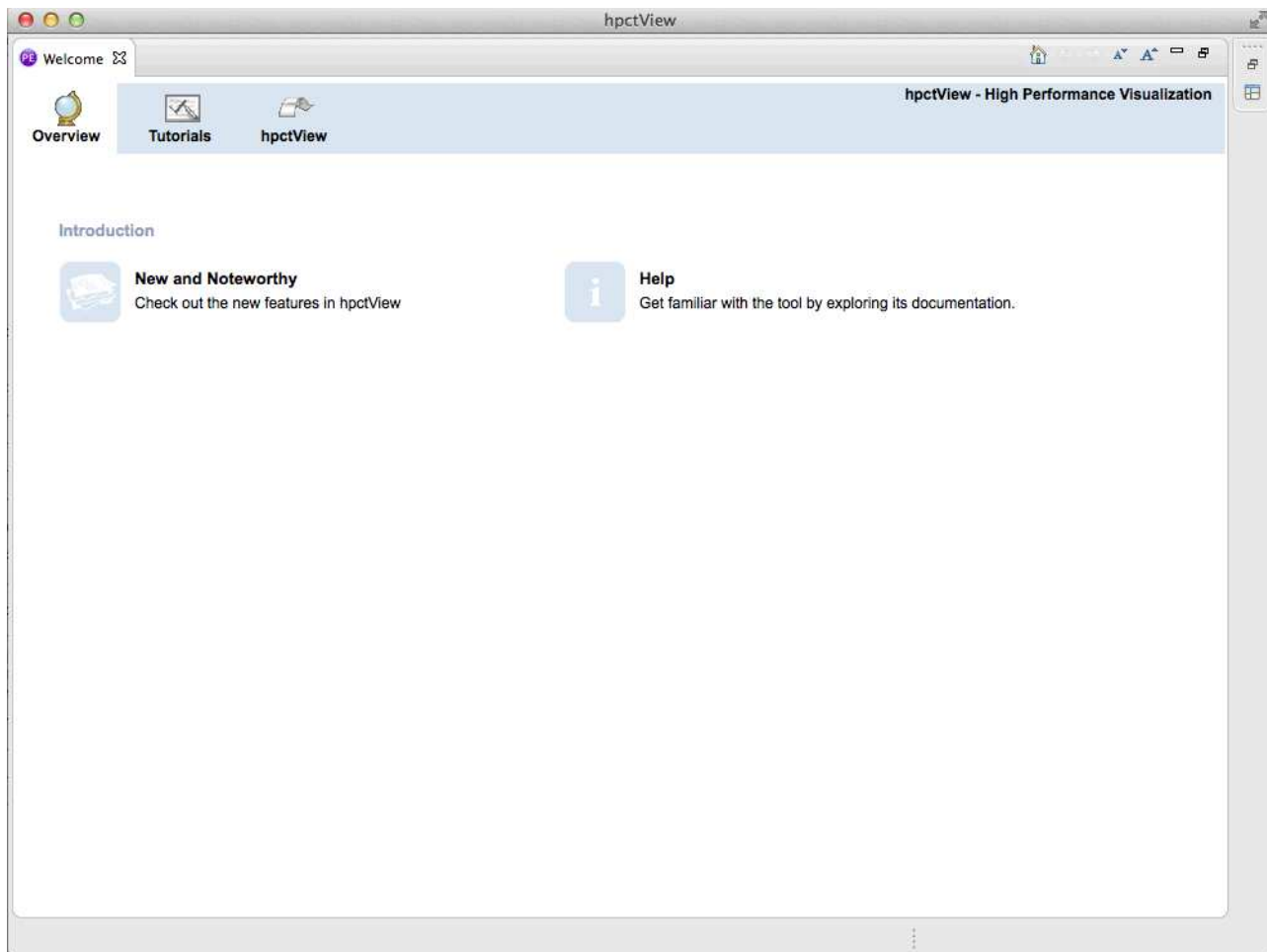


Figure 1. *hpctView* Welcome page

Preparing an application for analysis

You must compile and link your application using the **-g** compiler flag so that hpctView has 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.

Note: You must not compile your application with the **-pg** flag.

When you compile an application on a Power Linux system, you will need to use the following flags on the link step when compiling the application:

-Wl,--hash-style=sysv -Wl,--emit-stub-syms

When you try to load a FORTRAN Linux 64-bit application, the plug-in might be unable to locate the main entry point for your application. If the plug-in cannot locate the main entry point, it will issue a message suggesting that you set the **PSIGMA_MAIN** environment variable. If you see this message, you should set **PSIGMA_MAIN** to the main entry point of your application, or the name of the

main program of your FORTRAN application. Set the **PSIGMA_MAIN** environment variable, using the method suitable for your operating system, prior to launching the hpctView application.

Working with the application

When you start hpctView, the main window will look similar to Figure 2. On the left side of the window is the **Instrumentation** view. This is where you will select the instrumentation points used to gather performance data. In the middle is the **Performance Data** view where the results of profiling or tracing your application will be displayed. Other views may also become visible here as you use the features of hpctView. At the bottom is the **Console** view where output from your application runs will be displayed.

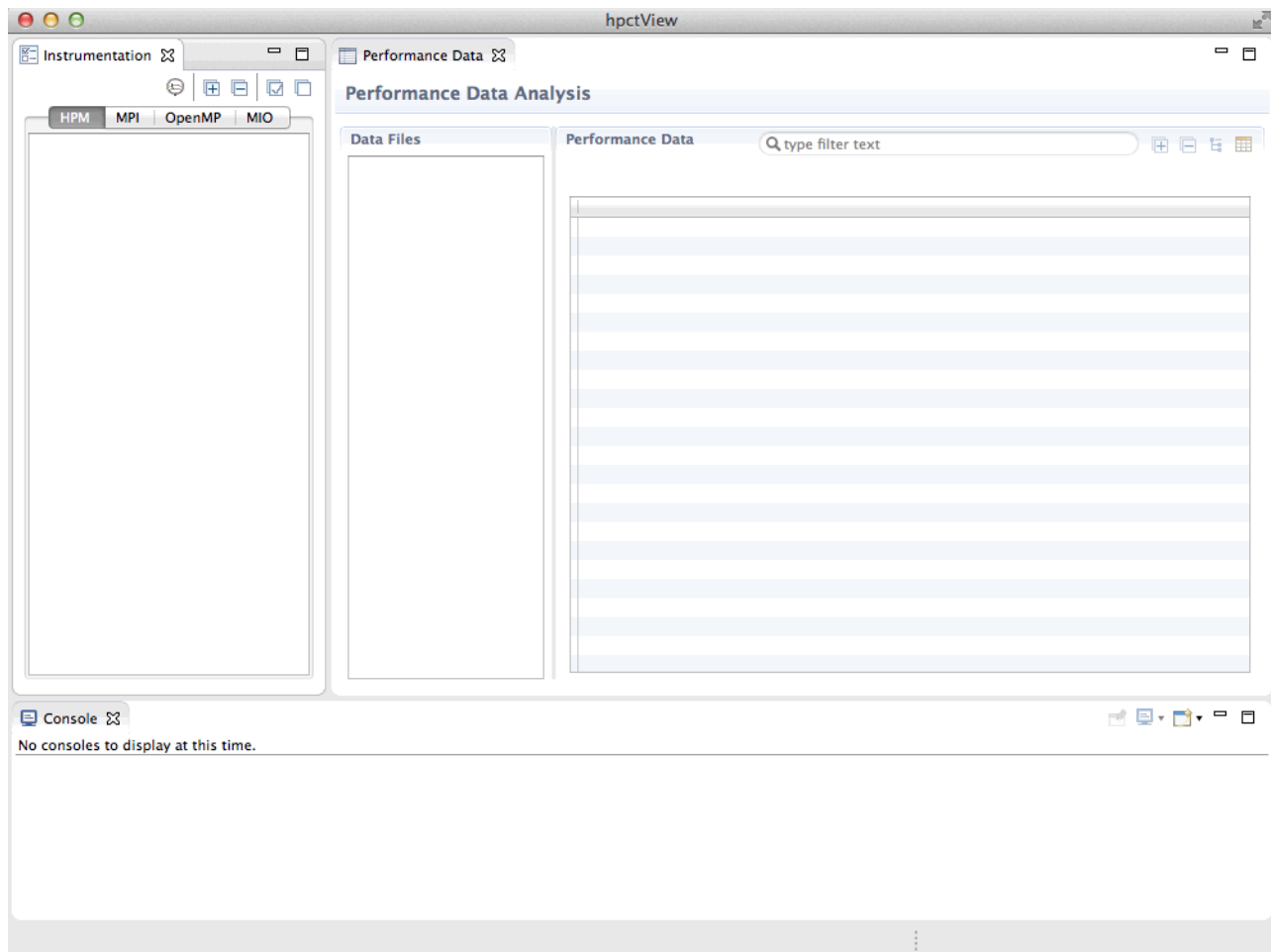


Figure 2. hpctView main window

Opening the application executable

In order to start using hpctView, the application executable must be opened. Before it can be opened, however, the executable must have been compiled as described in “Preparing an application for analysis” on page 24.

Using hpctView

Open the main **File** menu, then select **Open Executable....** This will display the remote file browser, as shown in Figure 3 on page 27.

The remote file browser will be used many times when working with applications that are located on a remote machine. Begin by clicking on the **New...** button to create a new connection. This will open a second dialog allowing the user to supply connection information (see Figure 4 on page 28). At a minimum, **Connection name**, **Host**, and **User** fields must be filled in. The **Connection name** field provides a memorable name that will be displayed in the connection name drop-down list next time the remote file browser is used. The **Host** field provides the hostname for the target system. The **User** field provides the username that will be used to log into the target system.



Figure 3. Remote file browser

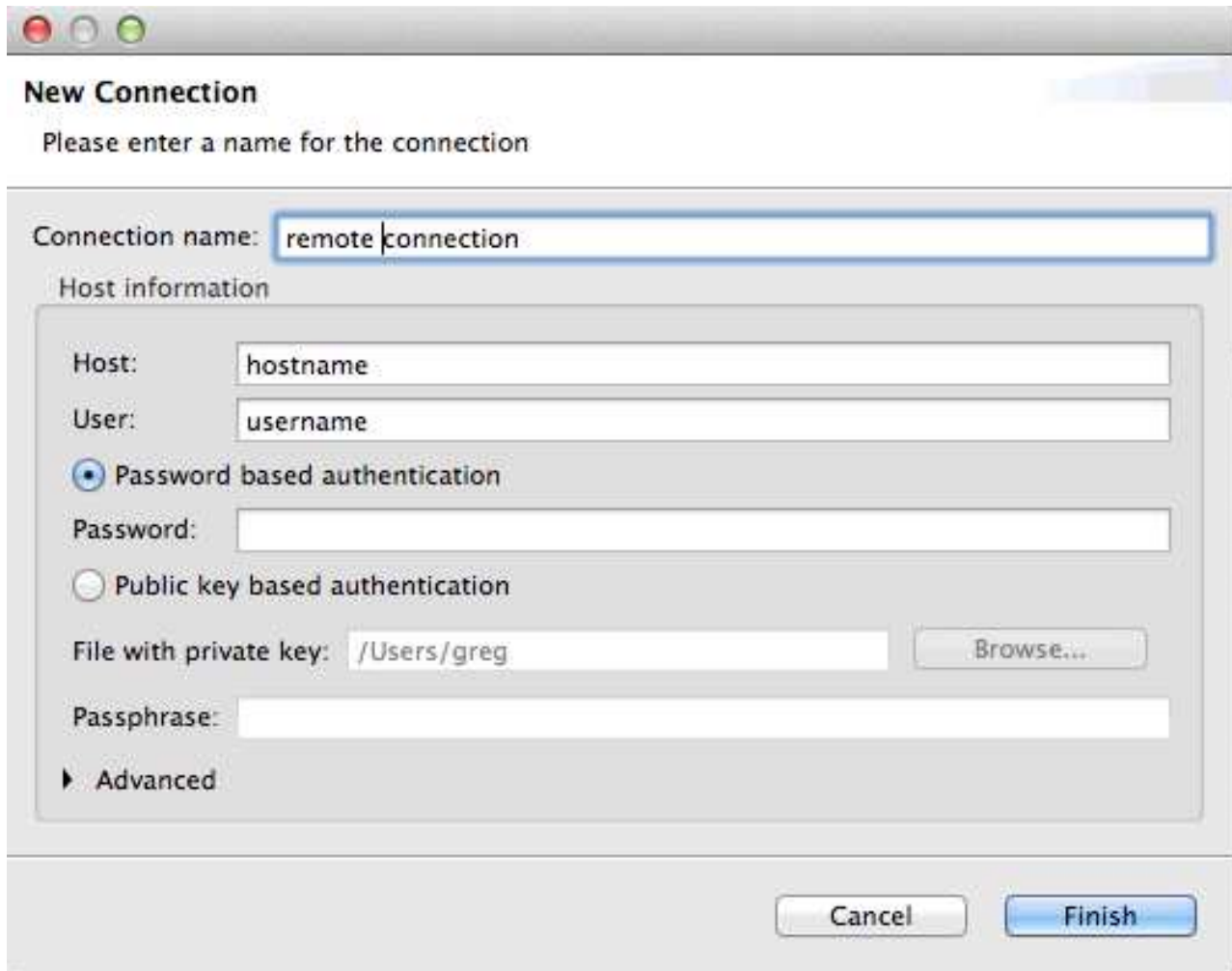


Figure 4. New connection dialog

If the connection is using a password for authentication, the **Password** field should be completed. This is not required if the user has already set up an **ssh** public key on the remote machine. For more information, see the **ssh(1)** man page (linux.die.net/man/1/ssh). If the user has not set an **ssh** public key, but wants to use a public key only for this connection, the **Public key based authentication** radio button should be selected and the file containing the user's private key entered into the **File with private key** field. The **Browse...** button can be used to locate the file using a file browser. If the private key is protected with a passphrase, then it will also need to be entered in the **Passphrase** field.

Once a new connection has been created, the remote file browser will populate with the files and directories in the user's home directory on the target system. The user can then navigate to the location of the application executable, select it, then press the **OK** button.

At this point the application executable will be opened, and **hpctView** will read and analyze the program. When completed, the **Instrumentation** view will populate with the available instrumentation points. Figure 5 on page 29 shows the **Instrumentation** view after opening a typical application executable.

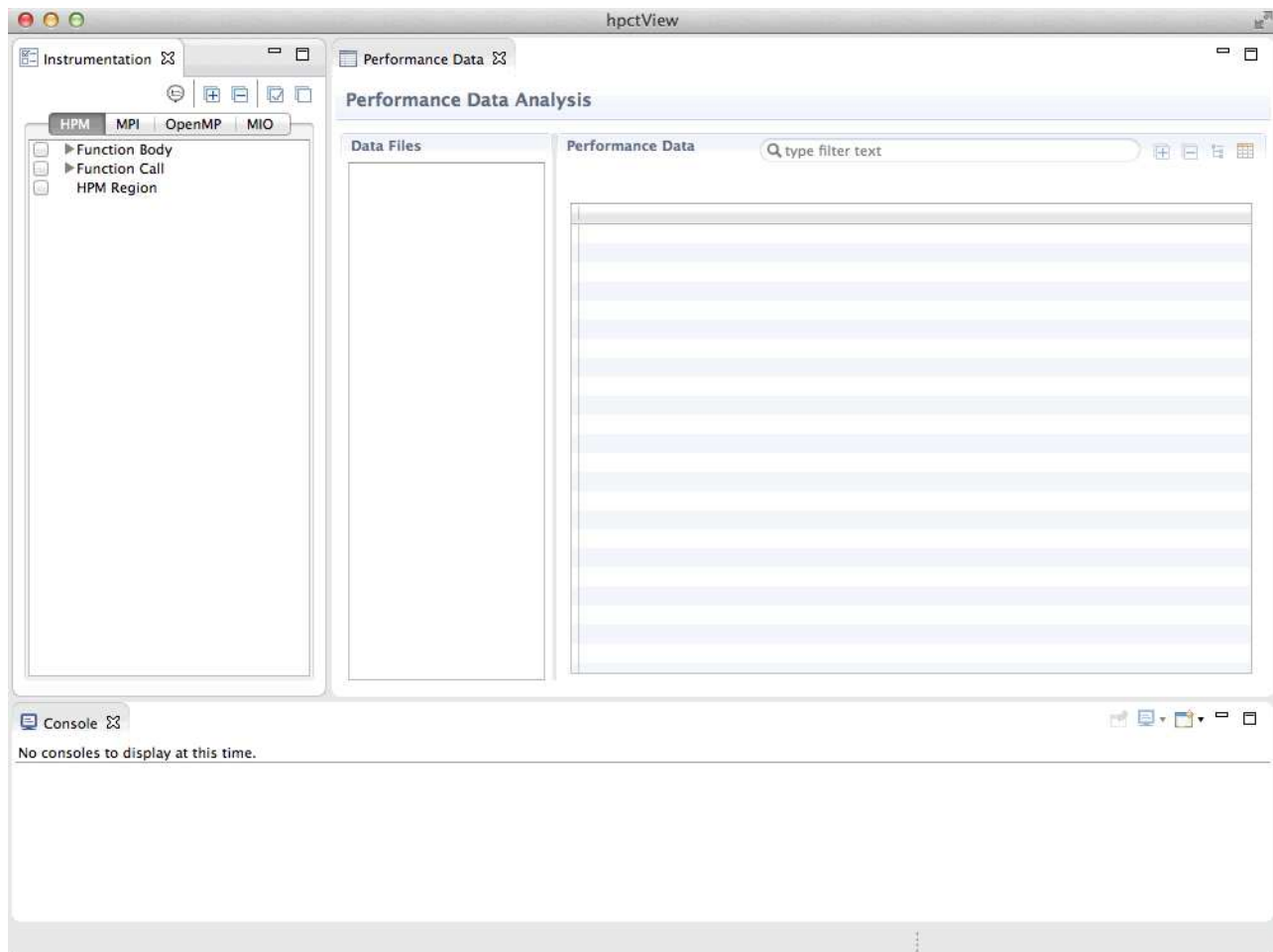


Figure 5. Initial instrumentation view

Using Eclipse PTP

To use Eclipse PTP, follow these steps:

1. Locate the executable in the **Project Explorer** view.
2. Right click over the executable name.
3. Move the mouse over the **HPCT** entry in the pop-up menu that is displayed.
4. Click the **Open Executable** menu entry that is displayed.

At this point the application executable is opened and hpctView will read and analyze the program. When completed, the **Instrumentation** view will populate with the available instrumentation points as shown in Figure 5.

Instrumenting the application

When an application is ready to be instrumented, the **HPM**, **MPI**, **OpenMP**, and **MIO** tabs in the **Instrumentation** view are updated with a tree showing the possible instrumentation points for each type of instrumentation. Nodes in the trees in each of these tabs can be expanded and collapsed by clicking the expand icons. All nodes in the tree can be expanded or collapsed by clicking on the + or - buttons, respectively, as well as by selecting the **Expand All** or **Collapse All** menus from the view's pop-up menu.

The source code for the instrumentation point can be viewed by double clicking on the instrumentation point in the view. This will open a source window with the appropriate region of code highlighted. The main window can be resized to provide more space for viewing the source code. Figure 6 shows the expanded instrumentation view with a number of points selected. It also shows the source code for the highlighted instrumentation point.

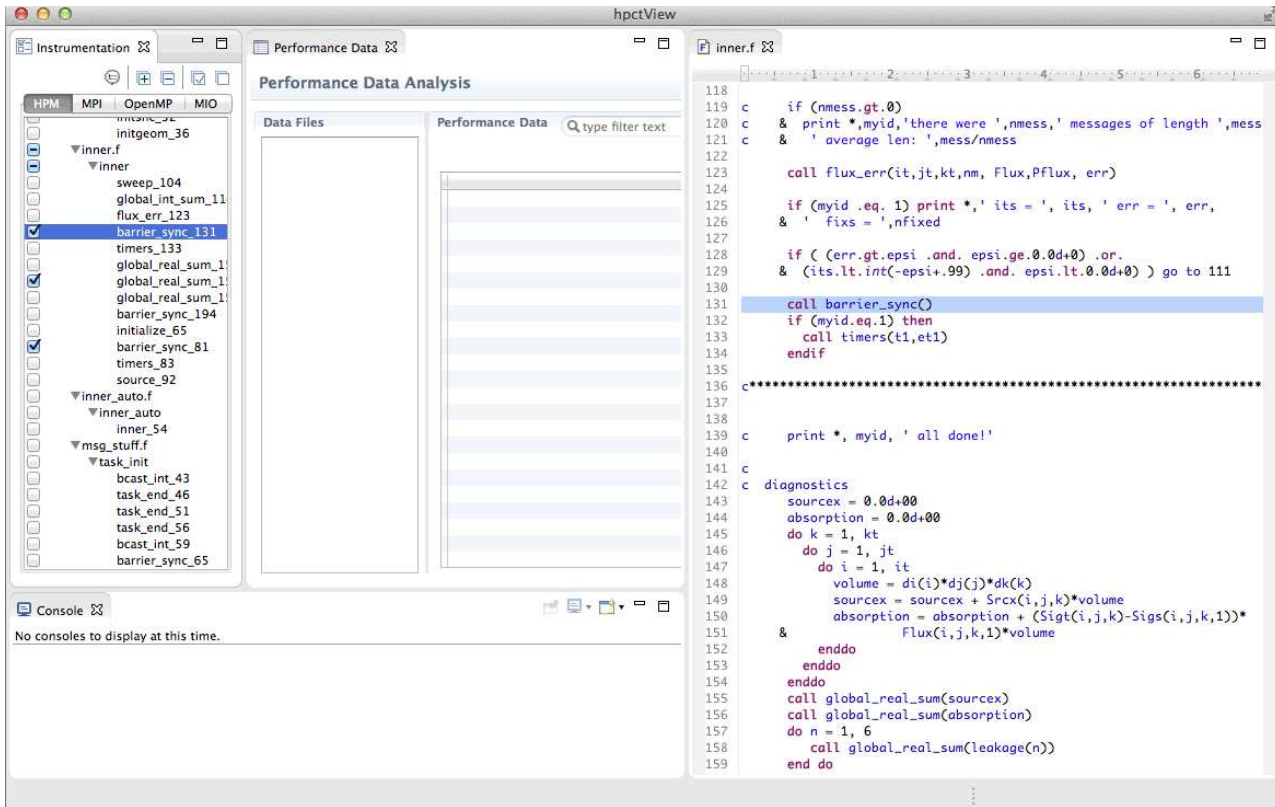


Figure 6. Expanded instrumentation view

An instrumentation point can be selected by checking the box next to the item in the tree. This allows the user to specify one or more points to be instrumented in the application executable simultaneously. The executable is not instrumented until the **Instrument executable** button is pressed, or the **Instrument Executable** option is selected from the view's pop-up menu.

During the instrumentation process, a dialog will be displayed to indicate the status of the instrumentation, or if the instrumentation fails for some reason.

Running the instrumented application

In order to run the instrumented application, a **Profile Configuration** must first be created that contains all the settings necessary to successfully submit the job and collect the required performance data. A profile configuration is only created once for each combination of settings, and can be reused for multiple runs of the application.

The **Profile Configurations** dialog is opened from the main **Run** menu. Figure 7 on page 31 shows the **Profile Configurations** dialog when it is first opened.

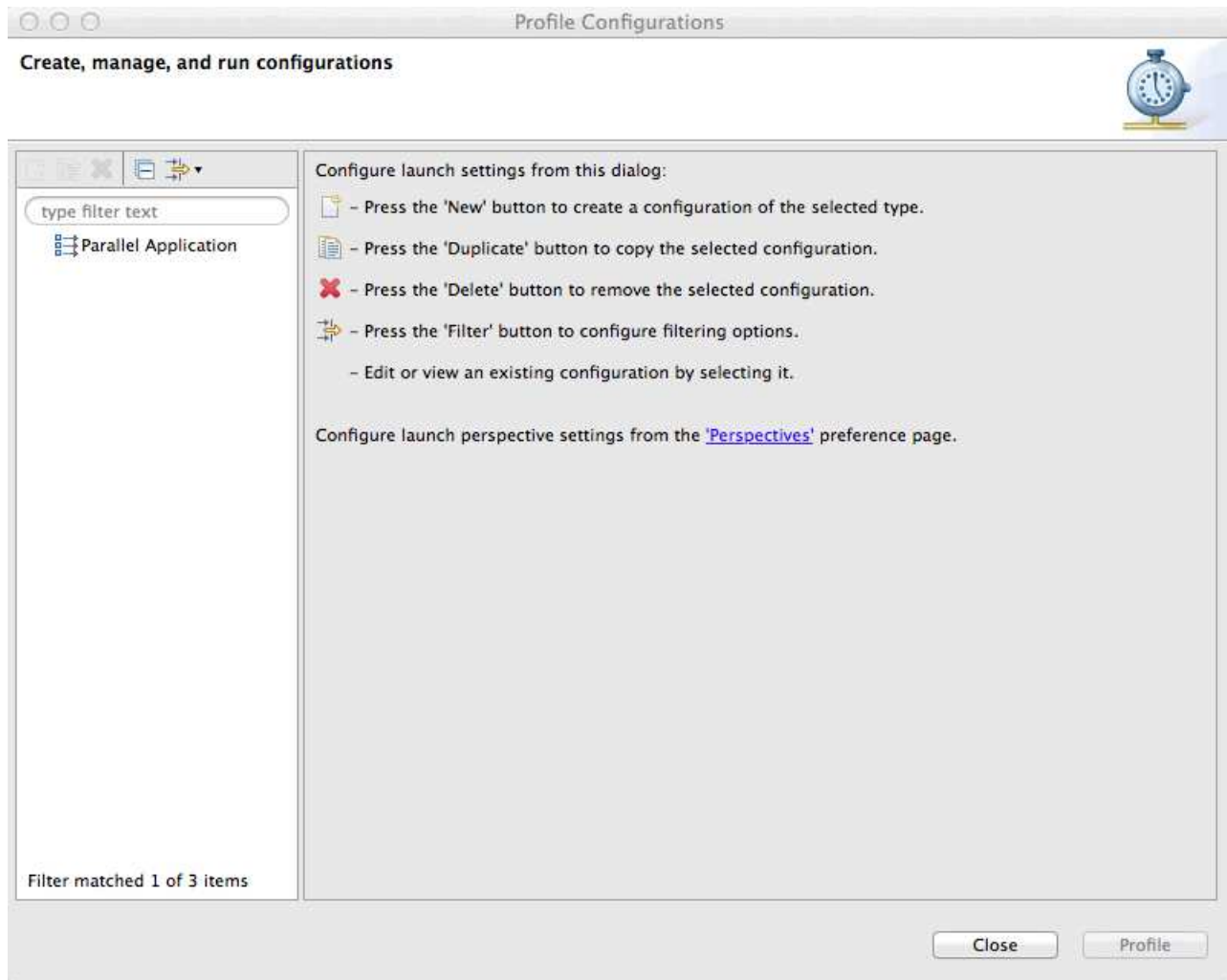


Figure 7. Profile Configurations dialog

If a profile configuration for your application does not exist, you can create a new configuration using this dialog. To do this, select **Parallel Application** from the list on the left side of the dialog, then click the **New** icon just above that list. You can have as many profile configurations as you want, and can have multiple profile configurations for the same application each with different settings.

Once you have created a profile configuration, you enter the necessary data in the **Resources**, **Application**, **Arguments**, **Environment**, and **Performance Analysis** tabs before running your application. There must not be any errors on any of the tabs within the configuration, or the application will not be able to be run.

The **Resources** tab is used to specify the type of target system and any necessary options required to run the application. Figure 8 on page 32 shows the **Resources** tab.

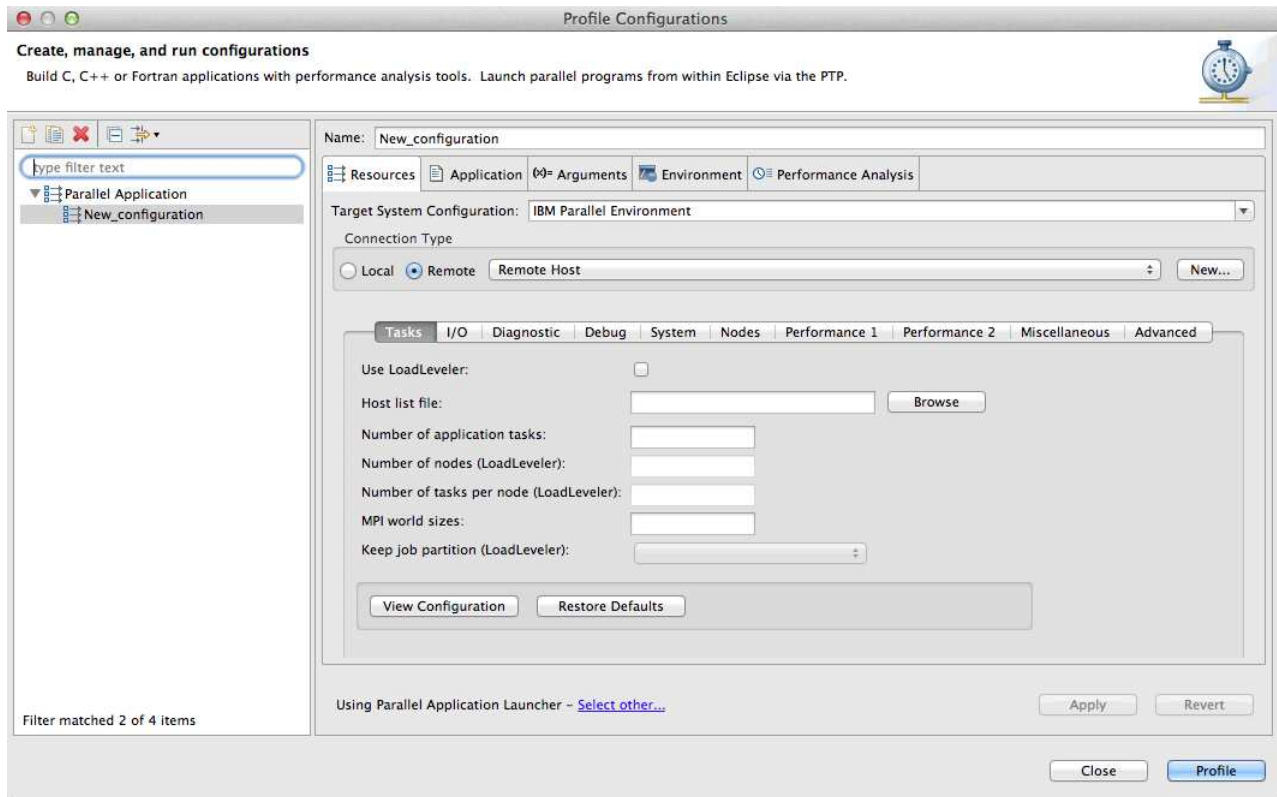


Figure 8. Resources tab

The first step in filling out the **Resources** tab is to ensure that the **Target System Configuration** is correctly selected from the drop-down list at the top of the panel. By default, the IBM Parallel Environment configuration will be selected. In most cases, this should be the correct configuration to use. If this is not correct for your system, you can select an alternative configuration from the drop-down list. Next, check that the connection you specified when opening the executable is selected in the **Remote** drop-down in the **Connection Type** section. If you have multiple systems on which instrumented executables are available, you could select a different connection here. Assuming that you are using the IBM Parallel Environment target system configuration, you should now enter the required IBM PE run-time options by clicking the tabs on the panel and filling in the required values. In particular, you will need to provide a **Host list file** and the **Number of application tasks** for the job to run.

Figure 9 on page 33 shows the **Application** tab, which specifies the application program.

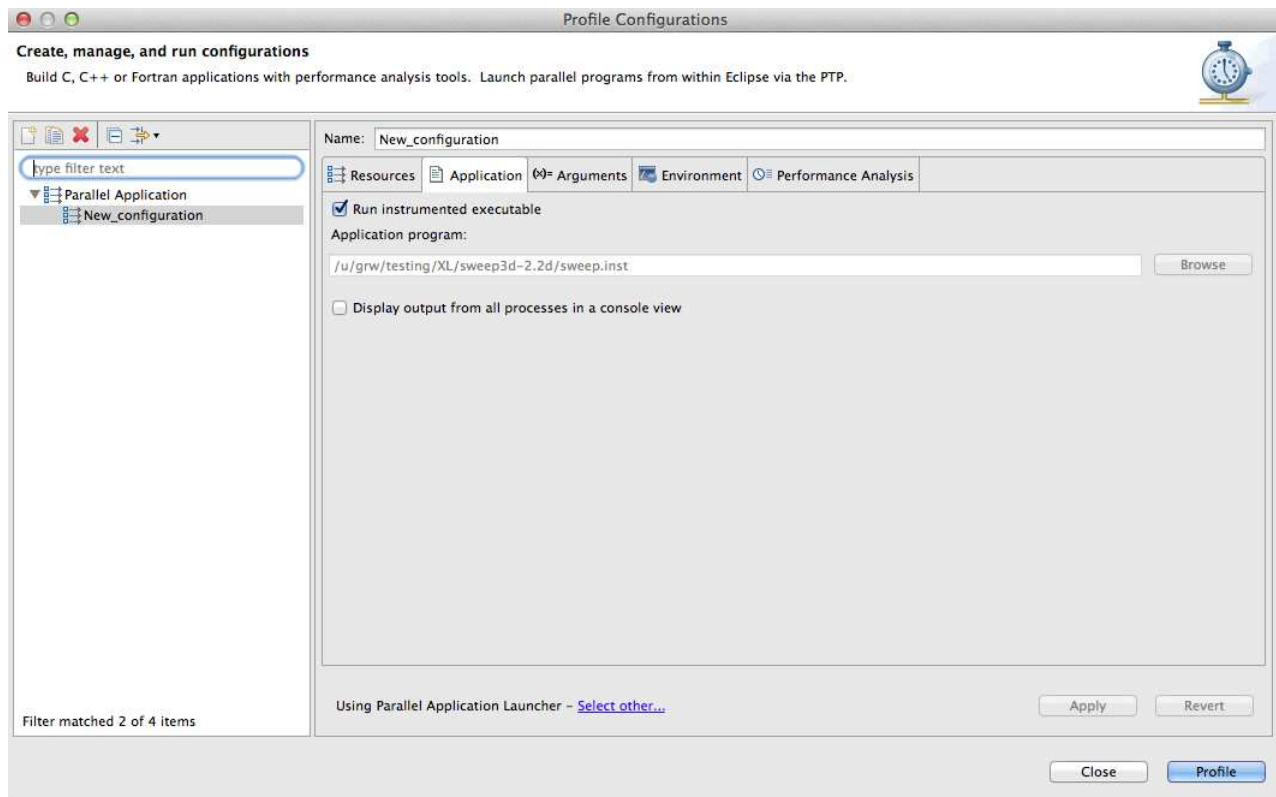


Figure 9. Application tab

The **Application** tab will be preconfigured to automatically run the instrumented executable, and will display the path to the executable in the **Application program** field. If this is not correct, or you want to run a different executable, deselect the **Run instrumented executable** checkbox and then enter the absolute path name to the executable on the target system. The **Browse** button can also be used to browse for the executable.

The **Arguments** tab, which looks like that shown in Figure 10 on page 34, can be used to specify any application program arguments by entering them in the **Program arguments** field. Leave this field empty if there are no arguments. This tab can also change the working directory used when the program is executed. This might be required if the program expects to access files in a particular location. By default, the working directory will be the directory containing the executable. If this needs to be changed, uncheck **Use default working directory** and enter or browse for the new location.

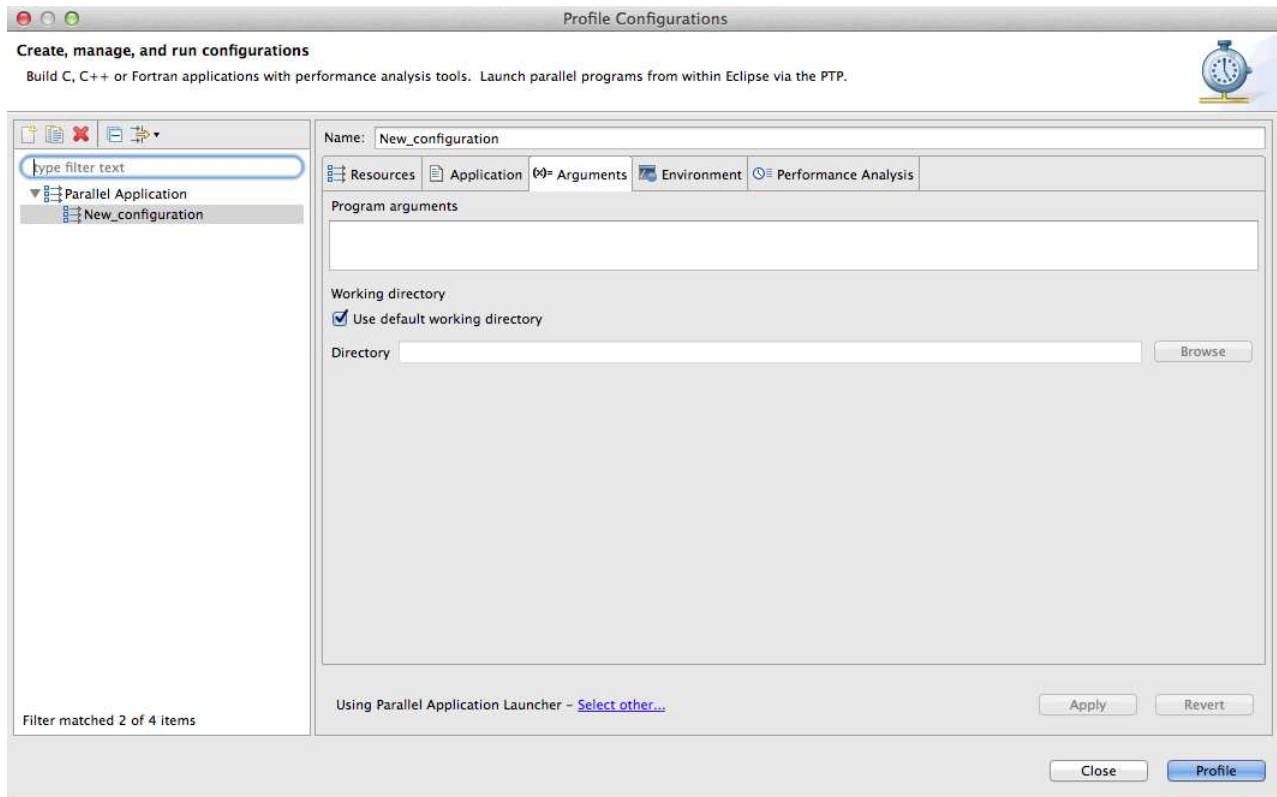


Figure 10. Arguments tab

The **Environment** tab, which looks like that shown in Figure 11 on page 35, can be used to specify any additional environment variables that are required by your application. You can add environment variables to the set passed to your application by clicking the **New** button and filling in the environment variable name and value in the pop-up dialog.

You can modify existing environment variables by clicking the environment variable in the list, clicking **Edit**, and modifying the setting in the pop-up dialog. You can remove environment variables from the set passed to your application by clicking the environment variable name in the list and clicking **Remove**.

The radio buttons at the bottom of the panel control how the environment variables interact with the target environment. If **Append environment to native environment** is selected, any environment variables you set will be appended to the target environment prior to executing the program. If **Replace native environment with specified environment**, only those environment variables set in this tab (and generated by other tabs) will be used when executing the program.

Note: Do not set IBM Parallel Environment or IBM HPC Toolkit environment variables in the **Environment** tab because those settings might conflict with values set on other panels.

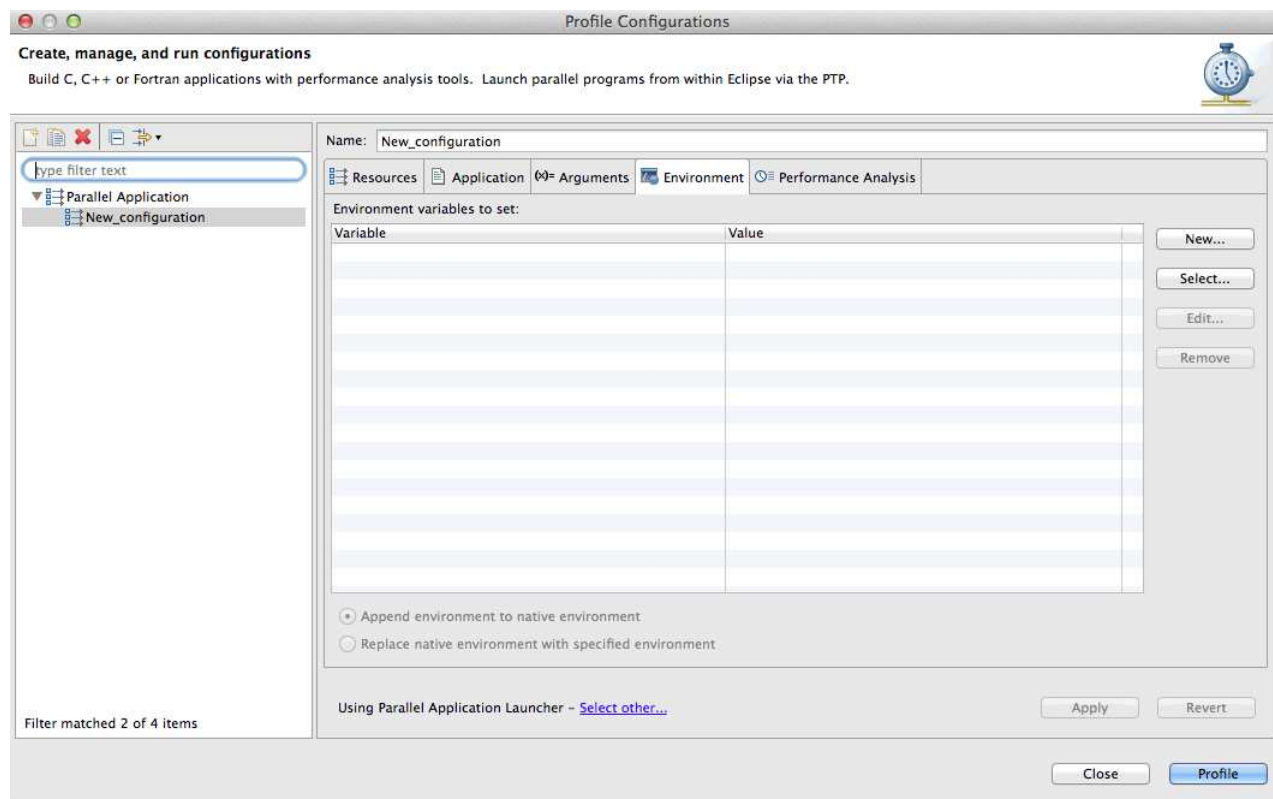


Figure 11. Environment tab

The **Performance Analysis** tab is used to specify the IBM HPC Toolkit run-time options, and is shown in Figure 12 on page 36.

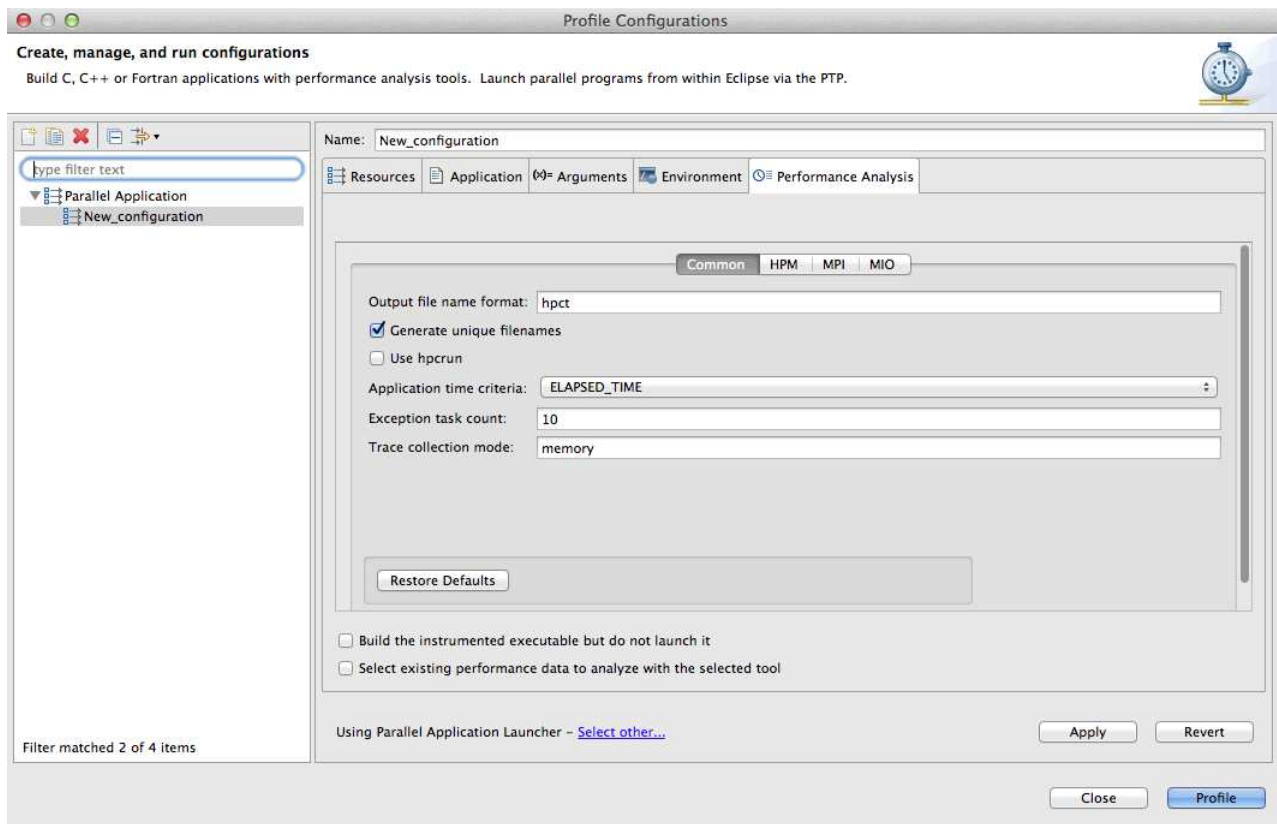


Figure 12. Performance Analysis tab

The **Performance Analysis** tab contains the subtabs, **Common**, **HPM**, **GPM**, **MPI**, and **MIO** that are used to set run-time options for the hardware performance counter, MPI analysis, and I/O analysis tools. The **HPM**, **GPM**, **MPI**, and **MIO** subtabs are described in more detail in the following topics:

- Chapter 7, “Using hardware performance counter profiling,” on page 41
- Chapter 8, “Using GPU hardware counter profiling,” on page 47
- Chapter 9, “Using MPI profiling,” on page 53
- Chapter 10, “Using I/O profiling,” on page 61

The **Common** subtab (shown in Figure 12) specifies settings that are common to all performance tools. These settings are as follows:

Output file name format

The format used to name output data files. This corresponds to the **HPC_OUTPUT_NAME** environment variable, and is normally a prefix like “hpct” that will be prepended to the file name.

