Software Development Kit for Multicore Acceleration
Version 3.1

Performance Tools Reference

SC34-2565-00

Software Development Kit for Multicore Acceleration
Version 3.1

Performance Tools Reference

SC34-2565-00

Note

Before using this information and the product it supports, read the information in “Notices” on page 73.

Edition notice

This edition applies to version 3, release 1, modification 0 of the IBM Software Development Kit for Multicore Acceleration (Product number 5724-S84) and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC33-8427-01.

Contents

Preface

The IBM® Software Development Kit for Multicore Acceleration Version 3.1 (SDK 3.1) is a complete package of tools to enable you to program applications for the Cell Broadband Engine™ (Cell/B.E.) processor. The Software Development Kit for Multicore Acceleration is composed of development tool chains, software libraries and sample source files, a system simulator, and a Linux® kernel, all of which fully support the capabilities of the Cell Broadband Engine Architecture.

About this publication

This publication describes the various performance tools provided to optimize your applications for the Cell Broadband Engine Architecture.

Supported platforms

Cell Broadband Engine Architecture applications can be developed on these platforms.

- X86
- X86_64
- 64-bit PowerPC® (PPC64)
- IBM BladeCenter® QS21
- IBM BladeCenter QS22

Supported languages

The supported languages are:

- C/C++
- Assembler
- Fortran
- ADA (Power Processing Element (PPE) Only)

Note: Although C++ and Fortran are supported, take care when you write code for the Synergistic Processing Units (SPUs) because many of the C++ and Fortran libraries are too large for the 256 KB local storage memory available.

Beta-level (unsupported) environments

This publication contains documentation that may be applied to certain environments on an "as-is" basis. Those environments are not supported by IBM, but wherever possible, workarounds to problems are provided in the respective forums.

Getting support

The SDK is supported through the CBEA architecture forum on the developerWorks® Web site at http://www.ibm.com/developerworks/power/cell/.

Commercial support from IBM is available if you purchased the SDK from Passport Advantage®.

The XL C/C++ compilers are supported through the XL compiler Web site. See http://www.ibm.com/software/awdtools/xlcpp/support/.

The XL Fortran compiler is supported through the XL compiler Web site. See http://www.ibm.com/software/awdtools/fortran/support/.

This version of the SDK supersedes all versions of the SDK that were available from alphaWorks®.

If you have a problem with the IBM BladeCenter QS21 or BladeCenter QS22 that you think is caused by running the Barcelona Supercomputing Center kernel on Fedora 9, report a bug to the public `cbe-oss-dev@ozlabs.org` mailing list. Archives and subscription information for this list are available from https://ozlabs.org/mailman/listinfo/cbe-oss-dev/. Since Fedora 9 is not a supported IBM product, IBM provides no guaranteed reply or target dates for fixes for this configuration. Commercial support is available for Red Hat Enterprise Linux (RHEL) 5.2.

Related documentation

This topic helps you find related information.

Document location

Links to documentation for the SDK are provided on the IBM developerWorks Web site located at:

`http://www.ibm.com/developerworks/power/cell/`

Click the **Docs** tab.

The following documents are available, organized by category:

Architecture

- *Cell Broadband Engine Architecture*
- *Cell Broadband Engine Registers*
- *SPU Instruction Set Architecture*

Standards

- *C/C++ Language Extensions for Cell Broadband Engine Architecture*
- *Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification*
- *SIMD Math Library Specification for Cell Broadband Engine Architecture*
- *SPU Application Binary Interface Specification*
- *SPU Assembly Language Specification*

Programming

- *Cell Broadband Engine Programmer's Guide*
- *Cell Broadband Engine Programming Handbook*
- *Cell Broadband Engine Programming Tutorial*

Library

- *Accelerated Library Framework for Cell Broadband Engine Programmer's Guide and API Reference*

- *Basic Linear Algebra Subprograms Programmer's Guide and API Reference*
- *Data Communication and Synchronization for Cell Broadband Engine Programmer's Guide and API Reference*
- *Example Library API Reference*
- *Fast Fourier Transform Library Programmer's Guide and API Reference*
- *LAPACK (Linear Algebra Package) Programmer's Guide and API Reference*
- *Mathematical Acceleration Subsystem (MASS)*
- *Monte Carlo Library Programmer's Guide and API Reference*
- *SDK 3.0 SIMD Math Library API Reference*
- *SPE Runtime Management Library*
- *SPE Runtime Management Library Version 1 to Version 2 Migration Guide*
- *SPU Runtime Extensions Library Programmer's Guide and API Reference*
- *Three dimensional FFT Prototype Library Programmer's Guide and API Reference*

Installation

- *SDK for Multicore Acceleration Version 3.1 Installation Guide*

Tools

- *Getting Started - XL C/C++ for Multicore Acceleration for Linux*
- *Compiler Reference - XL C/C++ for Multicore Acceleration for Linux*
- *Language Reference - XL C/C++ for Multicore Acceleration for Linux*
- *Programming Guide - XL C/C++ for Multicore Acceleration for Linux*
- *Installation Guide - XL C/C++ for Multicore Acceleration for Linux*
- *Getting Started - XL Fortran for Multicore Acceleration for Linux*
- *Compiler Reference - XL Fortran for Multicore Acceleration for Linux*
- *Language Reference - XL Fortran for Multicore Acceleration for Linux*
- *Optimization and Programming Guide - XL Fortran for Multicore Acceleration for Linux*
- *Installation Guide - XL Fortran for Multicore Acceleration for Linux*
- *Performance Analysis with the IBM Full-System Simulator*
- *IBM Full-System Simulator User's Guide*
- *IBM Visual Performance Analyzer User's Guide*

IBM PowerPC Base

- *IBM PowerPC Architecture Book*
  - *Book I: PowerPC User Instruction Set Architecture*
  - *Book II: PowerPC Virtual Environment Architecture*
  - *Book III: PowerPC Operating Environment Architecture*
- *IBM PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*

Chapter 1. Cell Broadband Engine Performance Debugging Tool (PDT)

This section describes the Cell Broadband Engine (Cell/B.E.) Performance Debugging Tool (PDT) usage, how to configure it, and how to enable the tool.