Generate unique filenames

If selected, this will append the process ID of the application to the filename. This corresponds to the **HPC_UNIQUE_FILE_NAME** environment variable, and is essential for parallel programs so that each task will generate a separate data file.

Note: For more information about file names for the **Output file name format** or **Generate unique filenames** options, see Appendix A, “Performance data file naming,” on page 197.

Use **hpcrun**

This enables use of the **hpcrun** command in order to collect performance data for a subset of application tasks.

When performance data is collected for a subset of application tasks, you must specify the collection criteria and number of tasks. See “hpcrun - Launch a program to collect profiling or trace data” on page 122 for more information.

Application time criteria

Only used if **Use hpcrun** is selected. Specifies the time criteria used to determine which tasks will have data collected.

ELAPSED_TIME

Determines the subset of tasks using elapsed wall clock time.

CPU_TIME

Determines the subset of tasks using CPU time.

Exception task count

Only used if **Use hpcrun** is selected. This field is used to specify the number of application tasks that for which performance data will be collected. This value will determine how many tasks with the minimum value for the time criteria and how many tasks with the maximum value for the time criteria will have their data collected. In addition, data will be collected for the task closest to the average for the time criteria, and for task zero.

Trace collection mode

Only used if **Use hpcrun** is selected. When collecting traces for MPI tracing or I/O profiling, you must select the collection model to use by specifying either memory or two path names separated by a comma. The first path name is used to hold MPI trace data and the second path name is used to hold I/O trace data. If you specify **memory**, then the trace for each application task is stored in application memory until **MPI_Finalize** is called. If you specify a pair of path names, then trace data is temporarily collected in those files until the application exits, then the final trace files are generated.

For more information about the settings on this panel, use Table 5 to reference the corresponding environment variable description in “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 153.

Table 5. Profile Configurations dialog Common environment variables

Profile configuration field	Environment variable name
Output file name	HPC_OUTPUT_NAME
Generate unique filenames	HPC_UNIQUE_FILE_NAME
Use hpcrun	HPC_USE_HPCRUN
Application time criteria	HPC_EXCEPTION_METRIC
Exception task count	HPC_EXCEPTION_COUNT
Trace collection mode	HPC_TRACE_STORE

Once you have filled in all the panels in the **Profile Configurations** dialog and have no error messages displayed, you can run your application by clicking the **Profile** button. Your application runs using the specified settings. If your application generates any output during its run, this output will be displayed in

the **Console** view. The **Console** view is also a good place to check that the program ran successfully. If the program failed, the reason for the failure should also be displayed here.

When your application completes, hpctView will display a pop-up dialog asking you if you want to view the performance data. If you click **Yes**, then the visualization files generated by the instrumented executable will be added to the list in the **Performance Data** view.

Viewing performance data

The performance data files created by your application are added to the list in the **Performance Data** view. If you did not respond **yes** to the pop-up dialog asking if the performance data files should be loaded, then you can load them manually in hpctView using the **Load Viz Files...** item in the main **File** menu. You can load them manually in Eclipse PTP by following these steps:

1. Select one or more performance data files in the **Project Explorer** view.
2. Right click over those files and move the mouse over the **HPCT** entry in that menu.
3. Click the **Open Performance Files** menu entry.

Once the performance files have been loaded, the **Performance Data** view will look like Figure 13.

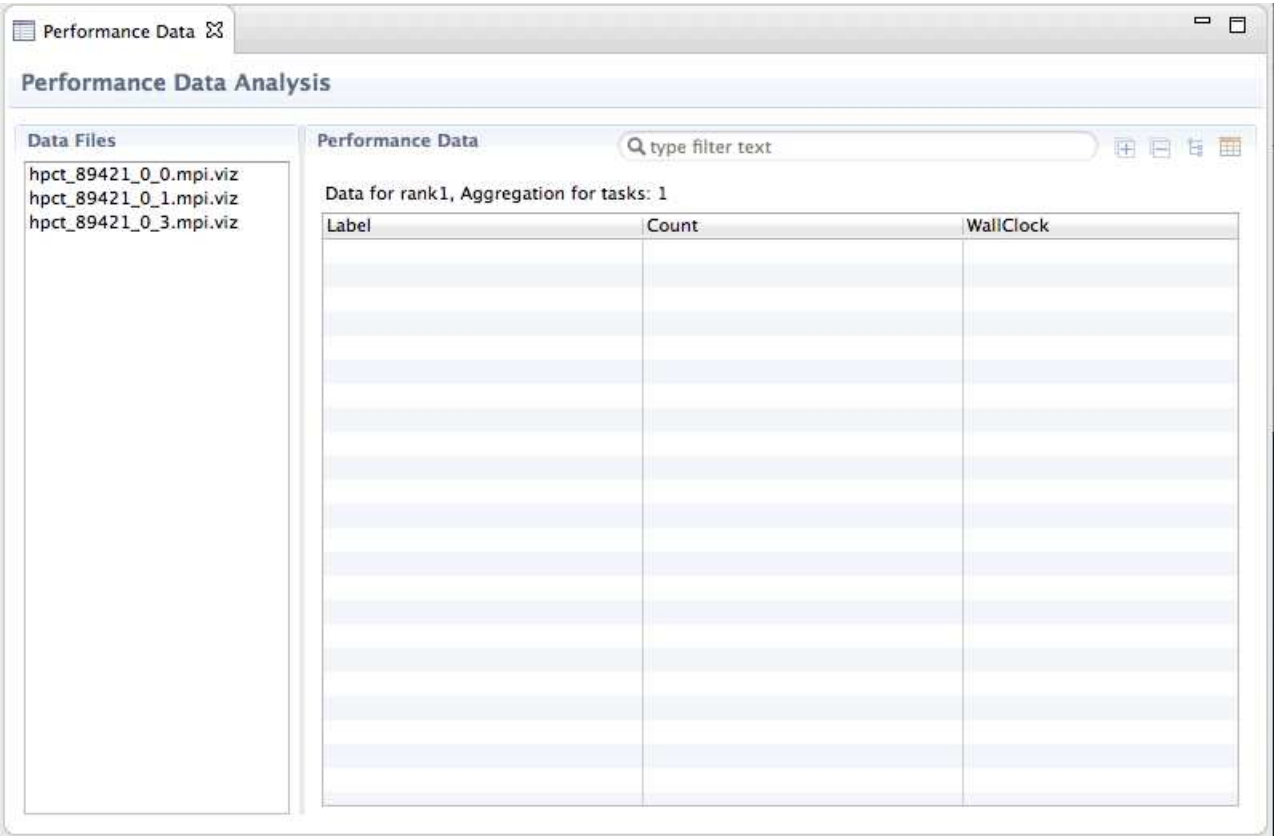


Figure 13. Performance Data view after loading performance files

The contents of the performance data files can be viewed by clicking one or more file names in the **Data Files** list. Figure 14 shows the performance data from rank 1 of the application.

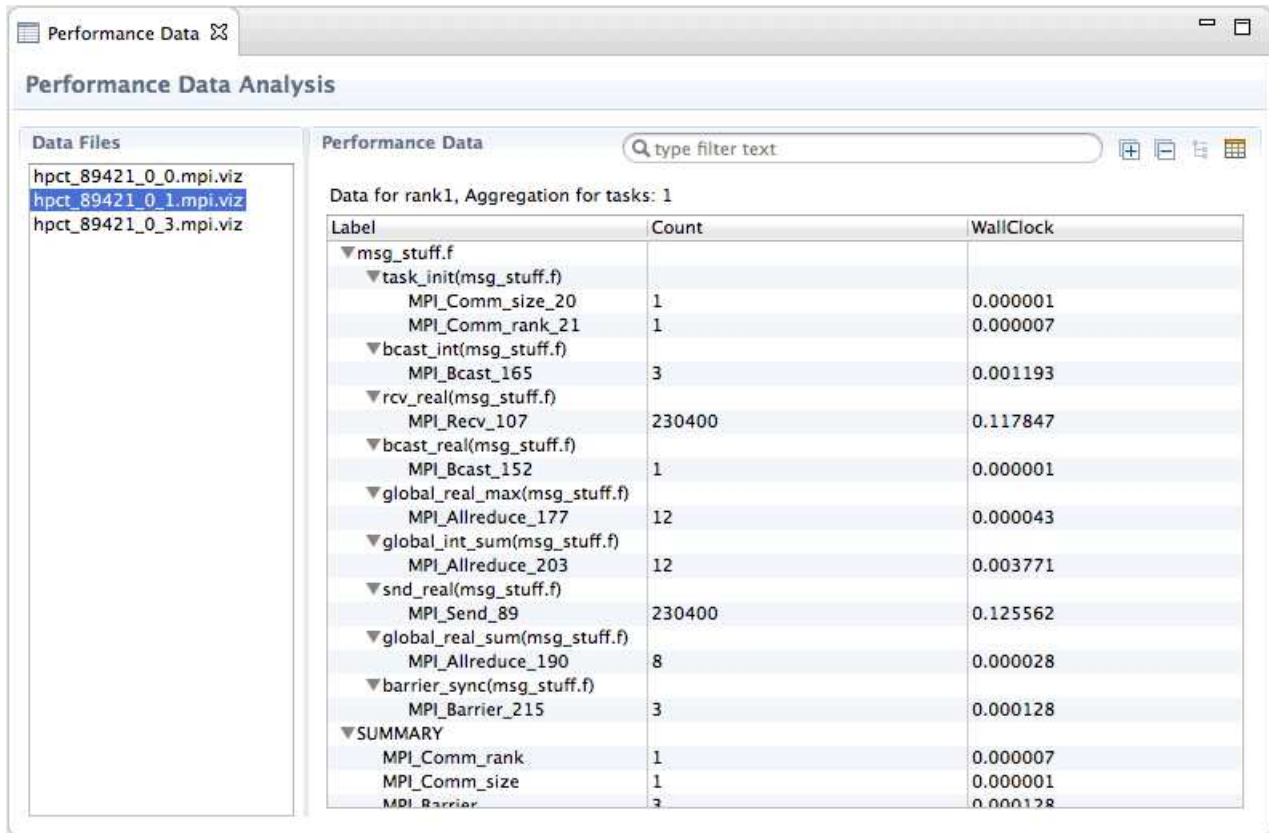


Figure 14. Performance Data view showing application data

If multiple files are selected, the data from the files will be aggregated where possible, and the results will show the average values.

Initially, the **Performance Data** view will show a tree view of the data collected. You can expand all nodes in the tree by clicking the **+** button or individual nodes by clicking the triangle next to the node. You can collapse all nodes by clicking the **-** button or individual nodes by clicking the triangle next to the nodes. You can switch to a tabular mode by clicking on the **Show Data as Flat Table** button and then switch back to the tree mode using the **Show Data as Tree** button.

Figure 15 on page 40 shows the same data viewed as a table.

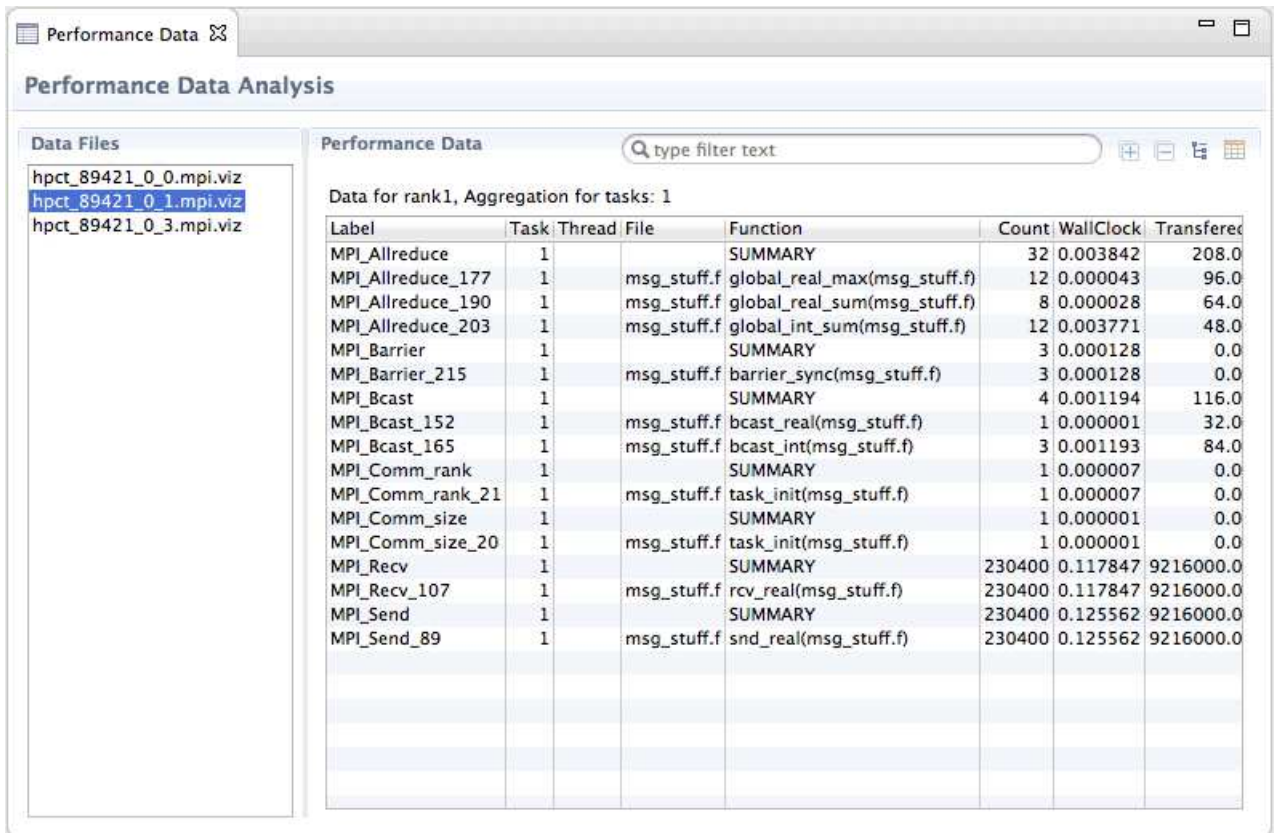


Figure 15. Performance Data view in tabular mode

Each row in the tree or table shows some of the data that was obtained for each instrumented location in the application that was executed. You can sort the tree or table by any column. To do this, click the column header of the column.

Rows in the tree or table can be filtered by entering text in the filter box. Only rows with label text that contains the text in the filter box will be displayed. This is an easy way to reduce the amount of information that is being displayed.

Source code for an instrumentation point can be examined by double clicking on the node, or right-clicking and selecting **Show source** from the pop-up menu. This will open the source file containing the code where the instrumentation point is located, and the corresponding code region will be highlighted in the source code window.

Chapter 7. Using hardware performance counter profiling

The following topics provide information about profiling an application using hardware performance counters.

Preparing an application for profiling

Note: The application must not be compiled with the `-pg` flag.

The application should not contain any calls to functions listed in Table 19 on page 131, because those calls might interfere with correct operation of the instrumentation inserted into the executable. The application should not be linked with the hardware performance counter library, (`libhpc.so`) and must be compiled and linked with the `-g` flag. When the application is compiled on a Power Linux system, the following compiler flags must be used:

`-Wl,--hash-style=sysv -WI,--emit-stub-syms`

Instrumenting the application

Start `hpctView` and load the application executable as described in “Working with the application” on page 25. Once the executable has been opened, make sure the **HPM** tab in the **Instrumentation view** is visible by selecting it (it is normally the default tab to be displayed).

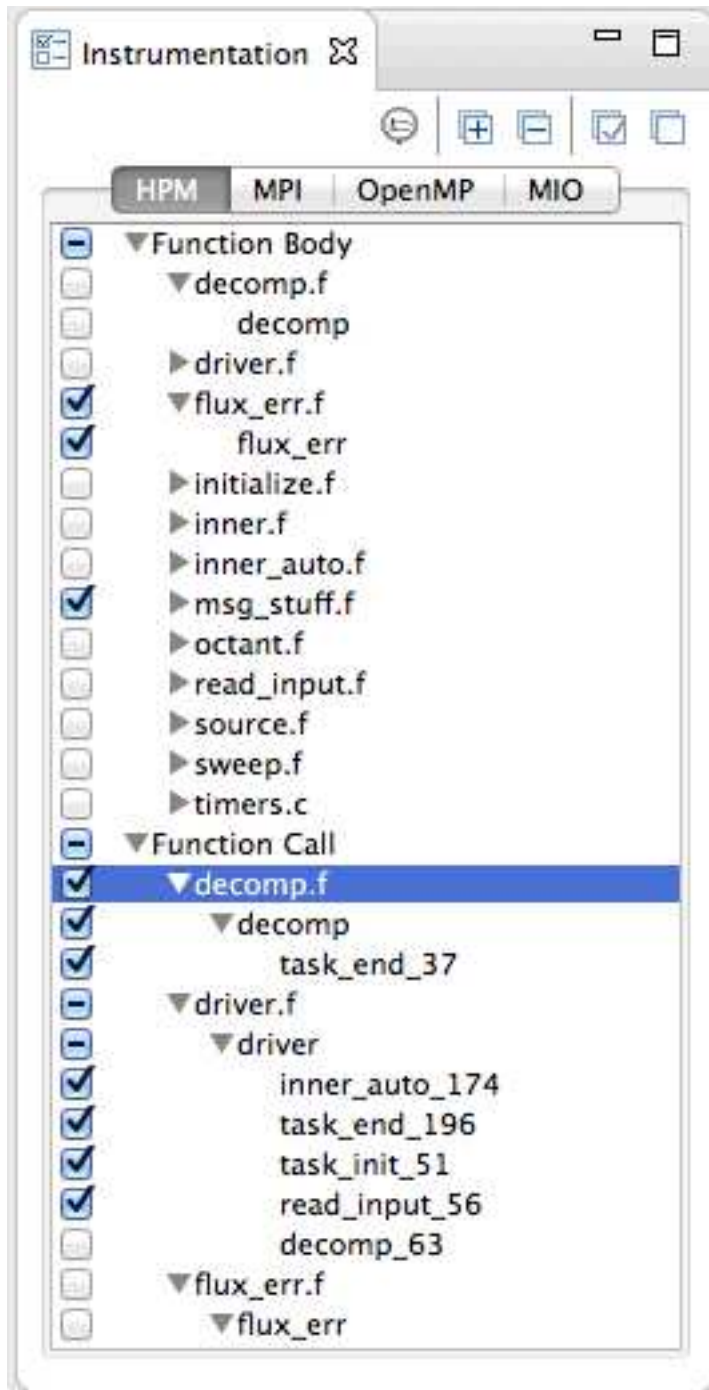


Figure 16. HPM instrumentation tab

Figure 16 shows the HPM tab with a number of instrumentation points selected. There are three classes of instrumentation points that can be shown in this tree:

1. The **Function Body** section lists all the function entry and exit points. Each item in this section is labeled with the name of the function.
2. The **Function Call** section lists all the function call sites, or locations where the functions are invoked. Each item in this section is labeled with the name of the function being called and with the line number of the function call.

3. The **HPM Region** section (not shown in Figure 16 on page 42) lists all the user-selected regions of code, such as a loop, that have been instrumented. The items in this section are labeled with the source file name and the starting and ending line numbers for the region.

Instrumenting a function call site obtains hardware performance counter measurements for that the specific instance of the function being called. The measurements are reported independently of any other call to the same function.

Instrumenting function entry and exit points obtains hardware performance counter measurements for that function every time that function is called, regardless of the caller.

Instrumenting a user-defined region obtains hardware performance measurements specifically for that region of code. Any region of code can be instrumented with the condition that the statements at both the start and end of the region must be executed each time the code region is executed. For example, when instrumenting a branch statement, the branch test and the statement following the branch must be instrumented to ensure the correct data is collected. There is no restriction on the number of user-defined regions that can be instrumented. User-defined regions can also overlap if desired.

User-defined regions are specified by highlighting a region of source code and then adding that region to the **HPM Region** section in the **Instrumentation** view. To add a region, first open the required source file in the source code window, left-click on the starting line of the region, then drag the mouse while the left mouse button is pressed until all required source code lines are highlighted. After the required lines are selected, right-click in the source code window and select the **Add to HPM** option from the pop-up menu.

Any combination of function call sites, function entry and exit points, and user-defined regions can be instrumented. You can select instrumentation by individually selecting items in the **Instrumentation** view or you can select a group of items by selecting a parent item in the tree.

After you have selected the set of instrumentation points that you require, the instrumented application is created by right-clicking in the **Instrumentation** view and selecting **Instrument Executable**, or by clicking the **Instrument** button on the toolbar in the **Instrumentation** view.

Running the instrumented application

Before running the instrumented application, create and configure a Profile Configuration as outlined in “Running the instrumented application” on page 30. In addition, all options in the **HPM** subtab of the **Performance Analysis** tab must be set correctly. Figure 17 on page 44 shows the **HPM** subtab settings.

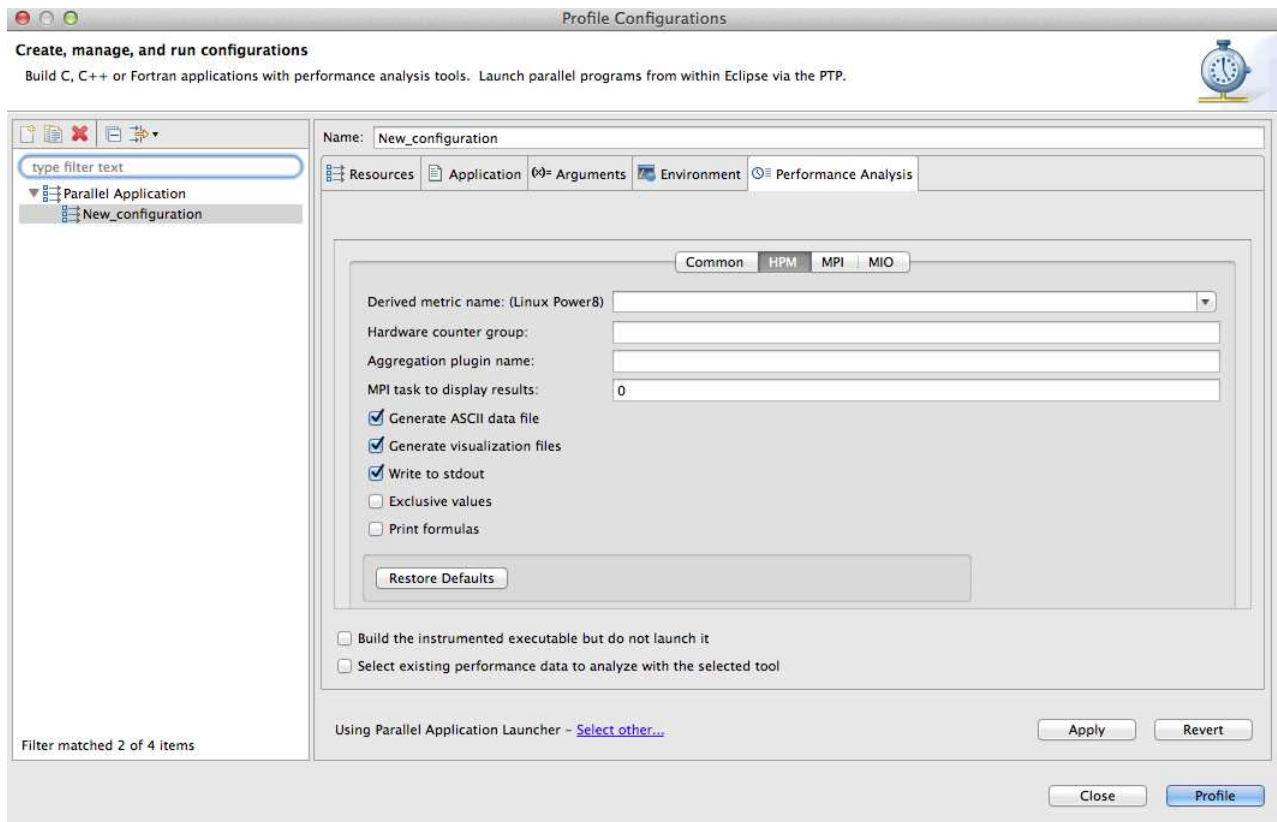


Figure 17. HPM subtab containing hardware counter settings

In order to generate performance data when the instrumented application is run, either the **Derived metric name** or a **Hardware performance counter group** must be selected and either the **Generate visualization files** or **Generate ASCII data file** options must be selected. If the **Generate visualization files** option is selected, the application profile information will be generated in a format that can be loaded into hpctView and displayed in the **Performance Data Analysis** view. If the **Generate OTF2 trace files** option is selected, files containing hardware event counter trace data will be generated in OTF2 format.

Note: Tracing will collect each event counter every time an instrumented function is executed, a call site is executed, or an instrumented region of code is executed. As a result, trace files can quickly become very large, so caution should be exercised when enabling this option.

For more information about settings on this panel, use Table 6 to reference the corresponding environment variable description in “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 153.

Table 6. HPM profile configuration dialog environment variables

Profile configuration field	Environment variable name
Hardware counter group	HPM_EVENT_SET
Aggregation plugin name	HPM_AGGREGATE
Time slice duration	HPM_SLICE_DURATION
MPI task to display results	HPM_PRINT_TASK

Table 6. HPM profile configuration dialog environment variables (continued)

Profile configuration field	Environment variable name
Generate ASCII data file	HPM_ASC_OUTPUT
Generate visualization files	HPM_VIZ_OUTPUT
Generate OTF2 trace files	HPM_TRACE
Write to stdout	HPM_STDOUT
Exclusive values	HPM_EXCLUSIVE_VALUES
Print formulas	HPM_PRINT_FORMULA

Viewing hardware performance counter data

Once the instrumented application has finished running, a dialog will be displayed indicating that the data files were generated and can be loaded. Select **yes** to load them into hpctView. If the files are not loaded at the completion of the run, they can be loaded manually at a later time.

“Viewing performance data” on page 38 describes how to open performance data files and view different presentations of that data.

Chapter 8. Using GPU hardware counter profiling

The following topics provide information about profiling an application using GPU hardware performance counters.

Preparing an application for profiling

Note: The application must not be compiled with the `-pg` flag.

The application should not contain any calls to functions listed in Table 19 on page 131, because those calls might interfere with correct operation of the instrumentation inserted into the executable. The application should not be linked with the hardware performance counter library, (`libhpc.so`) and must be compiled and linked with the `-g` flag. When the application is compiled on a Power Linux system, the following compiler flags must be used:

```
-Wl,--hash-style=sysv -WI,--emit-stub-syms
```

Instrumenting the application

To instrument the application, start `hpctView` and load the application executable as described in “Working with the application” on page 25. Once the executable has been opened, make sure the **HPM** tab in the **Instrumentation** view is visible by selecting it (it is normally the default tab to be displayed).

Figure 16 on page 42 shows the **HPM** tab with a number of instrumentation points selected. There are three classes of instrumentation points that can be shown in this tree:

1. The **Function Body** section lists all the function entry and exit points. Each item in this section is labeled with the name of the function.
2. The **Function Call** section lists all the function call sites or locations where the functions are invoked. Each item in this section is labeled with the name of the function being called and with the line number of the function call.
3. The **HPM Region** section (not shown in Figure 16 on page 42) lists all the user-selected regions of code, such as a loop, that have been instrumented. The items in this section are labeled with the source file name and the starting and ending line numbers for the region.

GPU kernels and CUDA functions are instrumented by identifying their location in the code, then instrumenting the function body, function call, or code region containing the kernel or function calls of interest.

Code regions are specified by highlighting a section of source code and then adding that region to the **HPM Region** section in the **Instrumentation** view. To add a region, do the following:

1. Open the required source file in the source code window.
2. Left-click on the starting line of the region, then drag the mouse while the left mouse button is pressed until all required source code lines are highlighted.
3. After the required lines are selected, right-click in the source code window and select the **Add to HPM** option from the pop-up menu.

Any combination of function call sites, function entry, and exit points, and user-defined regions can be instrumented. You can select instrumentation by individually selecting items in the **Instrumentation** view or you can select a group of items by selecting a parent item in the tree.

After you have selected the set of instrumentation points that you require, the instrumented application is created by right-clicking in the **Instrumentation** view and selecting **Instrument Executable**, or by clicking the **Instrument** button on the toolbar in the **Instrumentation** view.

Running the instrumented application

Before running the instrumented application, create and configure a Profile Configuration as outlined in “Running the instrumented application.” In addition, all options in the **GPM** subtab of the **Performance Analysis** tab must be set correctly. Figure 18 shows the GPM subtab settings.

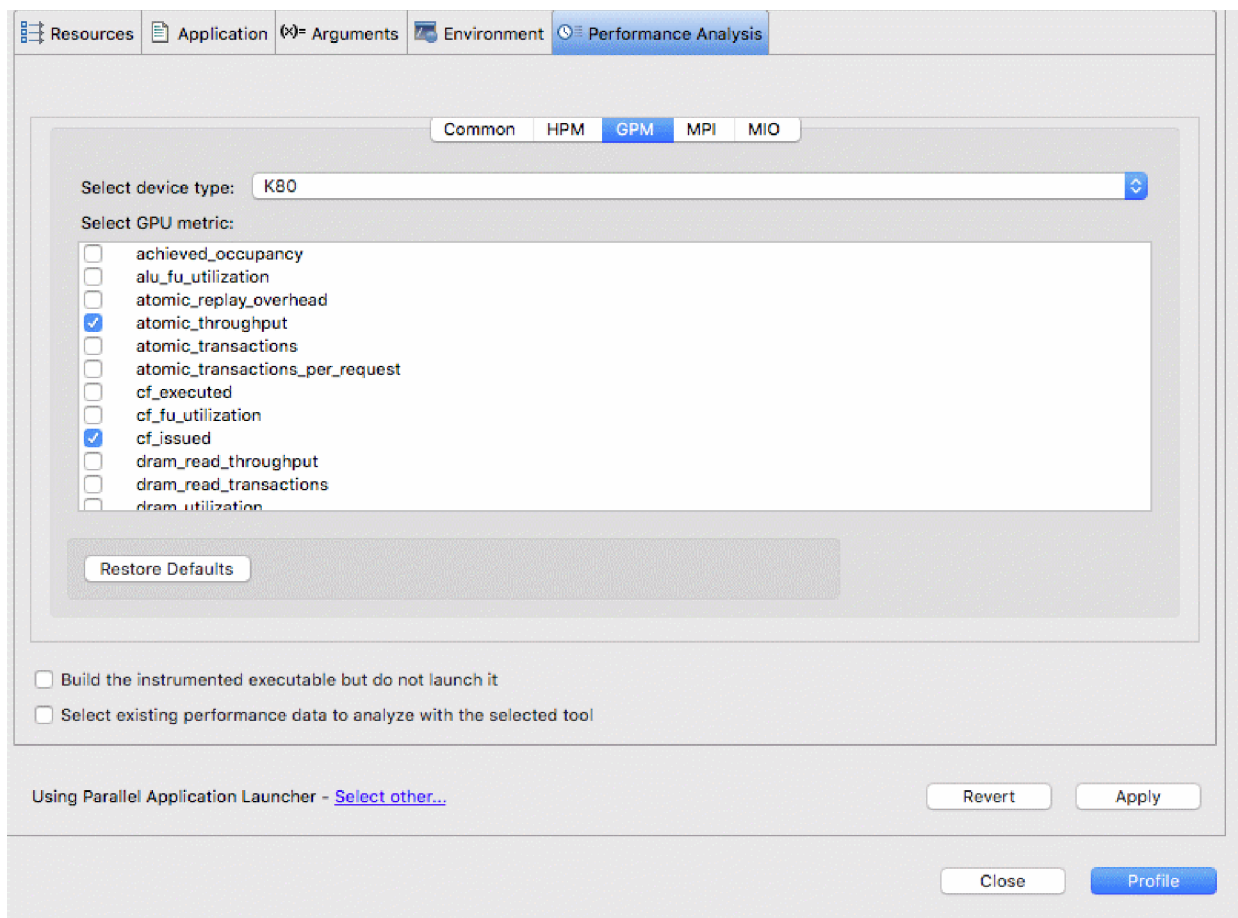


Figure 18. GPM subtab containing GPU metrics settings

In order to generate performance data when the instrumented application is run, a device type must be selected from the **Select device type** drop down. Once a device has been selected, metrics can be selected from the **Select GPU metric** list by clicking the checkbox next to the desired metric.

For more information about settings on this panel, use Table 7 to reference the corresponding environment variable description in “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 153

Table 7. GPM profile configuration dialog environment variable

Profile configuration field	Environment variable name
Select GPU metric	GPM_METRIC_SET

Viewing GPU hardware performance counter data

Once the instrumented application has finished running, a dialog will be displayed indicating that the data files were generated and can be loaded. Select **yes** to load them into hpctView. If the files are not loaded at the completion of the run, they can be loaded manually at a later time.

When the performance data is loaded, the **Trace View** will open automatically to display a visualization of the performance data. Figure 19 on page 50 shows example data being displayed in the **Trace View**.

The horizontal axis of the graph displays the timestamp associated with each trace record. The vertical axis displays the counter or metric value. A different colored line on the graph is used for each counter or metric, and a legend is displayed at the bottom of the view showing which color corresponds to the counter or metric value. Counters with widely varying values can also be displayed, as the left hand vertical axis displays a linear scale, while the right hand vertical axis displays a logarithmic scale. Very large or small values will be placed on the logarithmic scale rather than the linear scale so they can easily be viewed.

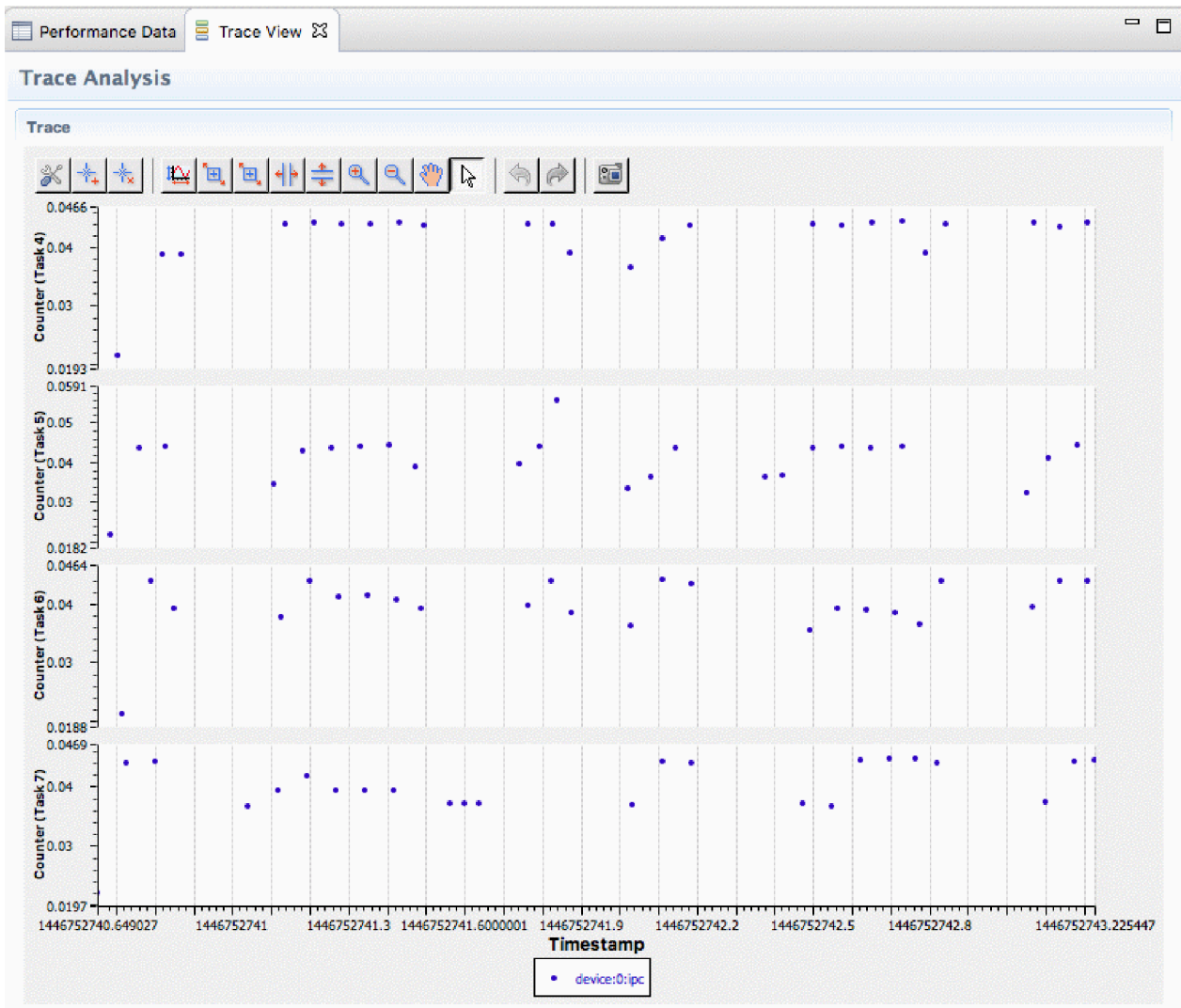


Figure 19. Trace View displaying GPU performance data from an MPI program

Displaying combined CPU and GPU performance data

It is possible to display combined CPU and GPU performance data on a single timeline. This is done by using the following three step process:

1. Instrument an application to collect either CPU or GPU (or both) performance data, and run the application. Once the application has completed execution, load the CPU or GPU performance data into the user interface using the **File→OTF2 Trace→Load Trace (*.otf2)** menu. At this point, a view should be displayed showing the trace information.
2. If the application was instrumented only to collect CPU or GPU performance data, instrument the application again for the alternative data to be collected, then run the application again. If the application was instrumented to collect both CPU and GPU data, skip to step 3.

3. Ensure that the **Trace View** that you want to display the combined data is visible. Add the additional performance data by using the **File→OTF2 Trace→Add Trace (*.otf2)** menu. The Trace View should now show the combined data.

Chapter 9. Using MPI profiling

The topics that follow provide information about profiling the MPI calls in an application.

Preparing an application for profiling

Note: The application must not be compiled with the `-pg` flag.

The application should not contain any calls to functions listed in Table 19 on page 131, because those calls might interfere with correct operation of the instrumentation inserted into the executable. The application should not be linked with the MPI profiling library, (**libmpitrace.so**) and must be compiled and linked with the `-g` flag. When the application is compiled on a Power Linux system, the following flags must be used:

`-Wl,--hash-style=sysv -WI,--emit-stub-syms`

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

Instrumenting the application

Start hpctView and load the application executable as described in “Working with the application” on page 25. Once the executable has been opened, make sure that the **MPI** tab is visible in the **Instrumentation** view by selecting it. Figure 20 on page 54 shows the **MPI** instrumentation tab.

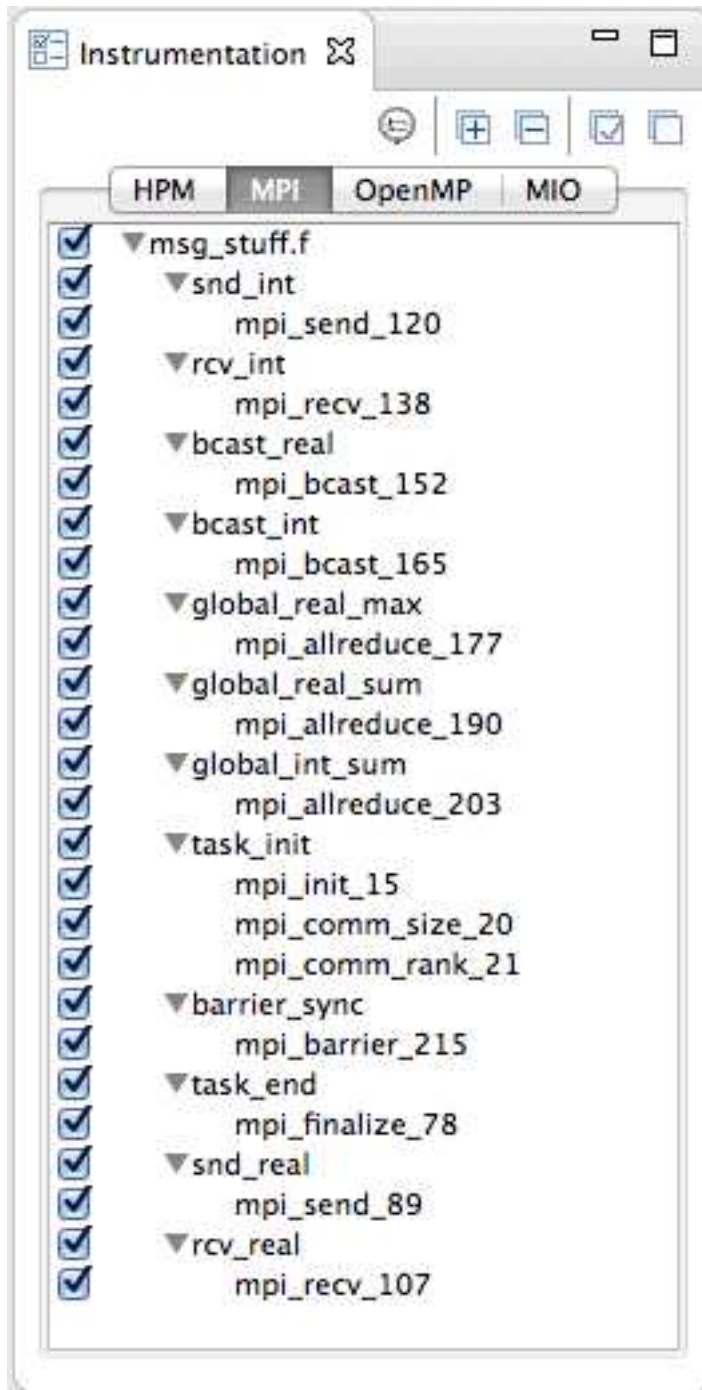


Figure 20. MPI instrumentation tab

The **MPI** instrumentation tab shows the instrumentation points tree expanded, displaying all MPI function calls that can be instrumented. Each instrumentation location is labeled with the name of the MPI function and the line number where the MPI function call is located.

Any number of MPI instrumentation points can be selected. If a leaf node in the tree is selected, only a single MPI function call is instrumented. If a node labeled with an application function name or file name is selected, all MPI function calls

within the file or function named by the enclosing node are instrumented. MPI function calls can be deselected by clicking the corresponding highlighted node in the instrumentation points tree.

MPI function calls can also be instrumented from the source view. With a source view open, left-click on a starting line and drag to the end of the region of interest while holding the left mouse button down. All MPI function calls in this region of code can then be instrumented by right-clicking and selecting **Select MPI**. The tree in the **Instrumentation** view will be updated with the instrumentation points accordingly. The same process can be used to deselect instrumentation points. To do this, select the region of code, right click, then choose **Deselect MPI** from the pop-up menu.

Once the set of instrumentation points has been selected the application executable is instrumented by right-clicking in the **Instrumentation** view, then clicking **Instrument Executable**, or by clicking the **Instrument** icon in the toolbar in the **Instrumentation** view.

Running the application

Before running the instrumented application, a Profile Configuration needs to be created and configured, as outlined in “Running the instrumented application” on page 30. An application with MPI instrumentation will always generate MPI profile information, however to generate MPI trace information, the options in the **MPI** subtab of the **Performance Analysis** tab must be set correctly. Figure 21 on page 56 shows the **MPI** subtab trace settings.

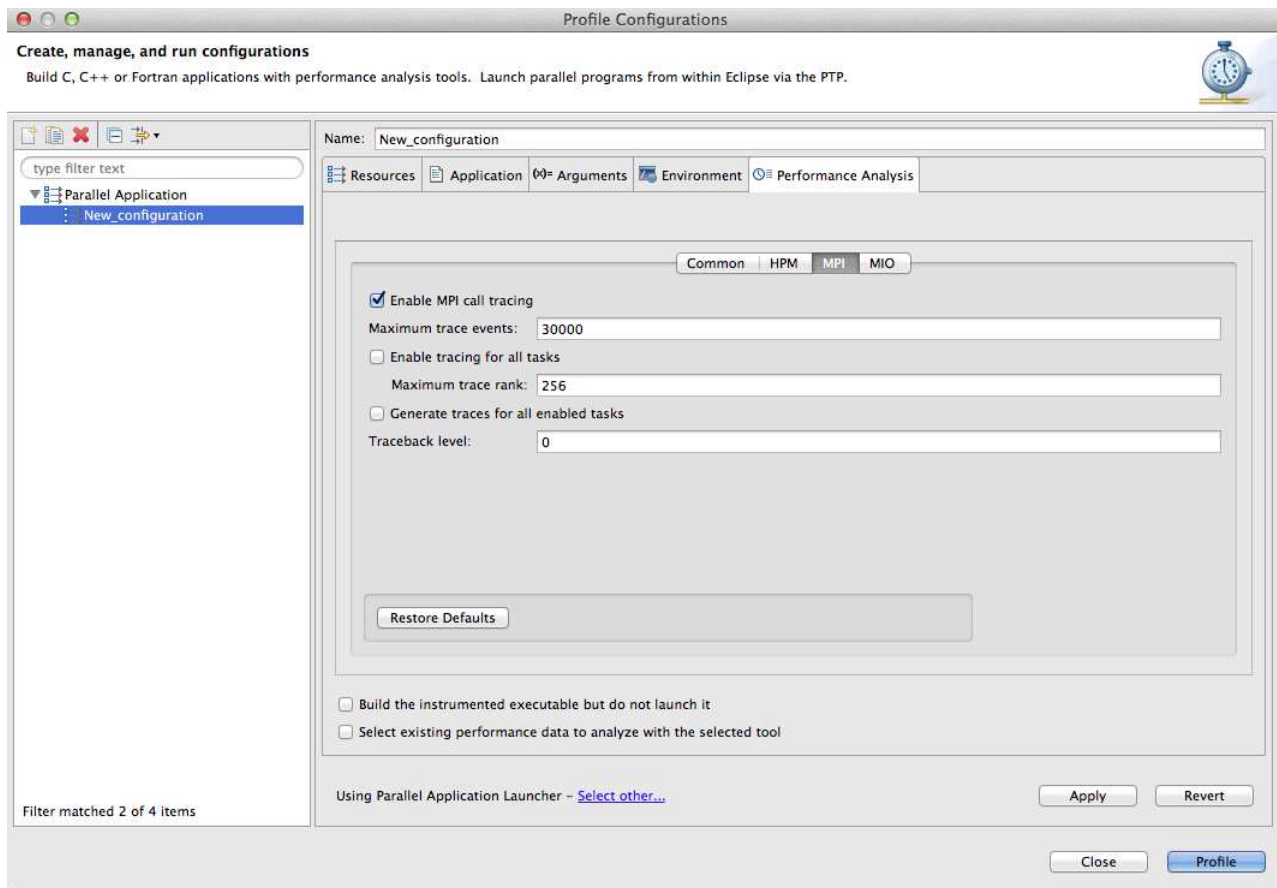


Figure 21. Subtab containing MPI trace settings

The **Enable MPI call tracing** setting is used to enable or disable MPI tracing. If the checkbox is unselected, the remaining settings will be disabled, and the instrumented application will not generate MPI trace data. If the checkbox is selected, then MPI tracing will be enabled, and the remaining settings will be used.

By default, the MPI profiling library enables tracing for only the first 256 tasks. If tracing is required for more tasks, set the task number with the highest rank in the **Maximum trace rank** field, or select **Enable tracing for all tasks**. In addition, the profiling library normally only generates trace files for the enabled tasks with the minimum, maximum, and median MPI communication time. A trace will also be generated for task zero if task zero is not the task with minimum, maximum, or median MPI communication time. To generate trace files for all enabled tasks, select the **Generate traces for all enabled tasks** option. If the application executes many MPI function calls, it might be necessary to increase the value of the **Maximum trace events** setting to a higher number than the default to avoid overwriting the event buffer. The application will normally display a message indicating that an event buffer overflow occurred.

For more information about the settings on this panel, use Table 8 to reference the corresponding environment variable description in “MT_trace_start, mt_trace_start - Start or resume the collection of trace events” on page 192.

Table 8. MPI trace profile configuration environment variables

Profile Configurations field name	Environment variable
Enable MPI call tracing	TRACE_ALL_EVENTS

Table 8. MPI trace profile configuration environment variables (continued)

Profile Configurations field name	Environment variable
Maximum trace events	MAX_TRACE_EVENTS
Enable tracing for all tasks	OUTPUT_ALL_RANKS
Maximum trace rank	MAX_TRACE_RANK
Generate traces for all enabled tasks	OUTPUT_ALL_RANKS
Traceback level	TRACEBACK_LEVEL

You can run your application as described in “Running the instrumented application” on page 30.

Viewing MPI profiling data

Once the instrumented application has finished running, a dialog will be displayed indicating that the data files were generated and can be loaded. Select **yes** to load them into hpctView. If the files are not loaded at the completion of the run, they can be loaded manually at a later time. Figure 22 shows the MPI profiling information displayed in the Performance Data view.

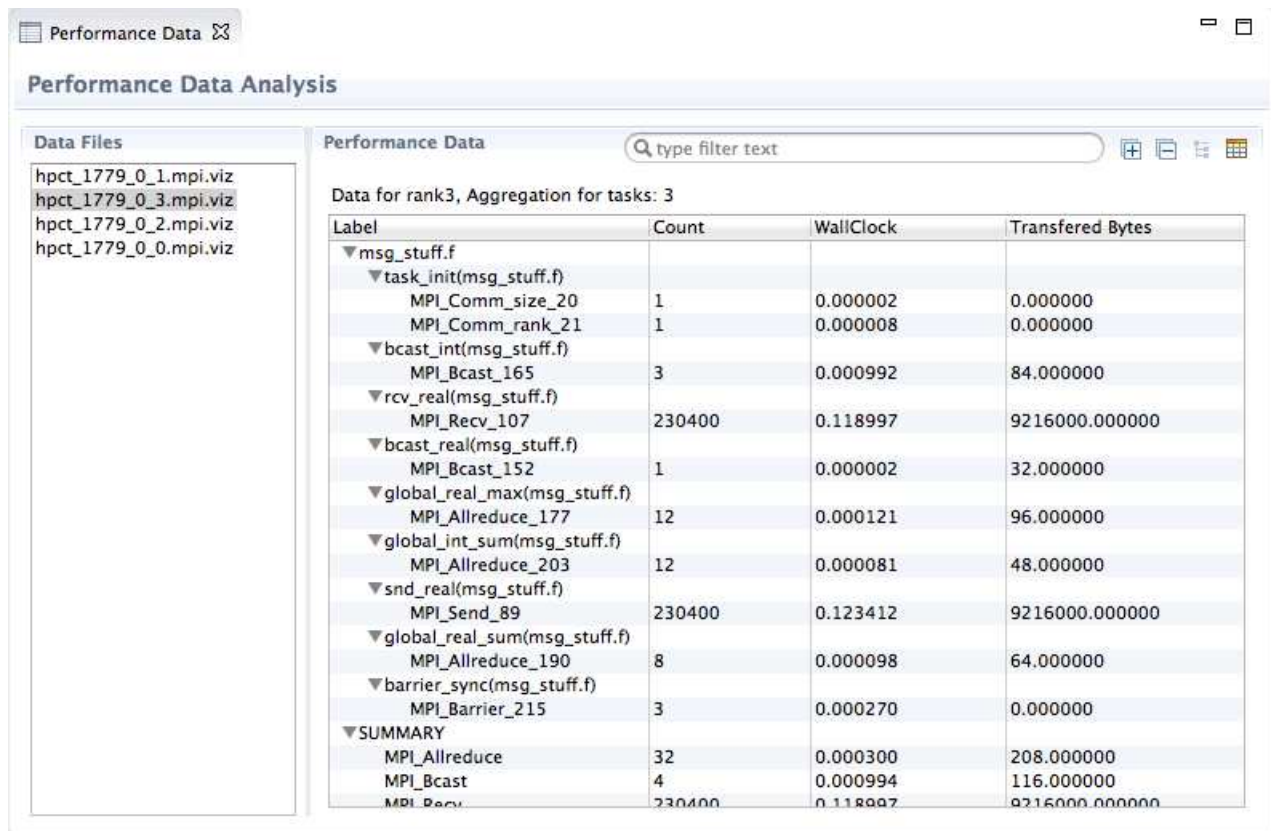


Figure 22. Performance Data view with MPI profiling data

This view 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. The data can also be viewed in tabular format by selecting the **Show Data as a Flat Table** button from the view toolbar.

The performance data view shows data from a single task at one time. If you want to see performance data from a different task, select that task from the **Data Files** panel in the view. If you select multiple files simultaneously the view will display the aggregated data.

Viewing MPI traces

A trace of the MPI activity in the application can be viewed by right-clicking in the **Data Files** panel and selecting **Load Trace Data** from the pop-up menu. The trace file will then be loaded and the **MPI Trace** view opened. This could take some time if the trace file is very large. Figure 23 shows the MPI trace view.



Figure 23. MPI trace view









The panel on the right side of the **MPI Trace View** is used to select which MPI functions are displayed in the trace on the left side of the view. Initially, all MPI function calls that were executed are displayed. Trace events for specific MPI functions can be hidden or displayed by clicking on the MPI function labels in the list. Trace events are displayed for MPI functions that have their labels checked while trace events are hidden for MPI functions with unchecked labels.

The left panel is the MPI trace viewer. This pane displays a timeline-based view of the 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 right pane. The numbers in the middle of the trace are timestamps identifying those points in the trace timeline.

The toolbar above the trace panel provides controls for navigating through the trace. These are described in more detail in Table 9:

Table 9. MPI trace timeline navigation

Button	Description	Action
	Push Left	Scrolls the trace left (decreasing time)
	Push Right	Scrolls the trace right (increasing time)
	Zoom In Horizontal	Zoom in on the X (time) axis
	Zoom Out Horizontal	Zoom out on the X (time) axis
	Restore Horizontal	Reset the X axis to its initial state
	Zoom In Vertical	Zoom in on the Y (task) axis
	Zoom Out Vertical	Zoom out on the Y (task) axis
	Restore Vertical	Reset the Y axis to its initial state

The trace can be scrolled horizontally or vertically by using the scroll bars at the bottom and right of the panel respectively. It is also possible to zoom in on the X axis by left-clicking at a point in the timeline, then dragging the mouse left or right to the desired ending point in the timeline then releasing the left mouse button.

If you double left-click over an MPI function's trace entry, the source file where the MPI function call was made will be displayed. If you right-click and hold the right mouse button over an MPI function's trace entry, it will display a pop-up with additional information about the function call.

Chapter 10. Using I/O profiling

The topics that follow provide information about profiling the I/O usage of an application.

Preparing an application for profiling

Note: The application must not be compiled with the `-pg` flag.

The application must be compiled and linked with the `-g` compiler flag. When the application is compiled on a Power Linux system, the following compiler flags must be used:

```
-Wl,--hash-style=sysv -WI,--emit-stub-syms
```

Instrumenting the application

Start hpctView and load the application executable as described in “Working with the application” on page 25. Once the executable has been opened, make sure the **MIO** tab in the **Instrumentation** view is visible by selecting it. Figure 24 on page 62 shows the MIO instrumentation view.

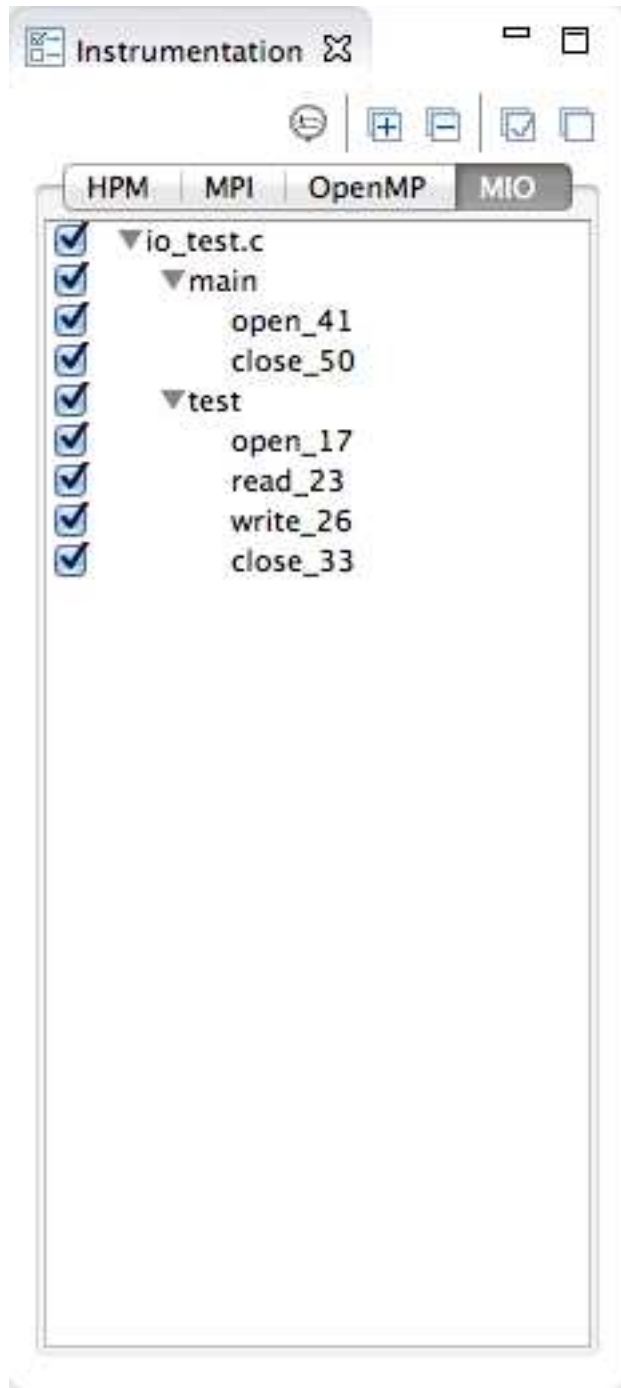


Figure 24. MIO instrumentation view

The **MIO instrumentation** view shows the instrumentation points in the 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. Selecting a leaf node will place instrumentation only at the specific instrumentation point. Selecting a non-leaf node will place instrumentation at all leaf nodes that are child nodes of the selected non-leaf node.

Note: In order for I/O profiling data to be collected for a file, the **open** and **close** system calls that open and close the file must be instrumented.

After you have selected the required instrumentation points, the application executable is instrumented by right-clicking in the **Instrumentation** view, then selecting **Instrument Executable**, or by clicking the **Instrument** button in the toolbar in the **Instrumentation** view.

Running the application

I/O profiling works by intercepting I/O system calls for any files for which performance measurements are to be obtained. In order to obtain the performance measurement data, the **I/O profiling** options setting on the **MIO** subtab of the **Performance Analysis** tab configured to use the following:

***[trace/xml/events]**

This specifies:

- A file-name-matching pattern (* in this case)
- The use of the **MIO** trace module
- The **xml** and **events** options for the trace module

The effect of this setting is to instruct MIO to apply the options to all files opened by the application, generate performance data in XML format, and to use the default naming for the I/O trace file. The **MIO** subtab of the **Performance Analysis** tab is shown in Figure 25.

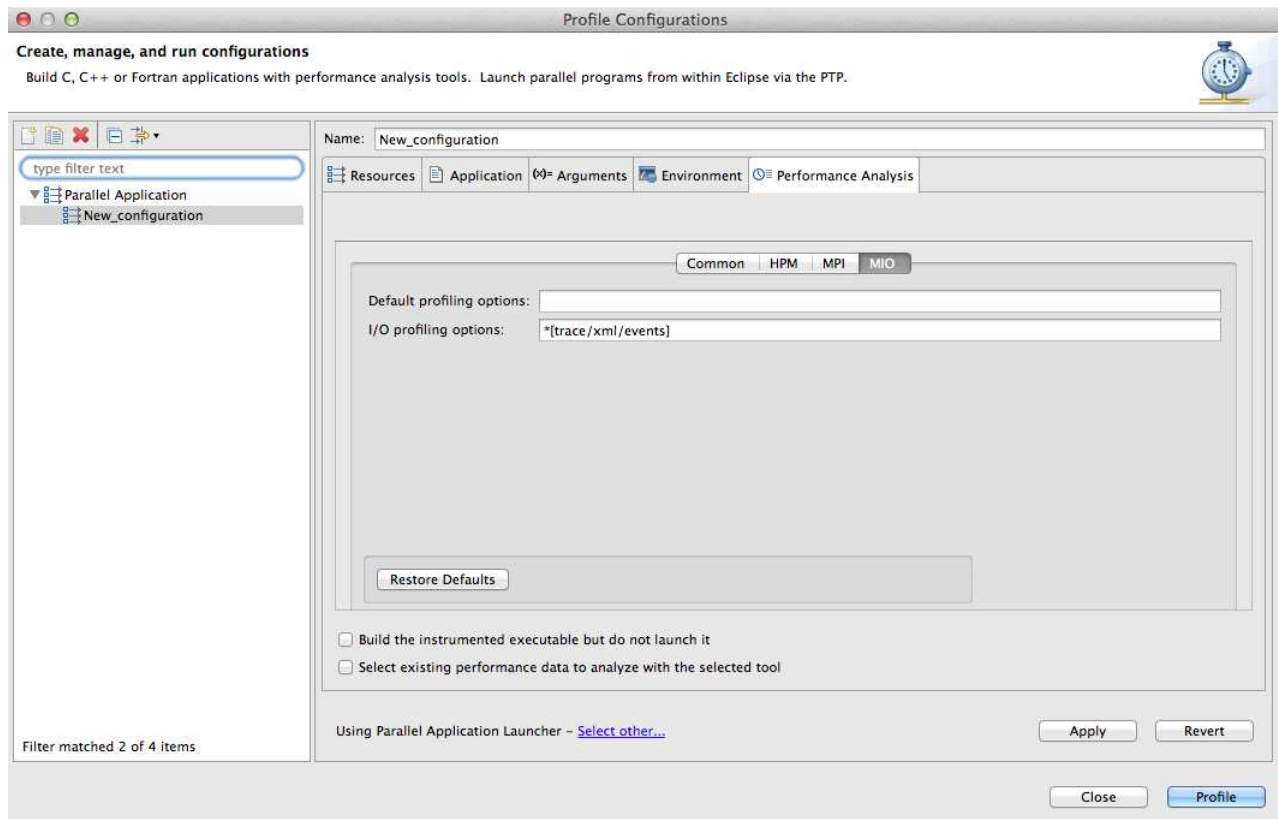


Figure 25. MIO subtab of the Performance Analysis tab

For more information about settings on this panel, use Table 10 on page 64 to reference the corresponding environment variable description.

Table 10. MIO profile configuration environment variables

Profile configuration field name	Environment variable
Default profiling options	MIO_DEFAULTS
I/O profiling options	MIO_FILES

Viewing I/O Data

Once the instrumented application has finished running, a dialog will be displayed indicating that the data files were generated and can be loaded. Select **yes** to load them into hpctView. If the files are not loaded at the completion of the run, they can be loaded manually at a later time. Figure 26 shows the **Performance Data** view with I/O profiling data.

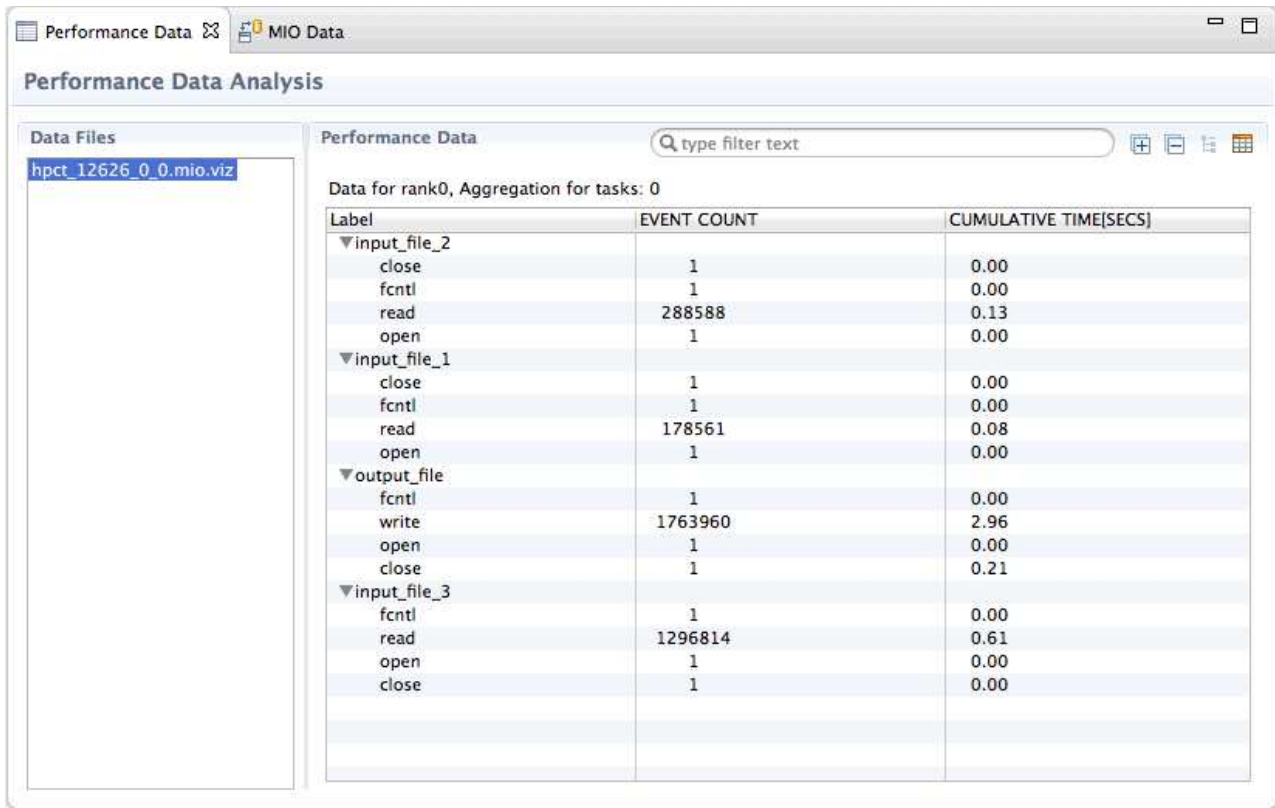


Figure 26. Performance Data view displaying the I/O profiling data.

Initially, the data is displayed in a tree view. Each non-leaf node represents a data file that was accessed by the application. Each leaf node shows, for a particular file, the number of times that function call was executed (**EVENT COUNT**) and the time spent in an I/O function call (**CUMULATIVE TIME**).

Selecting the **Show Data as a Flat Table** button switches to a table view. In addition to the previous information, the table view also displays the following (depending on which I/O operations are being profiled):

- Cumulative bytes requested
- Cumulative bytes delivered

- Minimum request size (bytes)
- Maximum request size (bytes)
- Rate (bytes)
- Suspend wait count
- Suspend wait time
- Forward seeks average
- Backward seeks average

I/O tracing information can be viewed by right-clicking the **Data Files** panel of the **Performance Data** view and selecting the **Load Trace Data** menu. The trace file will be loaded and then displayed in the **MIO Data** view. The MIO Data view displaying trace data is shown in Figure 27.

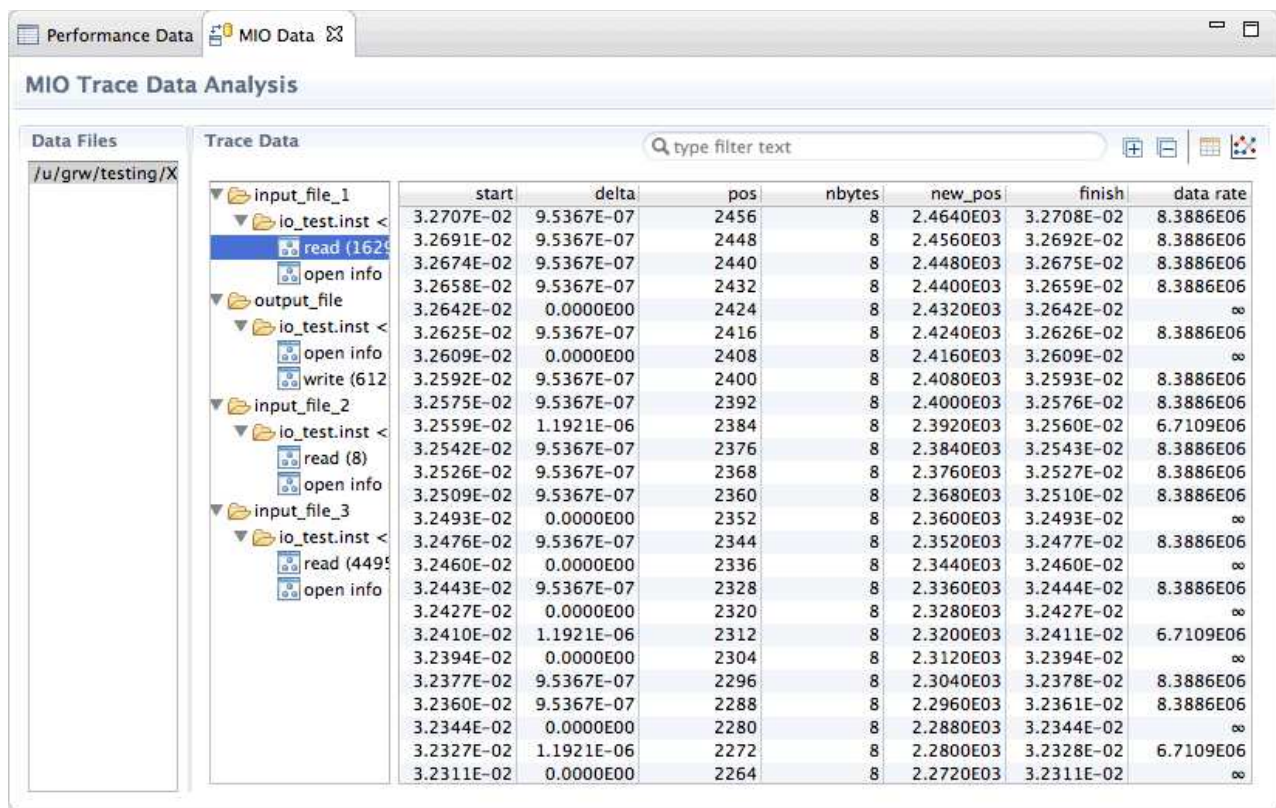


Figure 27. MIO Data view showing I/O trace data

The **MIO Data** view contains two panels:

1. A **Data Files** panel that lists the individual performance data files
2. A **Trace Data** panel that provides tree view of the trace data.

In the **Trace Data** panel, the top-level nodes in the tree 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. Selecting a leaf node will include the data from that nodes in the display. The remainder of the view displays the trace data either as a table (the default) or as a data plot. It is possible to switch between these views using the **Show Data as a Table** and **Show Data as a Graph** buttons. Figure 28 on page 66

shows the same trace data displayed as a graph. By selecting or deselecting nodes in the tree, it is possible to change the data that is included in the graph.

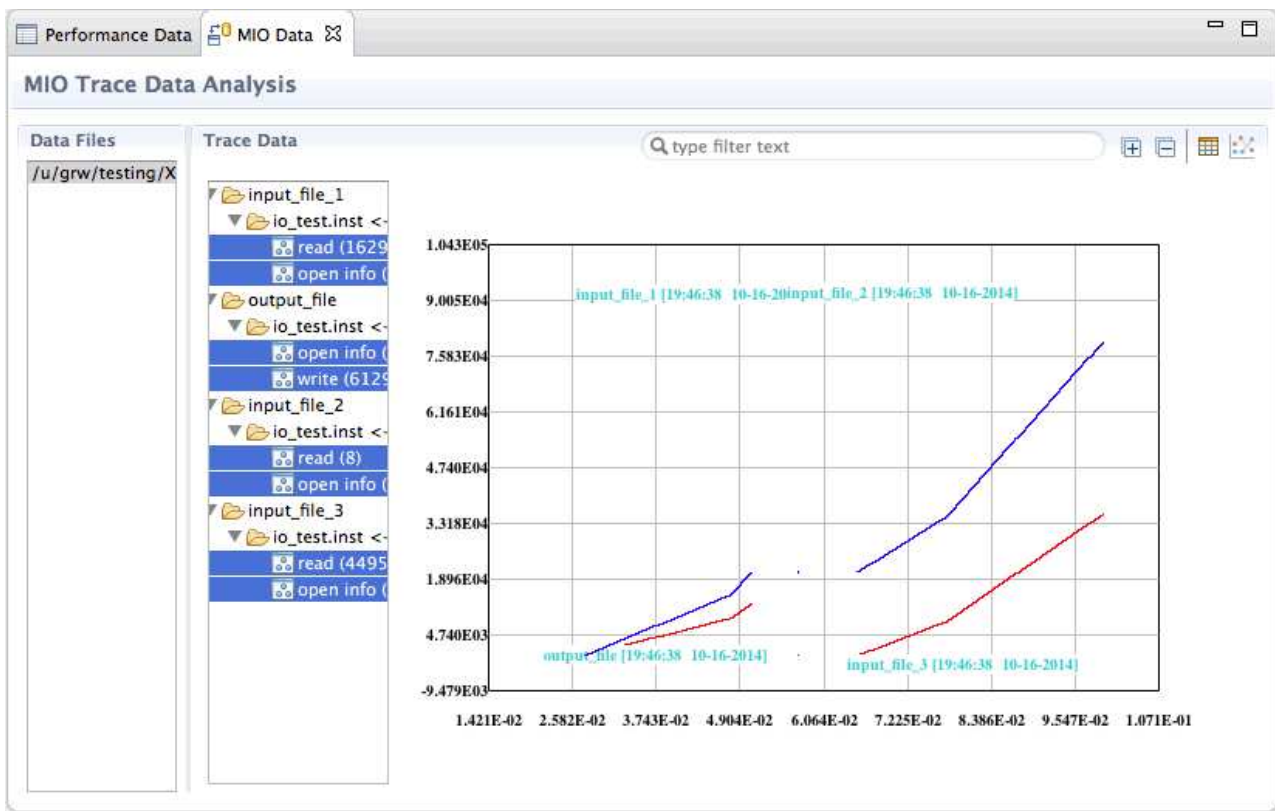


Figure 28. MIO Data view showing a graph of the trace data

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. It is possible to 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. Once the left mouse button is released, the graph will be redrawn showing the zoomed area. As you drag the mouse, 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 is displayed as text in the status bar area at the bottom of the view. The view can be restored to the unzoomed state by double clicking anywhere in the graph. It is possible to 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 view.

Chapter 11. Using Call Graph analysis

In addition to displaying profiling and trace data generated by the binary instrumentation tools, the hpctView application can also be used to display gprof-format call graph and histogram information. This topics that follow describe the steps necessary to collect and display this information.

Preparing the application

There are two methods for generating **gprof** data that can be viewed with hpctView:

1. The first method is to compile your application using the **-pg** compiler flag. This flag instructs the compiler to add special library functions to your code to generate the gprof-format data. Depending on the application type and the runtime system used, this might generate one or more **gmon.out** files. When running using the IBM Parallel Environment, an MPI application will generate one **gmon.out** file for each MPI task, with each **gmon.out** file being placed in a separate directory.

Note: Application executables compiled with **-pg** cannot be instrumented using the binary instrumentation tool discussed in the previous topics.

2. The second approach is to use a tool such as **opgprof** to convert the data generated from an **Oprofile** session into a **gprof** compatible format.

For more information, see the following:

- **opgprof(1)** manual page (man7.org/linux/man-pages/man1/opgprof.1.html)
- **oprofile(1)** manual page (man7.org/linux/man-pages/man1/oprofile.1.html)

Running the application

If the application has been compiled with the **-pg** flag, then all that is necessary is to run the application and the **gmon.out** files will be automatically created. The application can be run using the **Profile Configuration** used to run an instrumented application, or run manually on the target system.

Oprofile is a system-wide statistical profiling tool. To collect data using **Oprofile**, the profiling needs to be enabled while the application is being run. Consult the **oprofile(1)** manual page (man7.org/linux/man-pages/man1/oprofile.1.html) for more information on profiling a single application.

Viewing profile data

Once the gprof data has been generated, viewing the data is a two-step process. Before loading the **gmon.out** files, the executable that generated the data must first be opened. This is done in the same way that an executable is opened for instrumentation, using the **Open Executable...** option from the main **File** menu. Once this is done, a **gmon.out** file is loaded using the **Load Call Graph...** option from the main **File** menu.

At this point two views will be opened, the **Call Graph** view and the **gprof** view.

Viewing the Call Graph

The Call Graph view is shown in Figure 29.

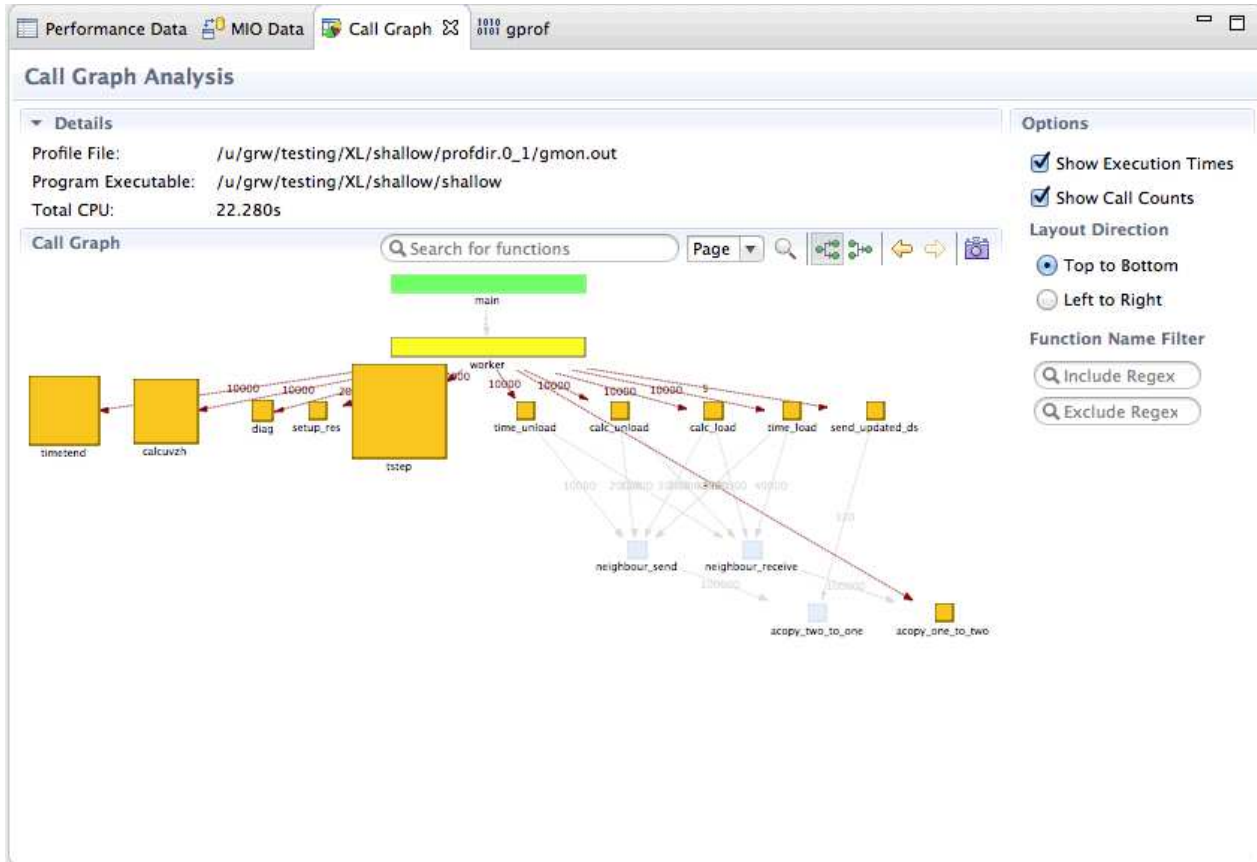


Figure 29. Call Graph view

The **Details** panel is used to display some information about the file being viewed. The **Profile File** field shows the path to the data file that was opened. The **Program Executable** field shows the path to the corresponding executable. The **Total CPU** field displays the total CPU time for the application run. This panel can be closed to provide more space to display the call graph.

The **Call Graph** panel displays a pictorial representation of the application call graph. Boxes are labeled with the name of the function, and arrows are used to indicate callers and callees, and are labeled with the number of calls. The shape of the box provides a 2-D representation of the execution time. The *height* of the box shows the time spent in the function itself. The *width* of the box shows the time spent in the function and all its descendents. This representation provides a very quick method for identifying those functions that may be causing a bottleneck in the application.

Boxes are color coded as follows:

Green The main function of the program.

Yellow

The currently selected function.

Orange

All functions being called by the currently selected function.

Gray

Functions that are not currently of interest.

This panel also provides a toolbar with a number of controls to manipulate the display. These controls are (from left to right):

Search for functions

Used to locate specific functions, particularly for large graphs. Entering text in this box will highlight any function with a name that contains the text.

Scale

Allows the diagram to be easily scaled in order to fit into the display area, or to provide a more detailed view of part of the graph.

Focus

Replaces the current graph with the sub-graph of the selected function. The same affect can be achieved by double-clicking on one of the function boxes.

Show Callees and Show Callers

Used to switch the graph from displaying the call tree for the top most function to showing which functions call the top most function in the current display.

Back and Forward

Used to navigate the hierarchy of functions when using the **Focus** button.

Snapshot

Used to create an image of the current display. This image can then be saved if desired.

The **Options** panel provides a number of options for controlling the graph display:

Show Execution Times

Controls displaying the execution time information using the shape of the boxes. Turning this off will use a single box size for all functions.

Show Call Counts

Controls displaying the call counts on the graph arrows.

Layout Direction

Enables the graph to be laid out either top-to-bottom or left-to-right.

Function Name Filter

These two fields can be used to filter out functions from the graph in order to reduce clutter. Both fields take regular expressions. The upper box will include any function with a name that matches the regular expression in the graph. The lower box will exclude any function with a name that matches the regular expression from the graph.

Viewing gprof data

The **gmon** view is shown in Figure 30 on page 70.

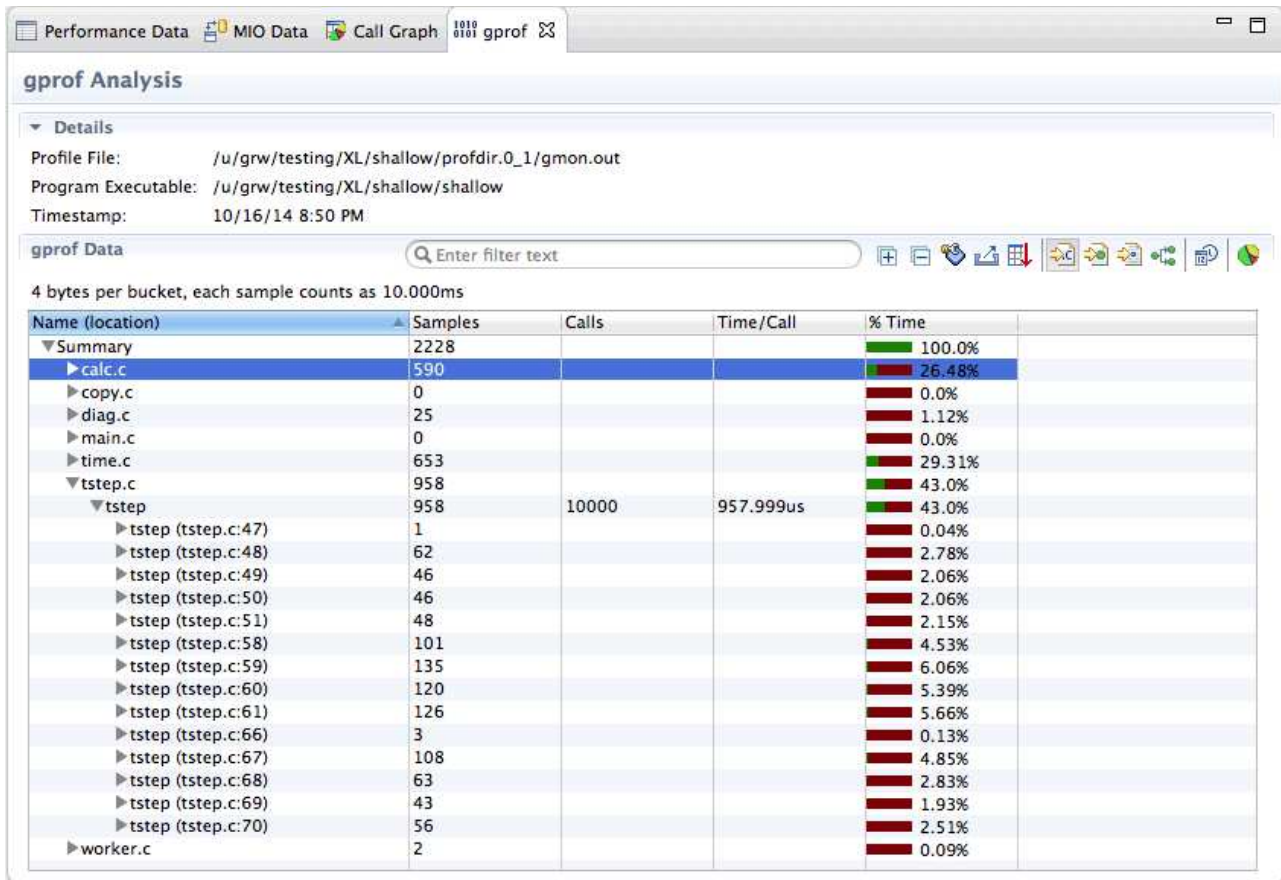


Figure 30. The gmon view

The **Details** panel is used to display some information about the file being viewed. The **Profile File** field shows the path to the data file that was opened. The **Program Executable** field shows the path to the corresponding executable. The **Timestamp** field displays the date and time that the data was collected.

The **gmon Data** panel displays a tabular representation of the sampled data. The first column of the table shows a tree representation of the program structure, with names representing the aggregation unit. At the top level is **Summary**, which displays aggregated data for the whole program. The next level is the file name, followed by the function name, lines within the function, and finally an address (or location) within the program. The other columns show the aggregated data for each node in the tree, and are as follows:

Samples

Shows the number of samples that were recorded for the program, function, line, or address.

Time

Only shown when the Switch Sample/Time button is activated. Shows the time spent in the program, function, line, or address.

Calls

Shows the number of times a function was called. Only displayed for the function level.

Time/Call

Shows the average time per call for the function. Only displayed for the function level.

% Time

Shows the percentage of the overall time that was spent in the program, function, line, or address. This is displayed as a numeric percentage and as a bar graph.

This panel provides a toolbar with a number of controls to manipulate the display. These controls are (from left to right):

Enter filter text

Used to filter nodes in the first column. This can be helpful for locating a known function or file if the list is very large.

Expand All and Collapse All

Provides a convenient way to expand and collapse all elements in the tree.

Show/Hide Columns

Enables columns to be removed (or re-added) to the table.

Export to CSV

Allows the table data to be exported as a CSV file.

Sorting

Opens a dialog that can be used to control sorting of the table columns.

The next four buttons control filtering of the data that is displayed in the table:

Samples Per File

Displays the sampled data for the file, function, line, and address levels.

Samples Per Function

Displays the sampled data for the function, line, and address levels.

Samples Per Line

Displays the sampled data for the line and address levels only.

Function Call Graph

Displays the samples as a call graph. Each function node will contain parent and children nodes that can be used to navigate the call graph.

The remaining two buttons are as follows:

Switch Sample/Time

Changes the **Samples** column to display **Time** and vice versa.

Create Chart...

Can be used to create a chart (bar graph/pie chart) from a selection of the sample data.

Part 4. The hardware performance counter tools

The topics in this part provide information about using the hardware performance counter tools.

Chapter 12. Using the hardware performance counter tools

The IBM HPC Toolkit provides a command-line tool called **hpccount** and a library called **libhpc** that 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. 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. 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 plug-ins 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 one of the setup scripts that are located in the top-level directory of your IBM HPC Toolkit installation. These setup scripts are located in the **/opt/ibmhpc/ppdev.hpct** directory. If you are using **sh**, **bash**, **ksh**, or a 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

The **hpccount** command 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, to **stdout**). In particular, it prints resource usage 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 usage 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 are not provided.

If you specify the **-l** flag, 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 **-g** flag, you can specify the hardware performance counter group from which you want to count events from. If you do not specify the **-g** flag, the **hpccount** command uses a default hardware counter group (for more information, see “**hpccount** - Report hardware performance counter statistics for an application” on page 116). 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 “Understanding CPU hardware counter multiplexing” on page 78 for more information.

You can obtain derived metric descriptions for your application by using the **-x** flag. Derived metrics are additional performance metrics computed from the hardware performance counter measurements you collected. See Appendix B, “Derived metrics, events, and groups supported on POWER8 architecture,” on page 203 for more information.

You can specify a data aggregation plug-in using the **-a** flag or the **HPM_AGGREGATE** environment variable. See “Considerations for MPI programs” on page 83 for more information.

Using the **hpcstat** command

The **hpcstat** is a simple system-wide monitor that is based on hardware performance counters. To run **hpcstat**, root privileges are required. The usage is very similar to that of the **vmstat** command. You can invoke **hpcstat** by issuing:

hpcstat

If you specify the **-l** flag, 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 **-g** flag, you can specify the hardware performance counter group from which you want to count events. If you do not specify the **-g** flag, the **hpcstat** command will use a default hardware counter group as described in “**hpcstat** - Reports a system-wide summary of hardware performance counter statistics” on page 124.

You can invoke **hpcstat** so that it periodically queries the requested hardware performance counter groups by using the **-I** flag to specify the number of times to query hardware performance counter groups and using the **-i** flag to specify the interval between queries.

The output of the **hpcstat** command is written to **stdout** and to a file, which consists of hardware performance counter information and derived hardware metrics.

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 within 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 region of code in which hardware performance counter events will be counted.
- **hpmStop()** for identifying the end of the instrumented region.

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.

The part of the application program between the start and stop of performance counting is called an instrumentation region. You assign a unique integer number as the region identifier. You specify this region identifier in the call to the **hpmStart()** function. A simple case of an instrumented program region 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( 0 );
```

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

The example program above provides an example of two properly nested instrumentation regions. For region 1 we can consider the *exclusive* time and *exclusive* counter values. By that we mean the difference of the values for region 1 and region 2. The original values for region 1 would be called *inclusive* for matter of distinction. The terms *inclusive* and *exclusive* for the embracing instrumentation region are chosen to indicate whether counter values and times for the contained regions are included or excluded. For more details see “Understanding inclusive and exclusive event counts” on page 80.

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 flag, 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 flag. Another option is to use the **.F** Fortran source file extension.

You must link your application with the **libhpc** library, using the **-lhpc** linking option. You must also link your application with the **libm** library, using the **-lm** linking option. When using the **libhpc** library, compile and link your application as a threaded program (for instance, using the **xlcr** or **xlfr** commands), or link with the **pthread** library using the **-lpthread** linking 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** linking option.

A sample compilation and link is:

```
xlcr -o testprog -g -q64 -I$(IHPCT_BASE)/include testprog.c -L$(IHPCT_BASE)/lib64 -lhpc
```

Like the **hpccount** command, you can obtain hardware performance counter measurements using multiple hardware performance groups as described in “Understanding CPU hardware counter multiplexing.” Derived metrics will be computed if the events within the group you selected contain the events necessary to compute the derived metric. If you are instrumenting an MPI program, you can use plug-ins to aggregate performance data from multiple tasks or to filter that data. See “Considerations for MPI programs” on page 83 for more detail about these plug-ins.

Note that **libhpc** collects information and performs summarization during run time. Thus, there could be a considerable overhead if instrumentation regions 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 “hpm_error_count, f_hpm_error - Verify a call to a libhpc function” on page 151.

Understanding CPU hardware counter multiplexing

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

Hardware Performance Monitor (HPM) supports multiplexing by specifying CPU groups as a comma separated list. If you are using the **hpccount** command, then you can specify the multiplexed hardware performance CPU counter groups one of two ways, for instance:

```
hpccount -g 1,2 program
```

or:

```
export HPM_EVENT_SET='1,2'  
hpccount program
```

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

```
export HPM_EVENT_SET='1,2'
```

Multiplexing means that none of the specified CPU 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.

On Linux, the kernel chooses the intervals used when multiple CPU groups are specified.

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

For MPI applications, the form of the output depends on the chosen aggregation plug-in as described in “Considerations for MPI programs” on page 83. Without specifying an aggregator plug-in (in other words, with the default plug-in), the data for each CPU hardware performance counter group is printed in a separate section with separate timing information for each CPU 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 “Plug-ins shipped with the tool kit” on page 84.

Understanding derived metrics

Some of the hardware counter events are difficult to interpret. Sometimes a combination of events provides better information. Such a combination of basic events is called a derived metric. HPM provides a list of derived metrics, which are defined in Appendix B, “Derived metrics, events, and groups supported on POWER8 architecture,” on page 203.

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 **-x** flag 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

For POWER8 servers in Little Endian (LE) mode, scalar FLOPS and scalar utilization rate are computed as derived metrics when the default hardware counter group is selected. There is no hardware counter group that includes all event counters for vector floating point operations, so a vector FLOPS rate cannot be computed automatically. Use groups containing event counters **PM_VSU_1FLOP**, **PM_VSU_2FLOP**, **PM_VSU_4FLOP**, **PM_VSU_8FLOP**,

PM_VSU_16FLOP and PM_VSU_2FLOP_DOUBLE to compute vector FLOPS. Note that when computing vector FLOPS, you must multiply the event count by the number in the counter name, for instance multiply PM_VSU_2FLOP by 2 to get the correct count.

Understanding inheritance

On 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. 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.

Suppose 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. Because **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 “Using the libhpc library” on page 77. That application fragment provides an example of two properly nested instrumentation regions. For region 1, exclusive time and exclusive counter values are the difference between the values for region 1 and region 2, or excluding the counts of events within the scope of region 2. The original values for region 1 would be called inclusive values since those values also include the count of events that occurred within the scope of region 2. The terms inclusive and exclusive for the enclosing instrumentation region are chosen to indicate whether counter values and times for the contained regions 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 “Understanding parent-child relationships” on page 81. The exact definition of *exclusive* is based on parent-child relationships among the instrumented regions. 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

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 be 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 equivalents).

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 (*.txt) 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. 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, you 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_hpmstart( instID, "computing meaning of life" )
!$OMP DO
do ...
do_work()
end do
call f_hpmstop( instID )
!$OMP END PARALLEL
```

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

HPM 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, 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

The follow topics discuss 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 “Understanding inheritance” on page 80). 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 subtopics. Consequently, each instance is writing its own output. For information about file naming conventions, see Appendix A, “Performance data file naming,” on page 197.