Introduction

Introduction to the Performance Debugging Tool

The Cell/B.E. environment enables several levels of parallelism:
- A cluster of Cell/B.E. processors executing a parallel application
- A Cell/B.E. running a parallel program that simultaneously utilizes the Power Processor Element (PPE) and the eight Synergistic Processor Elements (SPEs)
- A PPE or an SPE utilizing the vector units

Writing applications that utilize such multilevel parallelism effectively, and understanding the performance behavior of such a system, is a challenge. The objective of the Cell/B.E. PDT is to provide programmers with a means of analyzing the execution of such a system and tracking problems in order to optimize execution time and utilization of resources.

This version of the PDT addresses performance debugging of one Cell/B.E. processor with two PPEs that share the main memory, run under the same Linux operating system, and share up to 16 SPEs. The PDT also enables event tracing on the x86_64 (Opteron) as well as the Hybrid environment..

Performance analysis is usually based on profiling or tracing. The PDT provides tracing means for recording significant events during program execution and maintaining the sequential order of events. The main objective of the PDT is to provide the ability to trace events of interest, in real time, and record relevant data from the SPEs and PPE. This objective is achieved by instrumenting the code that implements key functions of the events on the SPEs and PPE and collecting the trace records. This instrumentation requires additional communication between the SPEs and PPE as trace records are collected in the PPE memory. Tracing 16 SPEs using one central PPE might lead to a heavy load on the PPE, and therefore, might influence the performance of your application. The PDT is designed to reduce the tracing execution load and provide a means for throttling the tracing activity on the PPE and each SPE. In addition, the SPE tracing code size is minimized so that it fits into the small SPE local store.

Tracing is enabled at the application level (user space). After the application has been enabled, the tracing facility trace data is gathered every time the application runs.

Note: Tracing can produce a very large amount of data.

Components high level description

The Cell/B.E. PDT package contains a tracing facility and a trace analyzer (TA) which is part of the Visual Performance Analyzer (VPA) tool.

In addition to the TA, other tools may process and analyze the trace files generated by the tracing facility. The SDK includes the PDT trace Reader/post-processor (PDTR) tool that provides trace-event listings and various summary reports, including lock analysis.

Tracing facility

How to enable tracing of events.

The following SDK libraries have trace-enabled versions available, that you can use for event tracing:
- On the PPE: DaCS, ALF, libspe2, and libsync
- On the SPE: DaCS, ALF, libsync, the `spu_mfcio` header file, and the overlay manager
- On X86_64 (Opteron): DaCS and ALF

Performance events are captured by the SDK functions that are already instrumented for tracing. These functions include:
- SPEs activation
- DMA transfers
- Synchronization
- Signaling
- User-defined events

A full list of events is found in the reference configuration file of PDT. You must compile and link statically-linked applications with the trace-enabled libraries. You do not need to rebuild applications which use shared libraries.

Note: The SPE code is always statically linked, and therefore must be recompiled and linked.

Before each application run, you can configure the PDT to trace events of interest. You can also use the PDT API to dynamically control the tracing.

During the application run, the PPE and SPE trace records are gathered in a memory-mapped (`mmap`) file in the PPE memory. These records are written into the file system when appropriate. The event-records order is maintained in this file. The SPEs use efficient DMA transfers to write the trace records into the `mmap` file. The trace records are written in the trace file using a format that is set by an external definition (using an XML file). The PDTR and TA tools, that use PDT traces as input, use the same format definition for visualization and analysis.

Trace processing

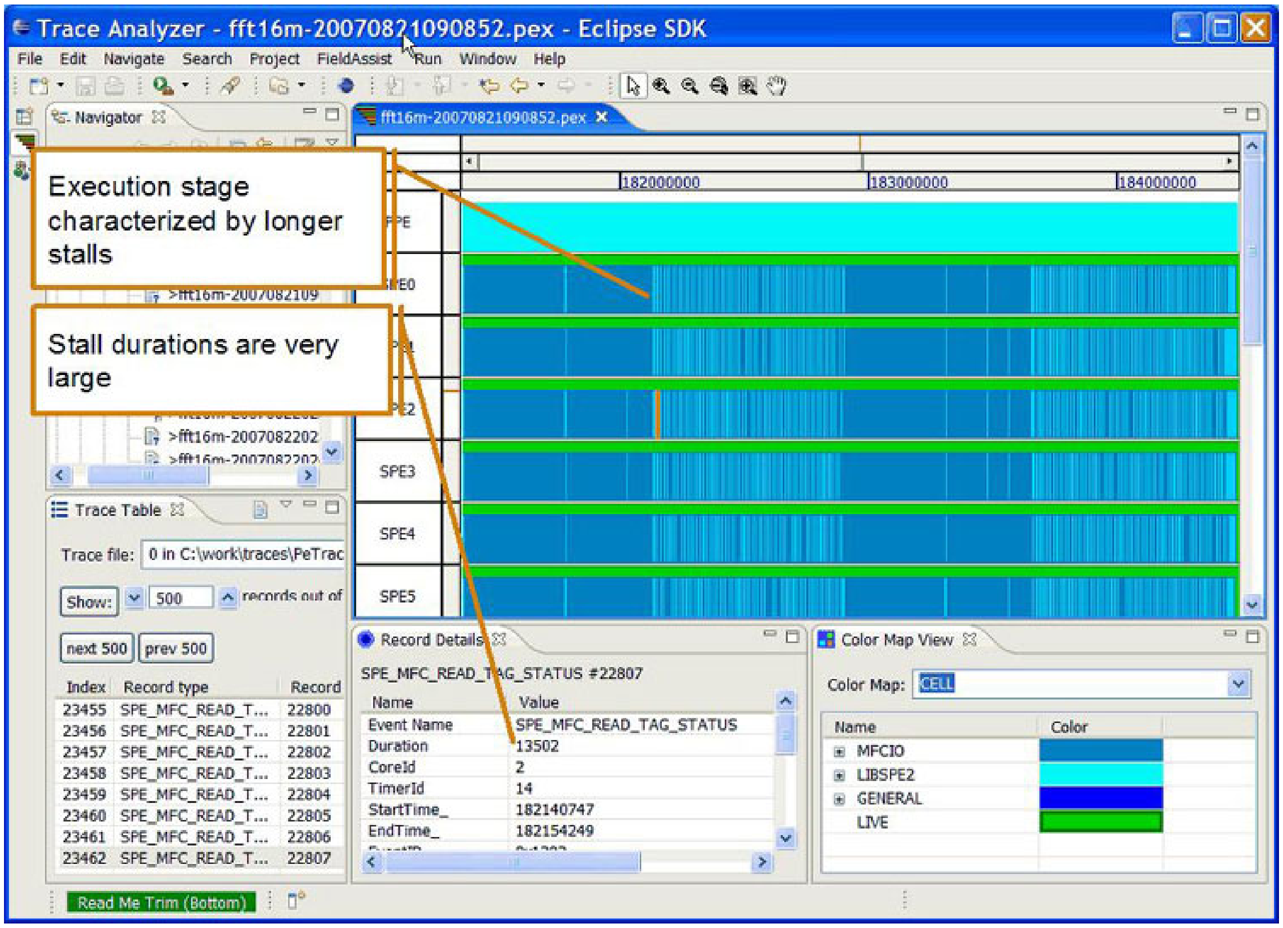
The TA processes the trace for analysis and visualization. This processing generates interval records from some of the event records in the trace (for example, SPE thread life intervals, wait intervals, and so on) as well as adding context parameters (for example, estimated wall clock time, unique SPE thread IDs, and so on) to individual records.

The SDK also provides the PDTR Trace Analyzer program. This command-line tool runs natively on the Cell/B.E.and is provided to view and post-process PDT traces (which enables local PDT trace analysis). The PDTR tool provides both sequential and event-by-event PDT trace text output. It also provides postprocessing summary reports based on specific instrumentation events.

Visualization

Traces can be viewed with the Eclipse-based VPA tool using the Trace Analyzer perspective. This tool provides a means for graphical and textual visualization of trace events over time. You can view the details that have been recorded in the trace for each event.

The graphical timeline view in the trace visualization has time as the *x* axis, and the PPE and SPEs as rows in the *y* axis. Each event interval is shown as a colored bar the width of which represents its time duration. The colors in the color legend determine the type of event interval. The following figure is a snapshot of the TA GUI for the FFT16M workload.


Trace Analyzer - fft16m-20070821090852.pex - Eclipse SDK
File Edit Navigate Search Project FieldAssist Run Window Help
Navigator
fft16m-20070821090852.pex
Execution stage characterized by longer stalls
Stall durations are very large
182000000
183000000
184000000
PPE
SPE0
SPE1
SPE2
SPE3
SPE4
SPE5
Trace Table
next 500
prev 500
Record Details
Color Map View
Read Me Trim (Bottom)

Trace file: 0 in C:\work\traces\PeTrac

Show: 500 records out of

Index	Record type	Record
23455	SPE_MFC_READ_T...	22800
23456	SPE_MFC_READ_T...	22801
23457	SPE_MFC_READ_T...	22802
23458	SPE_MFC_READ_T...	22803
23459	SPE_MFC_READ_T...	22804
23460	SPE_MFC_READ_T...	22805
23461	SPE_MFC_READ_T...	22806
23462	SPE_MFC_READ_T...	22807

SPE_MFC_READ_TAG_STATUS #22807

Name	Value
Event Name	SPE_MFC_READ_TAG_STATUS
Duration	13502
CoreId	2
TimerId	14
StartTime_	182140747
EndTime_	182154249

Color Map: CELL

Name	Color
MFCIO	🟦
LIBSPE2	🟦
GENERAL	🟦
LIVE	🟩

Figure 1. Trace Analyzer GUI for the FFT16M workload

The textual trace overview lists all the PPE and SPE events in order of appearance in the trace. If you select an event, it is highlighted the graphical timeline view and the fields of the event record are displayed in the record details view.

For additional information about trace visualization, refer to the *IBM Visual Performance Analyzer User Guide* available from IBM alphaWorks:

`http://www.alphaworks.ibm.com/tech/vpa`

PDT tracing-facility package directory structure

The tracing-facility package is part of the SDK. Use the following tables to locate the directories for the tracing-facility package.

Table 1. Tracing-facility directories on a Cell/B.E. system

Use	**Cell/B.E. Host**
PDT development trace includes	/usr/include/trace
PDT production trace libraries	/usr/lib/trace
PDT production trace 64 bit libraries	/usr/lib64/trace
PDT SPU development trace includes	/usr/spu/include/trace
PDT SPU development trace libraries	/usr/spu/lib/trace
PDT configuration	/usr/share/pdt/config
PDT examples	/opt/cell/sdk/src/pdt-cell-examples.tar

Table 2. Tracing-facility directories on a cross system

Use	**Cross x86 to Cell/B.E.**
PDT development trace includes	/opt/cell/sysroot/usr/include/trace
PDT production trace libraries	/opt/cell/sysroot/usr/lib/trace
PDT production trace 64 bit libraries	/opt/cell/sysroot/usr/lib64/trace
PDT SPU development trace includes	/opt/cell/sysroot/usr/spu/include/trace
PDT SPU development trace libraries	/opt/cell/sysroot/usr/spu/lib/trace
PDT configuration	/opt/cell/sysroot/usr/share/pdt/config
PDT examples	/opt/cell/sdk/src/pdt-cell-examples.tar

Table 3. Tracing-facility directories on an X86_64 (Opteron) system

Use	**Opteron Host**
PDT development trace includes	/usr/include/trace
PDT production trace libraries	/usr/lib/trace
PDT production trace 64 bit libraries	/usr/lib64/trace
PDT configuration	/usr/share/pdt/config
PDT examples	/opt/cell/sdk/src/pdt-opteron-example.tar

Configuring the PDT kernel module (Red Hat Enterprise Linux (RHEL) 5.2 only)

The PDT kernel module is a Linux-extension-kernel module that allows the PDT to be synchronized with the Linux SPE context switches. The kernel module is compiled and linked in the `pdt.ko` file, and is shipped in the `/usr/lib/modules/` directory.

The application loads the PDT kernel module before the tracing starts and removes it when the application ends. Because module insertion and removal require super user (root) permissions, this operation requires the `sudo` facility. The call to the `sudo` facility is integrated within the PDT.

To install the facility, update the `/etc/sudoers` file, using the visudo editor, as follows:

```
ALL ALL=(ALL) NOPASSWD: /sbin/insmod /usr/lib/modules/pdt.ko, /sbin/rmmod pdt
```

Note:

1. If an application terminates abnormally, the kernel module remains loaded. It is removed at the next run, and a new instance is inserted.
2. The context switch notification for RHEL 5.2 is implemented so that only one user can activate the tracing facility at a time. Therefore, multiuser usage is forbidden, but there is no protection against it.
3. If the kernel module is not installed, the TA does not show the SPE utilization correctly because the events are not aligned in time; however, a trace is created.
4. The kernel module is not required in Fedora 9 or newer versions.