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**.

Hardware performance counter plug-ins

The following topics provide information about the hardware performance counter plug-ins.

Note: Hardware performance counter plug-ins that use MPI calls are only supported when linking your application with **libhpc** or instrumenting your application with the **hpctInst** command. When using the **hpccount** command, hardware performance counter plug-ins such as **average.so** and **single.so** that use MPI calls are not supported.

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, 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. You can specify a plug-in using the **-a** flag to the **hpccount** command or by using the **HPM_AGGREGATE** environment variable.

Table 11. Plug-ins shipped with the tool kit

Plug-in name	Description
mirror.so	This plug-in 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	This plug-in 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 non-MPI context.
single.so	This plug-in 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.

Table 11. Plug-ins shipped with the tool kit (continued)

Plug-in name	Description
average.so	This plug-in calculates average values 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. You can control the cluster of groups assigned to each task, by setting the environment variable HPM_ROUND_ROBIN_CLUSTER to the number of groups desired per task. 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 “Understanding the distributor and aggregator interfaces.”

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/plugin** directory.

Understanding the distributor and aggregator interfaces

Each distributor and aggregator is a function returning an integer which is 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 **\${IHPCT_BASE}/include/hpc_agg.h** header file:

```
#include "hpc_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 `$(IHPCT_BASE)/include/hpc_data.h` header file.

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.

The following is from the **hpc_data.h** include file:

```
/*      NAME                      INDEX  DESCRIPTION                      */
#define HPM_NTIM                  8
#define HPM_TIME_WALLCLOCK        0 /* Wall clock time                */
#define HPM_TIME_CYCLE            1 /* User time generated from ticks  */
#define HPM_TIME_USER             2 /* User time as reported by rusage() */
#define HPM_TIME_SYSTEM           3 /* System time as reported by rusage() */
#define HPM_TIME_START            4 /* Start time stamp (volatile)     */
#define HPM_TIME_STOP             5 /* Stop time stamp (volatile)      */
#define HPM_TIME_OVERHEAD         6 /* Overhead time for counter start/stop */
#define HPM_TIME_INIT            7 /* Overhead time for initialization */

/*
 * These defines will be set depending on the group type
 * Each of the group types may have a different number of members
 */

#define HPM_GROUP_TYPE_UNSET      (0)
#define HPM_GROUP_TYPE_CPU       (1)
#define HPM_GROUP_TYPE_RUSAGE    (4)

typedef struct {
    int          num_data;          /* Number event info entries      */
    hpm_event_info *data;          /* Event info list                */
    double       times[HPM_NTIM]; /* Timing information              */
    int          group_type;        /* cpu, rusage                    */
    int          mpi_task_id;       /* MPI task id if available       */
    int          instr_id;          /* id of the instrumented code    */
    /* section if applicable (-1 otherwise) */
    int          count;            /* number of entries/exits to the */
    /* instrumented section (if applicable) */
    int          is_exclusive;      /* these are exclusive values    */
    int          xml_element_id;    /* arbitrary integer to identify */
    /* hpm_data_items that translate into */
    /* identical XML element shapes      */
    char         *description;      /* Description for this data item */
    char         *xml_descr;        /* Descriptive header for XML output */
} hpm_data_item;

typedef hpm_data_item *hpm_event_vector;
```

- The first element of the vector contains the data from a call to **getrusage()**. This vector element is the only element with its structure member **group_type** set to **HPM_GROUP_TYPE_RUSAGE** to distinguish it from 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 “Understanding inclusive and exclusive event counts” on page 80. 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/plugin/Makefile` using the command:

```
gmake
```

Note the following considerations when implementing your own plug-in:

- The Makefile distinguishes sequential and parallel plug-ins. 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.

Chapter 13. Using GPU hardware counters in HPCT

The IBM HPC Toolkit provides the capability for you to profile and trace GPU hardware counters through the GPU Performance Monitoring (GPM) component. The GPM component uses the CUPTI interface and the CUDA runtime and driver. CUPTI is an extraneous component of CUDA and must be installed on the nodes where the profiling is performed. Once installed, it is located in the `/usr/local/cuda/extras/CUPTI` directory.

By using the CUPTI interface, the GPM's profiling and tracing is discrete to GPU kernel executions. The events and metrics measured by GPM cover the entire execution of a GPU kernel, which is viewed as a black box by GPM. In addition, depending on what events and metrics are profiled or traced, the GPU kernels might be executed multiple times in order to measure the counters for all the events specified, or for the metrics specified.

The profiling and tracing of GPU hardware counters can be performed in three ways by using:

- The preload GPM library, `/opt/ibmhpc/ppdev.hpct/lib64/prelod/libgpm.so`
- The GPM API exported by the GPM library, `/opt/ibmhpc/ppdev.hpct/lib64/libgpm.so`
- GPM under the control of HPM

Regardless of which way the GPM component is used, the GPU events and metrics are specified by using the environment variables, `GPM_EVENT_SET` and `GPM_METRIC_SET`, respectively. The complete set of events and metrics supported by a specific device or version of CUPTI is generated by the `gpmlist` command (see “`gpmlist` - Lists the available events and metrics” on page 114).

You can specify the events and metrics as a comma-separated (no spaces) list using either their names or the following format:

`device:<device number>:<event/metric name>`

where *device number* is the device where the events and metrics are profiled.

If only the event or metric name is specified, the profiling is performed for all devices used by the application. The profiling or tracing of both events and metrics at the same time is not supported. You must use either `GPM_EVENT_SET` or `GPM_METRIC_SET`, but not both. GPM will display an error if both environment variables are set at the same time.

The output files produced by the GPM component follows the general HPC format:

`<tool name>-<MPI world #>.<MPI task #>.<component name>.<application name>.<extension>`

where:

tool name

Is **hpct** by default and it can be changed using the `HPC_OUTPUT_NAME` environment variable.

MPI world # and MPI_task #

Are the MPI world and tasks numbers.

component name

Is **gpm**.

application name

Is the name or the user's application that is being profiled or traced.

extension

Can be either **txt** for ASCII files, **viz** for XML formatted files, or **gpt** for trace directories.

In GPM, the error display is limited because the use of shared libraries makes it impractical to display errors to **stderr**. You can view a log of the GPM activity using the HPC Toolkit's log facility, by setting the **HPCTLOG** and **HPC_TEMPDIR** environment variables to the level of logging desired (level 1 logs errors) and the location of the generated log file, respectively. Each task in an MPI job produces a log file, named as follows:

<user name>.<process id>.log

where:

user name

Is the name of the user.

process id

Is the process id of the task or process producing the log file.

When profiling or tracing GPU hardware counters, you can use either event *or* metrics, but you cannot profile or trace both events and metrics at the same time.

The following topics describe the three modes of operation for GPM.

Using the GPM preload library

To use the GPM preload library, use the Linux **LD_PRELOAD** mechanism and set the **LD_PRELOAD** environment variable as follows:

```
LD_PRELOAD=/opt/ibmhpc/ppdev.hpct/lib64/preload/libgpm.so:$LD_PRELOAD
```

The **LD_PRELOAD** environment variable can be set either in the **poe** command line (before **poe**) or in a wrapper shell script that launches the user application. The advantage of using it in a wrapper shell is that it avoids an unnecessary initialization of the CUPTI and CUDA environment (**poe** does not use CUDA.)

The LULESH benchmarks of a wrapper shell script can be used to set the **LD_PRELOAD** environment variable and invoke the parallel application as shown in this example:

```
#!/bin/sh
LD_PRELOAD=/opt/ibmhpc/ppdev.hpct/lib64/preload/libgpm.so:$LD_PRELOAD
lulesh -s 100 -i 100
```

Then the wrapper shell script can be run with **poe** as follows:

```
$ MP_HOSTFILE=host.file MP_RESID=poe MP_PROCS=8 GPM_METRIC_SET=ipc poe lulesh.sh
```

When the GPM preload library is loaded by the dynamic linker, it:

1. Initializes the CUDA runtime environment
2. Reads the **HPM_EVENT_SET** or **HPM_METRIC_SET** environment variables
3. Sets up the appropriate CUPTI event groups or group sets
4. Starts the profiling or tracing of the GPU events and metrics

When the application finishes its execution, the GPM preload library stops the GPU event or metric profiling and tracing and terminates the GPM runtime environment. It is during this time that the output files or information is produced.

By default, the GPM preload library produces output to **stdout** and to an ASCII file:

- The output produced to **stdout** consists of measurements produced for each GPU kernel execution.
- The output produced in the ASCII file consists of cumulative measurements for all the GPU kernel executions.

Setting **GPM_STDOUT=*n*** suppresses the output to **stdout**. Setting the **GPM_VIZ_OUTPUT=*y*** produces an XML formatted output file equivalent to the ASCII text file that can be viewed with the **hpctView** command or with the HPCT plugin for Eclipse PTP.

If you want to generate trace files that can be visualized in the hpctView application, set **GPM_ENABLE_TRACE=*y***. The trace files produced by the GPM preload library are recorded in the OTF2 format. You can use the **otf2-print** command in the **/opt/ibmhpc/ppdev.hpct/bin** directory to see the content of the trace files. For example, if the **trace** subdirectory name in the local directory is **hpct_0.0.gpm.lulesh.gpt**, then the trace file can be output to **stdout** as follows:

```
otf2-print hpct_0.0.gpm.lulesh.gpt/lulesh.otf2
```

Considering that all tasks produce output, the GPM preload library is suitable for profiling and tracing small jobs. When using the GPM preload library, the entire execution of the application is profiled and traced for GPU.

Using the GPM API

To gain more control of what portion of the program is profiled or traced, you can use the **GPM API**, which does the following:

- Exports a minimal set of routines to initialize the GPM runtime
- Starts and stops the profiling for the entire process or for the current thread
- Terminates the GPM runtime and output results.

The following example shows how to use the **GPM API** with an small CUDA program:

```
#include <gpm.h>
...
int rc = 0;
// initialize the GPM runtime
rc = gpm_init();
// handle error code
rc = gpm_start();
// handle error code

... // program execution

rc = gpm_stop();
// handle error code
rc = gpm_terminate();
//handle error code
```

To build your application with the **GPM API**, you need to add the **GPM** header file path to the application's compile options and the **GPM** library path and name to your application's link options, as follows:

| -I/opt/ibmhpc/ppdev.hpct/include

| and:

| -L/opt/ibmhpc/ppdev.hpct/lib64 -lgpm

| Use the **GPM_EVENT_SET** or **GPM_METRIC_SET** environment variables to
| provide the set of events or metrics that you want to profile and trace.

| By default, the **GPM** API does not produce output to **stdout**. If you want to have
| output produced to **stdout**, use the **GPM_STDOUT** environment variable set to **y**
| (**GPM_STDOUT=y**). The rest of the output controls are identical to those used for
| the GPM preload library.

| Considering that all tasks in the job produce output, the **GPM** API is suitable to
| profile and trace small jobs.

| **Using GPM under the control of HPM**

| You can use HPCTs HPM component to drive the GPM component. In this mode,
| the HPM component does the following:

- Initializes the GPM runtime during its own initialization
- Starts and stops GPM profiling during its own start and stop routines
- Terminates the GPM runtime and produces GPM profile and trace output during its own termination

| For more information about how to use the HPM component for profiling and
| tracing, see Chapter 12, “Using the hardware performance counter tools,” on page
| 75.

| In order to enable the GPM component when using HPM, you need to set the
| **HPM_ENABLE_GPM** environment variable to **y** (**HPM_ENABLE_GPM=y**). In this
| case, HPM calls the **GPM** API to drive the GPU event and metric profiling and
| tracing. Consequently, the event or metric set that needs to be profiled and the
| output generated by the GPM component are controlled in the same manner as
| when using the GPM component.

| The IBM HPCT HPM component can be used to drive the GPM component either
| through its API or using binary instrumentation. For more information about how
| to use HPCT binary instrumentation, see Chapter 16, “Instrumenting your
| application using hpctInst,” on page 107.

| Considering that HPM allows the use of plug-ins to filter results and significantly
| reduce the number of tasks that produce output, you can use this mode to profile
| and trace GPU events or metrics with larger applications. For more information
| about how to use the HPM plug-ins, see Table 11 on page 84.

Part 5. The MPI and I/O profiling libraries

The topics in this part provide information about using the MPI and I/O profiling libraries.

Chapter 14. 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 one of the setup scripts that are located in the top-level directory of your IBM HPC Toolkit installation. These setup scripts are located in the **/opt/ibmhpc/ppdev.hpct** directory. If you are using **sh**, **bash**, **ksh**, or a 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**.

Compiling and linking with libmpitrace

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}/lib64** for 64-bit applications. Although it is not necessary for all linkers, it is recommended that the link command parameters be ordered such that the mpi program object files come first, before specifying the **-lmpitrace** linker flag.

In this release, only MPICH2 is supported and the **\${IHPCT_BASE}/lib64/libmpitrace.so** library is linked to the MPICH2-based library.

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

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 limited to 30,000 events or less. 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.

If you do not need to override the **MAX_TRACE_RANK** environment variable setting, you can set the **MT_BASIC_TRACE** environment variable to **yes**. If you do this, the trace library will operate with lower overhead for each MPI function call.

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 **MPI_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 “Understanding MPI profiling utility functions” on page 98 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 Table 12:

Table 12. MPI profiling utility functions

Function	Purpose
MT_get_mpi_counts	Determines how many times an MPI function is called.
MT_get_mpi_bytes	Determines the total number of bytes that are transferred by all calls to a specific MPI function.
MT_get_mpi_time	Determines the cumulative amount of time that is spent in all calls to a specific MPI function.
MT_get_mpi_name	Obtains the name of an MPI function, given the internal ID that is used by the IBM HPC Toolkit to refer to this MPI function.
MT_get_time	Determines the elapsed time since MPI_Init() was called.
MT_get_elapsed_time	Determines the elapsed time between calls to MPI_Init and MPI_Finalize .
MT_get_tracebufferinfo	Determines the size and current usage of the internal MPI trace buffer that is used by the IBM HPC Toolkit.
MT_get_calleraddress	Determines the address of the caller of a currently-active MPI function.
MT_get_callerinfo	Determines the source file and line number information for an MPI function call, using the address that is obtained by calling MT_get_calleraddress .
MT_get_environment	Obtains information about the MPI execution environment.
MT_get_allresults	Obtains statistical information about a specific MPI function call.

All of these functions are documented in Chapter 18, “Application programming interfaces,” on page 131.

Performance data file naming

By default, the **libmpitrace** library generates three sets of output files in the current working directory. For more information, see Appendix A, “Performance data file naming,” on page 197.

Chapter 15. Using the I/O profiling library

The topics that follow will provide information about 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 one of the setup scripts that are located in the top-level directory of your installation. These setup scripts are located in the `/opt/ibmhpc/ppdev.hpct` directory. If you are using `sh`, `bash`, `ksh`, or a 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, if you are not instrumenting your application using `hpctInst`, the `hpctView` application, or the HPC Toolkit Eclipse plug-in, you must link your application with the `libhpctkio` library using the `-L$IHPCT_BASE/lib64` and `-lhpctkio` linking options.

You must also set the `TKIO_ALTLIB` environment variable to the path name of an interface module used by the I/O profiling library before you invoke your application. The `TKIO_ALTLIB` environment variable should be set to `$IHPCT_BASE/lib64/get_hpcmio_ptr.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: There are no spaces between the interface library path name and the options. For instance, to load the interface library, 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_ptr.so/print/abort"
```

Setting I/O profiling environment variables

There are two 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 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 a pipe (|) character. For **MIO_DEFAULTS**, module specifications are delimited by commas (,).

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 **stats** option output for both **trace** and **pf** is written to a file with a subtype of **miostats**. For more information, see Appendix A, "Performance data file naming," on page 197. The setting of the **MIO_DEFAULTS** environment variable would be:

```
export MIO_DEFAULTS="trace/stats=miostats/mbytes,pf/stats=miostats/mbytes"
```

As an example of using the **MIO_FILES** environment variable, assume you have a program that does I/O to **/tmp/testdata** and some ***.txt** and ***.dat** files in your current working directory. The following setting will cause **/tmp/testdata** to use the **trace** module with the **events** option and files matching the patterns ***.txt** or ***.dat** to use the **trace** module with the **stats** and **events** options. The setting of the **MIO_FILES** environment variable would be:

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

After running the program the following performance data files will be created:

```
hpct.mio.tmp_testdata.iot
```

Contains the trace events for the **/tmp/testdata** file.

hpct.mio.txt

Contains the **trace** stats for the `/tmp/testdata` file.

hpct.mio.any_txt_or_dat.iot

Contains the trace events for the `*.txt` and `*.dat` files.

hpct.mio.any_txt_or_dat.txt

Contains the **trace** stats for the `*.txt` and `*.dat` files.

Specifying I/O profiling library module options

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

Table 13. MIO analysis modules

Module	Purpose
<code>mio</code>	The interface to the user program.
<code>pf</code>	A data prefetching module.
<code>trace</code>	A statistics gathering module.
<code>recov</code>	Analyzes failed I/O accesses and retries in case of failure.

The **mio** module has the following options:

Table 14. MIO module options

Option	Purpose
<code>mode=</code>	Override the file access mode in the open system call.
<code>nomode</code>	Do not override the file access mode.
<code>direct</code>	Set the O_DIRECT bit in the open system call.
<code>nodirect</code>	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:

Table 15. MIO pf module options

Option	Purpose
<code>norelease</code>	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 <code>norelease</code> option keeps the global cache opened and available, even if the file usage count goes to zero.
<code>release</code>	Free the global cache pages when the global cache file usage count goes to zero.
<code>private</code>	Use a private cache. Only the file that opens the cache might use it.
<code>global=</code>	Use global cache, where the number of global caches is specified as a value between 0 and 255. The default is <u>1</u> , which means that one global cache is used.
<code>asynchronous</code>	Use asynchronous calls to the child module.

Table 15. MIO pf module options (continued)

Option	Purpose
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 12 bytes.
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. The name, if specified will be the subtype portion of the file name. For more information, see Appendix A, "Performance data file naming," on page 197.
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/bytes/cache_size=64k/page_size=4k/
prefetch=1/asynchronous/global/release/stride=1/nolistio/notag
```

The trace module has the following options:

Table 16. MIO trace module options

Option	Purpose
stats=	Output trace statistics to the specified file name. The name, if specified will be the subtype portion of the file name. For more information, see Appendix A, "Performance data file naming," on page 197.
nostats	Do not output statistics on close.

Table 16. MIO trace module options (continued)

Option	Purpose
events=	Generate a binary events file. The name, if specified will be the subtype portion of the file name. For more information, see Appendix A, "Performance data file naming," on page 197.
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 the statistics file in a format that can be viewed using the Eclipse plug-in or the hpctView application. The name, if specified will be the subtype portion of the file name. For more information, see Appendix A, "Performance data file naming," on page 197.

The default options for the **trace** module are:

/stats/noevents/nointer/bytes

The **recov** module has the following options:

Table 17. MIO recov module options

Option	Purpose
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. The name, if specified will be the subtype portion of the file name. For more information, see Appendix A, "Performance data file naming," on page 197.
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 the hpctView application.

After you have compiled and linked your application as described in “Preparing your application” on page 99 and set the **MIO_FILES** and **MIO_DEFAULTS** 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 the hpctView application. You can add the HPCT plug-in to the Eclipse IDE to enable options for viewing the MIO trace files.

Performance data file naming

The names of the output files generated by the I/O profiling tool are dependent on the settings of options in the **MIO_FILES** or **MIO_DEFAULTS** environment variables and whether the application is a serial program, a MPI program, which does not use dynamic tasking, or a MPI program which uses dynamic tasking.

For information about how files are named, see Appendix A, “Performance data file naming,” on page 197.

Part 6. Using the **hpctlInst** command

The topics in this part provide information about how to instrument your application using the **hpctlInst** command.

Chapter 16. 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 flags 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 one of the setup scripts that are located in the top-level directory of your installation. These setup scripts are located in the `/opt/ibmhpc/ppdev.hpct` directory. If you are using `sh`, `bash`, `ksh`, or a 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 flag 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. You must link your application using the `-Wl,--hash-style=sysv -emit-stub-syms` flags.

Note: You must not compile your application with the `-pg` flag.

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 topics 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 only described in "hpctInst - Instrument applications to obtain performance data" on page 128.

Note: 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” on page 81 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 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 “Setting I/O profiling environment variables” on page 99.

If you want to instrument only specific I/O function calls in your application, use the Eclipse plug-in or the hpctView application to instrument your application.

Part 7. Command and API reference

The topics in this part provide information about the commands and APIs associated with the following:

- Application launch
- Hardware performance monitoring
- MPI profiling and trace (MPI trace APIs)
- Application instrumentation
- Performance data visualization

See Table 18 on page 113 and Table 19 on page 131 for more information.

Chapter 17. Commands

Table 18 lists the commands and what they are used for:

Table 18. Commands

Used for:	Command
Application launch	"hpcrun - Launch a program to collect profiling or trace data" on page 122
Hardware performance monitoring	"gpmList - Lists the available events and metrics" on page 114 "hpcCount - Report hardware performance counter statistics for an application" on page 116 "hpcstat - Reports a system-wide summary of hardware performance counter statistics" on page 124
Application instrumentation	"hpctInst - Instrument applications to obtain performance data" on page 128

gpmlist - Lists the available events and metrics

Use the **gpmlist** command to list the events and metrics available for specified or current NVIDIA GPU devices.

Syntax

```
gpmlist <-e> [-d <device model>] [-l]
gpmlist <-m> [-d <device model>] [-l]
gpmlist [-D] [-l]
```

Flags

- D** Lists the GPU devices that are supported by this command for static display of available events and metrics.
- d <device model>**
Lists the events or metrics for the specified GPU device (if this option is not provided, the command displays events and metrics for the GPU devices found on the host, if any).
- e** Lists all events available on the GPU devices.
- m** Lists all metrics available on the GPU devices.
- l** Valid only with the **-D**, **-e**, or **-m** flags, and it lists details about the GPU device, events, or metrics, respectively.
- h** Prints the help message.

Description

The **gpmlist** command lists the events and metrics available for a specific GPU device or the current GPU device. The command is provided by the **ppedev.runtime** package:

/opt/ibmhpc/ppedev.hpct/bin/gpmlist

The **gpmlist** command displays the list of events and metrics supported by a specified NVIDIA GPU device or the current device(s) on the host.

When invoked with the **-D** flag, the **gpmlist** command lists all the devices whose events and metrics can be displayed statically. The **-l** flag adds more detail about the GPU devices, such as their long names.

When invoked with the **-e** flag, the **gpmlist** command lists the events available on the device specified with the **-d** flag or the devices installed on the host. The **-l** flag adds more details, such as the names of the events.

When invoked with the **-m** flag, the **gpmlist** command lists the metrics available on the device specified with the **-d** flag or the devices installed on the host. The **-l** flag adds more details, such as the name of the metrics and the list of the events required for each metric.

Examples

1. To list the GPU devices available for a static listing of events and metrics, enter:
gpmlist -D -l

You should receive output similar to the following:

K40m - NVIDIA K40m (Tesla)

2. To list the events available for the K40m device, enter:

```
gpmlist -d K40m -e
```

You should receive output similar to the following:

```
Domain 0
  2082 tex0_cache_sector_queries
  2083 tex1_cache_sector_queries
  2084 tex2_cache_sector_queries
```

3. To list the metrics available for the current device(s), together with additional detail, enter:

```
gpmlist -m -l
```

You should receive output similar to the following:

```
Device 0 (Tesla K40m)
 1201 device:0:l1_cache_global_hit_rate - L1 Global Hit Rate
      2645 device:0:l1_global_load_hit - l1 global load hit
      2646 device:0:l1_global_load_miss - l1 global load miss
 1202 device:0:l1_cache_local_hit_rate - L1 Local Hit Rate
      2641 device:0:l1_local_load_hit - l1 local load hit
      2643 device:0:l1_local_store_hit - l1 local store hit
      2642 device:0:l1_local_load_miss - l1 local load miss
      2644 device:0:l1_local_store_miss - l1 local store miss
 1203 device:0:sm_efficiency - Multiprocessor Activity
      2629 device:0:active_cycles - active cycles
      2193 device:0:elapsed_cycles_sm - elapsed_cycles_sm
...
```

hpccount - Report hardware performance counter statistics for an application

Use the **hpccount** command to report summary hardware performance counter and resource usage statistics for an application.

Syntax

```
hpccount [-o name] [-u] [-n] [-x] [-g group[ ,group,...]] [-a plugin] program  
hpccount [-h]  
hpccount [-l]  
hpccount [-L]  
hpccount [-c]
```

Flags

- a 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 “Hardware performance counter plug-ins” on page 83 for more information.
- 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. If this flag or the **HPM_EVENT_SET** environment variable is not set, a processor specific default group is used. The default group for POWER8 is group number 226, defined as follows:

PM_FPU0_FCONV
Convert instruction executed.

PM_FPU0_FEST
Estimate instruction executed.

PM_FPU0_FRSP
Round to single precision instruction executed.

PM_LSU_LDF
FPU loads only on LS2/LS3 ie LU0/LU1.

PM_RUN_INST_CMPL
Run_Instructions.

PM_RUN_CYC
Run_cycles.

Note: Only CPU hardware counter groups can be specified when running this command.
- h Displays a usage message.
- l Lists the available hardware counter groups and the hardware counter events that are counted by each group.
- L Lists the available hardware counter groups and the counters contained in

each group. The individual counter information will include a short name, the hardware event ID and a short text description. This flag is only available when running Linux.

-n Suppresses **hpccount** output to stdout.

-o name

Writes output to a file with the prefix of *name*.

- The file prefix *name* can be specified using option **-o** or using the environment variable **HPC_OUTPUT_NAME**. The option takes precedence if there are conflicting specifications. If not specified, the file prefix is **hpct**.
 - The name *name* is expanded into different file names:
 - *name.hpmm.app.txt* is the file name for ASCII output, which is a one-to-one copy of the screen output.
 - *name.hpmm.app.viz* is the file name for the XML output. This file can be viewed using the hpctView application or the HPCT plugin.
- For more information about file naming conventions, see Appendix A, “Performance data file naming,” on page 197.
- Which of these output files are generated is governed by the additional environment variables, **HPM_ASC_OUTPUT** and **HPM_VIZ_OUTPUT**. If neither of these are set, only the ASCII output 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.

-u

Specifies that unique file names are used for generated ASCII and XML output files. The unique file name flag can be specified using option **-u** or using the environment variable **HPC_UNIQUE_FILE_NAME**. The option takes precedence if there are conflicting specifications. Unique file names are generated according to the following rules:

- For serial programs, a string *_processId* is appended to either the file prefix if no directory path is specified, or to the directory path if one is given.
- For MPI programs that do not use dynamic tasking, a string *_0_taskRank* is appended to either the file prefix if no directory path is specified, or to the directory path if one is given.
- For MPI programs that use dynamic tasking, a string *_worldId_worldRank* is appended to either the file prefix if no directory path is specified, or to the directory path if one is given.

For more information about file naming conventions, see Appendix A, “Performance data file naming,” on page 197.

-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.

If you are using the **hpccount** command to measure the performance of a parallel program, you should invoke **hpccount** as **poexec hpccount program**. If you invoke **hpccount** as **poexec hpccount program**, you will measure the performance of the **poexec** command, and not the performance of your application.

For information about file naming conventions, see Appendix A, “Performance data file naming,” on page 197.

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. If the **-g** flag is not used and **HPM_EVENT_SET** is not set, a processor-specific default group is used. The default group for POWER8 is group number 226, defined as follows:

PM_FPU0_FCONV

Convert instruction executed.

PM_FPU0_FEST

Estimate instruction executed.

PM_FPU0_FRSP

Round to single precision instruction executed.

PM_LSU_LDF

FPU loads only on LS2/LS3 ie LU0/LU1.

PM_RUN_INST_CMPL

Run_Instructions.

PM_RUN_CYC

Run_cycles.

Note: Only CPU hardware counter groups can be specified when running this command.

HPM_COUNTING_MODE

Specifies the CPU mode where counting will occur. Set this to a comma separated list of any combination of the following three possible values:

- **user**
Set to **user** to have user side events counted.
- **kernel**
Set to **kernel** to have kernel or system events counted.
- **hypervisor**
Set to **hypervisor** to have hypervisor events counted.

The default setting for **hpccount** is **user**.

Output control environment variables

HPM_ASC_OUTPUT

Determines whether or not to generate an ASCII output file. The output file name is:

name.hpm.app.txt

where:

app

Is the name of the executable.

Valid values are **yes** or **no**. The default value for **HPM_ASC_OUTPUT** is **true**, except if **HPM_VIZ_OUTPUT** has been set, in which case the default value for **HPM_ASC_OUTPUT** is **false**.

HPM_OUTPUT_NAME

Specifies the *name* prefix of the output files:

name.hpm.app.txt

and

name.hpm.app.viz

The *name.hpm.app.viz* file can be viewed using the hpctView application or the HPCT plugin. For more information about file naming conventions, see Appendix A, "Performance data file naming," on page 197.

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**. For more information about file naming conventions, see Appendix A, "Performance data file naming," on page 197.

HPM_VIZ_OUTPUT

Set to **yes** to generate an XML output file with the name:

name.hpm.app.viz

The default value for **HPM_VIZ_OUTPUT** is **false**. This file can be viewed using the hpctView application or the HPCT plugin.

Plug-in specific environment variables

HPM_ROUND_ROBIN_CLUSTER

HPM_ROUND_ROBIN_CLUSTER allows setting the number of groups distributed per task.

Without the environment variable **HPM_ROUND_ROBIN_CLUSTER** set, the **average.so** plug-in will distribute the group numbers from **HPM_EVENT_SET** in a round-robin fashion, one group to each of the MPI tasks in the application.

The default value for **HPM_ROUND_ROBIN_CLUSTER** is **1**. The default will be used if a value less than 1 is specified.

The number of groups spread out round robin-fashion to the tasks will be limited to the first "number of tasks times the setting of **HPM_ROUND_ROBIN_CLUSTER**" groups.

If a value greater than the number of groups in **HPM_EVENT_SET** is specified, **HPM_ROUND_ROBIN_CLUSTER** will be set to the number of groups specified in **HPM_EVENT_SET**.

It is possible that the number of groups does not distribute evenly to the tasks. The first task will get at most **HPM_ROUND_ROBIN_CLUSTER** of the groups in **HPM_EVENT_SET**. If there are more tasks and groups left, the second task will get at most **HPM_ROUND_ROBIN_CLUSTER** of the groups left in **HPM_EVENT_SET** and so on, until there are no groups unused in **HPM_EVENT_SET**. After the groups in **HPM_EVENT_SET** have been used once, and there are more tasks, the process will repeat until there are no more tasks.

The environment variable **HPM_ROUND_ROBIN_CLUSTER** 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

HPMAggregate

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 "Hardware performance counter plug-ins" on page 83 for more information.

HPCT_BASE

Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppdev.hpct**).

Files

name.hpmm.app.txt

The ASCII output from the **hpccount** invocation. This is a copy of the report displayed at the completion of **hpccount** execution.

name.hpmm.app.viz

An XML output file containing hardware performance counter data from **hpccount** execution. This file can be viewed using the hpctView application or the HPCT plugin.

Examples

1. To list available counter groups for your processor, enter:
hpccount -l
2. To report floating point unit (FPU) instructions on pipes 0 and 1, enter:
hpccount -o stats -g 127 testprog

3. To report double precision vector instructions issued on pipes 0 and 1, enter:
poe hpccount -o stats -g 129 -u testprog

hpcrun - Launch a program to collect profiling or trace data

Use the **hpcrun** command to launch a program and collect profiling or trace data from a subset of application tasks based on timing criteria.

Syntax

```
hpcrun [--exmetric ELAPSED_TIME | CPU_TIME] [--excount num_tasks]  
[--tracestore memory | mpi_fname, mio_fname] instrumented binary [args ...]  
hpcrun --help
```

Flags

--exmetric [ELAPSED_TIME | CPU_TIME]

Is the metric to be used to determine which task's data to collect. Specify **ELAPSED_TIME** to filter by elapsed (wall clock) time or **CPU_TIME** to filter by CPU time used. The default is **ELAPSED_TIME**.

--excount *num_tasks*

Is the number of tasks to be collected for minimum and maximum value of the metric specified by **--exmetric**. The default is **10** tasks.

--tracestore [**memory** | *mpi_fname, mio_fname*]

Specifies the mode for trace data collection. If **memory** is specified then trace data is held in each application task's memory until **MPI_Finalize** is called. If *mpi_fname, mio_fname* is specified, then this option specifies the path names of intermediate files where trace data from MPI trace and/or I/O trace are stored. The default is **memory**.

--help Displays a usage message.

Description

The **hpcrun** command is used to collect profiling and trace data from a subset of application tasks based on the metric the user specifies and the number of tasks that the user has selected for profiling and trace data collection. The **hpcrun** command can be invoked from the HPC Toolkit Eclipse plug-in, the hpctView application, or from the command line to run the application and collect performance data from the subset of application tasks.

Collecting profiling and trace data for a subset of tasks reduces the system resources required to process and store this data. It also reduces the volume of data that you have to analyze to evaluate your application's performance.

The subset of tasks is determined by querying each task for either the elapsed time or CPU time it consumed, depending on whether you specified the **ELAPSED_TIME** metric or the **CPU_TIME** metric.

The number of tasks is determined by the value the user specified for the **--exmetric** option. The **hpcrun** command will collect data for the number of tasks you specified with times closest to the minimum and maximum values for the metric you specified. Data will also be collected for tasks zero and the task closest to the average for the metric you specified if those tasks are not in the subsets closest to the minimum or maximum value of the metric you specified.

For example, if you specified **--excount 4** then the **hpcrun** command will collect data for the 4 tasks closest to the minimum metric value, the 4 tasks closest to the maximum metric value, and tasks zero and the task closest to the average metric value.

If you are collecting trace data, you have two options for handling that data. If you specify **--tracestore memory**, then the trace data for each application task is stored in that application's memory until **MPI_Finalize** is called. There is less processing overhead when trace data is stored in each task's memory, but this requires additional storage beyond what your application already uses.

For MPI trace, each traced MPI function call requires 52 bytes of memory. For I/O trace, each traced I/O system call requires approximately 44 bytes of memory.

If tracing requires too much memory, you can specify the **--tracestore** option with the *mpi_fname*, *mio_fname* parameters. If you specify this form of the **--tracestore** option, then MPI and/or I/O trace data will be stored in the respective user-specified temporary files where the pathname of the file is specified by the *mpi_fname* parameter for MPI trace and by the *mio_fname* parameter for I/O tracing.

Once trace processing is complete, **hpcrun** will process the temporary trace files to generate the final trace files and delete the temporary files.

Environment variables

HPC_EXCEPTION_COUNT

Equivalent to the **--excount** option. The **--excount** option overrides this environment variable.

HPC_EXCEPTION_METRIC

Equivalent to the **--exmetric** option. The **--exmetric** option overrides this environment variable.

HPC_TRACE_STORE

Equivalent to the **--tracestore** option. The **--tracestore** option overrides this environment variable.

HPC_TRACE_MAX_BUFFERS

Set this to a positive integer value that defines the number of in-memory buffers that are used to store trace events. When the in-memory buffers are full, a separate thread starts sending the data back to the home node. The default value is **1**. The values set by the environment variables **MAX_TRACE_EVENTS** (see “Controlling traced tasks” on page 96) and **HPC_TRACE_MAX_BUFFERS** buffers determine the maximum number of trace events that can be captured when an applications has been launched using **hpcrun**. The environment variable **HPC_TRACE_MAX_BUFFERS** only has an effect when you are *not* using the **memory** option of **HPC_TRACE_STORE** or **--tracestore**.

Miscellaneous environment variables

IHPCT_BASE

Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppdev.hpct**).

hpcstat - Reports a system-wide summary of hardware performance counter statistics

Use the **hpcstat** command to report a system-wide summary of hardware performance counter statistics.

Syntax

```
hpcstat [-o name [-n] [-x] [-k] [-u] [-I time] [-U time] [-C count] [-g group]
hpcstat [-h]
hpcstat [-l]
hpcstat [-L]
hpcstat [-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, or a comma-delimited list of hardware counter groups to be multiplexed. If this flag or the **HPM_EVENT_SET** environment variable is not set, a processor specific default group is used. The default group for POWER8 is group number 226, defined as follows:
 - PM_FPU0_FCONV**
Convert instruction executed.
 - PM_FPU0_FEST**
Estimate instruction executed.
 - PM_FPU0_FRSP**
Round to single precision instruction executed.
 - PM_LSU_LDF**
FPU loads only on LS2/LS3 ie LU0/LU1.
 - PM_RUN_INST_CMPL**
Run_Instructions.
 - PM_RUN_CYC**
Run_cycles.
- Note:** Only CPU hardware counter groups can be specified when running this command.
- h Displays a usage message.
 - I *time* Specifies the interval, in seconds, for reporting statistics. The default sleep time is 10 seconds.
 - 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.
 - L Lists the available hardware counter groups and the counters contained in

each group. The individual counter information will include a short name, the hardware event ID and a short text description. This flag is only available when running Linux.

-n Suppresses **hpcstat** output to stdout.

-o *name*

Writes output to a file with the prefix of *name*.

- The file prefix *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 file prefix may contain a directory path. If not specified, the file prefix is **hpct**.
- The name *name* is expanded into different file names:
 - *name.hpm.hpcstat.txt* is the file name for ASCII output, which is a one-to-one copy of the screen output.
 - *name.hpm.hpcstat.viz* is the file name for the XML output. This file can be viewed using the **hpctView** application or the **HPCT** plugin.
- Which of these output files are generated is governed by the additional environment variables, **HPM_ASC_OUTPUT** and **HPM_VIZ_OUTPUT**. If neither of these are set, only the ASCII output 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 reports system-wide hardware performance counter statistics and derived hardware metrics to **stdout** or to a file. If the **-C** flag and either the **-I** or **-U** flags are used, **hpcstat** reports hardware performance counter statistics on a periodic basis, similar to the **vmstat** command.

The **hpcstat** command requires the user to have **root** privileges.

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. If the **-g** flag is not used and **HPM_EVENT_SET** is not set, a processor-specific default group is used. The default group for POWER8 is group number 226, defined as follows:

PM_FPU0_FCONV

Convert instruction executed.

PM_FPU0_FEST

Estimate instruction executed.

PM_FPU0_FRSP

Round to single precision instruction executed.

PM_LSU_LDF

FPU loads only on LS2/LS3 ie LU0/LU1.

PM_RUN_INST_CMPL

Run_Instructions.

PM_RUN_CYC

Run_cycles.

Note: Only CPU hardware counter groups can be specified when running this command.

HPM_COUNTING_MODE

Specifies the CPU mode where counting will occur. Set this to a comma separated list of any combination of the following three possible values:

user Set to user to have user side events counted.

kernel Set to kernel to have kernel or system events counted.

hypervisor

Set to hypervisor to have hypervisor events counted.

The default setting for **hpcstat** is user and kernel.

Output control environment variables

HPM_ASC_OUTPUT

Determines whether or not to generate an ASCII output file. The output file name is:

name.hpm.hpcstat.txt

Valid values are **yes** or **no**. The default value for **HPM_ASC_OUTPUT** is true, except if **HPM_VIZ_OUTPUT** has been set, in which case the default value for **HPM_ASC_OUTPUT** is false.

HPC_OUTPUT_NAME

Specifies the *name* prefix of the output files:

name.hpm.hpcstat.txt

and

name.hpm.hpcstat.viz

The *name.hpm.hpcstat.viz* file can be viewed using the hpctView application or the HPCT plugin. For more information about file naming conventions, see Appendix A, "Performance data file naming," on page 197.

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.

HPC_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 **HPC_OUTPUT_NAME**. If **HPC_UNIQUE_FILE_NAME** is set to **yes**, the following rules apply:

- For serial programs, a string `_processId` is appended to either the file prefix if no directory path is specified, or to the directory path if one is given.
- For MPI programs that do not use dynamic tasking, a string `_0_taskRank` is appended to either the file prefix if no directory path is specified, or to the directory path if one is given.
- For MPI programs that use dynamic tasking, a string `_worldID_worldRank` is appended to either the file prefix if no directory path is specified, or to the directory path if one is given.

HPM_VIZ_OUTPUT

Set to **yes** to generate an XML output file with the name:

`name.hpm.hpcstat.viz`

The default value for **HPM_VIZ_OUTPUT** is **false**. This file can be viewed using the `hpctView` application or the HPCT plugin.

Miscellaneous environment variables

IHPCT_BASE

Specifies the path name of the directory in which the IBM HPC Toolkit is installed (`/opt/ibmhpc/ppdev.hpct`).

Files

`name.hpm.hpcstat.txt`

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

`name.hpm.hpcstat.viz`

An XML output file containing hardware performance counter data from **hpcstat** execution. This file can be viewed using the `hpctView` application or the HPCT plugin.

Examples

1. To list hardware performance counter groups available on your processor, enter:
hpcstat -l
2. To report floating point unit (FPU) instructions on pipes 0 and 1 for the system every 30 seconds for five minutes on POWER8 servers, enter:
hpcstat -u -C 10 -I 30 -g 127

hpctlInst - Instrument applications to obtain performance data

Use the **hpctlInst** command to instrument applications in order to obtain performance data.

Syntax

```
hpctlInst [-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_user{none | BeginEnd}] [-dpomp_userfunc file name]  
[-dpomp_loop{none | EnterExit | Chunks}]  
[-dlink args] binary  
[-h | --help]
```

Note: This command applies only to Power Architecture.

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 *file name*.

-dhpm_region *file name*

Instrument regions of code with HPM instrumentation, as specified by the contents of *file name*.

-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 *file name*.

-dmpi_region *file name*

Instrument MPI calls in regions of code in the application, as specified by *file name*.

-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 {none | BeginEnd}

Instrument user functions for OpenMP profiling.

-dpomp_userfunc *file name*

Instrument the user functions specified in *file name* 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 *file_name* 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 file system, 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 with the **-g** flag. In addition, the application must also be linked with these flags:

-Wl,--hash-style=sysv -emit-stub-syms

Environment variables

IHPCT_BASE

Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

LD_LIBRARY_PATH

Must be set to **\$IHPCT_BASE/lib64**.

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

1. To instrument all MPI calls in an application, enter:
hpctInst -dmpi testprog
2. To instrument call sites for function **testfunc** from file **main.c** lines 1 through 100 with HPM instrumentation, enter:
hpctInst -dhpm_func_call inst_spec testprog
where the **inst_spec** file contains the single line:
testfunc main.c 1 100
3. To instrument all parallel loops and parallel regions in an OpenMP application with **EnterExit** instrumentation, enter:
hpctInst -dpomp_loop -dpomp_parallel testprog

Chapter 18. Application programming interfaces

Table 19 lists the application programming interfaces (APIs) and what they are used for:

Table 19. APIs

Used for:	API
Hardware performance monitoring	<ul style="list-style-type: none"> • “gpm_init - Initialize the GPU Performance Monitor runtime environment” on page 133 • “gpm_start - Identify the starting point of an instrumented region of code” on page 136 • “gpm_stop - Identify the end point of an instrumented region of code” on page 139 • “gpm_terminate - Generate GPU Performance Monitoring statistics and trace files and shut down the GPM runtime environment” on page 142 • “gpm_Tstart - Identify the starting point of an instrumented region of code” on page 145 • “gpm_Tstop - Identify the end point of an instrumented region of code” on page 148 • “hpm_error_count, f_hpm_error - Verify a call to a libhpc function” on page 151 • “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 153 • “hpmTstart, f_hpmtstart - Identify the starting point for an instrumented region of code” on page 167 • “hpmStartx, f_hpmstartx - Identify the starting point for an instrumented region of code” on page 160 • “hpmStop, f_hpmstop - Identify the end point of an instrumented region of code” on page 163 • “hpmTerminate, f_hpmtterminate - Generate HPM statistic files and shut down HPM” on page 165 • “hpmTstart, f_hpmtstart - Identify the starting point for an instrumented region of code” on page 167 • “hpmTstartx, f_hpmtstartx - Identify the starting point for an instrumented region of code” on page 169 • “hpmTstop, f_hpmtstop - Identify the end point of an instrumented region of code” on page 172

Table 19. APIs (continued)

Used for:	API
MPI profiling customization	<ul style="list-style-type: none"> • “MT_get_allresults - Obtain statistical results” on page 174 • “MT_get_calleraddress - Obtain the address of the caller of an MPI function” on page 177 • “MT_get_callerinfo - Obtain source code information” on page 178 • “MT_get_elapsed_time - Obtains elapsed time” on page 180 • “MT_get_environment - Returns run-time environment information” on page 181 • “MT_get_mpi_bytes - Obtain the accumulated number of bytes transferred” on page 182 • “MT_get_mpi_counts - Obtain the the number of times a function was called” on page 183 • “MT_get_mpi_name - Returns the name of the specified MPI function” on page 184 • “MT_get_mpi_time - Obtain elapsed time” on page 185 • “MT_get_time - Get the elapsed time” on page 186 • “MT_get_tracebufferinfo - Obtain information about MPI trace buffer usage” on page 187 • “MT_output_text - Generate performance statistics” on page 188 • “MT_output_trace - Control whether an MPI trace file is created” on page 189 • “MT_trace_event - Control whether an MPI trace event is generated” on page 190 • “MT_trace_start, mt_trace_start - Start or resume the collection of trace events” on page 192 • “MT_trace_stop, mt_trace_stop - Suspend the collection of trace events” on page 194

gpm_init - Initialize the GPU Performance Monitor runtime environment

Use **gpm_init** to initialize the GPU Performance Monitor (GPM) runtime environment.

Library

libgpm.so (-lgpm)

C syntax

```
#include <gpm.h>
int gpm_init(void)
```

FORTRAN syntax

```
INCLUDE "h_gpm.inc"
INTEGER FUNCTION gpm_init()
```

Parameters

None.

Description

The **gpm_init()** function initializes the runtime environment of GPM for obtaining GPU hardware performance counter statistics and trace information. Applications must call this function before calling any other GPM functions.

During the initialization of the GPM runtime environment, this routine reads the set of GPU events or metrics that will be profiled by GPM. These events or metrics are specified with the **GPM_EVENT_SET** and **GPM_METRIC_SET** environment variables, respectively.

The include files shown in the syntax are located in the **\$(IHPCT_BASE)/include** directory.

Returns

Upon successful completion, **gpm_init** returns **GPM_OK** and initializes the GPM runtime environment.

If an error occurs, **gpm_init** returns one of the following:

GPM_ENOSTATIC

The application is statically linked to **libc**.

GPM_ENOLIBS

One of the CUDA or CUPTI libraries is not found.

GPM_ECUDA

An error occurred in the CUDA runtime library.

GPM_ECUPTI

An error occurred in the CUPTI library.

GPM_ENODEV

Unable to detect any GPU device.

GPM_ENOMEM

Memory allocation failure.

GPM_EMISC

Pthread related or other errors.