PDT example usage

The PDT package contains a sample application in the `/opt/cell/sdk/src/pdt-cell-examples.tar` file. After installation, it is recommended that you compile and run the application, and then use the TA and PDTR tools to examine the PDT output.

Each example includes a `pdt` script file that you can use to compile and run an available application. You can also study the script as a usage example, or modified it to run your own applications. For example, use the SDK `make.footer` file or an explicit `Makefile`. You can use the `Makefile` provided with the examples as a reference. It contains a sample configuration file, and a full reference configuration file named pdt_cbe_configuration.xml is located in the /usr/share/pdt/config directory. Copy the configuration file to your working directories and modify it as needed.

Enabling the PDT tracing facility for a new application

How to enable tracing in your application.

The PDT tracing facility is designed to minimize the effort that is needed to enable the tracing facility for a given application. In most cases, no code changes or additions are necessary. However, because the SPE code is statically linked and the PDT uses a different `spu_mfcio.h` file, you must recompile the SPE code. In addition, if the SPE executable is embedded in the PPE code, you must relink the PPE code.

Compilation and application building

You only need to change your source code if user-defined events or dynamic-trace control are used. For a cross-development environment, root (/) is defined as `/opt/cell/sysroot/`.

The examples provided as part of the PDT package can be used as reference to the following sections.

Compiling SPE code

How to compile SPE code

To compile SPE code, do the following:

1. Add the following compilation flags to your Makefile:

```
-Dmain=_pdt_main -Dexit=_pdt_exit -DMFCIO_TRACE
```

Note: Add `-DLIBSYNC_TRACE` if any libsync function (from the libsync.h include file) is used inline.

2. Add the compiler include option (`-I/usr/spu/include/trace`) as the first location in the compile command line.
3. Add the libtrace.a library (from the /usr/spu/lib/trace directory) and any other instrumented libraries, to the linkage of the executable file.
4. If overlays are used, add `spu_ovl.o` (from the `/usr/spu/lib/trace` directory) to the `spu` linking stage.

Certain SPU applications, in combination with certain libraries, may present a linking problem when using the PDT. For example, when instrumenting with the PDT, an SPU application that uses a wrapping library (such as ALF), can create a circular dependency. The solution is to specify the trace library twice: once before the wrapping library and once after it. For example:

```
spu-gcc -o alf_hello_world_spumain_spu.o -L/usr/spu/lib/trace -ltrace \
   -lalf -L/usr/spu/lib/trace -Wl, -N -ltrace
```

You can also use the following option to enable the linker for circular-dependencies search:

```
-Wl,-\( -lalf -ltrace -Wl,-\)
```

The following two statements give you more examples of how to compile SPE code.

```
spu-gcc -O -c spe_test.c -W -Wno-main -g -I/usr/spu/include/trace \
   -Dmain=_pdt_main -Dexit=_pdt_exit -DMFCIO_TRACE -DLIBSYNC_TRACE \
   -I. -I/opt/cell/sdk/usr/spu/include
```

```
spu-gcc -O spe_test.o -o spe_test -W -Wno-main -g \
   -I/usr/spu/include/trace -Dmain=_pdt_main -Dexit=_pdt_exit \
   -DMFCIO_TRACE -DLIBSYNC_TRACE -I. -I/opt/cell/sdk/usr/spu/include \
   -Wl,-N -Wl,-q -L/usr/spu/lib/trace -ltrace
```

Compiling PPE code

Compilation of the PPE code is needed only if the tracing API is used in the program or when inline instrumented library functions are used.

To compile PPE code, do the following:

1. If any libsync functions (from the libsync.h include file) is used inline, add the following compilation flags to your Makefile:

   ```
   -DLIBSYNC_TRACE
   ```

2. Add the compiler `include` option (`-I/usr/include/trace`) as the first location in the compile command line.
3. Add the `-L/usr/lib/trace` (or `-L/usr/lib64/trace` for 64 bit applications) flags to the linkage process. If using the trace-enabled libsync,also add `-L/opt/cell/sdk/usr/lib[64]/trace`

To enable the Trace Analyzer to link between events and the source code, rebuild the application using the linking relocation flags (for SPE and PPE). Use the `-g` flag for compilation and the `-g -Wl,-q` flags for linking. Do not use the `-s` stripping option.

A linking problem may occur when some SPU applications are combined with certain librarie whenusing the PDT. For example, when instrumenting with the PDT, an SPU application that uses a wrapping library, such as ALF, can create a circular dependency. The solution is to specify the trace library twice: once before the wrapping library and once after. For example:

```
spu-gcc -o alf_hello_world_spumain_spu.o -L/usr/spu/lib/trace -ltrace \
   -lalf -L/usr/spu/lib/trace -Wl, -N -ltrace
```

An alternative is to use the following option to enable the linker for circulate-dependencies search:

```
-Wl,-\( -lalf -ltrace -Wl,-\)
```

Running a program with trace-enabled PDT libraries

The PDT package provides a script file called *pdt* which is located in the pdt examples tar file, /opt/cell/sdk/src/pdt-examples.tar. You can use it to compile and run trace-enabled applications, or refer to it as a reference to the following explicit instructions.

You can copy the pdt script to your development environment, modify it if necessary, and run it. Here are the usage instructions for this:

```
pdt [options] execution_file [execution_parameters]

OPTIONS
 -h
  Print help

 -m  [[ -f make_file_ name] -b [32 | 64]]
 Build the application after clean, using Makefile or the optional make_file_name.
 Compilation target architecture width can be provided using the -b option (32 or 64 bits).
  Default is 64 bits. This option is based on the use of the COMPILATION_BITS eneivronment
  variable in the make file.

 -c configuration_file
  Set the PDT configuration file for this run. Default is
  /usr/share/pdt/config/pdt_cbe_configuration.

 -o output_directory
 Set the output directory for the trace files. Default is the current directory.

 -p prefix
  Set the prefix for the trace files. Default is none (use the prefix that is set
  in the configuration file).
```

To enable the program to use the PDT libraries after the rebuild process, do the following:

1. Set the following environment variables for the PDT before you run the program:

   LD_LIBRARY_PATH

   This is a colon separated list of directories that the runtime loader search for libraries. The full path to the trace library location is required: /usr/lib/trace (or /usr/lib64/trace for 64 bit applications).