More information about the error can be obtained from the PE DE log file. For information about how to generate a log file, see Chapter 5, “Generating a log file,” on page 21.

Environment variables

GPM_EVENT_SET

Specifies a comma-separated (no spaces) list of events to be measured. Events supported by the GPU devices can be listed with the **gpmlist -e** command.

Events to be profiled can be specified in two ways:

1. For all GPU devices
2. For a specific GPU device

Events to be profiled for all devices are specified using their name. Events to be profiled for specific devices are specified using the following syntax:

device:<device number>:<event name>

For example, to profile **inst_executed** on all GPU devices and **threads_launched** on GPU device 1, set **GPM_EVENT_SET** to the following:

GPM_EVENT_SET=inst_executed,device:1:threads_launched

GPM_METRIC_SET

Specifies a comma-separated (no spaces) list of metrics to be measured. Metrics supported by the GPU devices can be listed with the **gpmlist -m** command.

Note: Metrics specification follows the same syntax as for events.

IHPCT_BASE

Specifies the path name of the directory where the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Example

```
#include <gpm.h>
int main(int argc, char **argv)
{
    int rc = 0;
    ...
    rc = gpm_init();
    if (0 != rc) {
        /* process error code */
        ...
    }

    ...

    rc = gpm_terminate();
    if (0 != rc) {
        /* process error code */
        ...
    }
}
```

```
PROGRAM gpmtest
INCLUDE "f_gpm.inc"
```

```
|
|
|      RC = gpm_init()
|      if (RC != 0)
|      ! process return code
|
|      ...
|
|      RC = gpm_terminate()
|      if (RC != 0)
|      ! process return code
|
|      END PROGRAM
```

gpm_start - Identify the starting point of an instrumented region of code

Use **gpm_start** to identify the starting point of an instrumented region of code for which GPU hardware performance counter events and metrics are to be counted.

Library

libgpm.so (-lgpm)

C syntax

```
#include <gpm.h>
int gpm_start(void)
```

FORTRAN syntax

```
INCLUDE "h_gpm.inc"
INTEGER FUNCTION gpm_start()
```

Parameters

None.

Description

The **gpm_start()** function identifies a region of code in which GPU hardware performance counter events are to be counted. The end of the region is identified by a call to the **gpm_stop()** function.

The region of the code identified by calls to **gpm_start()** and **gpm_stop()** functions must be in code that runs on the CPU and not on the GPU. The effect of this call is to enable CUPTI callbacks used for counting the specified GPU hardware counter events and metrics. When enabled, the CUPTI callbacks will always remain enabled for the entire execution of any CUDA kernels.

In multithreaded applications, the counting of GPU hardware counters is performed for all GPU kernels launched from all POSIX threads in the process. If you need to measure GPU hardware counter events for GPU kernels launched from specific POSIX threads, use the **gpm_Tstart()** and **gpm_Tstop()** functions.

The **gpm_start()** and **gpm_stop()** functions must be called in pairs. Nesting of the instrumented code regions has no effect on the counting of the GPU hardware counting events.

The include files shown in the syntax are located in the **\$(IHPCT_BASE)/include** directory.

For a complete list of environment variables used by this function, see Appendix C, "HPC Toolkit environment variables," on page 271.

Returns

Upon successful completion, **gpm_start** returns 0.

If an error occurs, **gpm_start** returns the following:

GPM_EMISC

Improper use of **gpm_start** and **gpm_stop** pairing.

Environment variables

IHPCT_BASE

Specifies the path name of the directory where the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Example

```
#include <gpm.h>
int main(int argc, char **argv)
{
    int rc = 0;

    ...

    rc = gpm_init();
    if (0 != rc) {
        /* process error code */

        ...
    }

    /* start counting GPM hardware counter events */
    rc = gpm_start();
    if (0 != rc) {
        /* process error code here */

        ...
    }

    ...

    /* stop counting GPM hardware counter events */
    rc = gpm_stop();
    if (0 != rc) {
        /* process error code here */

        ...
    }

    rc = gpm_terminate();
    if (0 != rc) {
        /* process error code */

        ...
    }
}

PROGRAM gpmtest
INCLUDE "f_gpm.inc"

RC = gpm_init()
if (RC != 0)
! process return code
...
RC = gpm_start()
if (RC != 0)
! process return code
...

...

RC = gpm_stop()
if (RC != 0)
```

```
|      ! process return code  
|      ...  
|  
|      RC = gpm_terminate()  
|      if (RC != 0)  
|      ! process return code  
|      ...  
|  
|      END PROGRAM
```

gpm_stop - Identify the end point of an instrumented region of code

Use **gpm_stop** to identify the end point of an instrumented region of code for which GPU hardware performance counter events and metrics are to be counted.

Library

libgpm.so (-lgpm)

C syntax

```
#include <gpm.h>
int gpm_stop(void)
```

FORTRAN syntax

```
INCLUDE "h_gpm.inc"
INTEGER FUNCTION gpm_stop()
```

Parameters

None.

Description

The **gpm_stop()** function identifies the end of a region of code in which GPU hardware performance counter events are to be counted. The beginning of the instrumented region of code is identified by a call to the **gpm_start()** function.

The **gpm_start()** and **gpm_stop()** routines must be called in pairs. Nesting the instrumented regions has no effect on how the GPU hardware counter events and metrics are counted.

The region of the code identified by calls to the **gpm_start()** and **gpm_stop()** functions must be in code that runs on the CPU and not on the GPU. The effect of this call is to disable the CUPTI callbacks used for counting the specified GPU hardware counter events and metrics. When enabled, the CUPTI callbacks will always remain enabled for the entire execution of any CUDA kernels.

In multithreaded applications, the counting of GPU hardware counters is performed for all GPU kernels launched from all POSIX threads in the process. If you need to measure GPU hardware counter events for GPU kernels launched from specific POSIX threads, use the **gpm_Tstart()** and **gpm_Tstop()** functions.

The include files shown in the syntax are located in the **\$(IHPCT_BASE)/include** directory.

For a complete list of environment variables used by this function, see Appendix C, "HPC Toolkit environment variables," on page 271.

Returns

Upon successful completion, **gpm_stop** returns 0.

If an error occurs, **gpm_stop** returns the following:

GPM_EMISC

Improper use of the **gpm_start()** and **gpm_stop()** pairing.

Environment variables

HPCT_BASE

Specifies the path name of the directory where the IBM HPC Toolkit is installed (`/opt/ibmhpc/ppdev.hpct`).

Example

```
#include <gpm.h>
int main(int argc, char **argv)
{
    int rc = 0;

    ...

    rc = gpm_init();
    if (0 != rc) {
        /* process error code */
        ...
    }

    /* start counting GPM hardware counter events */
    rc = gpm_start();
    if (0 != rc) {

        /* process error code here */
        ...
    }

    ...

    /* stop counting GPM hardware counter events */
    rc = gpm_stop();
    if (0 != rc) {
        /* process error code here */
        ...
    }

    rc = gpm_terminate();
    if (0 != rc) {
        /* process error code */
        ...
    }
}
```

```
PROGRAM gpmtest
INCLUDE "f_gpm.inc"

RC = gpm_init()
if (RC != 0)
! process return code
...

RC = gpm_start()
if (RC != 0)
! process return code
...

...

RC = gpm_stop()
if (RC != 0)
! process return code
...
```



```
|      RC = gpm_terminate()
|      if (RC != 0)
|      ! process return code
|      ...
|
|      END PROGRAM
```

gpm_terminate - Generate GPU Performance Monitoring statistics and trace files and shut down the GPM runtime environment

Use **gpm_terminate** to terminate the GPU Performance Monitoring (GPM) runtime environment and generate statistics and trace files.

Library

libgpm.so (-lgpm)

C syntax

```
#include <gpm.h>
int gpm_terminate(void)
```

FORTRAN syntax

```
INCLUDE "h_gpm.inc"
INTEGER FUNCTION gpm_terminate()
```

Parameters

None.

Description

The **gpm_terminate** function generates statistics and output files for the events and metrics profiled, and terminates the GPM runtime environment.

This function must be called before the application terminates to generate statistics and trace information and must be the last GPM function called by the application.

This function produces several types of output, which users can control using environment variables.

The include files shown in the syntax section are located in the **\$(IHPCT_BASE)/include** directory.

For a complete list of environment variables used by this function, see Appendix C, "HPC Toolkit environment variables," on page 271.

Returns

Upon successful completion, **gpm_terminate** returns **GPM_OK**, terminates the GPM runtime environment, and produces statistics and trace data.

If an error occurs, **gpm_terminate** returns one of the following:

GPM_ECUPTI

An error occurred in the CUPTI library.

GPM_ENOMEM

Memory allocation failure.

GPM_EFOPEN

Error opening one of the output files.

More information about the error can be obtained from the PE DE log file. For information about how to generate a log file, see Chapter 5, “Generating a log file,” on page 21.

Environment variables

GPM_ASC_OUTPUT

Instructs the routine to produce an ASCII text file (.txt) containing the cumulative results of the event or metric profiling. The ASCII text file can be viewed with a text editor.

GPM_ENABLE_TRACE

Instructs the routine to produce a trace file containing all profiling records. The trace file can be visualized using the **hpctView** command.

GPM_PRINT

Instructs the routine to produce statistics and trace data in files.

GPM_STDOUT

Instructs the routine to produce statistics and profiling data to **stdout**. The results provided are not cumulative. Instead, each performance record is output to **stdout**.

GPM_VIZ_OUTPUT

Instructs the routine to produce an XML file (.viz) containing the cumulative results of the event or metric profiling. The XML visualization file can be viewed with the **hpctView** command.

IHPCT_BASE

Specifies the path name of the directory where the IBM HPC Toolkit is installed (/opt/ibmhpc/ppedev.hpct).

Example

```
#include <gpm.h>
int main(int argc, char **argv)
{
    int rc = 0;
    ...
    rc = gpm_init();
    if (0 != rc) {
        /* process error code */
        ...
    }

    ...

    rc = gpm_terminate();
    if (0 != rc) {
        /* process error code */
        ...
    }
}
```

```
PROGRAM gpctest
INCLUDE "f_gpm.inc"
```

```
RC = gpm_init()
if (RC != 0)
! process return code

...
```

```
|      RC = gpm_terminate()
|      if (RC != 0)
|      ! process return code
|
|      END PROGRAM
```

gpm_Tstart - Identify the starting point of an instrumented region of code

Use **gpm_Tstart** to identify the starting point of an instrumented region of code for which GPU hardware performance counter events and metrics are to be counted on a per-thread basis.

Library

libgpm.so (-lgpm)

C syntax

```
#include <gpm.h>
int gpm_Tstart(void)
```

FORTRAN syntax

```
INCLUDE "h_gpm.inc"
INTEGER FUNCTION gpm_Tstart()
```

Parameters

None.

Description

The **gpm_Tstart()** function identifies a region of code in which GPU hardware performance counter events are to be counted. The end of the region is identified by a call to the **gpm_Tstop()** function.

The region of the code identified by calls to the **gpm_Tstart()** and **gpm_Tstop()** functions must be in code that runs on the CPU and not on the GPU. The effect of this call is to enable the CUPTI callbacks used for counting the specified GPU hardware counter events and metrics. When enabled, the CUPTI callbacks will always remain enabled for the entire execution of any CUDA kernels.

In multithreaded applications, the counting of GPU hardware counters is performed for all GPU kernels launched from the POSIX thread in which the call to **gpm_Tstart()** was made. If you need to measure GPU hardware counter events for GPU kernels launched from all POSIX threads, use the **gpm_start()** and **gpm_stop()** functions.

The **gpm_Tstart()** and **gpm_Tstop()** functions must be called in pairs. Nesting of the instrumented code regions within the same POSIX thread has no effect on the counting of the GPU hardware counting events for that thread.

The include files shown in the syntax are located in the **\$(IHPCT_BASE)/include** directory.

For a complete list of environment variables used by this function, see Appendix C, “HPC Toolkit environment variables,” on page 271.

Returns

Upon successful completion, **gpm_Tstart** returns 0.

If an error occurs, **gpm_Tstart** returns the following:

GPM_EMISC

POSIX thread error or improper use of **gpm_Tstart()** and **gpm_Tstop()** pairing.

Environment variables

IHPCT_BASE

Specifies the path name of the directory where the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Example

```
#include <gpm.h>
int main(int argc, char **argv)
{
    int rc = 0;

    ...

    rc = gpm_init();
    if (0 != rc) {
        /* process error code */
        ...
    }

    /* start counting GPM hardware counter events */
    rc = gpm_Tstart();
    if (0 != rc) {
        /* process error code here */
        ...
    }

    ...

    /* stop counting GPM hardware counter events */
    rc = gpm_Tstop();
    if (0 != rc) {
        /* process error code here */
        ...
    }

    rc = gpm_terminate();
    if (0 != rc) {
        /* process error code */
        ...
    }
}

PROGRAM gpmtest
INCLUDE "f_gpm.inc"

RC = gpm_init()
if (RC != 0)
! process return code
...

RC = gpm_Tstart()
if (RC != 0)
! process return code
```

```

...

...

RC = gpm_Tstop()
if (RC != 0)

! process return code
...

RC = gpm_terminate()
if (RC != 0)
! process return code
...

END PROGRAM

```

gpm_Tstop - Identify the end point of an instrumented region of code

Use **gpm_Tstop** to identify the end point of an instrumented region of code identified using **gpm_Tstart()** for which GPU hardware performance counter events and metrics are to be counted.

Library

libgpm.so (-lgpm)

C syntax

```
#include <gpm.h>
int gpm_Tstop(void)
```

FORTRAN syntax

```
INCLUDE "h_gpm.inc"
INTEGER FUNCTION gpm_Tstop()
```

Parameters

None.

Description

The **gpm_Tstop()** function identifies the end of a region of code in which GPU hardware performance counter events are to be counted. The beginning of the instrumented region of code is identified by a call to the **gpm_Tstart()** function.

The **gpm_Tstart()** and **gpm_Tstop()** routines must be called in pairs in the same POSIX thread. Nesting the instrumented regions in the same POSIX thread has no effect on how the GPU hardware counter events and metrics are counted.

The region of the code identified by calls to the **gpm_Tstart()** and **gpm_Tstop()** functions must be in code that runs on the CPU and not on the GPU. The effect of this call is to disable the CUPTI callbacks used for counting the specified GPU hardware counter events and metrics. When enabled, the CUPTI callbacks will always remain enabled for the entire execution of any CUDA kernels.

In multithreaded applications, the counting of GPU hardware counters is performed only for the GPU kernels launched from the POSIX threads in which the **gpm_Tstart()** and **gpm_Tstop()** calls were made. If you need to measure GPU hardware counter events for GPU kernels launched from all POSIX threads, use the **gpm_start()** and **gpm_stop()** functions.

The include files shown in the syntax are located in the **\$(IHPCT_BASE)/include** directory.

For a complete list of environment variables used by this function, see Appendix C, "HPC Toolkit environment variables," on page 271.

Returns

Upon successful completion, **gpm_Tstop()** returns 0.

If an error occurs, **gpm_Tstop()** returns the following:

GPM_EMISC

Error from the POSIX thread or improper use of the **gpm_Tstart()** and **gpm_Tstop()** pairing.

Environment variables

HPCT_BASE

Specifies the path name of the directory where the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Example

```
#include <gpm.h>
int main(int argc, char **argv)
{
    int rc = 0;

    ...

    rc = gpm_init();
    if (0 != rc) {
        /* process error code */
        ...
    }

    /* start counting GPM hardware counter events */
    rc = gpm_Tstart();
    if (0 != rc) {
        /* process error code here */
        ...
    }

    ...

    /* stop counting GPM hardware counter events */
    rc = gpm_Tstop();
    if (0 != rc) {
        /* process error code here */
        ...
    }

    rc = gpm_terminate();
    if (0 != rc) {
        /* process error code */
        ...
    }
}

PROGRAM gpctest
INCLUDE "f_gpm.inc"

RC = gpm_init()
if (RC != 0)
! process return code
...

RC = gpm_Tstart()
if (RC != 0)
! process return code
...

...

RC = gpm_Tstop()
if (RC != 0)
```

```
|      ! process return code  
|      ...  
|  
|      RC = gpm_terminate()  
|      if (RC != 0)  
|      ! process return code  
|      ...  
|  
|      END PROGRAM
```

hpm_error_count, f_hpm_error - Verify a call to a libhpc function

Use **hpm_error_count** or **f_hpm_error** to verify that a call to a **libhpc** function was successful.

Library

-lhpc

C syntax

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

FORTRAN syntax

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

Notes:

1. Use **f_hpc.h** for programs compiled without **-qintsize=8** and use **f_hpc_i8.h** for programs compiled with **-qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xlfc** **-qsuffix** option, for instance, **-qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

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. The **f_hpm_error()** function 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_hpm_error()** returns **false**. If an HPM library call fails, **hpm_error_count** is set to a non zero value and **f_hpm_error()** returns **true**.

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

The include files shown in the syntax are located in the **\$(IHPCT_BASE)/include** directory.

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(0);
        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(0)
if (f_hpm_error() .eqv. .true.) then
    print *, 'f_hpmterminate error'
    stop 1
end if
end

```

hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment

Use **hpmInit** or **f_hpminit** to initialize the Hardware Performance Monitor (HPM) run-time environment.

Library

-lhpc

C syntax

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

FORTRAN syntax

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

Notes:

1. Use **f_hpc.h** for programs compiled without **-qintsize=8** and use **f_hpc_i8.h** for programs compiled with **-qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xlfc** **-qsuffix** option, for instance, **-qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

Parameters

my_ID Unused. Should be set to zero.

progName

Specifies the name of the program. If the **HPC_OUTPUT_NAME** environment variable is not set, this parameter is used as the file name prefix. If it is NULL, then the value **hpct** is used.

Description

This function initializes the run-time 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.

For more information about file naming conventions, see Appendix A, “Performance data file naming,” on page 197.

The include files shown in the syntax are located in the **\$(IHPCT_BASE)/include** directory.

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. When multiple CPU hardware counter groups are specified, they will be multiplexed. If

HPM_EVENT_SET is not set, a processor-specific default group is used. The default group for POWER8 is group number 226, defined as follows:

PM_FPU0_FCONV

Convert instruction executed.

PM_FPU0_FEST

Estimate instruction executed.

PM_FPU0_FRSP

Round to single precision instruction executed.

PM_LSU_LDF

FPU loads only on LS2/LS3 ie LU0/LU1.

PM_RUN_INST_CMPL

Run_Instructions.

PM_RUN_CYC

Run_cycles.

HPM_EXCLUSIVE_VALUES

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

HPM_COUNTING_MODE

Specifies the CPU mode where counting will occur. Set this to a comma-separated list of any combination of the following three possible values:

user Set to user to have user-side events counted.

kernel Set to kernel to have kernel or system events counted.

hypervisor

Set to hypervisor to have hypervisor events counted.

The default setting for **libhpc.a** is **user**.

Output control environment variables

HPM_ASC_OUTPUT

Determines whether or not to generate an ASCII output file. The output file name is:

name.**hpm**[*.progName*].**txt**

where:

name

Is obtained from the **HPC_OUTPUT_NAME** environment variable.

progName

Is specified if the second parameter to the **hpmInit()** call is non-NULL.

Valid values are **yes** or **no**.

If neither **HPM_ASC_OUTPUT** or **HPM_VIZ_OUTPUT** are set, both the ASCII and XML output files are 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

HPCT_OUTPUT_NAME

Specifies the *name* prefix of the output files:

name.**hpm**[*.progName*].**txt**

and

name.**hpm**[*.progName*].**viz**

The *name*.**hpm**[*.progName*].**viz** file can be viewed using the hpctView application or the HPCT plugin.

where:

progName

Is specified if the second parameter to the **hpmInit()** call is non-NULL.

If this environment variable is not set, then *name* is set to the value of the **hpct**. For more information about file naming conventions, see Appendix A, "Performance data file naming," on page 197.

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**.

HPCT_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 **HPCT_OUTPUT_NAME**. For more information about file naming conventions, see Appendix A, "Performance data file naming," on page 197.

HPM_VIZ_OUTPUT

Set to **yes** to generate an XML output file with the name:

name.**hpm**[*.progName*].**viz**

where:

name

Is specified by the **HPCT_OUTPUT_NAME** environment variable.

progName

Is specified if the second parameter to the **hpmInit()** call is non-NULL.

If neither **HPM_ASCII_OUTPUT** or **HPM_VIZ_OUTPUT** are set, both the ASCII and XML output files are generated. If at least one is set, the following rules apply:

- **HPM_ASCII_OUTPUT**, if set to **yes**, triggers the ASCII output.
- **HPM_VIZ_OUTPUT**, if set to **yes**, triggers the XML output. This file can be viewed using the hpctView application or the HPCT plugin.

Plug-in specific environment variables

HPM_ROUND_ROBIN_CLUSTER

HPM_ROUND_ROBIN_CLUSTER allows setting the number of groups distributed per task.

Without the environment variable **HPM_ROUND_ROBIN_CLUSTER** set, the **average.so** plug-in will distribute the group numbers from **HPM_EVENT_SET** in a round-robin fashion, one group to each of the MPI tasks in the application.

The default value for **HPM_ROUND_ROBIN_CLUSTER** is **1**. The default will be used if a value less than 1 is specified.

The number of groups spread out round-robin fashion to the tasks will be limited to the first "number of tasks times the setting of **HPM_ROUND_ROBIN_CLUSTER**" groups.

If a value greater than the number of groups in **HPM_EVENT_SET** is specified, **HPM_ROUND_ROBIN_CLUSTER** will be set to the number of groups specified in **HPM_EVENT_SET**.

It is possible that the number of groups does not distribute evenly to the tasks. The first task will get at most **HPM_ROUND_ROBIN_CLUSTER** of the groups in **HPM_EVENT_SET**. If there are more tasks and groups left, the second task will get at most **HPM_ROUND_ROBIN_CLUSTER** of the groups left in **HPM_EVENT_SET** and so on, until there are no groups unused in **HPM_EVENT_SET**. After the groups in **HPM_EVENT_SET** have been used once, and there are more tasks, this process will repeat until there are no more tasks.

The environment variable **HPM_ROUND_ROBIN_CLUSTER** 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 "Hardware performance counter plug-ins" on page 83 for more information.

IHPCT_BASE

Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Examples

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

program hpmtest
#include "f_hpc.h"
call f_hpminit(0, 'HPMTest')
```



```
.  
.   
.   
call f_hpmtterminate(0)  
end
```

hpmStart, f_hpmstart - Identify the starting point for an instrumented region of code

Use **hpmStart** or **f_hpmstart** to identify the starting point for an instrumented region of code in which hardware performance counter events are to be counted.

Library

-lhpc

C syntax

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

FORTRAN syntax

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

Notes:

1. Use **f_hpc.h** for programs compiled without **-qintsize=8** and use **f_hpc_i8.h** for programs compiled with **-qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xlfc** **-qsuffix** option, for instance, **-qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

Parameters

inst_ID

Specifies a unique value identifying the instrumented code region.

label

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()**.

The include files are located in the `${IHPCT_BASE}/include` directory.

Environment variables

See “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 153.

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(0);
}

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(0)
end
```

hpmStartx, f_hpmstartx - Identify the starting point for an instrumented region of code

Use **hpmStartx** or **f_hpmstartx** to identify the starting point for an instrumented region of code in which hardware performance counter events are to be counted, specifying explicit inheritance relationships for nested instrumentation regions.

Library

-lhpc

C syntax

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

FORTRAN syntax

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

Notes:

1. Use **f_hpc.h** for programs compiled without **-qintsize=8** and use **f_hpc_i8.h** for programs compiled with **-qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xlfc** **-qsuffix** option, for instance, **-qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

Parameters

inst_ID

Specifies a unique value identifying the instrumented code region.

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

label

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()**.

The include files shown in the syntax are located in the **\$(IHPCT_BASE)/include** directory.

Environment variables

See “**hpmInit**, **f_hpminit** - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 153.

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(0);
}

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(0)
end

```

hpmStop, f_hpmstop - Identify the end point of an instrumented region of code

Use **hpmStop** or **f_hpmstop** to identify the end point of an instrumented region of code starting with a call to **hpmStart()** or **hpmStartx()**, in which hardware performance counter events are to be counted. It also accumulates hardware performance counter events for that region.

Library

-lhpc

C syntax

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

FORTRAN syntax

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

Notes:

1. Use **f_hpc.h** for programs compiled without **-qintsize=8** and use **f_hpc_i8.h** for programs compiled with **-qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xlfc** **-qsuffix** option, for instance, **-qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

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.

The include files shown in the syntax are located in the **\$(IHPCT_BASE)/include** directory.

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(0);
}

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(0)
end

```

hpmTerminate, f_hpmterminate - Generate HPM statistic files and shut down HPM

Use **hpmStop** or **f_hpmstop** to generate Hardware Performance Monitor (HPM) statistics files and to shut down the HPM environment.

Library

-lhpc

C syntax

```
#include <libhpc.h>
void hpmTerminate(int my_ID)
```

FORTRAN syntax

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

Notes:

1. Use **f_hpc.h** for programs compiled without **-qintsize=8** and use **f_hpc_i8.h** for programs compiled with **-qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xlfc** **-qsuffix** option, for instance, **-qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

Parameters

my_ID Unused. Should be set to zero.

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.

The include files shown in the syntax are located in the **\$(IHPCT_BASE)/include** directory.

Environment variables

See "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 153.

Examples

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

```

        .
        .
        .
        hpmTerminate(0);
    }

program hpmtest
#include "f_hpc.h"
call f_hpminit(0, 'HPMTest')
.
.
.
call f_hpmterminate(0)
end

```

hpmTstart, f_hpmTstart - Identify the starting point for an instrumented region of code

Use **hpmTstart** or **f_hpmTstart** to identify the starting point for an instrumented region of code in which hardware performance counter events are to be counted on a per-thread basis.

Library

-lhpc

C syntax

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

FORTRAN syntax

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

Notes:

1. Use **f_hpc.h** for programs compiled without **-qintsize=8** and use **f_hpc_i8.h** for programs compiled with **-qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xlfc** **-qsuffix** option, for instance, **-qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

Parameters

inst_ID

Specifies a unique value identifying the instrumented code region.

label

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.

The include files shown in the syntax are located in the **\$(IHPCT_BASE)/include** directory.

Environment variables

See “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 153.

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(0);
}
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 hpmttest
#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(0)
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 - Identify the starting point for an instrumented region of code

Use **hpmStartx** or **f_hpmstartx** to identify the starting point for an instrumented 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

C syntax

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

FORTRAN syntax

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

Notes:

1. Use **f_hpc.h** for programs compiled without **-qintsize=8** and use **f_hpc_i8.h** for programs compiled with **-qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xl** **-qsuffix** option, for instance, **-qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

Parameters

inst_ID

Specifies a unique value identifying the instrumented code region.

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

label

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 **hpmTstop()**.

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 **hpmTsStart()** or **hpmTstartx()** call that is currently active, meaning the corresponding call to **hpmTstop()** has not been made for this instance of execution.

The include files shown in the syntax are located in the **\$(IHPCT_BASE)/include** directory.

Environment variables

See “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 153.

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(0);
}
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 hpmttest
#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(0)
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 - Identify the end point of an instrumented region of code

Use **hpmTstop** or **f_hpmtstop** to identify the end point of an instrumented 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

C syntax

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

FORTRAN syntax

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

Notes:

1. Use **f_hpc.h** for programs compiled without **-qintsize=8** and use **f_hpc_i8.h** for programs compiled with **-qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xlfc -qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

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 **hpmTstop()** function.

The include files shown in the syntax are located in the **\$(IHPCT_BASE)/include** directory.

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(0);
}

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(0)
end

```

MT_get_allresults - Obtain statistical results

Use **MT_get_allresults** to obtain statistical results from performance data for an MPI function.

Library

-lmpitrace

C syntax

```
#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 the **\$IHPCT_BASE/include/mpi_trace_ids.h** header file, 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

ELAPSEDTIME

double

You must cast the 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 just described. 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

Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Example

```
#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 - Obtain the address of the caller of an MPI function

Use `MT_get_calleraddress` to obtain the address of the caller of an MPI function.

Library

`-lmpitrace`

C syntax

```
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.

Example

```
#include <mpt.h>
int MT_trace_event(int id)
{
    unsigned long caller_addr;
    caller_addr = MT_get_calleraddress(3);
    return 1;
}
```

MT_get_callerinfo - Obtain source code information

Use **MT_get_callerinfo** to obtain source code information about the caller of an MPI function.

Library

-lmpitrace

C syntax

```
#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 *filename
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.

Example

```
#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_get_calleraddress(3);
    status = MT_get_callerinfo(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 - Obtains elapsed time

Use **MT_get_elapsed_time** to get the elapsed time in seconds between a call to **MPI_Init()** and a call to **MPI_Finalize()**.

Library

-lmpitrace

C syntax

```
#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.

Example

```
#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 - Returns run-time environment information

Use **MT_get_environment** to return information about the run-time environment for the application.

Library

-lmpitrace

C syntax

```
#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*

The number of tasks in the MPI application.

int *nmpi*

The maximum index allowed for *mpi_id* when calling **MT_get_mpi_counts()**, **MT_get_mpi_bytes()**, and **MT_get_mpi_time()**.

Environment variables

None.

Example

```
#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 - Obtain the accumulated number of bytes transferred

Use **MT_get_mpi_bytes** to obtain the accumulated number of bytes transferred by a specific MPI function.

Library

-lmpitrace

C syntax

```
#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 must be any of the values as specified in Table 20 on page 190 or as specified in the **\$IHPCT_BASE/include/mpi_trace_ids.h** header file. 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

Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppdev.hpct**).

Example

```
#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 - Obtain the the number of times a function was called

Use **MT_get_mpi_counts** to obtain the number of times the specified MPI function was called.

Library

-lmpitrace

C syntax

```
#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 must be any of the values as specified in Table 20 on page 190 or as specified in the **\$IHPCT_BASE/include/mpi_trace_ids.h** header file.

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

Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppdev.hpct**).

Example

```
#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 - Returns the name of the specified MPI function

Use `MT_get_mpi_name` to return the name of the specified MPI function.

Library

`-lmpitrace`

C syntax

```
#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 must be any of the values as specified in Table 20 on page 190 or as specified in the `$IHPCT_BASE/include/mpi_trace_ids.h` header file.

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

Specifies the path name of the directory in which the IBM HPC Toolkit is installed (`/opt/ibmhpc/ppdev.hpct`).

Example

```
#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 - Obtain elapsed time

Use `MT_get_mpi_time` to obtain the elapsed time, in seconds, spent in the specified MPI function.

Library

`-lmpitrace`

C syntax

```
#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 must be any of the values as specified in Table 20 on page 190 or as specified in the `$IHPCT_BASE/include/mpi_trace_ids.h` header file.

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`

Specifies the path name of the directory in which the IBM HPC Toolkit is installed (`/opt/ibmhpc/ppdev.hpct`).

Example

```
#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 - Get the elapsed time

Use **MT_get_time** to get the elapsed time, in seconds, since **MPI_Init()** was called.

Library

-lmpitrace

C syntax

```
#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.

Example

```
#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 - Obtain information about MPI trace buffer usage

Use **MT_get_tracebufferinfo** to obtain information about MPI trace buffer usage by the MPI trace library.

Library

-lmpitrace

C syntax

```
#include <mpt.h>
int MT_get_tracebufferinfo(struct MT_tracebufferstruct *info);
```

Parameters

info A pointer to an **MT_tracebufferstruct** structure allocated by the user, which also 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

The number of MPI trace events currently recorded in this buffer.

double total_buffer

The MPI trace buffer size, in megabytes.

double used_buffer

The amount of trace buffer used, in megabytes.

double free_buffer

The 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.

Example

```
#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 - Generate performance statistics

Use **MT_output_text** to generate the performance statistics for your application when your application calls **MPI_Finalize()**.

Library

-lmpitrace

C syntax

```
#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.

Example

```
#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 - Control whether an MPI trace file is created

Use **MT_output_trace** to control whether an MPI trace file is created for a specific task.

Library

-lmpitrace

C syntax

```
#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

- 0** If an MPI trace file is not to be generated for the MPI task from which this function is called.
- 1** If an MPI trace file is to be generated for the MPI task from which this function is called.

Environment variables

None.

Example

```
#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 - Control whether an MPI trace event is generated

Use **MT_trace_event** to control whether an MPI trace event is generated for a specific MPI function call

Library

-lmpitrace

C syntax

```
#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 the collection of MPI trace events for any function with an identifier as listed in Table 20:

Table 20. Controlling the collection of MPI trace events for any function with an identifier

Function	Function	Function
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_DETACH_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	ALLTOALLW_ID	ALLTOALLV_ID
ACCUMULATE_ID	GET_ID	PUT_ID
WIN_CREATE_ID	WIN_COMPLETE_ID	WIN_FENCE_ID
WIN_LOCK_ID	WIN_POST_ID	WIN_TEST_ID
WIN_UNLOCK_ID	WIN_WAIT_ID	COMPARE_AND_SWAP_ID

Table 20. Controlling the collection of MPI trace events for any function with an identifier (continued)

Function	Function	Function
FETCH_AND_OP_ID	GET_ACCUMULATE_ID	IALLGATHER_ID
IALLGATHERV_ID	IALLTOALL_ID	IALLTOALLV_ID
IALLTOALLW_ID	IBARRIER_ID	IBCAST_ID
IEXSCAN_ID	IGATHER_ID	IMPROBE_ID
INEIGHBOR_ALLGATHER_ID	INEIGHBOR_ALLGATHERV_ID	INEIGHBOR_ALLTOALL_ID
INEIGHBOR_ALLTOALLV_ID	INEIGHBOR_ALLTOALLW_ID	IREDUCE_ID
IALLREDUCE_ID	IREDUCE_SCATTER_ID	IREDUCE_SCATTER_BLOCK_ID
ISCAN_ID	ISCATTER_ID	ISCATTERV_ID
MPROBE_ID	NEIGHBOR_ALLGATHER_ID	NEIGHBOR_ALLGATHERV_ID
NEIGHBOR_ALLTOALL_ID	NEIGHBOR_ALLTOALLV_ID	NEIGHBOR_ALLTOALLW_ID
RACCUMULATE_ID	RGET_ID	RGET_ACCUMULATE_ID
RPUT_ID	WIN_ALLOCATE_ID	WIN_ALLOCATE_SHARED_ID
WIN_CREATE_DYNAMIC_ID	WIN_FLUSH_ID	WIN_FLUSH_ALL_ID
WIN_FLUSH_LOCAL_ID	WIN_FLUSH_LOCAL_ALL_ID	WIN_LOCK_ALL_ID
WIN_SYNC_ID	WIN_UNLOCK_ALL_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, mt_trace_start - Start or resume the collection of trace events” on page 192.

Example

```
#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 - Start or resume the collection of trace events

Use **MT_trace_start** or **mt_trace_start** to start or resume the collection of trace events for all MPI calls.

Library

-lmpitrace

C syntax

```
#include <mpt.h>
void MT_trace_start()
```

FORTTRAN syntax

```
subroutine mt_trace_start()
```

Parameters

None.

Description

MT_trace_start() is used to start the collection of MPI trace events or to resume the 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

Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

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 if **TRACE_ALL_TASKS** is set to **no**. The default is **256**.

MT_BASIC_TRACE

Specifies whether the **MAX_TRACE_RANK** environment variable is checked. If **MT_BASIC_TRACE** is set to **yes**, then **MAX_TRACE_RANK** is ignored and the trace is generated with less overhead. If **MT_BASIC_TRACE** is not set, then the setting of **MAX_TRACE_RANK** is honored.

OUTPUT_ALL_RANKS

Used to control which tasks will generate trace output files. Set to **yes** to generate trace files for all the enabled tasks specified by **TRACE_ALL_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

Used to control when the collection of trace events starts and stops. Set to **yes** to enable collection of tracing 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

Used to control how many tasks will be included in the tracing. Set to **yes** to enable MPI trace generation for all MPI tasks in the application. The default is no, which results in enabling trace generation for only the first **MAX_TRACE_RANK+1** tasks (task 0 through **MAX_TRACE_RANK**).

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.

Example

```
#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 - Suspend the collection of trace events

Use **MT_trace_stop** or **mt_trace_stop** to suspend the collection of trace events for all MPI calls.

Library

-lmpitrace

C syntax

```
#include <mpt.h>
void MT_trace_stop()
```

FORTRAN syntax

```
subroutine mt_trace_stop()
```

Parameters

None.

Description

MT_trace_stop() is used to suspend the 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, mt_trace_start - Start or resume the collection of trace events” on page 192.

Example

```
#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
```

Part 8. Appendixes

Appendix A. Performance data file naming

The names of the performance data files generated by the HPC Toolkit are dependent on the setting of the **HPC_OUTPUT_NAME** environment variable, the **HPC_UNIQUE_FILE_NAME** environment variable, whether the application is a serial program or a parallel MPI program, and whether MPI dynamic tasking is used.

File naming conventions

Table 21 shows the file naming conventions for various tools and their types. The value of the **HPC_OUTPUT_NAME** environment variable specifies the *base* performance data file name, or if the value contains one or more file path separators, it specifies both the *dir* and *base* components of the performance data file name. If **HPC_OUTPUT_NAME** is not set, then there is no *dir*, and the value of *base* is **hpct**. The first character of the value of the **HPC_UNIQUE_FILE_NAME** environment variable specifies **Y**, **y**, or **1** to enable unique file naming, otherwise it is disabled.

Table 21. File naming conventions for various tools and their types

Tool	File naming	
HPM	Type	Name
	ascii	[dir[_processId]/]base[_processId][_worldId_rankId].hpm[.subtype].txt
	viz	[dir[_processId]/]base[_processId][_worldId_rankId].hpm[.subtype].viz
MPI	Type	Name
	ascii	[dir[_processId]/]base[_processId][_worldId_rankId].mpi.txt
	viz	[dir[_processId]/]base[_processId][_worldId_rankId].mpi.viz
	trace	[dir[_processId]/]base[_processId][_worldId_rankId].mpi.mpt
OpenMP	Type	Name
	ascii	[dir[_processId]/]base[_processId][_worldId_rankId].omp.txt
	viz	[dir[_processId]/]base[_processId][_worldId_rankId].omp.viz
MIO	Type	Name
	stats	[dir[_processId]/]base[_processId][_worldId_rankId].mio[.subtype].txt
	viz	[dir[_processId]/]base[_processId][_worldId_rankId].mio[.subtype].viz
	event	[dir[_processId]/]base[_processId][_worldId_rankId].mio[.subtype].iot
Profiling	For applications compiled without -hpcprof and not run under the hpcrun command, the gmon.out file name follows the conventions of that environment. For example, Linux execution environment outputs a gmon.out file (unless the GPROF environment variable defines a different prefix than gmon), and for a IBM Parallel Environment (PE) Runtime Edition execution environment see the topic, "Profiling programs with the gprof commands" in <i>Parallel Environment Runtime Edition: Operation and Use</i> for the gmon output file naming convention.	

where:

dir

Is the optional directory path portion of the specified file name from the **HPC_OUTPUT_NAME** environment variable. The *dir* is appended with the process ID, *_processId*, only if the **HPC_UNIQUE_FILE_NAME** environment variable is set.

base

Is the base name portion of the specified file name from the **HPC_OUTPUT_NAME** environment variable. The base is appended with the process ID, *_processId*, only if the **HPC_UNIQUE_FILE_NAME** environment variable is set and if a directory name is not present. The base is further appended with the world ID and rank ID, *_worldId_rankId*, only for a MPI parallel execution. In other words, the world ID and rank ID are not appended for a serial execution.

processId

Is the optional process ID. It is the **getpid()** for a serial execution or the poe process ID for an MPI parallel execution.

worldId

Is the optional world ID. For MPI without dynamic tasking it is zero (0). For MPI with dynamic tasking it is the MPI world ID.

rankId

Is the optional rank ID. For MPI without dynamic tasking it is the rank ID. For MPI with dynamic tasking it is the rank ID within the world.

tool

Is the tool name. The various tools are: **hpm**, **mpi**, **omp**, **mio** and **gmon**.

subtype

Is the optional subtype. The subtype is appended only if one is provided or needed.

- For the HPM tool, the subtype depends on whether the **hpccount** command, the **hpcstat** command, **libhpc** library, or a HPM instrumented binary is used:
 - If the **hpccount** command is used, then the invoked executable's base name is the subtype.
 - If the **hpcstat** command is used, then **hpcstat** is the subtype.
 - If **libhpc** is used, then the *progName* argument of **hpmInit** is the subtype, unless the *progName* argument is NULL then the subtype is not defined.
 - If the HPM instrumented binary is used, then the binary name is the subtype.
- For the MIO tool, the subtype depends on the **MIO_FILES** and **MIO_DEFAULTS** environment variables. The subtype is defined only if the **MIO_FILES** or **MIO_DEFAULTS** environment variables specify an entry for the trace module with the setting for its **stats={subtype}**, **xml={subtype}** or **events={subtype}** keywords respectively.

type

Is the type of file, which is also known as the file extension. The various types are: **txt**, **viz**, **mpt**, **iot**, and **out**.

File naming examples

The file naming conventions are shown in the following examples. For MPI applications, these examples assume **MP_PROCS** is set to 5.

- For a serial program, named *app*, that is compiled, instrumented for HPM analysis, and executed with **HPC_OUTPUT_NAME** set to *sample*, and **HPC_UNIQUE_FILE_NAME** not set:
 - A hardware counter ASCII data output file is named:
sample.hpm.app.txt
 - A hardware counter visualization data output file is named:

sample.hpm.app.viz

- For a serial program that is compiled, instrumented for HPM analysis, and executed with **HPC_OUTPUT_NAME** set to *output/sample*, and **HPC_UNIQUE_FILE_NAME** not set:
 - A hardware counter ASCII data output file is named:
output/sample.hpm.app.txt
 - A hardware counter visualization data output file is named:
output/sample.hpm.app.viz
- For a serial program, named *app*, that is compiled, instrumented for HPM analysis, and executed with **HPC_OUTPUT_NAME** set to *sample*, **HPC_UNIQUE_FILE_NAME** set to *y*, and the process ID is 92878:
 - A hardware counter ASCII data output file is named:
sample_92878.hpm.app.txt
 - A hardware counter visualization data output file is named:
sample_92878.hpm.app.viz
- For a serial program, named *app*, that is compiled, instrumented for HPM analysis, and executed with **HPC_OUTPUT_NAME** set to *output/sample*, **HPC_UNIQUE_FILE_NAME** set to *y*, and the process ID is 185903:
 - A hardware counter ASCII data output file is named:
output_185903/sample.hpm.app.txt
 - A hardware counter visualization data output file is named:
output_185903/sample.hpm.app.viz
- For an MPI application, named *app*, that is compiled, instrumented for HPM analysis, and executed without using dynamic tasking and with **HPC_OUTPUT_NAME** set to *sample* and **HPC_UNIQUE_FILE_NAME** not set:
 - Hardware counter ASCII data output files are named:
sample_0_0.hpm.app.txt
sample_0_1.hpm.app.txt
sample_0_2.hpm.app.txt
sample_0_3.hpm.app.txt
sample_0_4.hpm.app.txt
 - Hardware counter visualization data output files are named:
sample_0_0.hpm.app.viz
sample_0_1.hpm.app.viz
sample_0_2.hpm.app.viz
sample_0_3.hpm.app.viz
sample_0_4.hpm.app.viz
- For an MPI application, named *app*, that is compiled, instrumented for HPM analysis, and executed without using dynamic tasking and with **HPC_OUTPUT_NAME** set to *sample*, **HPC_UNIQUE_FILE_NAME** set to *y*, and the process ID is 14150:
 - Hardware counter ASCII data output files are named:
sample_14150_0_0.hpm.app.txt
sample_14150_0_1.hpm.app.txt
sample_14150_0_2.hpm.app.txt
sample_14150_0_3.hpm.app.txt
sample_14150_0_4.hpm.app.txt
 - Hardware counter visualization data output files are named:
sample_14150_0_0.hpm.app.viz
sample_14150_0_1.hpm.app.viz
sample_14150_0_2.hpm.app.viz
sample_14150_0_3.hpm.app.viz
sample_14150_0_4.hpm.app.viz

- For an MPI application, named *app*, that is compiled, instrumented for HPM analysis, and executed with using dynamic tasking and with **HPC_OUTPUT_NAME** set to *sample*, **HPC_UNIQUE_FILE_NAME** not set, and the world ID is 7:
 - Hardware counter ASCII data output files are named:


```
sample_7_0.hpm.app.txt
sample_7_1.hpm.app.txt
sample_7_2.hpm.app.txt
sample_7_3.hpm.app.txt
sample_7_4.hpm.app.txt
```
 - Hardware counter visualization data output files are named:


```
sample_7_0.hpm.app.viz
sample_7_1.hpm.app.viz
sample_7_2.hpm.app.viz
sample_7_3.hpm.app.viz
sample_7_4.hpm.app.viz
```
- For an MPI application, named *app*, that is compiled, instrumented for HPM analysis, and executed with using dynamic tasking and with **HPC_OUTPUT_NAME** set to *sample*, **HPC_UNIQUE_FILE_NAME** set to *y*, the world ID is 7, and the process ID is 78727:
 - Hardware counter ASCII data output files are named:


```
sample_78727_7_0.hpm.app.txt
sample_78727_7_1.hpm.app.txt
sample_78727_7_2.hpm.app.txt
sample_78727_7_3.hpm.app.txt
sample_78727_7_4.hpm.app.txt
```
 - Hardware counter visualization data output files are named:


```
sample_78727_7_0.hpm.app.viz
sample_78727_7_1.hpm.app.viz
sample_78727_7_2.hpm.app.viz
sample_78727_7_3.hpm.app.viz
sample_78727_7_4.hpm.app.viz
```
- For an MPI application that is compiled, linked with the MPI tracing library to trace all MPI calls, and executed with **HPC_OUTPUT_NAME** and **HPC_UNIQUE_FILE_NAME** not set, the following output file names are generated. In this case, only output from ranks 0, 1, and 4 was generated: one for task 0, which also has the median MPI communications time, one for the task with the minimum MPI communications time, and one for the task with the maximum MP communications.
 - MPI ASCII data output files named:


```
hpct_0_0.mpi.txt
hpct_0_1.mpi.txt
hpct_0_4.mpi.txt
```
 - MPI visualization data output files named:


```
hpct_0_0.mpi.viz
hpct_0_1.mpi.viz
hpct_0_4.mpi.viz
```
 - A single MPI trace data output file named:


```
hpct_0_0.mpi.mpt
```

Note: For the same MPI application if **HPC_UNIQUE_FILE_NAME** is set to *y*, and the process ID is 31052, then each output file name that is generated contains the process ID (*processId*).

- MPI ASCII data output files named:

`hpct_31052_0_0.mpi.txt`
`hpct_31052_0_1.mpi.txt`
`hpct_31052_0_4.mpi.txt`

- MPI visualization data output files named:

`hpct_31052_0_0.mpi.viz`
`hpct_31052_0_1.mpi.viz`
`hpct_31052_0_4.mpi.viz`

- A single MPI trace data output file named:

`hpct_31052_0_0.mpi.mpt`

Appendix B. Derived metrics, events, and groups supported on POWER8 architecture

The following topics provide information about:

- “Derived metrics defined for POWER8 architecture”
- “Events and groups supported on POWER8 architecture” on page 204

Derived metrics defined for POWER8 architecture

There are 31 derived metrics that are defined for POWER8 architecture as described in Table 22:

Table 22. Derived metrics defined for POWER8 architecture

Number	Derived metric	Description
0	Utilization rate	$\text{Utilization rate} = 100.0 * \text{user_time} / \text{wall_clock_time}$
1	MIPS	$\text{MIPS} = \text{PM_INST_CMPL} * 0.000001 / \text{wall_clock_time}$
2	Instructions per cycle	$\text{Instructions per cycle} = (\text{double})\text{PM_INST_CMPL} / \text{PM_CYC}$
3	Instructions per run cycle	$\text{Instructions per run cycle} = (\text{double})\text{PM_RUN_INST_CMPL} / \text{PM_RUN_CYC}$
4	Percentage Instructions dispatched that completed	$\text{Percentage Instructions dispatched that completed} = 100.0 * \text{PM_INST_CMPL} / \text{PM_INST_DISP}$
5	Branches mispredicted percentage	$\text{Branches mispredicted percentage} = \text{PM_BR_MPRED_CMPL} / \text{PM_RUN_INST_CMPL} * 100$
6	Total Loads from local L2	$\text{Total Loads from local L2} = \text{tot_ld_l_L2} = (\text{double})\text{PM_DATA_FROM_L2} / (1024*1024)$
7	Local L2 load traffic	$\text{Local L2 load traffic} = \text{L1_cache_line_size} * \text{tot_ld_l_L2}$
8	Local L2 load bandwidth per processor	$\text{Local L2 load bandwidth per processor} = \text{L1_cache_line_size} * \text{tot_ld_l_L2} / \text{wall_clock_time}$
9	Percentage loads from local L2 per cycle	$\text{Percentage loads from local L2 per cycle} = 100.0 * (\text{double})\text{PM_DATA_FROM_L2} / \text{PM_RUN_CYC}$
10	Total Loads from local L3	$\text{Total Loads from local L3} = \text{tot_ld_l_L3} = (\text{double})\text{PM_DATA_FROM_L3} / (1024*1024)$
11	Local L3 load traffic	$\text{Local L3 load traffic} = \text{L2_cache_line_size} * \text{tot_ld_l_L3}$
12	Local L3 load bandwidth per processor	$\text{Local L3 load bandwidth per processor} = \text{L2_cache_line_size} * \text{tot_ld_l_L3} / \text{wall_clock_time}$
13	Percentage loads from local L3 per cycle	$\text{Percentage loads from local L3 per cycle} = 100.0 * (\text{double})\text{PM_DATA_FROM_L3} / \text{PM_RUN_CYC}$
14	Total Loads from memory	$\text{Total Loads from memory} = \text{tot_ld_mem} = (\text{PM_DATA_FROM_RMEM} + \text{PM_DATA_FROM_LMEM}) / (1024*1024)$
15	Memory load traffic	$\text{Memory load traffic} = \text{L3_cache_line_size} * \text{tot_ld_mem}$
16	Memory load bandwidth per processor	$\text{Memory load bandwidth per processor} = \text{L3_cache_line_size} * \text{tot_ld_mem} / \text{wall_clock_time}$
17	Total Loads from local memory	$\text{Total Loads from local memory} = \text{tot_ld_lmem} = (\text{double})\text{PM_DATA_FROM_LMEM} / (1024*1024)$
18	Local memory load traffic	$\text{Local memory load traffic} = \text{L3_cache_line_size} * \text{tot_ld_lmem}$

Table 22. Derived metrics defined for POWER8 architecture (continued)

Number	Derived metric	Description
19	Local memory load bandwidth per processor	Local memory load bandwidth per processor = $L3_cache_line_size * tot_ld_lmem / wall_clock_time$
20	CPU Utilization	CPU Utilization = $PM_RUN_CYC / PM_CYC * 100$
21	Instruction cache miss rate (per run instruction)	Instruction cache miss rate (per run instruction) = $PM_L1_ICACHE_MISS / PM_RUN_INST_CMPL * 100$
22	Completion Stall Cycles	Completion Stall Cycles = $PM_CMPLU_STALL / PM_RUN_INST_CMPL$
23	Cycles stalled by FXU	Cycles stalled by FXU = $PM_CMPLU_STALL_FXU / PM_RUN_INST_CMPL$
24	Cycles stalled by VSU Scalar Operations	Cycles stalled by VSU Scalar Operations = $PM_CMPLU_STALL_SCALAR / PM_RUN_INST_CMPL$
25	Cycles stalled by VSU Scalar Long Operations	Cycles stalled by VSU Scalar Long Operations = $PM_CMPLU_STALL_SCALAR_LONG / PM_RUN_INST_CMPL$
26	Cycles stalled by VSU Vector Operations	Cycles stalled by VSU Vector Operations = $PM_CMPLU_STALL_VECTOR / PM_RUN_INST_CMPL$
27	Cycles stalled by LSU	Cycles stalled by LSU = $PM_CMPLU_STALL_LSU / PM_RUN_INST_CMPL$
28	Cycles stalled by LSU Rejects	Cycles stalled by LSU Rejects = $PM_CMPLU_STALL_REJECT / PM_RUN_INST_CMPL$
29	Cycles stalled by ERAT Translation rejects	Cycles stalled by ERAT Translation rejects = $PM_CMPLU_STALL_ERAT_MISS / PM_RUN_INST_CMPL$
30	Cycles stalled by D-Cache Misses	Cycles stalled by D-Cache Misses = $PM_CMPLU_STALL_DCACHE_MISS / PM_RUN_INST_CMPL$

Events and groups supported on POWER8 architecture

On Linux POWER8, the number of CPU groups supported is 267. The following tables list the supported event groups listed by group ID number.

Group 0

Event	Description
PM_CYC	Cycles
PM_RUN_CYC	Run cycles
PM_INST_DISP	PPC Dispatched
PM_INST_CMPL	Number of PowerPC® Instructions that completed. PPC Instructions Finished (completed).
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 1

Event	Description
PM_RUN_SPURR	Run SPURR
PM_RUN_CYC	Run cycles
PM_CYC	Cycles
PM_RUN_PURR	Run_PURR
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 2

Event	Description
PM_IOPS_CMPL	Internal Operations completed
PM_CYC	Cycles
PM_IOPS_DISP	Internal Operations dispatched
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 3

Event	Description
PM_RUN_CYC_ST_MODE	Cycles run latch is set and core is in ST mode
PM_RUN_CYC_SMT2_SHRD_MODE	Cycles this threads run latch is set and the core is in SMT2 shared mode
PM_RUN_CYC_SMT2_MODE	Cycles run latch is set and core is in SMT2 mode
PM_RUN_CYC_SMT8_MODE	Cycles run latch is set and core is in SMT8 mode
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 4

Event	Description
PM_RUN_CYC_SMT2_SPLIT_MODE	Cycles run latch is set and core is in SMT2 split mode
PM_RUN_CYC_SMT4_MODE	Cycles this threads run latch is set and the core is in SMT4 mode
PM_RUN_CYC_SMT2_MODE	Cycles run latch is set and core is in SMT2 mode
PM_RUN_CYC_SMT8_MODE	Cycles run latch is set and core is in SMT8 mode
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 5

Event	Description
PM_THRD_GRP_CMPL_BOTH_CYC	Cycles group completed on both completion slots by any thread
PM_THRD_ALL_RUN_CYC	All Threads in run_cycles (was both threads in run_cycles)
PM_THRD_CONC_RUN_INST	PPC Instructions Finished when both threads in run_cycles
PM_THRD_PRIO_0_1_CYC	Cycles thread running at priority level 0 or 1
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 6

Event	Description
PM_THRD_PRIO_0_1_CYC	Cycles thread running at priority level 0 or 1
PM_THRD_PRIO_2_3_CYC	Cycles thread running at priority level 2 or 3
PM_THRD_PRIO_4_5_CYC	Cycles thread running at priority level 4 or 5
PM_THRD_PRIO_6_7_CYC	Cycles thread running at priority level 6 or 7
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 7

Event	Description
PM_ANY_THRD_RUN_CYC	One of threads in run_cycles
PM_THRD_REBAL_CYC	Cycles rebalance was active
PM_NEST_REF_CLK	Multiply by 4 to obtain the number of PB cycles
PM_RUN_INST_CMPL	Run instructions
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 8

Event	Description
PM_BR_PRED_BR0	Conditional Branch Completed on BR0 (1st branch in group) in which the HW predicted the Direction or Target
PM_BR_PRED_BR1	Conditional Branch Completed on BR1 (2nd branch in group) in which the HW predicted the Direction or Target. Note: BR1 can only be used in Single Thread Mode. In all of the SMT modes, only one branch can complete, thus BR1 is unused.
PM_BR_UNCOND_BR0	Unconditional Branch Completed on BR0. HW branch prediction was not used for this branch. This can be an I form branch, a B form branch with BO field set to branch always, or a B form branch which was converted to a Resolve.
PM_BR_UNCOND_BR1	Unconditional Branch Completed on BR1. HW branch prediction was not used for this branch. This can be an I form branch, a B form branch with BO field set to branch always, or a B form branch which was converted to a Resolve.
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 9

Event	Description
PM_BR_PRED_CCACHE_BR0	Conditional Branch Completed on BR0 that used the Count Cache for Target Prediction
PM_BR_PRED_CCACHE_BR1	Conditional Branch Completed on BR1 that used the Count Cache for Target Prediction
PM_BR_PRED_LSTACK_BR0	Conditional Branch Completed on BR0 that used the Link Stack for Target Prediction
PM_BR_PRED_LSTACK_BR1	Conditional Branch Completed on BR1 that used the Link Stack for Target Prediction
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 10

Event	Description
PM_BR_PRED_CR_BR0	Conditional Branch Completed on BR0 that had its direction predicted. I form branches do not set this event. In addition, B form branches which do not use the BHT do not set this event these are branches with BO field set to 'always taken' and branches
PM_BR_PRED_CR_BR1	Conditional Branch Completed on BR1 that had its direction predicted. I form branches do not set this event. In addition, B form branches which do not use the BHT do not set this event these are branches with BO field set to 'always taken' and branches

Event	Description
PM_BR_PRED_TA_BR0	Conditional Branch Completed on BR0 that had its target address predicted. Only XL form branches set this event.
PM_BR_PRED_TA_BR1	Conditional Branch Completed on BR1 that had its target address predicted. Only XL form branches set this event.
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 11

Event	Description
PM_BRU_FIN	Branch Instruction Finished
PM_BR_TAKEN_CMPL	New event for Branch Taken
PM_BR_PRED_BR_CMPL	Completion Time Event. This event can also be calculated from the direct bus as follows: if_pc_br0_br_pred(0) OR if_pc_br0_br_pred(1).
PM_BR_CMPL	Branch Instruction completed
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 12

Event	Description
PM_BR_BC_8_CONV	Pairable BC+8 branch that was converted to a Resolve Finished in the BRU pipeline.
PM_BR_BC_8	Pairable BC+8 branch that has not been converted to a Resolve Finished in the BRU pipeline
PM_BR_UNCOND_BR0	Unconditional Branch Completed on BR0. HW branch prediction was not used for this branch. This can be an I form branch, a B form branch with BO field set to branch always, or a B form branch which was converted to a Resolve.
PM_BR_2PATH	Two path branch
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 13

Event	Description
PM_BR_MPRED_LSTACK	Conditional Branch Completed that was Mispredicted due to the Link Stack Target Prediction
PM_BR_MPRED_CCACHE	Conditional Branch Completed that was Mispredicted due to the Count Cache Target Prediction
PM_BR_MPRED_CR	Conditional Branch Completed that was Mispredicted due to the BHT Direction Prediction (taken/not taken).
PM_BR_MPRED_TA	Conditional Branch Completed that was Mispredicted due to the Target Address Prediction from the Count Cache or Link Stack. Only XL form branches that resolved Taken set this event.
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 14

Event	Description
PM_FLUSH_DISP_SYNC	Dispatch flush: Sync
PM_FLUSH_BR_MPRED	Flush caused by branch mispredict

Event	Description
PM_FLUSH_DISP_SB	Dispatch Flush: Scoreboard
PM_FLUSH	Flush (any type)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 15

Event	Description
PM_FLUSH_DISP	Dispatch flush
PM_FLUSH_PARTIAL	Partial flush
PM_FLUSH_COMPLETION	Completion Flush
PM_BR_MPRED_CMPL	Number of Branch Mispredicts
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 16

Event	Description
PM_DTLB_MISS_16G	Data TLB Misses, page size 16G
PM_DTLB_MISS_4K	Data TLB Misses, page size 4k
PM_DTLB_MISS_64K	Data TLB Miss page size 64K
PM_DTLB_MISS_16M	Data TLB Miss page size 16M
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 17

Event	Description
PM_DERAT_MISS_4K	Data ERAT Miss (Data TLB Access) page size 4K
PM_DERAT_MISS_64K	Data ERAT Miss (Data TLB Access) page size 64K
PM_DERAT_MISS_16M	Data ERAT Miss (Data TLB Access) page size 16M
PM_DERAT_MISS_16G	Data ERAT Miss (Data TLB Access) page size 16G
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 18

Event	Description
PM_FLUSH_DISP_SYNC	Dispatch flush: Sync
PM_FLUSH_DISP_TLBIE	Dispatch Flush: TLBIE
PM_FLUSH_DISP_SB	Dispatch Flush: Scoreboard
PM_FLUSH	Flush (any type)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 19

Event	Description
PM_FLUSH_DISP	Dispatch flush
PM_CYC	Cycles
PM_FLUSH_COMPLETION	Completion Flush

Event	Description
PM_FLUSH	Flush (any type)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 20

Event	Description
PM_FXU_IDLE	FXU0 idle and FXU1 idle
PM_FXU_BUSY	FXU0 busy and FXU1 busy.
PM_FXU0_BUSY_FXU1_IDLE	FXU0 busy and FXU1 idle
PM_FXU1_BUSY_FXU0_IDLE	FXU0 idle and FXU1 busy.
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 21

Event	Description
PM_FXU0_FIN	The fixed point unit Unit 0 finished an instruction. Instructions that finish may not necessary complete.
PM_RUN_CYC	Run cycles
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_FXU1_FIN	FXU1 finished
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 22

Event	Description
PM_CYC	Cycles
PM_FXU_BUSY	FXU0 busy and FXU1 busy.
PM_FXU0_BUSY_FXU1_IDLE	FXU0 busy and FXU1 idle
PM_FXU1_BUSY_FXU0_IDLE	FXU0 idle and FXU1 busy.
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 23

Event	Description
PM_FXU_IDLE	FXU0 idle and FXU1 idle
PM_FXU_BUSY	FXU0 busy and FXU1 busy.
PM_CYC	Cycles
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 24

Event	Description
PM_DISP_CLB_HELD_BAL	Dispatch/CLB Hold: Balance
PM_DISP_CLB_HELD_RES	Dispatch/CLB Hold: Resource

Event	Description
PM_DISP_CLB_HELD_TLBIE	Dispatch Hold: Due to TLBIE
PM_DISP_CLB_HELD_SYNC	Dispatch/CLB Hold: Sync type instruction
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 25

Event	Description
PM_DISP_CLB_HELD_SB	Dispatch/CLB Hold: Scoreboard
PM_DISP_HELD_IQ_FULL	Dispatch held due to Issue q full
PM_DISP_HELD_SRQ_FULL	Dispatch held due SRQ no room
PM_DISP_HELD_SYNC_HOLD	Dispatch held due to SYNC hold
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 26

Event	Description
PM_DISP_HELD_MAP_FULL	Dispatch for this thread was held because the Mappers were full
PM_INST_DISP	PPC Dispatched
PM_GRP_DISP	Group dispatch
PM_IPLUS_PPC_DISP	Cycles at least one Instr Dispatched
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 27

Event	Description
PM_IPLUS_PPC_CMPL	1 or more ppc instructions finished
PM_NTCG_ALL_FIN	Cycles after all instructions have finished to group completed
PM_GRP_CMPL	Group completed
PM_CYC	Cycles
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 28

Event	Description
PM_CMPLU_STALL	Completion stall
PM_CMPLU_STALL_FXU	Completion stall due to FXU
PM_CMPLU_STALL_FLUSH	Completion stall due to flush by own thread
PM_SHL_ST_DISABLE	Store Hit Load Table Read Hit with entry Disabled (entry was disabled due to the entry shown to not prevent the flush)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 29

Event	Description
PM_CMPLU_STALL_THRD	Completion Stalled due to thread conflict. Group ready to complete but it was another thread's turn
PM_CMPLU_STALL_BRU_CRU	Completion stall due to IFU
PM_CMPLU_STALL_COQ_FULL	Completion stall due to CO q full
PM_CMPLU_STALL_BRU	Completion stall due to a Branch Unit
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 30

Event	Description
PM_DATA_FROM_L2	The processor's data cache was reloaded from local core's L2 due to a demand load
PM_CMPLU_STALL_DCACHE_MISS	Completion stall by Dcache miss
PM_CMPLU_STALL_HWSYNC	Completion stall due to hwsync
PM_CMPLU_STALL_DMISS_L2L3_CONFLICT	Completion stall due to cache miss that resolves in the L2 or L3 with a conflict
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 31

Event	Description
PM_DATA_FROM_L3_NO_CONFLICT	The processor's data cache was reloaded from local core's L3 without conflict due to a demand load
PM_CMPLU_STALL_DMISS_L2L3	Completion stall by Dcache miss which resolved on chip (excluding local L2/L3)
PM_CMPLU_STALL_MEM_ECC_DELAY	Completion stall due to memory ECC delay
PM_CMPLU_STALL_DMISS_L3MISS	Completion stall due to cache miss resolving missed the L3
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 32

Event	Description
PM_DISP_HELD	Dispatch Held
PM_CMPLU_STALL_DMISS_L2L3	Completion stall by Dcache miss which resolved in L2/L3
PM_CMPLU_STALL_OTHER_CMPL	Instructions core completed while this tread was stalled
PM_CMPLU_STALL_DMISS_LMEM	Completion stall due to cache miss that resolves in local memory
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 33

Event	Description
PM_FLOP	Floating Point Operation Finished
PM_CMPLU_STALL_DMISS_REMOTE	Completion stall by Dcache miss which resolved from remote chip (cache or memory)
PM_DISP_WT	Dispatched Starved
PM_CMPLU_STALL_ERAT_MISS	Completion stall due to LSU reject ERAT miss
PM_RUN_INST_CMPL	Run instructions

Event	Description
PM_RUN_CYC	Run cycles

Group 34

Event	Description
PM_GCT_NOSLOT_CYC	Pipeline empty (No itags assigned , no GCT slots used)
PM_CMPLU_STALL_LSU	Completion stall by LSU instruction
PM_FXU0_BUSY_FXU1_IDLE	FXU0 busy and FXU1 idle
PM_CMPLU_STALL_FXLONG	Completion stall due to a long latency fixed point instruction
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 35

Event	Description
PM_DATA_ALL_PUMP_CPRED	Pump prediction correct. Counts across all types of pumps for either demand loads or data prefetch
PM_CMPLU_STALL_NTCG_FLUSH	Completion stall due to ntcg flush
PM_L3_CO_MEPF	L3 castouts in Mepf state
PM_CMPLU_STALL_LOAD_FINISH	Completion stall due to a Load finish
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 36

Event	Description
PM_IC_DEMAND_CYC	Cycles when a demand ifetch was pending
PM_CMPLU_STALL_REJECT_LHS	Completion stall due to reject (load hit store)
PM_L3_SW_PREF	Data stream touch to L3
PM_CMPLU_STALL_REJECT	Completion stall due to LSU reject
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 37

Event	Description
PM_L1_DCACHE_RELOADED_ALL	L1 data cache reloaded for demand or prefetch
PM_CMPLU_STALL_SCALAR_LONG	Completion stall due to VSU scalar long latency instruction
PM_LSU_LMQ_SRQ_EMPTY_ALL_CYC	ALL threads lsu empty (lmq and srq empty)
PM_CMPLU_STALL_REJ_LMQ_FULL	Completion stall due to LSU reject LMQ full
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 38

Event	Description
PM_L2_TM_REQ_ABORT	TM abort
PM_CMPLU_STALL_STORE	Completion stall by stores this includes store agen finishes in pipe LS0/LS1 and store data finishes in LS2/LS3

Event	Description
PM_MRK_STALL_CMPLU_CYC	Marked Group completion Stall
PM_CMPLU_STALL_SCALAR	Completion stall due to VSU scalar instruction
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 39

Event	Description
PM_L3_CO_MEPF	L3 castouts in Mepf state
PM_CMPLU_STALL_VECTOR	Completion stall due to VSU vector instruction
PM_MRK_ST_CMPL	Marked store completed and sent to nest
PM_CMPLU_STALL_ST_FWD	Completion stall due to store forward
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 40

Event	Description
PM_L3_LD_PREF	L3 Load Prefetches
PM_CMPLU_STALL_VSU	Completion stall due to VSU instruction
PM_ST_MISS_L1	Store Missed L1
PM_CMPLU_STALL_VECTOR_LONG	Completion stall due to VSU vector long instruction
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 41

Event	Description
PM_LSU0_REJECT	LSU0 reject
PM_GCT_NOSLOT_DISP_HELD_ISSQ	Gct empty for this thread due to dispatch hold on this thread due to Issue q full
PM_IFU_L2_TOUCH	L2 touch to update MRU on a line
PM_GCT_NOSLOT_BR_MPRED	Gct empty for this thread due to branch mispredict
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 42

Event	Description
PM_LSU_REJECT_LMQ_FULL	LSU reject due to LMQ full (4 per cycle)
PM_GCT_NOSLOT_DISP_HELD_OTHER	Gct empty for this thread due to dispatch hold on this thread due to sync
PM_LSU_FIN	LSU Finished an instruction (up to 2 per cycle)
PM_GCT_NOSLOT_BR_MPRED_ICMISS	Gct empty for this thread due to Icache Miss and branch mispredict
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 43

Event	Description
PM_DATA_PUMP_CPRED	Pump prediction correct. Counts across all types of pumps for a demand load
PM_GCT_NOSLOT_DISP_HELD_SRQ	Gct empty for this thread due to dispatch hold on this thread due to SRQ full
PM_FLUSH_LSU	Flush initiated by LSU
PM_GCT_NOSLOT_DISP_HELD_MAP	Gct empty for this thread due to dispatch hold on this thread due to Mapper full
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 44

Event	Description
PM_MRK_ST_CMPL	Marked store completed and sent to nest
PM_GCT_NOSLOT_IC_MISS	Gct empty for this thread due to Icache Miss
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_GCT_NOSLOT_IC_L3MISS	Gct empty for this thread due to icach L3 miss
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 45

Event	Description
PM_MEM_LOC_THRESH_IFU	Local Memory above threshold for IFU speculation control
PM_CMPLU_STALL_NO_NTF	Completion stall due to nop
PM_L1_DCACHE_RELOAD_VALID	DL1 reloaded due to Demand Load
PM_DATA_FROM_OFF_CHIP_CACHE	The processor's data cache was reloaded either shared or modified data from another core's L2/L3 on a different chip (remote or distant) due to a demand load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 46

Event	Description
PM_CMPLU_STALL_LWSYNC	Completion stall due to isync/lwsync
PM_MEM_PREF	Memory prefetch for this lpar. Includes L4
PM_UP_PREF_L3	Micropartition prefetch
PM_UP_PREF_POINTER	Micropartition pointer prefetches
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 47

Event	Description
PM_DC_PREF_STREAM_ALLOC	Stream marked valid. The stream could have been allocated through the hardware prefetch mechanism or through software. This is combined LS0 and LS1
PM_DC_PREF_STREAM_CONF	A demand load referenced a line in an active prefetch stream. The stream could have been allocated through the hardware prefetch mechanism or through software. Combine up + down

Event	Description
PM_DC_PREF_STREAM_STRIDED_CONF	A demand load referenced a line in an active strided prefetch stream. The stream could have been allocated through the hardware prefetch mechanism or through software.
PM_DC_PREF_STREAM_FUZZY_CONF	A demand load referenced a line in an active fuzzy prefetch stream. The stream could have been allocated through the hardware prefetch mechanism or through software. Fuzzy stream confirm (out of order effects, or pf cant keep up)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 48

Event	Description
PM_LD_CMPL	Count of Loads completed
PM_L3_ST_PREF	L3 store Prefetches
PM_L3_SW_PREF	Data stream touch to L3
PM_L3_PREF_ALL	Total HW L3 prefetches(Load+store)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 49

Event	Description
PM_DATA_FROM_L2	The processor's data cache was reloaded from local core's L2 due to a demand load
PM_DATA_FROM_L2MISS	Demand LD L2 Miss (not L2 hit)
PM_DATA_FROM_L3MISS	Demand LD L3 Miss (not L2 hit and not L3 hit)
PM_DATA_FROM_L3	The processor's data cache was reloaded from local core's L3 due to a demand load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 50

Event	Description
PM_DATA_FROM_L2_NO_CONFLICT	The processor's data cache was reloaded from local core's L2 without conflict due to a demand load
PM_DATA_FROM_L2_MEPF	The processor's data cache was reloaded from local core's L2 hit without dispatch conflicts on Mepf state due to a demand load
PM_DATA_FROM_L2_DISP_CONFLICT_LDHITST	The processor's data cache was reloaded from local core's L2 with load hit store conflict due to a demand load
PM_DATA_FROM_L2_DISP_CONFLICT_OTHER	The processor's data cache was reloaded from local core's L2 with dispatch conflict due to a demand load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 51

Event	Description
PM_DATA_FROM_L3_NO_CONFLICT	The processor's data cache was reloaded from local core's L3 without conflict due to a demand load

Event	Description
PM_DATA_FROM_L3_MEPF	The processor's data cache was reloaded from local core's L3 without dispatch conflicts hit on Mepf state due to a demand load
PM_DATA_FROM_L3_DISP_CONFLICT	The processor's data cache was reloaded from local core's L3 with dispatch conflict due to a demand load
PM_DATA_FROM_L3MISS_MOD	The processor's data cache was reloaded from a location other than the local core's L3 due to a demand load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 52

Event	Description
PM_DATA_FROM_L31_SHR	The processor's data cache was reloaded with Shared (S) data from another core's L3 on the same chip due to a demand load
PM_DATA_FROM_L31_MOD	The processor's data cache was reloaded with Modified (M) data from another core's L3 on the same chip due to a demand load
PM_DATA_FROM_L31_ECO_SHR	The processor's data cache was reloaded with Shared (S) data from another core's ECO L3 on the same chip due to a demand load
PM_DATA_FROM_L31_ECO_MOD	The processor's data cache was reloaded with Modified (M) data from another core's ECO L3 on the same chip due to a demand load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 53

Event	Description
PM_DATA_FROM_L2MISS_MOD	The processor's data cache was reloaded from a location other than the local core's L2 due to a demand load
PM_DATA_FROM_LMEM	The processor's data cache was reloaded from the local chip's Memory due to a demand load
PM_DATA_FROM_RMEM	The processor's data cache was reloaded from another chip's memory on the same Node or Group (Remote) due to a demand load
PM_DATA_FROM_DMEM	The processor's data cache was reloaded from another chip's memory on the same Node or Group (Distant) due to a demand load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 54

Event	Description
PM_DATA_FROM_ON_CHIP_CACHE	The processor's data cache was reloaded either shared or modified data from another core's L2/L3 on the same chip due to a demand load
PM_DATA_FROM_RL2L3_MOD	The processor's data cache was reloaded with Modified (M) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to a demand load
PM_DATA_FROM_L21_SHR	The processor's data cache was reloaded with Shared (S) data from another core's L2 on the same chip due to a demand load
PM_DATA_FROM_L21_MOD	The processor's data cache was reloaded with Modified (M) data from another core's L2 on the same chip due to a demand load

Event	Description
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 55

Event	Description
PM_DATA_FROM_LL4	The processor's data cache was reloaded from the local chip's L4 cache due to a demand load
PM_DATA_FROM_RL4	The processor's data cache was reloaded from another chip's L4 on the same Node or Group (Remote) due to a demand load
PM_DATA_FROM_DL4	The processor's data cache was reloaded from another chip's L4 on a different Node or Group (Distant) due to a demand load
PM_DATA_FROM_MEM	The processor's data cache was reloaded from a memory location including L4 from local remote or distant due to a demand load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 56

Event	Description
PM_DATA_FROM_RL2L3_SHR	The processor's data cache was reloaded with Shared (S) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to a demand load
PM_DATA_FROM_MEMORY	The processor's data cache was reloaded from a memory location including L4 from local remote or distant due to a demand load
PM_DATA_FROM_DL2L3_SHR	The processor's data cache was reloaded with Shared (S) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to a demand load
PM_DATA_FROM_DL2L3_MOD	The processor's data cache was reloaded with Modified (M) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to a demand load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 57

Event	Description
PM_DATA_ALL_FROM_L2	The processor's data cache was reloaded from local core's L2 due to either demand loads or data prefetch
PM_FLOP_SUM_SCALAR	Flops summary scalar instructions
PM_FLOP_SUM_VEC	Flops summary vector instructions
PM_DATA_ALL_FROM_L3	The processor's data cache was reloaded from local core's L3 due to either demand loads or data prefetch
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 58

Event	Description
PM_DATA_ALL_FROM_L2_NO_CONFLICT	The processor's data cache was reloaded from local core's L2 without conflict due to either demand loads or data prefetch

Event	Description
PM_DATA_ALL_FROM_L2_MEPF	The processor's data cache was reloaded from local core's L2 hit without dispatch conflicts on Mepf state due to either demand loads or data prefetch
PM_DATA_ALL_FROM_L2_DISP_CONFLICT_LDHITST	The processor's data cache was reloaded from local core's L2 with load hit store conflict due to either demand loads or data prefetch
PM_DATA_ALL_FROM_L2_DISP_CONFLICT_OTHER	The processor's data cache was reloaded from local core's L2 with dispatch conflict due to either demand loads or data prefetch
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 59

Event	Description
PM_DATA_ALL_FROM_L3_NO_CONFLICT	The processor's data cache was reloaded from local core's L3 without conflict due to either demand loads or data prefetch
PM_DATA_ALL_FROM_L3_MEPF	The processor's data cache was reloaded from local core's L3 without dispatch conflicts hit on Mepf state due to either demand loads or data prefetch
PM_DATA_ALL_FROM_L3_DISP_CONFLICT	The processor's data cache was reloaded from local core's L3 with dispatch conflict due to either demand loads or data prefetch
PM_DATA_ALL_FROM_L3MISS_MOD	The processor's data cache was reloaded from a location other than the local core's L3 due to either demand loads or data prefetch
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 60

Event	Description
PM_DATA_ALL_FROM_L31_SHR	The processor's data cache was reloaded with Shared (S) data from another core's L3 on the same chip due to either demand loads or data prefetch
PM_DATA_ALL_FROM_L31_MOD	The processor's data cache was reloaded with Modified (M) data from another core's L3 on the same chip due to either demand loads or data prefetch
PM_DATA_ALL_FROM_L31_ECO_SHR	The processor's data cache was reloaded with Shared (S) data from another core's ECO L3 on the same chip due to either demand loads or data prefetch
PM_DATA_ALL_FROM_L31_ECO_MOD	The processor's data cache was reloaded with Modified (M) data from another core's ECO L3 on the same chip due to either demand loads or data prefetch
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 61

Event	Description
PM_DATA_ALL_FROM_L2MISS_MOD	The processor's data cache was reloaded from a location other than the local core's L2 due to either demand loads or data prefetch
PM_DATA_ALL_FROM_LMEM	The processor's data cache was reloaded from the local chip's Memory due to either demand loads or data prefetch
PM_DATA_ALL_FROM_RMEM	The processor's data cache was reloaded from another chip's memory on the same Node or Group (Remote) due to either demand loads or data prefetch

Event	Description
PM_DATA_ALL_FROM_DMEM	The processor's data cache was reloaded from another chip's memory on the same Node or Group (Distant) due to either demand loads or data prefetch
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 62

Event	Description
PM_DATA_ALL_FROM_ON_CHIP_CACHE	The processor's data cache was reloaded either shared or modified data from another core's L2/L3 on the same chip due to either demand loads or data prefetch
PM_DATA_ALL_FROM_RL2L3_MOD	The processor's data cache was reloaded with Modified (M) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to either demand loads or data prefetch
PM_DATA_ALL_FROM_L21_SHR	The processor's data cache was reloaded with Shared (S) data from another core's L2 on the same chip due to either demand loads or data prefetch
PM_DATA_ALL_FROM_L21_MOD	The processor's data cache was reloaded with Modified (M) data from another core's L2 on the same chip due to either demand loads or data prefetch
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 63

Event	Description
PM_DATA_ALL_FROM_LL4	The processor's data cache was reloaded from the local chip's L4 cache due to either demand loads or data prefetch
PM_DATA_ALL_FROM_RL4	The processor's data cache was reloaded from another chip's L4 on the same Node or Group (Remote) due to either demand loads or data prefetch
PM_DATA_ALL_FROM_DL4	The processor's data cache was reloaded from another chip's L4 on a different Node or Group (Distant) due to either demand loads or data prefetch
PM_DATA_ALL_FROM_OFF_CHIP_CACHE	The processor's data cache was reloaded either shared or modified data from another core's L2/L3 on a different chip (remote or distant) due to either demand loads or data prefetch
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 64

Event	Description
PM_DATA_ALL_FROM_RL2L3_SHR	The processor's data cache was reloaded with Shared (S) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to either demand loads or data prefetch
PM_DATA_ALL_FROM_MEMORY	The processor's data cache was reloaded from a memory location including L4 from local remote or distant due to either demand loads or data prefetch
PM_DATA_ALL_FROM_DL2L3_SHR	The processor's data cache was reloaded with Shared (S) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to either demand loads or data prefetch

Event	Description
PM_DATA_ALL_FROM_DL2L3_MOD	The processor's data cache was reloaded with Modified (M) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to either demand loads or data prefetch
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 65

Event	Description
PM_INST_FROM_L2	The processor's Instruction cache was reloaded from local core's L2 due to an instruction fetch (not prefetch)
PM_INST_FROM_L2_MEPF	The processor's Instruction cache was reloaded from local core's L2 hit without dispatch conflicts on Mepf state. due to an instruction fetch (not prefetch)
PM_INST_FROM_L2_DISP_CONFLICT_LDHITST	The processor's Instruction cache was reloaded from local core's L2 with load hit store conflict due to an instruction fetch (not prefetch)
PM_INST_FROM_L2_DISP_CONFLICT_OTHER	The processor's Instruction cache was reloaded from local core's L2 with dispatch conflict due to an instruction fetch (not prefetch)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CY	Run cycles

Group 66

Event	Description
PM_INST_FROM_L2_NO_CONFLICT	The processor's Instruction cache was reloaded from local core's L2 without conflict due to an instruction fetch (not prefetch)
PM_INST_FROM_L31_MOD	The processor's Instruction cache was reloaded with Modified (M) data from another core's L3 on the same chip due to an instruction fetch (not prefetch)
PM_INST_FROM_L21_SHR	The processor's Instruction cache was reloaded with Shared (S) data from another core's L2 on the same chip due to an instruction fetch (not prefetch)
PM_INST_FROM_L21_MOD	The processor's Instruction cache was reloaded with Modified (M) data from another core's L2 on the same chip due to an instruction fetch (not prefetch)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 67

Event	Description
PM_INST_FROM_L3_NO_CONFLICT	The processor's Instruction cache was reloaded from local core's L3 without conflict due to an instruction fetch (not prefetch)
PM_INST_FROM_L3_MEPF	The processor's Instruction cache was reloaded from local core's L3 without dispatch conflicts hit on Mepf state. due to an instruction fetch (not prefetch)
PM_INST_FROM_L3_DISP_CONFLICT	The processor's Instruction cache was reloaded from local core's L3 with dispatch conflict due to an instruction fetch (not prefetch)
PM_INST_FROM_L3	The processor's Instruction cache was reloaded from local core's L3 due to an instruction fetch (not prefetch)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 68

Event	Description
PM_INST_FROM_L2MISS	The processor's Instruction cache was reloaded from a location other than the local core's L2 due to an instruction fetch (not prefetch)
PM_INST_FROM_MEMORY	The processor's Instruction cache was reloaded from a memory location including L4 from local remote or distant due to an instruction fetch (not prefetch)
PM_INST_FROM_L3MISS	Marked instruction was reloaded from a location beyond the local chiplet
PM_INST_FROM_L3MISS_MOD	The processor's Instruction cache was reloaded from a location other than the local core's L3 due to a instruction fetch
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 69

Event	Description
PM_INST_FROM_L31_SHR	The processor's Instruction cache was reloaded with Shared (S) data from another core's L3 on the same chip due to an instruction fetch (not prefetch)
PM_INST_FROM_RL2L3_MOD	The processor's Instruction cache was reloaded with Modified (M) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to an instruction fetch (not prefetch)
PM_INST_FROM_L31_ECO_SHR	The processor's Instruction cache was reloaded with Shared (S) data from another core's ECO L3 on the same chip due to an instruction fetch (not prefetch)
PM_INST_FROM_L31_ECO_MOD	The processor's Instruction cache was reloaded with Modified (M) data from another core's ECO L3 on the same chip due to an instruction fetch (not prefetch)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 70

Event	Description
PM_INST_FROM_ON_CHIP_CACHE	The processor's Instruction cache was reloaded either shared or modified data from another core's L2/L3 on the same chip due to an instruction fetch (not prefetch)
PM_INST_FROM_LMEM	The processor's Instruction cache was reloaded from the local chip's Memory due to an instruction fetch (not prefetch)
PM_INST_FROM_RMEM	The processor's Instruction cache was reloaded from another chip's memory on the same Node or Group (Remote) due to an instruction fetch (not prefetch)
PM_INST_FROM_OFF_CHIP_CACHE	The processor's Instruction cache was reloaded either shared or modified data from another core's L2/L3 on a different chip (remote or distant) due to an instruction fetch (not prefetch)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 71

Event	Description
PM_INST_FROM_RL2L3_SHR	The processor's Instruction cache was reloaded with Shared (S) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to an instruction fetch (not prefetch)

Event	Description
PM_UTHROTTL	Cycles in which instruction issue throttle was active in ISU
PM_INST_FROM_DL2L3_SHR	The processor's Instruction cache was reloaded with Shared (S) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to an instruction fetch (not prefetch)
PM_INST_FROM_DL2L3_MOD	The processor's Instruction cache was reloaded with Modified (M) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to an instruction fetch (not prefetch)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 72

Event	Description
PM_INST_FROM_LL4	The processor's Instruction cache was reloaded from the local chip's L4 cache due to an instruction fetch (not prefetch)
PM_INST_FROM_RL4	The processor's Instruction cache was reloaded from another chip's L4 on the same Node or Group (Remote) due to an instruction fetch (not prefetch)
PM_INST_FROM_DL4	The processor's Instruction cache was reloaded from another chip's L4 on a different Node or Group (Distant) due to an instruction fetch (not prefetch)
PM_INST_FROM_DMEM	The processor's Instruction cache was reloaded from another chip's memory on the same Node or Group (Distant) due to an instruction fetch (not prefetch)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 73

Event	Description
PM_INST_ALL_FROM_L2	The processor's Instruction cache was reloaded from local core's L2 due to instruction fetches and prefetches
PM_INST_ALL_FROM_L2_MEPF	The processor's Instruction cache was reloaded from local core's L2 hit without dispatch conflicts on Mepf state. due to instruction fetches and prefetches
PM_INST_ALL_FROM_L2_DISP_CONFLICT_LDHITST	The processor's Instruction cache was reloaded from local core's L2 with load hit store conflict due to instruction fetches and prefetches
PM_INST_ALL_FROM_L2_DISP_CONFLICT_OTHER	The processor's Instruction cache was reloaded from local core's L2 with dispatch conflict due to instruction fetches and prefetches
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 74

Event	Description
PM_INST_ALL_FROM_L2_NO_CONFLICT	The processor's Instruction cache was reloaded from local core's L2 without conflict due to instruction fetches and prefetches
PM_INST_ALL_FROM_L31_MOD	The processor's Instruction cache was reloaded with Modified (M) data from another core's L3 on the same chip due to instruction fetches and prefetches
PM_INST_ALL_FROM_L21_SHR	The processor's Instruction cache was reloaded with Shared (S) data from another core's L2 on the same chip due to instruction fetches and prefetches

Event	Description
PM_INST_ALL_FROM_L21_MOD	The processor's Instruction cache was reloaded with Modified (M) data from another core's L2 on the same chip due to instruction fetches and prefetches
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 75

Event	Description
PM_INST_ALL_FROM_L3_NO_CONFLICT	The processor's Instruction cache was reloaded from local core's L3 without conflict due to instruction fetches and prefetches
PM_INST_ALL_FROM_L3_MEPF	The processor's Instruction cache was reloaded from local core's L3 without dispatch conflicts hit on Mepf state. due to instruction fetches and prefetches
PM_INST_ALL_FROM_L3_DISP_CONFLICT	The processor's Instruction cache was reloaded from local core's L3 with dispatch conflict due to instruction fetches and prefetches
PM_INST_ALL_FROM_L3	The processor's Instruction cache was reloaded from local core's L3 due to instruction fetches and prefetches
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 76

Event	Description
PM_INST_ALL_FROM_L2MISS	The processor's Instruction cache was reloaded from a location other than the local core's L2 due to instruction fetches and prefetches
PM_INST_ALL_FROM_MEMORY	The processor's Instruction cache was reloaded from a memory location including L4 from local remote or distant due to instruction fetches and prefetches
PM_ISLB_MISS	Instruction SLB Miss Total of all segment sizes
PM_INST_ALL_FROM_L3MISS_MOD	The processor's Instruction cache was reloaded from a location other than the local core's L3 due to a instruction fetch
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 77

Event	Description
PM_INST_ALL_FROM_L31_SHR	The processor's Instruction cache was reloaded with Shared (S) data from another core's L3 on the same chip due to instruction fetches and prefetches
PM_INST_ALL_FROM_RL2L3_MOD	The processor's Instruction cache was reloaded with Modified (M) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to instruction fetches and prefetches
PM_INST_ALL_FROM_L31_ECO_SHR	The processor's Instruction cache was reloaded with Shared (S) data from another core's ECO L3 on the same chip due to instruction fetches and prefetches
PM_INST_ALL_FROM_L31_ECO_MOD	The processor's Instruction cache was reloaded with Modified (M) data from another core's ECO L3 on the same chip due to instruction fetches and prefetches
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 78

Event	Description
PM_INST_ALL_FROM_ON_CHIP_CACHE	The processor's Instruction cache was reloaded either shared or modified data from another core's L2/L3 on the same chip due to instruction fetches and prefetches
PM_INST_ALL_FROM_LMEM	The processor's Instruction cache was reloaded from the local chip's Memory due to instruction fetches and prefetches
PM_INST_ALL_FROM_RMEM	The processor's Instruction cache was reloaded from another chip's memory on the same Node or Group (Remote) due to instruction fetches and prefetches
PM_INST_ALL_FROM_OFF_CHIP_CACHE	The processor's Instruction cache was reloaded either shared or modified data from another core's L2/L3 on a different chip (remote or distant) due to instruction fetches and prefetches
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 79

Event	Description
PM_INST_ALL_FROM_RL2L3_SHR	The processor's Instruction cache was reloaded with Shared (S) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to instruction fetches and prefetches
PM_HV_CYC	Cycles in which msr_hv is high. Note that this event does not take msr_pr into consideration
PM_INST_ALL_FROM_DL2L3_SHR	The processor's Instruction cache was reloaded with Shared (S) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to instruction fetches and prefetches
PM_INST_ALL_FROM_DL2L3_MOD	The processor's Instruction cache was reloaded with Modified (M) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to instruction fetches and prefetches
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 80

Event	Description
PM_INST_ALL_FROM_LL4	The processor's Instruction cache was reloaded from the local chip's L4 cache due to instruction fetches and prefetches
PM_INST_ALL_FROM_RL4	The processor's Instruction cache was reloaded from another chip's L4 on the same Node or Group (Remote) due to instruction fetches and prefetches
PM_INST_ALL_FROM_DL4	The processor's Instruction cache was reloaded from another chip's L4 on a different Node or Group (Distant) due to instruction fetches and prefetches
PM_INST_ALL_FROM_DMEM	The processor's Instruction cache was reloaded from another chip's memory on the same Node or Group (Distant) due to instruction fetches and prefetches
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 81

Event	Description
PM_MRK_LD_MISS_EXPOSED_CYC	Marked Load exposed Miss cycles
PM_MRK_LD_MISS_L1	Marked DL1 Demand Miss counted at exec time

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_LD_MISS_L1_CYC	Marked ld latency
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 82

Event	Description
PM_MRK_DATA_FROM_L2	The processor's data cache was reloaded from local core's L2 due to a marked load
PM_BR_UNCOND_CMPL	Completion Time Event. This event can also be calculated from the direct bus as follows: if_pc_br0_br_pred=00 AND if_pc_br0_completed.
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_L2_CYC	Duration in cycles to reload from local core's L2 due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 83

Event	Description
PM_BR_PRED_TA_CMPL	Completion Time Event. This event can also be calculated from the direct bus as follows: if_pc_br0_br_pred(0)='1'.
PM_MRK_DATA_FROM_L2_DISP_CONFLICT_LDHITST_CYC	Duration in cycles to reload from local core's L2 with load hit store conflict due to a marked load
PM_MRK_DATA_FROM_L2_DISP_CONFLICT_LDHITST	The processor's data cache was reloaded from local core's L2 with load hit store conflict due to a marked load
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 84

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_L2_DISP_CONFLICT_OTHER_CYC	Duration in cycles to reload from local core's L2 with dispatch conflict due to a marked load
PM_MRK_ST_CMPL_INT	Marked store finished with intervention
PM_MRK_DATA_FROM_L2_DISP_CONFLICT_OTHER	The processor's data cache was reloaded from local core's L2 with dispatch conflict due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 85

Event	Description
PM_MRK_DPTEG_FROM_L2_NO_CONFLICT	A Page Table Entry was loaded into the TLB from local core's L2 without conflict due to a marked data side request

Event	Description
PM_MRK_DATA_FROM_L2_MEPF	The processor's data cache was reloaded from local core's L2 hit without dispatch conflicts on Mepf state. due to a marked load
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_L2_MEPF_CYC	Duration in cycles to reload from local core's L2 hit without dispatch conflicts on Mepf state. due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 86

Event	Description
PM_MRK_DATA_FROM_L2_NO_CONFLICT	The processor's data cache was reloaded from local core's L2 without conflict due to a marked load
PM_MRK_DATA_FROM_L21_SHR_CYC	Duration in cycles to reload with Shared (S) data from another core's L2 on the same chip due to a marked load
PM_MRK_DATA_FROM_L21_SHR	The processor's data cache was reloaded with Shared (S) data from another core's L2 on the same chip due to a marked load
PM_MRK_DATA_FROM_L2_NO_CONFLICT_CYC	Duration in cycles to reload from local core's L2 without conflict due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 87

Event	Description
PM_DATA_ALL_FROM_L3_NO_CONFLICT	The processor's data cache was reloaded from local core's L3 without conflict due to either demand loads or data prefetch
PM_MRK_DATA_FROM_L3_DISP_CONFLICT_CYC	Duration in cycles to reload from local core's L3 with dispatch conflict due to a marked load
PM_MRK_DATA_FROM_L3_DISP_CONFLICT	The processor's data cache was reloaded from local core's L3 with dispatch conflict due to a marked load