   Add also the directories of other trace-enabled libraries such as libsync, located in /opt/cell/sdk/usr/lib/trace (or /opt/cell/sdk/usr/lib64/trace for 64 bit applications). The list is colon separated. For example, `LD_LIBRARY_PATH=/usr/lib64/trace:/opt/cell/sdk/usr/lib64/trace`.

   PDT_CONFIG_FILE

   The full path to the PDT configuration file for the application run. The PDT package contains a pdt_cbe_configuration.xml file in the /usr/share/pdt/config directory that can be used without modification, or copied and modified for each application run. For more information on PDT configuration, see "Configuring the PDT for an application run" on page 9 below.

PDT_TRACE_OUTPUT (optional)
: The full path to the PDT output directory. The directory must exist before the application runs.

PDT_OUTPUT_PREFIX (optional)
: This variable is used to add a prefix to the PDT output file names.

2. Modify the pdt configuration file for the application, if desired.
3. Run the program.

The PDT libraries produces trace files in a directory that is defined by the environment variable *PDT_TRACE_OUTPUT*. If this environment variable is not defined, the output location is taken from the definition provided by the **output_dir** attribute in the PDT configuration file. If neither is defined, the current path is used. The output directory must exist before the application runs, and the user must have write access to this directory. PDT creates the following files in that output directory at each run.

Table 4. Output directory files

File Name	Description
`<prefix>-<app_name>-yyyymmddhhmmss.pex`	Meta file of the trace event definitions
`<prefix>-<app_name>-yyyymmddhhmmss.maps`	This is a copy of the maps file from /proc/<pid>/. It is used for address-to-name resolution done by the PDTR tool (pdtr command).
`<prefix>-<app_name>-yyyymmddhhmmss.<N>.trace`	Trace file or files

Note:
1. `<prefix>` is provided by the optional *PDT_OUTPUT_PREFIX* environment variable.
2. `<app_name>` is a string provided in the PDT configuration file **application_name** attribute.
3. `yyyymmddhhmmss` is the date and time when the application started (`trace_init()` time).
4. `<N>` is the sequential number of the trace file. The maximum size of each trace file is 32 MB.

Running a program with SPE profiling

The PDT libraries provide an option to produce event records by profiling SPE programs. You can profile PPE programs with oProfile. The PDTR tool has the ability to process these records and show the profile information.

To perform SPE profiling on a program, do the following:

1. Compile your program with the Tracing Facility enabled.
2. Activate profiling in the SPE. To do this, modify the configuration XML file as follows:
   a. Locate the SPE *<configuration name="SPE">* tag.
   b. Under the that tag, set the profile statement to *<profile active="true" rate="100"/>*.
   c. The profiling sampling rate that is set here is limited to 20000 samples per second. Minimize other event tracing during profiling at rates higher than 2000 samples per second.

3. Run your program, then run the PDTR tool on the trace results.

Note: When you use SPE profiling with PDT, ensure that atomic and DMA operations in application code and any library code in use are interrupt safe. Disable interrupts during these operations.

Configuring the PDT for an application run

An XML configuration file is used to configure the PDT. The PDT tracing facility that is built into the application at run time reads the configuration file defined by the *PDT_CONFIG_FILE* environment variable. The /usr/share/pdt/config directory contains a reference configuration file, pdt_cbe_configuration.xml. Copy this file and modify it for the requirements of your application.

If you have installed the DaCS, ALF, or libsync libraries, the /usr/share/pdt/config directory contains additional reference configuration files for each installed library.

Table 5. Reference files for additional libraries

Reference file	Library
pdt_dacs_config_cell.xml	DaCS
pdt_dacs_config_hybrid.xml	DaCS for Hybrid
pdt_alf_config_cell.xml	ALF
pdt_alf_config_hybrid.xml	ALF for Hybrid
pdt_libsync_config.xml	libsync

The first line of the configuration file contains the application name. This name is used as a prefix for the PDT output files. To correlate the output name with a specific run, the name can be changed before each run. The PDT output directory is also defined in the **output_dir** attribute. This location will be used if the *PDT_TRACE_OUTPUT* environment variable is not defined.

The first section of the file, `<groups>`, defines the groups of events for the run. The events of each group are defined in other definition files (which are also in XML format), and included in the configuration file. These files reside in the /usr/share/pdt/config directory. They are provided with the instrumented library and you should not modify them. Each of these files contains a list of events with the definition of the trace-record data for each event. Note that some of the events define an interval with *StartTime* and *EndTime*, and some are single events in which the *StartTime* is 0 and the *EndTime* is set to the event time. The names of the trace-record fields match the names defined by the API functions, and each event is related to an API function. There are two types of records: one for the PPE and one for the SPE. Each of these record types has a different header that is defined in a separate file: pdt_ppe_event_header.xml for the PPE and pdt_spe_event_header.xml for the SPE.

The SDK provides instrumentation for the following libraries. The traced events with a description of each record are provided in the following XML files:

GENERAL (pdt_general.xml)
: These are the general trace events such as trace start, trace stop, etc. Tracing of these events is always active.

LIBSPE2 (pdt_lbspe2.xml)
These are the LIBSPE2 events.

SPU_MFCIO (pdt_mfcio.xml)
These are the spu_mfcio events that are defined in the /usr/spu/include/trace/spu_mfcio.h header file.

LIBSYNC (pdt_libsync.xml)
These are the mutex events that are part of the libsync library.

DACS (pdt_dacs*.xml)
These are the DaCS events, separated into three groups of events. For more information, see the Data Communication and Synchronization programmer's guide and API reference.

ALF (pdt_alf*.xml)
These are the ALF events, separated into three groups of events. For more information, see the Data Communication and Synchronization programmer's guide and API reference.

The second section of the file contains the tracing control definitions for each type of processor. The PDT is made ready for the hybrid environment so each processor has a host, `<host>`. On each processor, several groups of events can be activated in the group control, <*groupControl*>. Each group is divided into subgroups, and each subgroup, <*subgroup*>, has a set of events. Each group, subgroup, and event has an *active* attribute that can be either true or false. This attribute affects tracing as follows:

- If a group is active, all of its events will be traced.
- If a group is not active, and the subgroup is active, all of its subgroup's events will be traced.
- If a group and subgroup are not active, and an event is active, that event will be traced.

You can create new group of events for new libraries that are in use. Defined these groups using XML files like those above. Give each group a unique name and ID. Once created, add them to the other files in the /usr/share/pdt/config directory. Next, add a reference to these group to the two sections in the configuration file used by the application. Instrument the libraries using the library developer API functions described below,

Note: It is highly recommended that tracing be enabled only for those events that are of interest. Depending on the number of processors involved, programs might produce events at a high rate. If this scenario occurs, the number of traced events might also be very high.

Using the tracing API

The tracing API used by the PDT is a generic API. It enables any implementation of a tracing facility: the PDT is only one possible implementation. For example, you can implement a tracing facility that only prints a trace.

The PDT API is intended for library developers who want to add the tracing facility to their libraries. Because tracing is done invisibly to users, application programmers use only a subset of the API. This subset provides an interface for user defined events and dynamic trace control.

Essential definitions

The `trace_basic_defs.h` and `trace_defs.h` files contain the definitions for the PDT API parameters located in the /usr/include/trace directory.

Application programmer API

Use this API only if you want to create user defined events records in a trace, or if you need dynamic trace control at run time.

User-defined events

These APIs are for user-defined events.

Include the trace_user.h file in your program.

void trace_user_event(trace_payload_p payload)
: This function writes a trace record with the provided payload. The user event ID is defined by the user and should be the first element (long) in **trace_ payload_t**, pointed by payload. On the SPE, the payload array must be aligned on a 16-byte boundary.

trace_interval_p trace_user_interval_entry()
: This function initiates a user-defined interval that terminates when trace_user_interval_exit() is called. This function does not write a trace record. The function returns a pointer to **trace_interval** type that must be used as a parameter to the trace_user_interval_exit() function.

void trace_user_interval_exit(trace_interval_p user_interval, trace_payload_p payload)
: This function terminates a user-defined interval that was initiated by trace_user_interval_entry(). The trace_user_interval_entry() function provides the trace interval pointer. This function writes a trace record with the provided payload. On the SPE, the payload array must be aligned on a 16-byte boundary.

Dynamic trace control

These APIs control which events are traced at run time.

Include the trace_dynamic.h file in your program.

void trace_event_control(trace_event_id_t event_id, trace_bool_t value);
: This API changes the control state of an event according to the requested value: `trace_false = off` or `trace_true = on`. The event IDs are provided in the events-groups XML files.

void trace_group_control(trace_group_t group, trace_bool_t value);
: This API changes the state of all the group's events according to the requested value: `trace_false = off` or `trace_true = on`. The event IDs are provided in the events-groups XML files.

trace_bool_t trace_event_get_control(trace_event_id_t event_id);
: This API returns the current control state of an event.

Generic profiling interface with user defined payload

The PDT enables an application to activate a generic profiling. This service allows the application to define a set of variables to be periodically recorded in the trace.

Include the trace_profile.h header file in your application.

Service activation interface

Syntax:

```
trace_profile_handle trace_profile_register(
   trace_event_id_t event_id, int tm_sec,
   void (*callback)(trace_payload_p payload_ptr)
```

The following table shows the interface parameters.

Table 6. trace_profile_handle trace_profile_register parameters

Parameter	Explanation
event_id	The event ID that has been defined in the group.
tm_sec	An interval in 1 millisecond units.
callback	A callback function that is called by the service with two output parameters, to be filled by the callback function.
payload_ptr	A pointer to the trace record payload of size *trace_payload_t* which is defined as 80 characters. The callback function should fill the payload with data.

The function returns the *trace_profile_handle* handle to be used for service control. *NULL* is returned if the registration failed.

Service termination interface

Syntax:

```
void trace_profile_unregister(
   trace_profile_handle handle)
```

The following table shows the interface parameters.

Table 7. trace_profile_unregister parameters

Parameter	Explanation
handle	The handle returned by the trace_profile_register() function.

Profile trace start interface

Syntax:

```
void trace_profile_start(trace_profile_handle handle)
```

Profile trace pause interface

Syntax:

```
void trace_profile_pause(trace_profile_handle handle)
```

Using the trace functions

Setting a callback function through this API enables this function to manage the recorded information as you require, for example to gather data from various sources and use mutex operation if necessary.

You can activate several profiling intervals, but the resource usage is high. It's best to use only one interval, or one service activation per run. Because the profiling time resolution is 1 msec, we recommend that you use an interval of more than 10 msec (profiling of less than 100 times a second) to reduce the influence of the profiling code on your application.

On the SPE, the profiling timing is based on the SPU timer which enables up to four intervals, while the application can use some or all of those intervals. Take care in using this resource.

Library developer API

An extended API is provided for library developers. This API is used to instrument a library with generic tracing code. A library may be assigned with one or more groups of events. Each group ID should be obtained from IBM. This requirement enables the usage of any combination of groups in the same run. The PDT can handle up 256 groups: currently 10 groups are in use. Each group can have up to 64 events.

Trace facility control

Use this function to initiate the tracing facility.

Include the trace_control.h header file in your applications.

void trace_init(void);
This function initiates the tracing facility. Call it before you call any traced event. You can call it more than one time within an application, but it will be activated only once.

Events recording

Use these functions to create a single trace record and a trace record that defines an interval.

long trace_event(trace_event_id_t event_id, int argc, trace_payload_p payload, const char *format, unsigned int level);
This function writes a trace record with the provided payload and returns an event count. On the SPE, this array must be aligned on a 16-byte boundary.

event id
This is the event identifier. In the PDT, the **event id** is combined from the **group id** (one byte) and the **specific id** within this group (0-63).

argc The number of parameters in the payload.

format
A string that describes the payload parameters using printf format.

pavload
A pointer to the data to be recorded in the trace record.

level The number of calls from the application until this function is called. It enables the tracing facility to provide the program counter at the application level in order to link between the event and the source code.

This function returns a sequential event count.

trace_interval_p trace_interval_entry(trace_event_id_t event_id, unsigned int level); This function initiates an interval that terminates when trace_interval_exit() is called. This function does not write a trace record.

event id
This is the event identifier. In the PDT, the **event id** is combined from the **group id** (one byte) and the **specific id** within this group (0-63).

level The number of calls from the application until this function is called. It enables the tracing facility to provide the program counter at the application level in order to link between the event and the source code.

The function returns a pointer to a trace_interval type that must be used as a parameter to the trace_interval_exit function.

long trace_interval_exit(trace_interval_p interval, int argc, trace_payload_p payload, const char *format);
This function terminates an interval that was initiated when trace_interval_entry() was called. The pointer to the trace interval type is provided by the trace_interval_entry() function.