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 88

Event	Description
PM_MRK_DATA_FROM_L2MISS	Data cache reload L2 miss
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_BR_MPRED_CMPL	Marked Branch Mispredicted
PM_MRK_DATA_FROM_L2MISS_CYC	Duration in cycles to reload from a location other than the local core's L2 due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 89

Event	Description
PM_SYNC_MRK_L2MISS	Marked L2 Miss that can throw a synchronous interrupt
PM_MRK_DATA_FROM_L3_CYC	Duration in cycles to reload from local core's L3 due to a marked load
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_L3	The processor's data cache was reloaded from local core's L3 due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 90

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_L3_MEPF	The processor's data cache was reloaded from local core's L3 without dispatch conflicts hit on Mepf state. due to a marked load
PM_ST_MISS_L1	Store Missed L1
PM_MRK_DATA_FROM_L3_MEPF_CYC	Duration in cycles to reload from local core's L3 without dispatch conflicts hit on Mepf state. due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 91

Event	Description
PM_MRK_DATA_FROM_L3_NO_CONFLICT	The processor's data cache was reloaded from local core's L3 without conflict due to a marked load
PM_MRK_DATA_FROM_L31_ECO_SHR_CYC	Duration in cycles to reload with Shared (S) data from another core's ECO L3 on the same chip due to a marked load
PM_MRK_DATA_FROM_L31_ECO_SHR	The processor's data cache was reloaded with Shared (S) data from another core's ECO L3 on the same chip due to a marked load
PM_MRK_DATA_FROM_L3_NO_CONFLICT_CYC	Duration in cycles to reload from local core's L3 without conflict due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 92

Event	Description
PM_SYNC_MRK_L3MISS	Marked L3 misses that can throw a synchronous interrupt
PM_MRK_DATA_FROM_L3MISS_CYC	Duration in cycles to reload from a location other than the local core's L3 due to a marked load
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_L3MISS	The processor's data cache was reloaded from a location other than the local core's L3 due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 93

Event	Description
PM_MRK_DATA_FROM_ON_CHIP_CACHE	The processor's data cache was reloaded either shared or modified data from another core's L2/L3 on the same chip due to a marked load
PM_BACK_BR_CMPL	Branch instruction completed with a target address less than current instruction address
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_ON_CHIP_CACHE_CYC	Duration in cycles to reload either shared or modified data from another core's L2/L3 on the same chip due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 94

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_L21_MOD_CYC	Duration in cycles to reload with Modified (M) data from another core's L2 on the same chip due to a marked load
PM_L1_DCACHE_RELOAD_VALID	DL1 reloaded due to Demand Load
PM_MRK_DATA_FROM_L21_MOD	The processor's data cache was reloaded with Modified (M) data from another core's L2 on the same chip due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 95

Event	Description
PM_IPTEG_FROM_L2_NO_CONFLICT	A Page Table Entry was loaded into the TLB from local core's L2 without conflict due to a instruction side request
PM_MRK_DATA_FROM_L31_ECO_MOD_CYC	Duration in cycles to reload with Modified (M) data from another core's ECO L3 on the same chip due to a marked load
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_L31_ECO_MOD	The processor's data cache was reloaded with Modified (M) data from another core's ECO L3 on the same chip due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 96

Event	Description
PM_INST_ALL_PUMP_CPRED	Pump prediction correct. Counts across all types of pumps for instruction fetches and prefetches
PM_MRK_DATA_FROM_L31_MOD	The processor's data cache was reloaded with Modified (M) data from another core's L3 on the same chip due to a marked load
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_L31_MOD_CYC	Duration in cycles to reload with Modified (M) data from another core's L3 on the same chip due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 97

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_RMEM_CYC	Duration in cycles to reload from another chip's memory on the same Node or Group (Remote) due to a marked load
PM_MRK_DATA_FROM_RMEM	The processor's data cache was reloaded from another chip's memory on the same Node or Group (Remote) due to a marked load
PM_SYS_PUMP_MPRED_RTY	Final Pump Scope (system) ended up larger than Initial Pump Scope (Chip/Group) for all data types excluding data prefetch (demand load, instruction prefetch, instruction fetch, xlate)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 98

Event	Description
PM_MRK_DATA_FROM_L31_SHR	The processor's data cache was reloaded with Shared (S) data from another core's L3 on the same chip due to a marked load
PM_DATA_ALL_FROM_L31_MOD	The processor's data cache was reloaded with Modified (M) data from another core's L3 on the same chip due to either demand loads or data prefetch
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_L31_SHR_CYC	Duration in cycles to reload with Shared (S) data from another core's L3 on the same chip due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 99

Event	Description
PM_MRK_DATA_FROM_LL4	The processor's data cache was reloaded from the local chip's L4 cache due to a marked load
PM_MRK_DATA_FROM_DL4_CYC	Duration in cycles to reload from another chip's L4 on a different Node or Group (Distant) due to a marked load
PM_MRK_DATA_FROM_DL4	The processor's data cache was reloaded from another chip's L4 on a different Node or Group (Distant) due to a marked load
PM_MRK_DATA_FROM_LL4_CYC	Duration in cycles to reload from the local chip's L4 cache due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 100

Event	Description
PM_LD_L3MISS_PEND_CYC	Cycles L3 miss was pending for this thread
PM_MRK_DATA_FROM_LMEM	The processor's data cache was reloaded from the local chip's Memory due to a marked load
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_LMEM_CYC	Duration in cycles to reload from the local chip's Memory due to a marked load
PM_RUN_INST_CMPL	Run instructions

Event	Description
PM_RUN_CYC	Run cycles

Group 101

Event	Description
PM_LD_REF_L1	All L1 D cache load references counted at finish, gated by reject
PM_MRK_DATA_FROM_MEMORY	The processor's data cache was reloaded from a memory location including L4 from local remote or distant due to a marked load
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_MEMORY_CYC	Duration in cycles to reload from a memory location including L4 from local remote or distant due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 102

Event	Description
PM_INST_PUMP_CPRED	Pump prediction correct. Counts across all types of pumps for an instruction fetch
PM_MRK_DATA_FROM_OFF_CHIP_CACHE_CYC	Duration in cycles to reload either shared or modified data from another core's L2/L3 on a different chip (remote or distant) due to a marked load
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_OFF_CHIP_CACHE	The processor's data cache was reloaded either shared or modified data from another core's L2/L3 on a different chip (remote or distant) due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 103

Event	Description
PM_INST_ALL_PUMP_CPRED	Pump prediction correct. Counts across all types of pumps for instruction fetches and prefetches
PM_MRK_DATA_FROM_RL2L3_MOD	The processor's data cache was reloaded with Modified (M) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to a marked load
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_RL2L3_MOD_CYC	Duration in cycles to reload with Modified (M) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 104

Event	Description
PM_MRK_DATA_FROM_RL2L3_SHR	The processor's data cache was reloaded with Shared (S) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to a marked load

Event	Description
PM_MRK_DATA_FROM_DL2L3_SHR_CYC	Duration in cycles to reload with Shared (S) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to a marked load
PM_MRK_DATA_FROM_DL2L3_SHR	The processor's data cache was reloaded with Shared (S) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to a marked load
PM_MRK_DATA_FROM_RL2L3_SHR_CYC	Duration in cycles to reload with Shared (S) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 105

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_RL4	The processor's data cache was reloaded from another chip's L4 on the same Node or Group (Remote) due to a marked load
PM_ALL_SYS_PUMP_CPRED	Initial and Final Pump Scope was system pump for all data types (demand load, data prefetch, instruction prefetch, instruction fetch, translate)
PM_MRK_DATA_FROM_RL4_CYC	Duration in cycles to reload from another chip's L4 on the same Node or Group (Remote) due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 106

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_DL2L3_MOD_CYC	Duration in cycles to reload with Modified (M) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to a marked load
PM_I2_SYS_PUMP	RC requests that were system pump attempts
PM_MRK_DATA_FROM_DL2L3_MOD	The processor's data cache was reloaded with Modified (M) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to a marked load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 107

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DATA_FROM_DMEM_CYC	Duration in cycles to reload from another chip's memory on the same Node or Group (Distant) due to a marked load
PM_IPTEG_FROM_I2_DISP_CONFLICT_LDHITST	A Page Table Entry was loaded into the TLB from local core's L2 with load hit store conflict due to a instruction side request
PM_MRK_DATA_FROM_DMEM	The processor's data cache was reloaded from another chip's memory on the same Node or Group (Distant) due to a marked load
PM_RUN_INST_CMPL	Run instructions

Event	Description
PM_RUN_CYC	Run cycles

Group 108

Event	Description
PM_L1MISS_LAT_EXC_256	L1 misses that took longer than 256 cycles to resolve (miss to reload)
PM_L1MISS_LAT_EXC_32	L1 misses that took longer than 32 cycles to resolve (miss to reload)
PM_L1MISS_LAT_EXC_1024	L1 misses that took longer than 1024 cycles to resolve (miss to reload)
PM_L1MISS_LAT_EXC_2048	L1 misses that took longer than 2048 cycles to resolve (miss to reload)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 109

Event	Description
PM_MRK_ST_L2DISP_TO_CMPL_CYC	Cycles from L2 RC dispatch to L2 RC completion
PM_MRK_ST_NEST	Marked store sent to nest
PM_MRK_ST_DRAIN_TO_L2DISP_CYC	Cycles to drain st from core to L2
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 110

Event	Description
PM_L2_ST	All successful D side store dispatches for this thread
PM_L2_CHIP_PUMP	RC requests that were local on chip pump attempts
PM_L2_SYS_PUMP	RC requests that were system pump attempts
PM_L2_RTY_LD	RC retries on PB for any load from core
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 111

Event	Description
PM_L2_ST_MISS	All successful D side store dispatches for this thread that were L2 Miss
PM_L2_GROUP_PUMP	RC requests that were on Node Pump attempts
PM_L2_RTY_ST	RC retries on PB for any store from core
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 112

Event	Description
PM_RC_LIFETIME_EXC_256	Number of times the RC machine for a sampled instruction was active for more than 256 cycles
PM_RC_LIFETIME_EXC_32	Number of times the RC machine for a sampled instruction was active for more than 32 cycles
PM_RC_LIFETIME_EXC_1024	Number of times the RC machine for a sampled instruction was active for more than 1024 cycles
PM_RC_LIFETIME_EXC_2048	Number of times the RC machine for a sampled instruction was active for more than 2048 cycles
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 113

Event	Description
PM_RC0_BUSY	RC mach 0 Busy. Used by PMU to sample average RC lifetime (mach0 used as sample point)
PM_SN0_BUSY	SN mach 0 Busy. Used by PMU to sample average RC lifetime (mach0 used as sample point)
PM_RC_USAGE	Continuous 16 cycle(2to1) window where this signals rotates thru sampling each L2 RC machine busy. PMU uses this wave to then do 16 cyc count to sample total number of machs running
PM_SN_USAGE	Continuous 16 cycle(2to1) window where this signals rotates thru sampling each L2 SN machine busy. PMU uses this wave to then do 16 cyc count to sample total number of machs running
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 114

Event	Description
PM_CO0_BUSY	CO mach 0 Busy. Used by PMU to sample average RC lifetime (mach0 used as sample point)
PM_L1PF_L2MEMACC	Valid when first beat of data comes in for an L1pref where data came from mem(or L4)
PM_CO_USAGE	Continuous 16 cycle(2to1) window where this signals rotates thru sampling each L2 CO machine busy. PMU uses this wave to then do 16 cyc count to sample total number of machs running
PM_ISIDE_L2MEMACC	Valid when first beat of data comes in for an i side fetch where data came from mem(or L4)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 115

Event	Description
PM_RC0_ALLOC	RC mach 0 Busy. Used by PMU to sample average RC lifetime (mach0 used as sample point)
PM_SN0_ALLOC	SN mach 0 Busy. Used by PMU to sample average RC lifetime (mach0 used as sample point)
PM_L3_SN0_ALLOC	Lifetime, sample of snoop machine 0 valid
PM_L3_RD0_ALLOC	Lifetime, sample of RD machine 0 valid
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 116

Event	Description
PM_CO0_ALLOC	CO mach 0 Busy. Used by PMU to sample average RC lifetime (mach0 used as sample point)
PM_ST_CMPL	Store completion count
PM_L3_CO0_ALLOC	Lifetime, sample of CO machine 0 valid
PM_L3_PF0_ALLOC	Lifetime, sample of PF machine 0 valid
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 117

Event	Description
PM_L3_PF_MISS_L3	L3 Prefetch missed in L3
PM_L3_CO_L31	L3 CO to L3.1 OR of port 0 and 1 (lossy)
PM_L3_PF_ON_CHIP_CACHE	L3 Prefetch from On chip cache
PM_L3_PF_ON_CHIP_MEM	L3 Prefetch from On chip memory
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 118

Event	Description
PM_L3_LD_PREF	L3 Load Prefetches
PM_L3_CO_MEM	L3 CO to memory OR of port 0 and 1 (lossy)
PM_L3_PF_OFF_CHIP_CACHE	L3 Prefetch from Off chip cache
PM_L3_PF_OFF_CHIP_MEM	L3 Prefetch from Off chip memory
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 119

Event	Description
PM_L3_SN_USAGE	Rotating sample of 8 snoop valids
PM_L3_RD_USAGE	Rotating sample of 16 RD actives
PM_L3_SN0_BUSY	Lifetime, sample of snooper machine 0 valid
PM_L3_RD0_BUSY	Lifetime, sample of RD machine 0 valid
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 120

Event	Description
PM_L3_CI_USAGE	Rotating sample of 16 CI or CO actives
PM_L3_PF_USAGE	Rotating sample of 32 PF actives
PM_L3_CO0_BUSY	Lifetime, sample of CO machine 0 valid
PM_L3_PF0_BUSY	Lifetime, sample of PF machine 0 valid
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 121

Event	Description
PM_VSU0_1FLOP	One flop (fadd, fmul, fsub, fcmp, fsel, fabs, fnabs, fres, fsqrte, fneg) operation finished
PM_VSU1_1FLOP	One flop (fadd, fmul, fsub, fcmp, fsel, fabs, fnabs, fres, fsqrte, fneg) operation finished
PM_VSU0_2FLOP	Two flops operation (scalar fmadd, fnmadd, fmsub, fnmsub and DP vector versions of single flop instructions)
PM_VSU1_2FLOP	Two flops operation (scalar fmadd, fnmadd, fmsub, fnmsub and DP vector versions of single flop instructions)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 122

Event	Description
PM_VSU0_4FLOP	Four flops operation (scalar fdiv, fsqrt, DP vector version of fmadd, fnmadd, fmsub, fnmsub, SP vector versions of single flop instructions)
PM_VSU1_4FLOP	Four flops operation (scalar fdiv, fsqrt, DP vector version of fmadd, fnmadd, fmsub, fnmsub, SP vector versions of single flop instructions)
PM_VSU0_8FLOP	Eight flops operation (DP vector versions of fdiv,fsqrt and SP vector versions of fmadd,fnmadd,fmsub,fnmsub)
PM_VSU1_8FLOP	Eight flops operation (DP vector versions of fdiv,fsqrt and SP vector versions of fmadd,fnmadd,fmsub,fnmsub)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 123

Event	Description
PM_VSU0_COMPLEX_ISSUED	Complex VMX instruction issued
PM_VSU1_COMPLEX_ISSUED	Complex VMX instruction issued
PM_VSU0_SIMPLE_ISSUED	Simple VMX instruction issued
PM_VSU1_SIMPLE_ISSUED	Simple VMX instruction issued
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 124

Event	Description
PM_VSU0_DP_FMA	DP vector version of fmadd,fnmadd,fmsub,fnmsub
PM_VSU1_DP_FMA	DP vector version of fmadd,fnmadd,fmsub,fnmsub
PM_VSU0_FMA	Two flops operation (fmadd, fnmadd, fmsub, fnmsub) Scalar instructions only!
PM_VSU1_FMA	Two flops operation (fmadd, fnmadd, fmsub, fnmsub) Scalar instructions only!
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 125

Event	Description
PM_VSU1_DD_ISSUED	64BIT Decimal Issued

Event	Description
PM_VSU0_DD_ISSUED	64BIT Decimal Issued
PM_VSU0_CY_ISSUED	Cryptographic instruction RFC02196 Issued
PM_VSU1_CY_ISSUED	Cryptographic instruction RFC02196 Issued
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 126

Event	Description
PM_VSU1_FSQRT_FDIV	Four flops operation (fdiv,fsqrt) Scalar Instructions only!
PM_VSU0_SQ	Store Vector Issued
PM_VSU1_SQ	Store Vector Issued
PM_VSU0_16FLOP	Sixteen flops operation (SP vector versions of fdiv,fsqrt)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 127

Event	Description
PM_VSU0_SINGLE	FPU single precision
PM_VSU1_SINGLE	FPU single precision
PM_VSU0_FIN	VSU0 Finished an instruction
PM_VSU1_FIN	VSU1 Finished an instruction
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 128

Event	Description
PM_VSU0_EX_ISSUED	Direct move 32/64b VRFtoGPR RFC02206 Issued
PM_VSU1_EX_ISSUED	Direct move 32/64b VRFtoGPR RFC02206 Issued
PM_VSU0_DQ_ISSUED	128BIT Decimal Issued
PM_VSU1_DQ_ISSUED	128BIT Decimal Issued
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 129

Event	Description
PM_VSU0_VECTOR_DP_ISSUED	Double Precision vector instruction issued on Pipe0
PM_VSU1_VECTOR_DP_ISSUED	Double Precision vector instruction issued on Pipe1
PM_VSU0_VECTOR_SP_ISSUED	Single Precision vector instruction issued (executed)
PM_VSU1_VECTOR_SP_ISSUED	Single Precision vector instruction issued (executed)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 130

Event	Description
PM_VSU0_SCALAR_DP_ISSUED	Double Precision scalar instruction issued on Pipe0
PM_VSU1_SCALAR_DP_ISSUED	Double Precision scalar instruction issued on Pipe1

Event	Description
PM_VSU0_DP_FSQRT_FDIV	DP vector versions of fdiv,fsqrt
PM_VSU1_DP_FSQRT_FDIV	DP vector versions of fdiv,fsqrt
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 131

Event	Description
PM_VSU0_FPSCR	Move to/from FPSCR type instruction issued on Pipe 0
PM_VSU1_FPSCR	Move to/from FPSCR type instruction issued on Pipe 0
PM_VSU0_DP_2FLOP	DP vector version of fmul, fsub, fcmp, fsel, fabs, fnabs, fres ,fsqrte, fneg
PM_VSU1_DP_2FLOP	DP vector version of fmul, fsub, fcmp, fsel, fabs, fnabs, fres ,fsqrte, fneg
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 132

Event	Description
PM_VSU0_STF	FPU store (SP or DP) issued on Pipe0
PM_VSU1_STF	FPU store (SP or DP) issued on Pipe1
PM_VSU0_PERMUTE_ISSUED	Permute VMX Instruction Issued
PM_VSU1_PERMUTE_ISSUED	Permute VMX Instruction Issued
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 133

Event	Description
PM_VSU0_FSQRT_FDIV	Four flops operation (fdiv,fsqrt) Scalar Instructions only!
PM_VSU1_FSQRT_FDIV	Four flops operation (fdiv,fsqrt) Scalar Instructions only!
PM_VSU1_16FLOP	Sixteen flops operation (SP vector versions of fdiv,fsqrt)
PM_IPTEG_FROM_L3	A Page Table Entry was loaded into the TLB from local core's L3 due to a instruction side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 134

Event	Description
PM_DFU	Finish DFU (all finish)
PM_DFU_DCFIX	Convert from fixed opcode finish (dcffix,dcffixq)
PM_DFU_DENBCD	BCD >DPD opcode finish (denbcd, denbcdq)
PM_DFU_MC	Finish DFU multicycle
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 135

Event	Description
PM_DPTEG_FROM_L2_NO_CONFLICT	A Page Table Entry was loaded into the TLB from local core's L2 without conflict due to a data side request
PM_DPTEG_FROM_L2_MEPF	A Page Table Entry was loaded into the TLB from local core's L2 hit without dispatch conflicts on Mepf state, due to a data side request
PM_DPTEG_FROM_L2_DISP_CONFLICT_LDHITST	A Page Table Entry was loaded into the TLB from local core's L2 with load hit store conflict due to a data side request
PM_DPTEG_FROM_L2_DISP_CONFLICT_OTHER	A Page Table Entry was loaded into the TLB from local core's L2 with dispatch conflict due to a data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 136

Event	Description
PM_DPTEG_FROM_L3_NO_CONFLICT	A Page Table Entry was loaded into the TLB from local core's L3 without conflict due to a data side request
PM_DPTEG_FROM_L3_MEPF	A Page Table Entry was loaded into the TLB from local core's L3 without dispatch conflicts hit on Mepf state, due to a data side request
PM_DPTEG_FROM_L3_DISP_CONFLICT	A Page Table Entry was loaded into the TLB from local core's L3 with dispatch conflict due to a data side request
PM_DPTEG_FROM_L3	A Page Table Entry was loaded into the TLB from local core's L3 due to a data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 137

Event	Description
PM_DPTEG_FROM_L31_SHR	A Page Table Entry was loaded into the TLB with Shared (S) data from another core's L3 on the same chip due to a data side request
PM_DPTEG_FROM_L31_MOD	A Page Table Entry was loaded into the TLB with Modified (M) data from another core's L3 on the same chip due to a data side request
PM_DPTEG_FROM_L31_ECO_SHR	A Page Table Entry was loaded into the TLB with Shared (S) data from another core's ECO L3 on the same chip due to a data side request
PM_DPTEG_FROM_L31_ECO_MOD	A Page Table Entry was loaded into the TLB with Modified (M) data from another core's ECO L3 on the same chip due to a data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 138

Event	Description
PM_DPTEG_FROM_RL2L3_SHR	A Page Table Entry was loaded into the TLB with Shared (S) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to a data side request
PM_DPTEG_FROM_RL2L3_MOD	A Page Table Entry was loaded into the TLB with Modified (M) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to a data side request
PM_DPTEG_FROM_DL2L3_SHR	A Page Table Entry was loaded into the TLB with Shared (S) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to a data side request

Event	Description
PM_DPTEG_FROM_DL2L3_MOD	A Page Table Entry was loaded into the TLB with Modified (M) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to a data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 139

Event	Description
PM_DPTEG_FROM_ON_CHIP_CACHE	A Page Table Entry was loaded into the TLB either shared or modified data from another core's L2/L3 on the same chip due to a data side request
PM_DPTEG_FROM_MEMORY	A Page Table Entry was loaded into the TLB from a memory location including L4 from local remote or distant due to a data side request
PM_DPTEG_FROM_RMEM	A Page Table Entry was loaded into the TLB from another chip's memory on the same Node or Group (Remote) due to a data side request
PM_DPTEG_FROM_OFF_CHIP_CACHE	A Page Table Entry was loaded into the TLB either shared or modified data from another core's L2/L3 on a different chip (remote or distant) due to a data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 140

Event	Description
PM_DPTEG_FROM_LL4	A Page Table Entry was loaded into the TLB from the local chip's L4 cache due to a data side request
PM_DPTEG_FROM_RL4	A Page Table Entry was loaded into the TLB from another chip's L4 on the same Node or Group (Remote) due to a data side request
PM_DPTEG_FROM_DL4	A Page Table Entry was loaded into the TLB from another chip's L4 on a different Node or Group (Distant) due to a data side request
PM_DPTEG_FROM_DMEN	A Page Table Entry was loaded into the TLB from another chip's memory on the same Node or Group (Distant) due to a data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 141

Event	Description
PM_DPTEG_FROM_L2MISS	A Page Table Entry was loaded into the TLB from a location other than the local core's L2 due to a data side request
PM_DPTEG_FROM_LMEM	A Page Table Entry was loaded into the TLB from the local chip's Memory due to a data side request
PM_DPTEG_FROM_RMEM	A Page Table Entry was loaded into the TLB from another chip's memory on the same Node or Group (Remote) due to a data side request
PM_DPTEG_FROM_L3MISS	A Page Table Entry was loaded into the TLB from a location other than the local core's L3 due to a data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 142

Event	Description
PM_DPTEG_FROM_L2	A Page Table Entry was loaded into the TLB from local core's L2 due to a data side request
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_DPTEG_FROM_L21_SHR	A Page Table Entry was loaded into the TLB with Shared (S) data from another core's L2 on the same chip due to a data side request
PM_DPTEG_FROM_L21_MOD	A Page Table Entry was loaded into the TLB with Modified (M) data from another core's L2 on the same chip due to a data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 143

Event	Description
PM_IPTEG_FROM_L3_NO_CONFLICT	A Page Table Entry was loaded into the TLB from local core's L3 without conflict due to a instruction side request
PM_IPTEG_FROM_L3_MEPF	A Page Table Entry was loaded into the TLB from local core's L3 without dispatch conflicts hit on Mepf state. due to a instruction side request
PM_IPTEG_FROM_L3_DISP_CONFLICT	A Page Table Entry was loaded into the TLB from local core's L3 with dispatch conflict due to a instruction side request
PM_IPTEG_FROM_L2_DISP_CONFLICT_OTHER	A Page Table Entry was loaded into the TLB from local core's L2 with dispatch conflict due to a instruction side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 144

Event	Description
PM_IPTEG_FROM_L31_SHR	A Page Table Entry was loaded into the TLB with Shared (S) data from another core's L3 on the same chip due to a instruction side request
PM_IPTEG_FROM_L31_MOD	A Page Table Entry was loaded into the TLB with Modified (M) data from another core's L3 on the same chip due to a instruction side request
PM_IPTEG_FROM_L31_ECO_SHR	A Page Table Entry was loaded into the TLB with Shared (S) data from another core's ECO L3 on the same chip due to a instruction side request
PM_IPTEG_FROM_L31_ECO_MOD	A Page Table Entry was loaded into the TLB with Modified (M) data from another core's ECO L3 on the same chip due to a instruction side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 145

Event	Description
PM_IPTEG_FROM_RL2L3_SHR	A Page Table Entry was loaded into the TLB with Shared (S) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to a instruction side request

Event	Description
PM_IPTEG_FROM_RL2L3_MOD	A Page Table Entry was loaded into the TLB with Modified (M) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to a instruction side request
PM_IPTEG_FROM_DL2L3_SHR	A Page Table Entry was loaded into the TLB with Shared (S) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to a instruction side request
PM_IPTEG_FROM_DL2L3_MOD	A Page Table Entry was loaded into the TLB with Modified (M) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to a instruction side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 146

Event	Description
PM_IPTEG_FROM_ON_CHIP_CACHE	A Page Table Entry was loaded into the TLB either shared or modified data from another core's L2/L3 on the same chip due to a instruction side request
PM_IPTEG_FROM_MEMORY	A Page Table Entry was loaded into the TLB from a memory location including L4 from local remote or distant due to a instruction side request
PM_IPTEG_FROM_RMEM	A Page Table Entry was loaded into the TLB from another chip's memory on the same Node or Group (Remote) due to a instruction side request
PM_IPTEG_FROM_OFF_CHIP_CACHE	A Page Table Entry was loaded into the TLB either shared or modified data from another core's L2/L3 on a different chip (remote or distant) due to a instruction side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 147

Event	Description
PM_IPTEG_FROM_LL4	A Page Table Entry was loaded into the TLB from the local chip's L4 cache due to a instruction side request
PM_IPTEG_FROM_RL4	A Page Table Entry was loaded into the TLB from another chip's L4 on the same Node or Group (Remote) due to a instruction side request
PM_IPTEG_FROM_DL4	A Page Table Entry was loaded into the TLB from another chip's L4 on a different Node or Group (Distant) due to a instruction side request
PM_IPTEG_FROM_L3MISS	A Page Table Entry was loaded into the TLB from a location other than the local core's L3 due to a instruction side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 148

Event	Description
PM_IPTEG_FROM_L2MISS	A Page Table Entry was loaded into the TLB from a location other than the local core's L2 due to a instruction side request
PM_IPTEG_FROM_LMEM	A Page Table Entry was loaded into the TLB from the local chip's Memory due to a instruction side request

Event	Description
PM_IPTEG_FROM_RMEM	A Page Table Entry was loaded into the TLB from another chip's memory on the same Node or Group (Remote) due to a instruction side request
PM_IPTEG_FROM_DMEN	A Page Table Entry was loaded into the TLB from another chip's memory on the same Node or Group (Distant) due to a instruction side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 149

Event	Description
PM_IPTEG_FROM_L2	A Page Table Entry was loaded into the TLB from local core's L2 due to a instruction side request
PM_IPTEG_FROM_L2_MEPF	A Page Table Entry was loaded into the TLB from local core's L2 hit without dispatch conflicts on Mepf state. due to a instruction side request
PM_IPTEG_FROM_L21_SHR	A Page Table Entry was loaded into the TLB with Shared (S) data from another core's L2 on the same chip due to a instruction side request
PM_IPTEG_FROM_L21_MOD	A Page Table Entry was loaded into the TLB with Modified (M) data from another core's L2 on the same chip due to a instruction side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 150

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DPTEG_FROM_L2_MEPF	A Page Table Entry was loaded into the TLB from local core's L2 hit without dispatch conflicts on Mepf state. due to a marked data side request
PM_MRK_DPTEG_FROM_L2_DISP_CONFLICT_LDHITST	A Page Table Entry was loaded into the TLB from local core's L2 with load hit store conflict due to a marked data side request
PM_MRK_DPTEG_FROM_L2_DISP_CONFLICT_OTHER	A Page Table Entry was loaded into the TLB from local core's L2 with dispatch conflict due to a marked data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 151

Event	Description
PM_MRK_DPTEG_FROM_L3_NO_CONFLICT	A Page Table Entry was loaded into the TLB from local core's L3 without conflict due to a marked data side request
PM_MRK_DPTEG_FROM_L3_MEPF	A Page Table Entry was loaded into the TLB from local core's L3 without dispatch conflicts hit on Mepf state. due to a marked data side request
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DPTEG_FROM_L3	A Page Table Entry was loaded into the TLB from local core's L3 due to a marked data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 152

Event	Description
PM_MRK_DPTEG_FROM_L31_SHR	A Page Table Entry was loaded into the TLB with Shared (S) data from another core's L3 on the same chip due to a marked data side request
PM_MRK_DPTEG_FROM_L31_MOD	A Page Table Entry was loaded into the TLB with Modified (M) data from another core's L3 on the same chip due to a marked data side request
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DPTEG_FROM_L31_ECO_MOD	A Page Table Entry was loaded into the TLB with Modified (M) data from another core's ECO L3 on the same chip due to a marked data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 153

Event	Description
PM_MRK_DPTEG_FROM_L2	A Page Table Entry was loaded into the TLB from local core's L2 due to a marked data side request
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DPTEG_FROM_L21_SHR	A Page Table Entry was loaded into the TLB with Shared (S) data from another core's L2 on the same chip due to a marked data side request
PM_MRK_DPTEG_FROM_L21_MOD	A Page Table Entry was loaded into the TLB with Modified (M) data from another core's L2 on the same chip due to a marked data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 154

Event	Description
PM_MRK_DPTEG_FROM_RL2L3_SHR	A Page Table Entry was loaded into the TLB with Shared (S) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to a marked data side request
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DPTEG_FROM_DL2L3_SHR	A Page Table Entry was loaded into the TLB with Shared (S) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to a marked data side request
PM_MRK_DPTEG_FROM_DL2L3_MOD	A Page Table Entry was loaded into the TLB with Modified (M) data from another chip's L2 or L3 on a different Node or Group (Distant), as this chip due to a marked data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 155

Event	Description
PM_MRK_DPTEG_FROM_ON_CHIP_CACHE	A Page Table Entry was loaded into the TLB either shared or modified data from another core's L2/L3 on the same chip due to a marked data side request
PM_MRK_DPTEG_FROM_LMEM	A Page Table Entry was loaded into the TLB from the local chip's Memory due to a marked data side request

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DPTEG_FROM_OFF_CHIP_CACHE	A Page Table Entry was loaded into the TLB either shared or modified data from another core's L2/L3 on a different chip (remote or distant) due to a marked data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 156

Event	Description
PM_MRK_DPTEG_FROM_LL4	A Page Table Entry was loaded into the TLB from the local chip's L4 cache due to a marked data side request
PM_MRK_DPTEG_FROM_RL4	A Page Table Entry was loaded into the TLB from another chip's L4 on the same Node or Group (Remote) due to a marked data side request
PM_MRK_DPTEG_FROM_DL4	A Page Table Entry was loaded into the TLB from another chip's L4 on a different Node or Group (Distant) due to a marked data side request
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 157

Event	Description
PM_MRK_DPTEG_FROM_L2MISS	A Page Table Entry was loaded into the TLB from a location other than the local core's L2 due to a marked data side request
PM_MRK_DATA_FROM_MEM	The processor's data cache was reloaded from a memory location including L4 from local remote or distant due to a marked load
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DPTEG_FROM_L3MISS	A Page Table Entry was loaded into the TLB from a location other than the local core's L3 due to a marked data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 158

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DPTEG_FROM_MEMORY	A Page Table Entry was loaded into the TLB from a memory location including L4 from local remote or distant due to a marked data side request
PM_MRK_DPTEG_FROM_RMEM	A Page Table Entry was loaded into the TLB from another chip's memory on the same Node or Group (Remote) due to a marked data side request
PM_MRK_DPTEG_FROM_DMEM	A Page Table Entry was loaded into the TLB from another chip's memory on the same Node or Group (Distant) due to a marked data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 159

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DPTEG_FROM_RL2L3_MOD	A Page Table Entry was loaded into the TLB with Modified (M) data from another chip's L2 or L3 on the same Node or Group (Remote), as this chip due to a marked data side request
PM_MRK_DPTEG_FROM_L3_DISP_CONFLICT	A Page Table Entry was loaded into the TLB from local core's L3 with dispatch conflict due to a marked data side request
PM_MRK_DPTEG_FROM_L2_DISP_CONFLICT_OTHER	A Page Table Entry was loaded into the TLB from local core's L2 with dispatch conflict due to a marked data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 160

Event	Description
PM_MRK_DPTEG_FROM_L31_SHR	A Page Table Entry was loaded into the TLB with Shared (S) data from another core's L3 on the same chip due to a marked data side request
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DPTEG_FROM_L31_ECO_SHR	A Page Table Entry was loaded into the TLB with Shared (S) data from another core's ECO L3 on the same chip due to a marked data side request
PM_MRK_DPTEG_FROM_L31_ECO_MOD	A Page Table Entry was loaded into the TLB with Modified (M) data from another core's ECO L3 on the same chip due to a marked data side request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 161

Event	Description
PM_LSU_FX_FIN	LSU Finished a FX operation (up to 2 per cycle)
PM_ST_FIN	Store Instructions Finished
PM_LSU_FIN	LSU Finished an instruction (up to 2 per cycle)
PM_LD_MISS_L1	Load Missed L1
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 162

Event	Description
PM_LSU_LRQ_S0_ALLOC	Per thread use edge detect to count allocates On a per thread basis, level signal indicating Slot 0 is valid. By instrumenting a single slot we can calculate service time for that slot. Previous machines required a separate signal indicating the slot was allocated. Because any signal can be routed to any counter in P8, we can count level in one PMC and edge detect in another PMC using the same signal
PM_LSU_LRQ_S43_VALID	LRQ slot 43 was busy
PM_LSU_LRQ_S43_ALLOC	LRQ slot 43 was released
PM_LSU_LRQ_S0_VALID	Slot 0 of LRQ valid
PM_RUN_INST_CMPL	Run instructions

Event	Description
PM_RUN_CYC	Run cycles

Group 163

Event	Description
PM_LSU_LMQ_FULL_CYC	LMQ full
PM_LSU_LMQ_SRQ_EMPTY_CYC	LSU empty (lmq and srq empty)
PM_LSU_LMQ_S0_VALID	Per thread use edge detect to count allocates On a per thread basis, level signal indicating Slot 0 is valid. By instrumenting a single slot we can calculate service time for that slot. Previous machines required a separate signal indicating the slot was allocated. Because any signal can be routed to any counter in P8, we can count level in one PMC and edge detect in another PMC using the same signal.
PM_LSU_LMQ_S0_ALLOC	Slot 0 of LMQ valid
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 164

Event	Description
PM_LSU_SRQ_S39_ALLOC	SRQ slot 39 was released
PM_LSU_SRQ_S0_VALID	Slot 0 of SRQ valid
PM_LSU_SRQ_S0_ALLOC	Per thread use edge detect to count allocates On a per thread basis, level signal indicating Slot 0 is valid. By instrumenting a single slot we can calculate service time for that slot. Previous machines required a separate signal indicating the slot was allocated. Because any signal can be routed to any counter in P8, we can count level in one PMC and edge detect in another PMC using the same signal.
PM_LSU_SRQ_S39_VALID	SRQ slot 39 was busy
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 165

Event	Description
PM_LSU_SRQ_FULL_CYC	Storage Queue is full and is blocking dispatch
PM_REAL_SRQ_FULL	Out of real srq entries
PM_DISP_HELD_SRQ_FULL	Dispatch held due SRQ no room
PM_LSU0_STORE_REJECT	LS0 store reject
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 166

Event	Description
PM_LSU0_LMQ_LHR_MERGE	LS0 Load Merged with another cacheline request
PM_LSU1_LMQ_LHR_MERGE	LS1 Load Merge with another cacheline request
PM_LSU2_LMQ_LHR_MERGE	LS0 Load Merged with another cacheline request
PM_LSU3_LMQ_LHR_MERGE	LS1 Load Merge with another cacheline request
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 167

Event	Description
PM_LD_REF_L1_LSU0	LS0 L1 D cache load references counted at finish, gated by reject
PM_LD_REF_L1_LSU1	LS1 L1 D cache load references counted at finish, gated by reject
PM_LD_REF_L1_LSU2	LS2 L1 D cache load references counted at finish, gated by reject
PM_LD_REF_L1_LSU3	LS3 L1 D cache load references counted at finish, gated by reject
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 168

Event	Description
PM_LS0_L1_PREF	LS0 L1 cache data prefetches
PM_LS1_L1_PREF	LS1 L1 cache data prefetches
PM_LS0_L1_SW_PREF	Software L1 Prefetches, including SW Transient Prefetches
PM_LS1_L1_SW_PREF	Software L1 Prefetches, including SW Transient Prefetches
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 169

Event	Description
PM_LSU0_SRQ_STFWD	LS0 SRQ forwarded data to a load
PM_LSU1_SRQ_STFWD	LS1 SRQ forwarded data to a load
PM_LSU2_SRQ_STFWD	LS2 SRQ forwarded data to a load
PM_LSU3_SRQ_STFWD	LS3 SRQ forwarded data to a load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 170

Event	Description
PM_LSU0_FLUSH_UST	LS0 Flush: Unaligned Store
PM_LSU1_FLUSH_UST	LS1 Flush: Unaligned Store
PM_FREQ_DOWN	Power Management: Below Threshold B
PM_FREQ_UP	Power Management: Above Threshold A
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 171

Event	Description
PM_LSU0_FLUSH_LRQ	LS0 Flush: LRQ
PM_LSU1_FLUSH_LRQ	LS1 Flush: LRQ
PM_LSU2_FLUSH_LRQ	LS2 Flush: LRQ
PM_LSU3_FLUSH_LRQ	LS3 Flush: LRQ
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 172

Event	Description
PM_LSU0_FLUSH_ULD	LS0 Flush: Unaligned Load
PM_LSU1_FLUSH_ULD	LS1 Flush: Unaligned Load
PM_LSU2_FLUSH_ULD	LS3 Flush: Unaligned Load
PM_LSU3_FLUSH_ULD	LS14 Flush: Unaligned Load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 173

Event	Description
PM_LSU0_FLUSH_SRQ	LS0 Flush: SRQ
PM_LSU1_FLUSH_SRQ	LS1 Flush: SRQ
PM_LSU2_FLUSH_SRQ	LS2 Flush: SRQ
PM_LSU3_FLUSH_SRQ	LS13 Flush: SRQ
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 174

Event	Description
PM_PTE_PREFETCH	PTE prefetches
PM_TABLEWALK_CYC_PREF	Tablewalk qualified for pte prefetches
PM_LSU_FOUR_TABLEWALK_CYC	Cycles when four tablewalks pending on this thread
PM_LSU_TWO_TABLEWALK_CYC	Cycles when two tablewalks pending on this thread
RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 175

Event	Description
PM_LSU0_PRIMARY_ERAT_HIT	Primary ERAT hit
PM_LSU1_PRIMARY_ERAT_HIT	Primary ERAT hit
PM_LSU2_PRIMARY_ERAT_HIT	Primary ERAT hit
PM_LSU3_PRIMARY_ERAT_HIT	Primary ERAT hit
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 176

Event	Description
PM_LSU2_LDF	LS2 Scalar Loads
PM_LSU3_LDF	LS3 Scalar Loads
PM_LSU2_LDX	LS0 Vector Loads
PM_LSU3_LDX	LS1 Vector Loads
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 177

Event	Description
PM_LSU0_NCLD	LS0 Non cachable Loads counted at finish
PM_LSU1_NCLD	LS1 Non cachable Loads counted at finish
PM_LSU_NCST	Non cachable Stores sent to nest
PM_SNOOP_TLBIE	TLBIE snoop
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 178

Event	Description
PM_DSLB_MISS	Data SLB Misses, total of all segment sizes
PM_ISLB_MISS	Instruction SLB Miss Total of all segment sizes
PM_LS0_ERAT_MISS_PREF	LS0 Erat miss due to prefetch
PM_LS1_ERAT_MISS_PREF	LS1 Erat miss due to prefetch
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 179

Event	Description
PM_LSU_SRQ_SYNC	A sync in the SRQ ended
PM_LSU_SET_MPRED	Line already in cache at reload time
PM_LSU_SRQ_SYNC_CYC	A sync is in the SRQ (edge detect to count)
PM_SEC_ERAT_HIT	Secondary ERAT Hit
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 180

Event	Description
PM_UP_PREF_L3	Micropartition prefetch
PM_UP_PREF_POINTER	Micropartition pointer prefetches
PM_DC_COLLISIONS	DATA Cache collisions
PM_TEND_PEND_CYC	TEND latency per thread
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 181

Event	Description
PM_TM_FAIL_DISALLOW	TM fail disallow
PM_TM_FAIL_TLBIE	TLBIE hit bloom filter
PM_TM_FAIL_TX_CONFLICT	Transactional conflict from LSU, whatever gets reported to TEXAS
PM_TM_FAIL_NON_TX_CONFLICT	Non transactional conflict from LSU, whatever gets reported to TEXAS
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 182

Event	Description
PM_LSU0_TM_L1_HIT	Load TM hit in L1
PM_LSU1_TM_L1_HIT	Load TM hit in L1
PM_LSU2_TM_L1_HIT	Load TM hit in L1
PM_LSU3_TM_L1_HIT	Load TM hit in L1
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 183

Event	Description
PM_LSU0_TM_L1_MISS	Load TM L1 miss
PM_LSU1_TM_L1_MISS	Load TM L1 miss
PM_LSU2_TM_L1_MISS	Load TM L1 miss
PM_LSU3_TM_L1_MISS	Load TM L1 miss
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 184

Event	Description
PM_LSU0_TMA_REQ_L2	Addr only req to L2 only on the first one indication that Load footprint is not expanding
PM_LSU1_TMA_REQ_L2	Address only req to L2 only on the first one, indication that Load footprint is not expanding
PM_LSU2_TMA_REQ_L2	Addr only req to L2 only on the first one, indication that Load footprint is not expanding
PM_LSU3_TMA_REQ_L2	Addr only req to L2 only on the first one, indication that Load footprint is not expanding
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 185

Event	Description
PM_LSU0_L1_CAM_CANCEL	LS0 L1 tm cam cancel
PM_LSU1_L1_CAM_CANCEL	LS1 L1 TM cam cancel
PM_LSU2_L1_CAM_CANCEL	LS2 L1 TM cam cancel
PM_LSU3_L1_CAM_CANCEL	LS3 L1 TM cam cancel
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 186

Event	Description
PM_TM_FAIL_CON_TM	TEXAS fail reason @ completion
PM_FAV_TBEGIN	Dispatch time Favored tbegin
PM_TM_FAIL_FOOTPRINT_OVERFLOW	TEXAS fail reason @ completion
PM_TM_FAIL_SELF	TEXAS fail reason @ completion
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 187

Event	Description
PM_TM_TRANS_RUN_CYC	Run cycles in transactional state
PM_TM_TRESUME	TM resume
PM_TM_TRANS_RUN_INST	Instructions completed in transactional state
PM_TM_TSUSPEND	TM suspend
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 188

Event	Description
PM_TM_TBEGIN	TM nested tbegin
PM_TM_TX_PASS_RUN_CYC	Cycles spent in successful transactions
PM_TM_TRANS_RUN_INST	Instructions completed in transactional state
PM_TM_TX_PASS_RUN_INST	Run instructions spent in successful transactions.
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 189

Event	Description
PM_TM_FAIL_CONF_NON_TM	TEXAS fail reason @ completion
PM_TM_BEGIN_ALL	TM any tbegin
PM_L2_TM_ST_ABORT_SISTER	TM marked store abort
PM_TM_END_ALL	TM any tend
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 190

Event	Description
PM_NESTED_TEND	Completion time nested tend
PM_NON_FAV_TBEGIN	Dispatch time non favored tbegin
PM_OUTER_TBEGIN	Completion time outer tbegin
PM_OUTER_TEND	Completion time outer tend
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 191

Event	Description
PM_MEM_READ	Reads from Memory from this lpar (includes data/instruction/translate/I1 prefetch/instruction prefetch). Includes L4
PM_MEM_PREF	Memory prefetch for this lpar. Includes L4
PM_MEM_RWITM	Memory rwitm for this lpar
PM_MEM_CO	Memory castouts from this lpar
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 192

Event	Description
PM_CHIP_PUMP_CPRED	Initial and Final Pump Scope was chip pump (prediction=correct) for all data types excluding data prefetch (demand load, instruction prefetch, instruction fetch, translate)
PM_GRP_PUMP_CPRED	Initial and Final Pump Scope and data sourced across this scope was group pump for all data types excluding data prefetch (demand load, instruction prefetch, instruction fetch, translate)
PM_SYS_PUMP_CPRED	Initial and Final Pump Scope was system pump for all data types excluding data prefetch (demand load, instruction prefetch, instruction fetch, translate)
PM_PUMP_MPRED	Pump misprediction. Counts across all types of pumps for all data types excluding data prefetch (demand load, instruction prefetch, instruction fetch, xlate)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 193

Event	Description
PM_PUMP_CPRED	Pump prediction correct. Counts across all types of pumps for all data types excluding data prefetch (demand load/instruction prefetch/instruction fetch/translate)
PM_GRP_PUMP_MPRED	Final Pump Scope (Group) ended up either larger or smaller than Initial Pump Scope for all data types excluding data prefetch (demand load, instruction prefetch, instruction fetch, translate)
PM_SYS_PUMP_MPRED	Final Pump Scope (system) mispredicted. Either the original scope was too small (Chip/Group) or the original scope was System and it should have been smaller. Counts for all data types excluding data prefetch (demand load, instruction prefetch, instruction fetch, translate)
PM_SYS_PUMP_MPRED_RTY	Final Pump Scope (system) ended up larger than Initial Pump Scope (Chip/Group) for all data types excluding data prefetch (demand load, instruction prefetch, instruction fetch, xlate)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 194

Event	Description
PM_DATA_CHIP_PUMP_CPRED	Initial and Final Pump Scope was chip pump (prediction=correct) for a demand load
PM_DATA_GRP_PUMP_CPRED	Initial and Final Pump Scope was group pump (prediction=correct) for a demand load
PM_DATA_SYS_PUMP_CPRED	Initial and Final Pump Scope was system pump (prediction=correct) for a demand load
PM_DATA_SYS_PUMP_MPRED_RTY	Final Pump Scope (system) ended up larger than Initial Pump Scope (Chip/Group) for a demand load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 195

Event	Description
PM_GRP_PUMP_MPRED_RTY	Final Pump Scope (Group) ended up larger than Initial Pump Scope (Chip) for all data types excluding data prefetch (demand load, instruction prefetch, instruction fetch, translate)
PM_DATA_GRP_PUMP_MPRED	Final Pump Scope (Group) ended up either larger or smaller than Initial Pump Scope for a demand load
PM_DATA_SYS_PUMP_MPRED	Final Pump Scope (system) mispredicted. Either the original scope was too small (Chip/Group) or the original scope was System and it should have been smaller. Counts for a demand load
PM_DATA_PUMP_MPRED	Pump misprediction. Counts across all types of pumps for a demand load
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 196

Event	Description
PM_INST_CHIP_PUMP_CPRED	Initial and Final Pump Scope was chip pump (prediction=correct) for an instruction fetch
PM_INST_GRP_PUMP_CPRED	Initial and Final Pump Scope was group pump (prediction=correct) for an instruction fetch
PM_INST_SYS_PUMP_CPRED	Initial and Final Pump Scope was system pump (prediction=correct) for an instruction fetch
PM_INST_SYS_PUMP_MPRED_RTY	Final Pump Scope (system) ended up larger than Initial Pump Scope (Chip/Group) for an instruction fetch
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 197

Event	Description
PM_INST_GRP_PUMP_MPRED_RTY	Final Pump Scope (Group) ended up larger than Initial Pump Scope (Chip) for an instruction fetch
PM_INST_GRP_PUMP_MPRED	Final Pump Scope (Group) ended up either larger or smaller than Initial Pump Scope for an instruction fetch
PM_INST_SYS_PUMP_MPRED	Final Pump Scope (system) mispredicted. Either the original scope was too small (Chip/Group) or the original scope was System and it should have been smaller. Counts for an instruction fetch
PM_INST_PUMP_MPRED	Pump misprediction. Counts across all types of pumps for an instruction fetch
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 198

Event	Description
PM_ALL_CHIP_PUMP_CPRED	Initial and Final Pump Scope was chip pump (prediction=correct) for all data types (demand load, data prefetch, instruction prefetch, instruction fetch, translate)
PM_ALL_GRP_PUMP_CPRED	Initial and Final Pump Scope and data sourced across this scope was group pump for all data types (demand load, data prefetch, instruction prefetch, instruction fetch, translate)
PM_ALL_SYS_PUMP_CPRED	Initial and Final Pump Scope was system pump for all data types (demand load, data prefetch, instruction prefetch, instruction fetch, translate)

Event	Description
PM_ALL_PUMP_MPRED	Pump misprediction. Counts across all types of pumps for all data types (demand load, data prefetch, instruction prefetch, instruction fetch, translate)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 199

Event	Description
PM_ALL_PUMP_CPRED	Pump prediction correct. Counts across all types of pumps for all data types (demand load, data prefetch, instruction prefetch, instruction fetch, translate).
PM_ALL_GRP_PUMP_MPRED	Final Pump Scope (Group) ended up either larger or smaller than Initial Pump Scope for all data types (demand load, data prefetch, instruction prefetch, instruction fetch, translate)
PM_ALL_SYS_PUMP_MPRED	Final Pump Scope (system) mispredicted. Either the original scope was too small (Chip/Group) or the original scope was System and it should have been smaller. Counts for all data types (demand load, data prefetch, instruction prefetch, instruction fetch, translate)
PM_ALL_SYS_PUMP_MPRED_RTY	Final Pump Scope (system) ended up larger than Initial Pump Scope (Chip/Group) for all data types (demand load, data prefetch, instruction prefetch, instruction fetch, translate)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 200

Event	Description
PM_DATA_ALL_CHIP_PUMP_CPRED	Initial and Final Pump Scope was chip pump (prediction=correct) for either demand loads or data prefetch
PM_DATA_ALL_GRP_PUMP_CPRED	Initial and Final Pump Scope was group pump (prediction=correct) for either demand loads or data prefetch
PM_DATA_ALL_SYS_PUMP_CPRED	Initial and Final Pump Scope was system pump (prediction=correct) for either demand loads or data prefetch
PM_DATA_ALL_SYS_PUMP_MPRED_RTY	Final Pump Scope (system) ended up larger than Initial Pump Scope (Chip/Group) for either demand loads or data prefetch
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 201

Event	Description
PM_ALL_GRP_PUMP_MPRED_RTY	Final Pump Scope (Group) ended up larger than Initial Pump Scope (Chip) for all data types (demand load, data prefetch, instruction prefetch, instruction fetch, translate)
PM_DATA_ALL_GRP_PUMP_MPRED	Final Pump Scope (Group) ended up either larger or smaller than Initial Pump Scope for either demand loads or data prefetch
PM_DATA_ALL_SYS_PUMP_MPRED	Final Pump Scope (system) mispredicted. Either the original scope was too small (Chip/Group) or the original scope was System and it should have been smaller. Counts for either demand loads or data prefetch
PM_DATA_ALL_PUMP_MPRED	Pump misprediction. Counts across all types of pumps for either demand loads or data prefetch

Event	Description
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 202

Event	Description
PM_INST_ALL_CHIP_PUMP_CPRED	Initial and Final Pump Scope was chip pump (prediction=correct) for instruction fetches and prefetches
PM_INST_ALL_GRP_PUMP_CPRED	Initial and Final Pump Scope was group pump (prediction=correct) for instruction fetches and prefetches
PM_INST_ALL_SYS_PUMP_CPRED	Initial and Final Pump Scope was system pump (prediction=correct) for instruction fetches and prefetches
PM_INST_ALL_SYS_PUMP_MPRED_RTY	Final Pump Scope (system) ended up larger than Initial Pump Scope (Chip/Group) for instruction fetches and prefetches
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 203

Event	Description
PM_INST_ALL_GRP_PUMP_MPRED_RTY	Final Pump Scope (Group) ended up larger than Initial Pump Scope (Chip) for instruction fetches and prefetches
PM_INST_ALL_GRP_PUMP_MPRED	Final Pump Scope (Group) ended up either larger or smaller than Initial Pump Scope for instruction fetches and prefetches
PM_INST_ALL_SYS_PUMP_MPRED	Final Pump Scope (system) mispredicted. Either the original scope was too small (Chip/Group) or the original scope was System and it should have been smaller. Counts for instruction fetches and prefetches
PM_INST_ALL_PUMP_MPRED	Pump misprediction. Counts across all types of pumps for instruction fetches and prefetches
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 204

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_FAB_RSP_BKILL_CYC	Cycles L2 RC took for a bkill
PM_MRK_FAB_RSP_CLAIM_RTY	Sampled store did a rwitm and got a rty
PM_MRK_FAB_RSP_BKILL	Marked store had to do a bkill
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 205

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_FAB_RSP_DCLAIM_CYC	Cycles L2 RC took for a dclaim
PM_MRK_FAB_RSP_DCLAIM	Marked store had to do a dclaim
PM_MRK_FAB_RSP_RD_RTY	Sampled L2 reads retry count
PM_RUN_INST_CMPL	Run instructions

Event	Description
PM_RUN_CYC	Run cycles

Group 206

Event	Description
PM_MRK_FAB_RSP_RD_T_INTV	Sampled Read got a T intervention
PM_MRK_FAB_RSP_RWITM_RTY	Sampled store did a rwitm and got a rty
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_FAB_RSP_RWITM_CYC	Cycles L2 RC took for a rwitm
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 207

Event	Description
PM_LSU0_LARX_FIN	Larx finished in LSU pipe0
PM_LSU1_LARX_FIN	Larx finished in LSU pipe1
PM_LSU2_LARX_FIN	Larx finished in LSU pipe2
PM_LSU3_LARX_FIN	Larx finished in LSU pipe3
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 208

Event	Description
PM_STCX_FAIL	STCX failed
PM_STCX_LSU	STCX executed reported at sent to nest
PM_LARX_FIN	Larx finished
PM_LSU_SRQ_EMPTY_CYC	ALL threads srq empty
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 209

Event	Description
PM_IERAT_RELOAD	Number of I ERAT reloads
PM_IERAT_RELOAD_4K	IERAT Miss
PM_IERAT_RELOAD_64K	IERAT Reloaded (Miss) for a 64k page
PM_IERAT_RELOAD_16M	IERAT Reloaded (Miss) for a 16M page
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 210

Event	Description
PM_CYC	Cycles
PM_LSU_DERAT_MISS	DERAT Reloaded due to a DERAT miss
PM_DTLB_MISS	Data PTEG reload
PM_ITLB_MISS	ITLB Reloaded
PM_RUN_INST_CMPL	Run instructions

Event	Description
PM_RUN_CYC	Run cycles

Group 211

Event	Description
PM_DATA_PUMP_CPRED	Pump prediction correct. Counts across all types of pumps for a demand load
PM_TLB_MISS	TLB Miss (I + D)
PM_TLBIE_FIN	TLBIE finished
PM_LSU_ERAT_MISS_PREF	Erat miss due to prefetch, on either pipe
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 212

Event	Description
PM_TABLEWALK_CYC	Cycles when a tablewalk (I or D) is active
PM_TABLEWALK_CYC_PREF	Tablewalk qualified for pte prefetches
PM_DATA_TABLEWALK_CYC	Tablewalk Cycles (could be 1 or 2 active)
PM_LD_MISS_L1	Load Missed L1
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 213

Event	Description
PM_INST_IMC_MATCH_CMPL	IMC Match Count (Not architected in P8)
PM_INST_FROM_L1	Instruction fetches from L1
PM_INST_IMC_MATCH_DISP	Matched Instructions Dispatched
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 214

Event	Description
PM_EE_OFF_EXT_INT	EE off and external interrupt
PM_EXT_INT	External interrupt
PM_TB_BIT_TRANS	Timebase event
PM_CYC	Cycles
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 215

Event	Description
PM_L1_DEMAND_WRITE	Instruction demand sectors written into IL1
PM_IC_PREF_WRITE	Instruction prefetch written into IL1
PM_IBUF_FULL_CYC	Cycles No room in ibuff

Event	Description
PM_BANK_CONFLICT	Read blocked due to interleave conflict. The ifar logic will detect an interleave conflict and kill the data that was read that cycle.
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 216

Event	Description
PM_IC_DEMAND_L2_BHT_REDIRECT	L2 I cache demand request due to BHT redirect, branch redirect (2 bubbles 3 cycles)
PM_IC_DEMAND_L2_BR_REDIRECT	L2 I cache demand request due to branch Mispredict (15 cycle path)
PM_IC_DEMAND_REQ	Demand Instruction fetch request
PM_IC_INVALIDATE	Ic line invalidated
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 217

Event	Description
PM_GRP_IC_MISS_NONSPEC	Group experienced non speculative I cache miss
PM_L1_ICACHE_MISS	Demand iCache Miss
PM_IC_RELOAD_PRIVATE	Reloading line was brought in private for a specific thread. Most lines are brought in shared for all eight threads. If RA does not match then invalidates and then brings it shared to other thread. In P7 line brought in private, then line was invalidate
PM_LSU_L1_SW_PREF	Software L1 Prefetches, including SW Transient Prefetches, on both pipes
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 218

Event	Description
PM_IC_PREF_CANCEL_HIT	Prefetch canceled due to icache hit
PM_IC_PREF_CANCEL_L2	L2 Squashed request
PM_IC_PREF_CANCEL_PAGE	Prefetch canceled due to page boundary
PM_IC_PREF_REQ	Instruction prefetch requests
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 219

Event	Description
PM_DATA_ALL_GRP_PUMP_MPRED_RTY	Final Pump Scope (Group) ended up larger than Initial Pump Scope (Chip) for either demand loads or data prefetch
PM_ST_FWD	Store forwards that finished
PM_L1_ICACHE_RELOADED_PREF	Counts all Icache prefetch reloads (includes demand turned into prefetch)
PM_L1_ICACHE_RELOADED_ALL	Counts all Icache reloads includes demand, prefetchm prefetch turned into demand and demand turned into prefetch

Event	Description
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 220

Event	Description
PM_1LPAR_CYC	Number of cycles in single lpar mode. All threads in the core are assigned to the same lpar.
PM_2LPAR_CYC	Cycles in 2 lpar mode. Threads 0 3 belong to Lpar0 and threads 4 7 belong to Lpar1
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_4LPAR_CYC	Number of cycles in 4 LPAR mode. Threads 0 1 belong to lpar0, threads 2 3 belong to lpar1, threads 4 5 belong to lpar2, and threads 6 7 belong to lpar3
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 221

Event	Description
PM_FUSION_TOC_GRP0_1	One pair of instructions fused with TOC in Group0
PM_FUSION_TOC_GRP0_2	Two pairs of instructions fused with TOC in Group0
PM_FUSION_TOC_GRP0_3	Three pairs of instructions fused with TOC in Group0
PM_FUSION_TOC_GRP1_1	One pair of instructions fused with TOX in Group1
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 222

Event	Description
PM_FUSION_VSX_GRP0_1	One pair of instructions fused with VSX in Group0
PM_FUSION_VSX_GRP0_2	Two pairs of instructions fused with VSX in Group0
PM_FUSION_VSX_GRP0_3	Three pairs of instructions fused with VSX in Group0
PM_FUSION_VSX_GRP1_1	One pair of instructions fused with VSX in Group1
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 223

Event	Description
PM_GCT_UTIL_1_2_ENTRIES	GCT Utilization 1 2 entries
PM_GCT_UTIL_3_6_ENTRIES	GCT Utilization 3 6 entries
PM_GCT_UTIL_7_10_ENTRIES	GCT Utilization 7 10 entries
PM_GCT_UTIL_11_14_ENTRIES	GCT Utilization 11 14 entries
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 224

Event	Description
PM_GCT_UTIL_15_17_ENTRIES	GCT Utilization 15 17 entries

Event	Description
PM_GCT_UTIL_18_ENTRIES	GCT Utilization 18+ entries
PM_DISP_HOLD_GCT_FULL	Dispatch Hold Due to no space in the GCT
PM_GCT_MERGE	Group dispatched on a merged GCT empty. GCT entries can be merged only within the same thread
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 225

Event	Description
PM_STALL_END_GCT_EMPTY	Count ended because GCT went empty
PM_GCT_EMPTY_CYC	No itags assigned either thread (GCT Empty)
PM_CYC	Cycles
PM_FLUSH_DISP	Dispatch flush
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 226

Event	Description
PM_FPU0_FCONV	Convert instruction executed
PM_FPU0_FEST	Estimate instruction executed
PM_FPU0_FRSP	Round to single precision instruction executed
PM_LSU_LDF	FPU loads only on LS2/LS3 ie LU0/LU1
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 227

Event	Description
PM_FPU1_FCONV	Convert instruction executed
PM_FPU1_FEST	Estimate instruction executed
PM_FPU1_FRSP	Round to single precision instruction executed
PM_LSU_LDX	Vector loads can issue only on LS2/LS3
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 228

Event	Description
PM_GRP_NON_FULL_GROUP	GROUPs where we did not have 6 non branch instructions in the group(ST mode), in SMT mode 3 non branches
PM_GRP_TERM_2ND_BRANCH	There were enough instructions in the Ibuffer, but 2nd branch ends group
PM_GRP_TERM_FPU_AFTER_BR	There were enough instructions in the Ibuffer, but FPU OP IN same group after a branch terminates a group, cant do partial flushes
PM_GRP_TERM_NOINST	Do not fill every slot in the group, Not enough instructions in the Ibuffer. This includes cases where the group started with enough instructions, but some got knocked out by a cache miss or branch redirect (which would also empty the Ibuffer).
PM_RUN_INST_CMPL	Run instructions

Event	Description
PM_RUN_CYC	Run cycles

Group 229

Event	Description
PM_SHL_CREATED	Store Hit Load Table Entry Created
PM_SHL_ST_CONVERT	Store Hit Load Table Read Hit with entry Enabled
PM_SHL_ST_DISABLE	Store Hit Load Table Read Hit with entry Disabled (entry was disabled due to the entry shown to not prevent the flush)
PM_EAT_FULL_CYC	Cycles No room in EAT
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 230

Event	Description
PM_GRP_BR_MPRED_NONSPEC	Group experienced non speculative branch redirect
PM_GRP_TERM_OTHER	There were enough instructions in the Ibuffer, but the group terminated early for some other reason, most likely due to a First or Last.
PM_GRP_TERM_SLOT_LIMIT	There were enough instructions in the Ibuffer, but 3 src RA/RB/RC , 2 way crack caused a group termination
PM_EAT_FORCE_MISPRED	XL form branch was mispredicted due to the predicted target address missing from EAT. The EAT forces a mispredict in this case since there is no predicated target to validate. This is a rare case that may occur when the EAT is full and a branch is issue
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 231

Event	Description
PM_CLB_HELD	CLB Hold: Any Reason
PM_LINK_STACK_INVALID_PTR	A flush were LS ptr is invalid, results in a pop , A lot of interrupts between push and pops
PM_LINK_STACK_WRONG_ADD_PRED	Link stack predicts wrong address, because of link stack design limitation
PM_ISU_REF_FXU	FXU ISU reject from either pipe
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 232

Event	Description
PM_DATA_GRP_PUMP_MPRED_RTY	Final Pump Scope (Group) ended up larger than Initial Pump Scope (Chip) for a demand load
PM_UTHROTTL	Cycles in which instruction issue throttle was active in ISU
PM_IFETCH_THROTTL	Cycles in which Instruction fetch throttle was active
PM_IFU_L2_TOUCH	L2 touch to update MRU on a line
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 233

Event	Description
PM_ISU_REJECTS_ALL	All isu rejects could be more than 1 per cycle
PM_ISU_REJECT_SAR_BYPASS	Reject because of SAR bypass
PM_ISU_REJECT_SRC_NA	ISU reject due to source not available
PM_ISU_REJECT_RES_NA	ISU reject due to resource not available
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 234

Event	Description
PM_LSU0_STORE_REJECT	LS0 store reject
PM_LSU1_STORE_REJECT	LS1 store reject
PM_LSU2_REJECT	LSU2 reject
PM_LSU_REJECT_LHS	LSU Reject due to LHS (up to 4 per cycle)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 235

Event	Description
PM_LSU_REJECT_LMQ_FULL	LSU reject due to LMQ full (4 per cycle)
PM_LSU_REJECT_ERAT_MISS	LSU Reject due to ERAT (up to 4 per cycles)
PM_LSU2_REJECT	LSU2 reject
PM_LSU3_REJECT	LSU3 reject
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 236

Event	Description
PM_LSU_REJECT	LSU Reject (up to 4 per cycle)
PM_LSU1_REJECT	LSU1 reject
PM_MRK_LSU_REJECT_ERAT_MISS	LSU marked reject due to ERAT (up to 2 per cycle)
PM_MRK_LSU_REJECT	LSU marked reject (up to 2 per cycle)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 237

Event	Description
PM_ISU_REF_FX0	FX0 ISU reject
PM_ISU_REF_LS0	LS0 ISU reject
PM_ISU_REF_LS1	LS1 ISU reject
PM_ISU_REF_LS2	LS2 ISU reject
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 238

Event	Description
PM_ISU_REF_LS3	LS3 ISU reject
PM_ISU_REJ_VS0	VS0 ISU reject
PM_ISU_REJ_VS1	VS1 ISU reject
PM_ISU_REF_FX1	FX1 ISU reject
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 239

Event	Description
PM_SWAP_CANCEL	SWAP cancel, rtag not available
PM_SWAP_CANCEL_GPR	SWAP cancel , rtag not available for gpr
PM_SWAP_COMPLETE	SWAP cast in completed
PM_SWAP_COMPLETE_GPR	SWAP cast in completed fpr gpr
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 240

Event	Description
PM_MEM_LOC_THRESH_LSU_MED	Local memory above threshold for data prefetch
PM_CASTOUT_ISSUED	Castouts issued
PM_CASTOUT_ISSUED_GPR	Castouts issued GPR
PM_MEM_LOC_THRESH_LSU_HIGH	Local memory above threshold for LSU medium
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 241

Event	Description
PM_TEND_PEND_CYC	TEND latency per thread
PM_TABORT_TRECLAIM	Completion time tabortnoncd, tabortcd, treclaim
PM_LSU_NCLD	Count at finish so can return only on LS0 or LS1
PM_CYC	Cycles
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 242

Event	Description
PM_ISYNC	Isync count per thread
PM_LWSYNC	Threaded version, IC Misses where we got EA dir hit but no sector valids were on. ICBI took line out.
PM_LWSYNC_HELD	LWSYNC held at dispatch
PM_FLUSH_DISP_SYNC	Dispatch Flush: Sync
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 243

Event	Description
PM_MRK_INST_ISSUED	Marked instruction issued
PM_MRK_INST_DECODED	Marked instruction decoded
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_INST_FROM_L3MISS	Marked instruction was reloaded from a location beyond the local chiplet
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 244

Event	Description
PM_MRK_INST_DISP	The thread has dispatched a randomly sampled marked instruction
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_INST_FIN	Marked instruction finished
PM_MRK_INST_TIMEO	Marked Instruction finish timeout (instruction lost)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 245

Event	Description
PM_GRP_MRK	Instruction Marked
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_LSU_MRK_DERAT_MISS	DERAT Reloaded (Miss)
PM_MRK_GRP_CMPL	Marked instruction finished (completed)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 246

Event	Description
PM_GRP_MRK	Instruction Marked
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_GRP_NTC	Marked group ntc cycles.
PM_MRK_GRP_IC_MISS	Marked Group experienced I cache miss
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 247

Event	Description
PM_MRK_L1_ICACHE_MISS	Sampled Instruction suffered an icache Miss
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MULT_MRK	Multiple marked instructions
PM_MRK_INST_TIMEO	Marked Instruction finish timeout (instruction lost)
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 248

Event	Description
PM_MRK_BR_CMPL	Branch Instruction completed
PM_MRK_BRU_FIN	BRU marked instruction finish
PM_MRK_BACK_BR_CMPL	Marked branch instruction completed with a target address less than current instruction address
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 249

Event	Description
PM_MRK_BR_TAKEN_CMPL	Marked Branch Taken completed
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_BR_MPRED_CMPL	Marked Branch Mispredicted
PM_BR_MRK_2PATH	Marked two path branch
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 250

Event	Description
PM_SYNC_MRK_BR_LINK	Marked Branch and link branch that can cause a synchronous interrupt
PM_BR_PRED_LSTACK_CMPL	Completion Time Event. This event can also be calculated from the direct bus as follows: if_pc_br0_br_pred(0) AND (not if_pc_br0_pred_type).
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_INST_CMPL	Marked instruction completed
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 251

Event	Description
PM_SYNC_MRK_BR_MPRED	Marked Branch mispredict that can cause a synchronous interrupt
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_BR_PRED_CR_CMPL	Completion Time Event. This event can also be calculated from the direct bus as follows: if_pc_br0_br_pred(1)='1'.
PM_BR_PRED_CCACHE_CMPL	Completion Time Event. This event can also be calculated from the direct bus as follows: if_pc_br0_br_pred(0) AND if_pc_br0_pred_type.
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 252

Event	Description
PM_MRK_ST_CMPL	Marked store completed and sent to nest

Event	Description
PM_MRK_L2_RC_DISP	Marked Instruction RC dispatched in L2
PM_MRK_ST_FWD	Marked st forwards
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 253

Event	Description
PM_MRK_LSU_FLUSH_LRQ	Flush: (marked) LRQ
PM_MRK_LSU_FLUSH_SRQ	Flush: (marked) SRQ
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_LSU_FLUSH_UST	Unaligned Store Flush on either pipe
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 254

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_LSU_FLUSH	Flush: (marked) : All Cases
PM_MRK_LSU_FLUSH_ULD	Flush: (marked) Unaligned Load
PM_MRK_LSU_FLUSH_UST	Flush: (marked) Unaligned Store
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 255

Event	Description
PM_MRK_FIN_STALL_CYC	Marked instruction Finish Stall cycles (marked finish after NTC) (use edge detect to count)
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_STCX_FAIL	Marked stcx failed
PM_MRK_LARX_FIN	Larx finished
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 256

Event	Description
PM_MRK_RUN_CYC	Marked run cycles
PM_MRK_DFU_FIN	Decimal Unit marked Instruction Finish
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_LSU_FIN	LSU marked instr finish
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 257

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_FXU_FIN	FXU marked instruction finish
PM_MRK_CRU_FIN	IFU non branch finished
PM_CRU_FIN	IFU Finished a (non branch) instruction
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 258

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_NTF_FIN	Marked next to finish instruction finished
PM_MRK_VSU_FIN	VSU marked instr finish
PM_ISU_REJ_VSU	VSU ISU reject from either pipe
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 259

Event	Description
PM_MRK_L1_RELOAD_VALID	Marked demand reload
PM_LSU_L1_PREF	HW initiated, include SW streaming forms as well, include SW streams as a separate event
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DCACHE_RELOAD_INTV	Combined Intervention event
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 260

Event	Description
PM_SYNC_MRK_L2HIT	Marked L2 Hits that can throw a synchronous interrupt
PM_MRK_L2_RC_DISP	Marked Instruction RC dispatched in L2
PM_MRK_L2_RC_DONE	Marked RC done
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 261

Event	Description
PM_SYNC_MRK_PROBE_NOP	Marked probeNops which can cause synchronous interrupts
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_BACK_BR_CMPL	Marked branch instruction completed with a target address less than current instruction address
PM_PROBE_NOP_DISP	ProbeNops dispatched
PM_RUN_INST_CMPL	Run instructions

Event	Description
PM_RUN_CYC	Run cycles

Group 262

Event	Description
PM_SYNC_MRK_FX_DIVIDE	Marked fixed point divide that can cause a synchronous interrupt
PM_MRK_FXU_FIN	FXU marked instruction finish
PM_ISU_REF_FX0	FX0 ISU reject
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 263

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DERAT_MISS_64K	Marked Data ERAT Miss (Data TLB Access) page size 64K
PM_MRK_DERAT_MISS_16M	Marked Data ERAT Miss (Data TLB Access) page size 16M
PM_MRK_DERAT_MISS_16G	Marked Data ERAT Miss (Data TLB Access) page size 16G
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 264

Event	Description
PM_MRK_DERAT_MISS_4K	Marked Data ERAT Miss (Data TLB Access) page size 4K
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DERAT_MISS	Erat Miss (TLB Access) All page sizes
PM_MRK_DTLB_MISS_16M	Marked Data TLB Miss page size 16M
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 265

Event	Description
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_MRK_DTLB_MISS_4K	Marked Data TLB Miss page size 4k
PM_MRK_DTLB_MISS_64K	Marked Data TLB Miss page size 64K
PM_MRK_DTLB_MISS	Marked dtlb miss
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Group 266

Event	Description
PM_MRK_DTLB_MISS_16G	Marked Data TLB Miss page size 16G
PM_MRK_DTLB_MISS_4K	Marked Data TLB Miss page size 4k
PM_DTLB_MISS	Data PTEG reload
PM_INST_CMPL	Number of PowerPC Instructions that completed. PPC Instructions Finished (completed).
PM_RUN_INST_CMPL	Run instructions
PM_RUN_CYC	Run cycles

Appendix C. HPC Toolkit environment variables

Table 23 is an alphabetical listing all of the HPC Toolkit environment variables and the command or API for which they are used. For a description of each environment variable, see the referenced command or API.

Table 23. HPC Toolkit environment variables

Environment variable	Used by:
GPM_EVENT_SET GPM_METRIC_SET	"gpmlist - Lists the available events and metrics" on page 114
GPM_ASC_OUTPUT GPM_ENABLE_TRACE GPM_PRINT GPM_STDOUT GPM_VIZ_OUTPUT	"gpm_terminate - Generate GPU Performance Monitoring statistics and trace files and shut down the GPM runtime environment" on page 142
HPC_EXCEPTION_COUNT	"hpcrun - Launch a program to collect profiling or trace data" on page 122
HPC_EXCEPTION_METRIC	"hpcrun - Launch a program to collect profiling or trace data" on page 122
HPC_OUTPUT_NAME	<ul style="list-style-type: none"> "hpccount - Report hardware performance counter statistics for an application" on page 116 "hpcstat - Reports a system-wide summary of hardware performance counter statistics" on page 124 "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 153
HPC_TRACE_MAX_BUFFERS	"hpcrun - Launch a program to collect profiling or trace data" on page 122
HPC_TRACE_STORE	"hpcrun - Launch a program to collect profiling or trace data" on page 122
HPC_UNIQUE_FILE_NAME	<ul style="list-style-type: none"> "hpccount - Report hardware performance counter statistics for an application" on page 116 "hpcstat - Reports a system-wide summary of hardware performance counter statistics" on page 124 "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 153
HPM_AGGREGATE	<ul style="list-style-type: none"> "hpccount - Report hardware performance counter statistics for an application" on page 116 "hpcrun - Launch a program to collect profiling or trace data" on page 122 "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 153
HPM_ASC_OUTPUT	<ul style="list-style-type: none"> "hpccount - Report hardware performance counter statistics for an application" on page 116 "hpcstat - Reports a system-wide summary of hardware performance counter statistics" on page 124 "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 153

Table 23. HPC Toolkit environment variables (continued)

Environment variable	Used by:
HPM_COUNTING_MODE	<ul style="list-style-type: none"> “hpccount - Report hardware performance counter statistics for an application” on page 116 “hpcstat - Reports a system-wide summary of hardware performance counter statistics” on page 124 “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 153
HPM_ENABLE_GPM	Instructs the HPM module to drive GPU event and metric profiling.
HPM_EVENT_SET	<ul style="list-style-type: none"> “hpccount - Report hardware performance counter statistics for an application” on page 116 “hpcstat - Reports a system-wide summary of hardware performance counter statistics” on page 124 “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 153
HPM_EXCLUSIVE_VALUES	“hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 153
HPM_PRINT_FORMULA	<ul style="list-style-type: none"> “hpccount - Report hardware performance counter statistics for an application” on page 116 “hpcstat - Reports a system-wide summary of hardware performance counter statistics” on page 124 “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 153
HPM_PRINT_TASK	<ul style="list-style-type: none"> “hpccount - Report hardware performance counter statistics for an application” on page 116 “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 153
HPM_ROUND_ROBIN_CLUSTER	<ul style="list-style-type: none"> “hpccount - Report hardware performance counter statistics for an application” on page 116 “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 153
HPM_STDOUT	<ul style="list-style-type: none"> “hpccount - Report hardware performance counter statistics for an application” on page 116 “hpcstat - Reports a system-wide summary of hardware performance counter statistics” on page 124 “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 153
HPM_VIZ_OUTPUT	<ul style="list-style-type: none"> “hpccount - Report hardware performance counter statistics for an application” on page 116 “hpcstat - Reports a system-wide summary of hardware performance counter statistics” on page 124 “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 153

Table 23. HPC Toolkit environment variables (continued)

Environment variable	Used by:
IHPCT_BASE	<ul style="list-style-type: none"> • “gpm_init - Initialize the GPU Performance Monitor runtime environment” on page 133 • “gpm_start - Identify the starting point of an instrumented region of code” on page 136 • “gpm_stop - Identify the end point of an instrumented region of code” on page 139 • “gpm_terminate - Generate GPU Performance Monitoring statistics and trace files and shut down the GPM runtime environment” on page 142 • “gpm_Tstart - Identify the starting point of an instrumented region of code” on page 145 • “gpm_Tstop - Identify the end point of an instrumented region of code” on page 148 • “hpcrun - Launch a program to collect profiling or trace data” on page 122 • “hpcstat - Reports a system-wide summary of hardware performance counter statistics” on page 124 • “hpctInst - Instrument applications to obtain performance data” on page 128 • “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 153 • “MT_get_allresults - Obtain statistical results” on page 174 • “MT_get_mpi_bytes - Obtain the accumulated number of bytes transferred” on page 182 • “MT_get_mpi_counts - Obtain the the number of times a function was called” on page 183 • “MT_get_mpi_name - Returns the name of the specified MPI function” on page 184 • “MT_get_time - Get the elapsed time” on page 186 • “MT_trace_start, mt_trace_start - Start or resume the collection of trace events” on page 192
LD_LIBRARY_PATH	“hpctInst - Instrument applications to obtain performance data” on page 128
MAX_TRACE_EVENTS	“MT_trace_start, mt_trace_start - Start or resume the collection of trace events” on page 192
MAX_TRACE_RANK	“MT_trace_start, mt_trace_start - Start or resume the collection of trace events” on page 192
MT_BASIC_TRACE	“MT_trace_start, mt_trace_start - Start or resume the collection of trace events” on page 192
OUTPUT_ALL_RANKS	“MT_trace_start, mt_trace_start - Start or resume the collection of trace events” on page 192
POMP_LOOP	“hpctInst - Instrument applications to obtain performance data” on page 128
POMP_PARALLEL	“hpctInst - Instrument applications to obtain performance data” on page 128
POMP_USER	“hpctInst - Instrument applications to obtain performance data” on page 128
TRACE_ALL_EVENTS	“MT_trace_start, mt_trace_start - Start or resume the collection of trace events” on page 192
TRACE_ALL_TASKS	“MT_trace_start, mt_trace_start - Start or resume the collection of trace events” on page 192
TRACEBACK_LEVEL	“MT_trace_start, mt_trace_start - Start or resume the collection of trace events” on page 192

Accessibility features for IBM PE Developer Edition

Accessibility features help users who have a disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

The following list includes the major accessibility features in IBM PE Developer Edition:

- Keyboard-only operation
- Interfaces that are commonly used by screen readers

The *IBM Knowledge Center*, and its related publications, are accessibility-enabled. The accessibility features are described in IBM Knowledge Center (www.ibm.com/support/knowledgecenter).

Keyboard navigation

This product uses standard Microsoft Windows navigation keys.

IBM and accessibility

See the IBM Human Ability and Accessibility Center (www.ibm.com/able) for more information about the commitment that IBM has to accessibility.

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

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, MD-NC119
Armonk, NY 10504-1785
US*

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:

*Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan*

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 jurisdictions 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 websites are provided for convenience only and do not in any manner serve as an endorsement of those

websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide 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:

*IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US*

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.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

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.

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

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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 actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate 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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

© (your company name) (year).

Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_.

Programming interface information

The IBM MPICH product is a complete MPI implementation, based on the MPICH open source project, designed to comply with all the requirements of the Message Passing Interface standard, MPI: A Message-Passing Interface Standard, Version 3.0, University of Tennessee, Knoxville, Tennessee, September 21, 2012. If you believe that IBM MPICH does not comply with the MPI-3 standard, please contact IBM Service.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Intel, Intel Inside (logos), MMX and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of the Open Group in the United States and other countries.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display

or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at <http://www.ibm.com/privacy> and IBM's Online Privacy Statement at <http://www.ibm.com/privacy/details> in the section entitled "Cookies, Web Beacons and Other Technologies", and the "IBM

Privacy policy considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, See IBM’s Privacy Policy at <http://www.ibm.com/privacy> and IBM’s Online Privacy Statement at <http://www.ibm.com/privacy/details> the section entitled “Cookies, Web Beacons and Other Technologies” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at <http://www.ibm.com/software/info/product-privacy>.

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com)® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" (www.ibm.com/legal/copytrade.shtml).

Intel is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Index

A

- about this information xi
- accessibility 275
 - keyboard 275
 - shortcut keys 275
- additional trace controls 96
- aggregator functions
 - understanding 84
- alphabetical listing
 - HPC Toolkit environment variables 271
- APIs 131
 - f_hpm_error 151
 - f_hpminit 153
 - f_hpmstart 158
 - f_hpmstartx 160
 - f_hpmstop 163
 - f_hpmterminate 165
 - f_hpmtstart 167
 - f_hpmtstartx 169
 - f_hpmtstop 172
 - gpm_init 133
 - gpm_start 136
 - gpm_stop 139
 - gpm_terminate 142
 - gpm_tstart 145
 - gpm_tstop 148
 - hpm_error_count 151
 - hpmInit 153
 - hpmStart 158
 - hpmStartx 160
 - hpmStop 163
 - hpmTerminate 165
 - hpmTstart 167
 - hpmTstartx 169
 - hpmTstop 172
 - MT_get_allresults 174
 - MT_get_calleraddress 177
 - MT_get_callerinfo 178
 - MT_get_elapsed_time 180
 - MT_get_environment 181
 - MT_get_mpi_bytes 182
 - MT_get_mpi_counts 183
 - MT_get_mpi_name 184
 - MT_get_mpi_time 185
 - MT_get_time 186
 - MT_get_tracebufferinfo 187
 - MT_output_text 188
 - MT_output_trace 189
 - MT_trace_event 190
 - mt_trace_start 192
 - MT_trace_start 192
 - mt_trace_stop 194
 - MT_trace_stop 194

C

- Call Graph analysis
 - using 67
- collecting performance data
 - from a subset of application tasks 4

- command and API reference 111
- commands 113
 - gpmList 114
 - hpccount 75, 116
 - hpcrun 122
 - hpcstat 76, 124
 - hpctInst 128
- compiling and linking with libmpitrace 95
- considerations for MPI programs 83, 84, 85
 - distributor and aggregator interfaces 85
 - plug-ins shipped with the tool kit 84
- controlling traced tasks 96
- conventions
 - file naming 197
- conventions and terminology xi
- CPU hardware counter multiplexing
 - understanding 78
- customizing MPI profiling data 97

D

- data file naming
 - HPC_OUTPUT_NAME 197
 - HPC_UNIQUE_FILE_NAME 197
- derived metrics 203
 - defined for POWER8 architecture 203
 - understanding 79
- descriptions
 - derived metrics 203
- disability 275
- displaying combined performance data
 - GPU hardware counter profiling 50
- distributor and aggregator interfaces
 - understanding 85
- distributor functions
 - understanding 83

E

- Eclipse PTP
 - using 29

F

- f_hpm_error API 151
- f_hpminit API 153
- f_hpmstart API 158
- f_hpmstartx API 160
- f_hpmstop API 163
- f_hpmterminate API 165
- f_hpmtstart API 167
- f_hpmtstartx API 169
- f_hpmtstop API 172
- file naming conventions 197
- file naming examples 198

G

- generating a log file 21
- getting the plug-ins to work 87
- GPM API
 - using 91
- GPM preload library
 - using 90
- GPM under the control of HPM
 - using 92
- gpm_init API 133
- gpm_start API 136
- gpm_stop API 139
- gpm_terminate API 142
- gpm_tstart API 145
- gpm_tstop API 148
- gpmlist command 114
- GPU hardware counter profiling
 - displaying combined performance data 50
 - instrumenting the application 47
 - preparing an application 47
 - running the instrumenting application 48
 - viewing data 49
- GPU hardware counters
 - using in HPCT 89
- GPU hardware counters profiling
 - using 47
- graphical performance analysis tools
 - hpctView application 19
 - using 19

H

- handling multithreaded program instrumentation issues 82
 - handling of overlap issues 81
 - hardware performance counter data
 - viewing 45
 - hardware performance counter plug-ins 83
 - hardware performance counter tools 75
 - CPU hardware counter multiplexing 78
 - derived metrics 79
 - hpccount command 75
 - hpcstat command 76
 - inclusive and exclusive event counts 80
 - inheritance, understanding 80
 - measurement overhead 82
 - MFlop issues 79
 - multithreaded program instrumentation 82
 - overlap issues 81
 - parent-child relationships 81
 - using the libhpc library 77
 - hardware performance counters
 - instrumenting your application 107
 - HPC Toolkit
 - introduction 3
 - HPC Toolkit environment variables
 - alphabetical listing 271
 - hpccount command 116
 - using 75
 - hpcrun command 122
 - hpcstat command 124
 - using 76
 - hpctlnst command 128
 - hpctlnst utility
 - instrumenting your application 107
 - hpctView
 - using 26
 - hpctView application
 - instrumenting 29
 - opening the application executable 25
 - preparing an application for analysis 24
 - using 23
 - viewing performance data 38
 - working with the application 25
 - hpm_error_count API 151
 - hpmInit API 153
 - hpmStart API 158
 - hpmStartx API 160
 - hpmStop API 163
 - hpmTerminate API 165
 - hpmTstart API 167
 - hpmTstartx API 169
 - hpmTstop API 172
- ## I
- I/O profiling
 - instrumenting your application 109
 - using 61
 - I/O profiling environment variables
 - setting 99
 - I/O profiling library
 - using 99
 - I/O profiling library module options
 - performance data file naming 104
 - running your application 104
 - specifying 101
 - IBM High Performance Computing Toolkit
 - introduction 3
 - IBM HPC Toolkit
 - introduction 3
 - IBM HPC Toolkit hpctView application
 - installing 17
 - IBM HPC Toolkit xCAT kit
 - installing 16
 - IBM HPCT plug-in for Eclipse PTP
 - installing 17
 - inclusive and exclusive event counts
 - understanding 80
 - inheritance
 - understanding 80
 - installation media contents 13
 - installing
 - hpctView application 15
 - IBM HPC Toolkit hpctView application 17
 - IBM HPCT plug-in for Eclipse PTP 17
 - runtime and the CLI instrumentation tools 15
 - the IBM HPC Toolkit xCAT kit 16
 - installing the IBM HPC Toolkit on Linux systems 15
 - instrumentation models 5
 - instrumented application
 - running 30
 - instrumenting
 - hpctView application 29
 - instrumenting the application
 - GPU hardware counter profiling 47
 - using I/O profiling 61
 - using MPI profiling 53
 - instrumenting your application for hardware performance counters 107
 - instrumenting your application for I/O profiling 109
 - instrumenting your application for MPI profiling 108
 - instrumenting your application using the hpctlnst utility 107

introduction
 IBM High Performance Computing Toolkit 3
introduction to the IBM HPC Toolkit 3

L

libhpc library
 using 77
library module options
 specifying 101
limitations and restrictions xvii
Linux systems
 installing the IBM HPC Toolkit 15
log file
 generating 21

M

measurement overhead
 understanding 82
media contents
 installation 13
MFlop issues
 understanding 79
MPI profiling
 instrumenting your application 108
 using 53
MPI profiling data
 customizing 97
MPI profiling library
 using 95
MPI profiling utility functions
 understanding 98
MPI programs
 considerations 83
 general considerations 83
MPI trace tool
 performance data file naming 98
MT_get_allresults API 174
MT_get_calleraddress API 177
MT_get_callerinfo API 178
MT_get_elapsed_time API 180
MT_get_environment API 181
MT_get_mpi_bytes API 182
MT_get_mpi_counts API 183
MT_get_mpi_name API 184
MT_get_mpi_time API 185
MT_get_time API 186
MT_get_tracebufferinfo API 187
MT_output_text API 188
MT_output_trace API 189
MT_trace_event API 190
mt_trace_start API 192
MT_trace_start API 192
mt_trace_stop API 194
MT_trace_stop API 194
multithreaded program instrumentation
 handling issues 82

O

overlap issues
 handling 81
overview
 instrumentation models 5
 performance measurement tools 4

P

parent-child relationships
 understanding 81
performance data file naming
 HPC_OUTPUT_NAME 197
 HPC_UNIQUE_FILE_NAME 197
 I/O profiling library module options 104
 understanding 98
performance measurement tools 4
platforms and software levels
 supported 11
plug-ins
 getting them to work 87
 hardware performance counter 83
plug-ins shipped
 with the tool kit 84
POWER8 architecture
 derived metrics 203
 derived metrics, events, groups 203
 supported events and groups 204
preparing an application
 for GPU hardware counter profiling 47
preparing an application for analysis 24
preparing an application for MPI profiling 53
preparing an application for profiling 41
 using I/O profiling 61
preparing you application
 using Call Graph Analysis 67
prerequisite information xii, xiii
profiling
 hardware performance counter 41
 preparing an application 41
profiling the MPI calls in an application 53

R

related information xii
 Parallel Tools Platform component xiii
restrictions and limitations xvii
running the application
 using Call Graph Analysis 67
 using I/O profiling 63
 using MPI profiling 55
running the instrumented application 30
 using hardware performance counter profiling 43
running the instrumenting application
 GPU hardware counter profiling 48
running your application
 I/O profiling library module options 104

S

setting I/O profiling environment variables 99
setting the user environment 6
shortcut keys
 keyboard 275
supported events and groups
 POWER8 architecture 204
supported platforms and software levels 11

T

tool kit
 plug-ins shipped 84

- tools
 - hardware performance counter 75
- traced tasks
 - controlling 96
- trademarks 281

U

- understanding
 - distributor and aggregator interfaces 85
- understanding aggregator functions 84
- understanding CPU hardware counter multiplexing 78
- understanding derived metrics 79
- understanding distributor functions 83
- understanding inclusive and exclusive event counts 80
- understanding inheritance 80
- understanding measurement overhead 82
- understanding MFlop issues 79
- understanding MPI profiling utility functions 98
- understanding parent-child relationships 81
- understanding why user-defined plug-ins are useful 85
- user environment
 - setting up 6
- user-defined plug-ins
 - understanding 85
 - why they are useful 85
- using
 - hpctView application 23
- using Call Graph analysis 67
 - viewing gprof data 69
 - viewing the Call Graph 68
- using Call Graph Analysis
 - preparing you application 67
 - running the application 67
 - viewing profile data 67
- using Eclipse PTP 29
- using GPU hardware counter profiling 47
- using GPU hardware counters in HPCT 89
 - using GPM under the control of HPM 92
 - using the GPM API 91
 - using the GPM preload library 90
- using hardware performance counter profiling 41
 - running the instrumented application 43
- using hardware performance counters in profiling
 - instrumenting the application 41
- using hpctView 26
- using I/O profiling 61
 - instrumenting the application 61
 - preparing an application for profiling 61
 - running the application 63
 - viewing I/O Data 64
- using MPI profiling 53
 - instrumenting the application 53
 - preparing an application 53
 - running the application 55
 - viewing MPI profiling data 57
 - viewing MPI traces 58
- using the I/O profiling library 99
 - collecting performance data from a subset of application tasks 4
 - performance data file naming 104
 - running your application 104
 - setting I/O profiling environment variables 99
 - specifying module options 101
- using the libhpc library 77
- using the MPI profiling library 95, 96
 - additional trace controls 96

- using the MPI profiling library (*continued*)
 - compiling and linking with libmpitrace 95
 - customizing MPI profiling data 97
 - performance data file naming 98
 - understanding MPI profiling utility functions 98

V

- viewing data
 - GPU hardware counter profiling 49
- viewing gprof data
 - using Call Graph analysis 69
- viewing hardware performance counter data 45
- viewing I/O Data
 - using I/O profiling 64
- viewing MPI profiling data
 - using MPI profiling 57
- viewing MPI traces
 - using MPI profiling 58
- viewing performance data
 - hpctView application 38
- viewing profile data
 - using Call Graph Analysis 67
- viewing the Call Graph
 - using Call Graph analysis 68

W

- what are derived metrics 79
- who should read this information xi
- working with the hpctView application 25



Product Number: 5765-PD2

Printed in USA

SC23-7287-01