Interval
This pointer to the **trace_interval** type is provided by the trace_interval_entry() function.

argc The number of parameters in the payload.

format
A string that describes the payload parameters using printf format.

pavload
A pointer to the data to be recorded in the trace record.

This function writes a trace record with the provided payload. On the SPE, this array must be aligned on a 16–byte boundary. The function returns a sequential event count.

Note: On the SPE, interrupts are disabled during the functions that create trace records. This is essential because the interrupts handler may create a traced event that can override the record creation.

Define event and interval class

The additional trace record attribute *pdtRecordClass* is enabled as an option. This attribute states the event or interval record class. The class enables the trace analyzer (PDTR or TA) to perform special processing on the events of the same class.

You can define the following types of classes (lowercase):

blocking_wait_interval
An interval caused by a blocking event

busy_wait_interval
A polling interval

working_interval
While processing is performed

user_interval
Allow summarization of user intervals

profile_event
Direct the Trace Analyzer to provide a graphical timeline visualization of the payload data

You can define additional classes.

The following is the addition to the record definition in the XML file:

```
<recordType ... pdtRecordClass="blocking_wait_interval" ... >
```

Installing and using the PDT trace facility on the x86_64 (Opteron)

The tracing-facility package on the x86_64 is almost identical to the one used on the PPE. Events tracing is enabled by instrumenting selected function of the DaCS and ALF SDK libraries.

Table 8. Tracing-facility directories on x86_64

Use	Host X86
PDT development trace includes	/usr/include/trace
PDT production trace libraries	/usr/lib/trace
PDT production trace 64 bit libraries	/usr/lib64/trace

The /usr/share/pdt/config directory contains reference configuration files for applications that are use the DaCS and ALF libraries: pdt_dacs_config_hybrid.xml for DaCS and pdt_alf_config_hybrid.xml for ALF. The instrumented libraries are part of ALF and DaCS packages.

The instrumented events for the X86_64 libraries are defined in the following files:

GENERAL (pdt_general.xml)
These are the general trace events such as trace start, trace stop, and so on. Tracing of these events is always active.

DACS (pdt_dacs*.xml)
These are the DaCS events (separated into two groups of events). Refer to the Data Communication and Synchronization programmer's guide and API reference for more details.

ALF (pdt_alf*.xml)
These are the ALF events (separated into two groups of events). Refer to the Data Communication and Synchronization programmer's guide and API reference for more details.

Using the PDT on Hybrid-x86 example

The PDT package contains a sample application in the /opt/cell/sdk/src/pdt-opteron-example.tar file. After installation, compile and run the example, then examine the PDT output using the TA and PDTR tools.

The example directory contains a Makefile that you can use as a reference, and a pdt script file that is similar to the one used for the Cell/B.E. environment. A set of full-reference-configuration files (pdt_x86_64_configuration.xml) is provided in the /usr/share/pdt/config directory. You can copy these files to user directories and modify them as necessary. The trace files that are produced during the application run have the same characteristics as those generated on the PPE.

Note: When an application is run on a hybrid environment using DaCS or ALF, the time on each processor in the hybrid system must be synchronized by the operating system. Accurate time synchronization is required to compare traces from each processor in the hybrid system. The trace data contains ″heart beats″ that record the time of day. These heart beats can be used by the TA and other tools to synchronize the traces.

PDT Restrictions

The PDT has certain restrictions

The following restrictions apply to using the PDT.

- The context-switch notification on Red Hat Enterprise Linux (RHEL) 5.2 is implemented so that only one user can activate the tracing facility at a time. Therefore, RHEL 5.2 multiuser usage of PDT is forbidden, but there is no protection against it.
- The PDT and Oprofile cannot be used at the same time.
- For SPE applications, the SPU tag manager must be used for DMA-tag control.
- If decrementer usage is needed, use the spu_timer API. Do not directly modify the SPU decrementer during the run.

The following restrictions apply to using the Opteron PDT.

- PDT on Opteron is using the RDTSCP to atomically read the TS register with the processor ID. AMD processors former to the AMD NPT Family 0Fh do not provide atomic reads of TS and physical processor ID. Therefore, on older processors it is not possible to guarantee that a thread will not be switched out between consecutive reads of these data elements with two instructions. As a result, PDT on Opteron is limited to the AMD NPT Family 0Fh processors or newer.

Using the PDTR tool (pdtr command)

About this task

The PDTR tool (pdtr command) is a command-line tool that provides both viewing and postprocessing of PDT traces on the target (client) machine. To use this tool, you must instrument your application by building it with the PDT. After the instrumented application has run and created the trace output files, the pdtr command can be run to show the trace output. For example, given PDT instrumented application output files:

```
20070604073422.pex     (the xml trace meta file)
20070604073422.1.trace (the binary trace data)
20070604073422.map     (long strings data)
20070604073422.maps    (copy of /proc/<pid>/maps data)
```

use the pdtr command to generate text-based output for this trace as follows:

```
pdtr [options] 20070604073422
```

which produces:

```
20070604073422.pep
```

Calling pdtr with no arguments produces a usage summary:

```
pdtr [options] name

    -trc  Sequential per-event trace output
    -trc0     Sequential reduced trace output
    -meta     Dump meta file output (name.meta)
    -map      Dump address maps
```

```
-ip path  Full path to trace input files
-op path  Full path to output files
-tb TB    Use timebase frequency TB
-w level  Set the warning level for unaligned or small DMAs
-sw       Suppress warnings
-z         Show zero count events
-dc       Disable repetative event compression
-sf lspe  Include output for only logical spe lspe
-psc      Per spe profile counts
-mp       Enable spe micro-profiling
```

Where **name** is the PDT trace prefix, for example *foo* for `trace foo.pex, foo.maps, foo.1.trace`.

The pdtr output file contains a summary report for preselected events, such as mutex locking and DMA. If you use the optional `-trc` flag, the file will also include a time-stamped event-by-event sequential-trace listing. The following example is a partial sequential-output trace.

```
----- Trace File(s) ------------------------------------------------------------------
Rec# TimeStamp DeltaTime  Proc EvID EventName  Event Parameters ...
-------------------------------------------------------------------------------------
1 0.000000 0.000ms PPE

2     1.035853  1035.853ms PPE  0200 HEART_BEAT EventID=200 Processor=2 PhysicalID=0 EventCount=1
     CallingThread=F53DF4B0 StartTime=101248800FFA0B08 EndTime=BABA597C8E7 ProgramCounter=FF8A788
     time_of_day=DBC0600000000

3    1.045290  9.436ms    PPE  0001 CONTEXT_CREATE EventID=1 Processor=2 PhysicalID=0 EventCount=2
     CallingThread=F6F8F4B0 StartTime=F7FA3150F7FA3150 EndTime=BABA599D8AB ProgramCounter=10001C50
     gang=0 spe=100202C8 flags=0 run_spu_thread()
:
:
38    98 6.845us          SPE  0302  SPE_MFC_GET   EventID=302 Processor=3 PhysicalID=0 EventCount=2
     SPEcontext=100202C8 StartTime=0 EndTime=6D PPEcreateContextEventCount=12 ProgramCounter=1754
     ea=6D80 ls=10012680 size=80 tagid=1E tid=0 rid=0 main() lspe=1 Size: 0x80 (128), Tag: 0x1e (30)

39   111 0.908us          SPE  1202  SPE_MFC_READ_TAG_STATUS EventID=1202 Processor=3 PhysicalID=0
     EventCount=3 SPEcontext=100202C8 StartTime=70 EndTime=7A PPEcreateContextEventCount=12
     ProgramCounter=1754 _update_type=2 _current_mask=40000000 tag_status=40000000 main() lspe=1
     {DMA done[tag=30,0x1e] rec:38 0.908us 141.0MB/s}

40   124 0.908us          SPE  0503  SPE_MUTEX_LOCK EventID=503 Processor=3 PhysicalID=0 EventCount=4
     SPEcontext=100202C8 StartTime=7D EndTime=87 PPEcreateContextEventCount=12 ProgramCounter=65C
     lock=10012580 miss=0 main() lspe=1 lock:mylock

:
45   196 0.698us         SPE   0703  SPE_MUTEX_UNLOCK EventID=703 Processor=3 PhysicalID=0 EventCount=9
     SPEcontext=100202C8 StartTime=BE EndTime=CF PPEcreateContextEventCount=12 ProgramCounter=880
     lock=10012580 main() lspe=1 lock:mylock rec:40 hold=5.0us
:
:
```

See the PDTR man page for additional output examples and usage details.

The following example shows a lock report summary. This report shows summary information for a single lock, shr_lock at address 0x10012180. It shows the total number of accesses to that lock, the hit and miss counts and ratio, and the minimum, average and maximum hold (after the lock is acquired) and wait (waiting on a miss) times. Following this line are the individual callers of the lock (procedure name, address, and logical SPE (lspe if from SPE code) and the associated hit, miss, hold, and wait times per caller. The asterisk (*) character indicates each lock that was not explicitly initialized with a mutex_init() call.

```
=====================================================================================================================

     Accesses           Hits          Misses       Hit hold time (uS)   Miss wait time (uS)
 Acount  %Total       Count %Acount  Count %Acount   min,   avg,    max   min,   avg,    max     Name
---------------------------------------------------------------------------------------------------------------------
*   600 (100.0 )        3 (  0.5 )   597 ( 99.5 )    100.8, 184.6, 402.4   13.3, 264.4,  568.0   shr_lock (0x10012180)
                        2 ( 66.7 )   298 ( 49.9 )    100.8, 101.1, 101.5  181.8, 249.7,  383.6   main (0x68c)(lspe=1)
                        1 ( 33.3 )   199 ( 33.3 )    200.7, 201.3, 202.5   13.3, 315.2,  568.0   main (0x68c)(lspe=2)
                        0 (  0.0 )   100 ( 16.8 )      0.0,   0.0,   0.0  205.0, 206.8,  278.5   main (0x68c)(lspe=3)
* - Implicitly initialized locks (used before/without mutex_init)
```

If SPE profiling events are enabled in the PDT configuration file, these profile events are summarized as follows:

```
Profile:
========
Total SPE profile samples: 426
lspe:1 context:01015D698
   /home/user/pep/pt3/gtst1/spu:0
      231 ( 54.2%) 00668-0068B procA
      110 ( 25.8%) 00690-006B3 procB
       57 ( 13.4%) 006B8-006DB procC
       28 (  6.6%) 006E0-00703 procD
```

The preceding summary shows that of the 426 total sample events, 231 (spe decrementer based) sample events (54.2% of the total) occurred in procA, 110 (25.8% of the total) occurred in procB, etc.

Chapter 2. Feedback Directed Program Restructuring (FDPR-Pro)

This section describes FDPR-Pro. It covers the following topics:

Introduction

The Post-link Optimization for Linux on POWER™ tool (FDPR-Pro or *fdprpro*) is a performance tuning utility that reduces the execution time and the real memory utilization of user space application programs. It optimizes the executable image of a program by collecting information on the behavior of the program under a workload. It then creates a new version of that program optimized for that workload. The new program typically runs faster and uses less real memory than the original program.

The fdprpro tool applies advanced optimization techniques to a program. Some aggressive optimizations might produce programs that do not behave as expected. You should test the resulting optimized program with the same test suite used to test the original program. You cannot re-optimize an optimized program by passing it as input to fdprpro.

The post-link optimizer builds an optimized executable program in three distinct phases:

1. Instrumentation phase

   The optimizer creates an instrumented executable program and an empty template profile file. Type the command `fdprpro` and specify the instrumentation action as follows:

   ```
   fdprpro -a instr myprog
   ```

   The instrumentation phase creates an instrumented file and a profile file. The default filename suffix appended to the instrumented file is `.instr` and the default filename suffix appended to the profile file is `.nprof`. Therefore, the preceding command would generate the files myprog.instr and myprog.nprof.
2. Training phase

   The instrumented program is executed with a representative workload and as it runs it updates the profile file.
3. Optimization phase

   The optimizer generates the optimized executable program file. You can control the behavior of the optimizer with options specified on the command line. Type the command `fdprpro` and specify the optimization action, the (same) input program, the profile file, and the desired optimization flags. The following is an example.

```
$ fdprpro -a opt -f myprog.nprof [<opts> ...] myprog
```

The default suffix for the output file name is `.fdpr`. The preceding command creates an optimized file named myprog.fdpr.

An instrumented executable, created in the instrumentation phase and run in the training phase, typically runs several times slower than the original program. This slowdown is caused by the increased execution time required by the instrumentation. Select a lighter workload to reduce training time to a reasonable value, while still fully exercising the desired code areas.

Input files

The input to the fdprpro command must be an executable or a shared library (for PPE files) produced by the Linux linker. fdprpro supports 32-bit or 64-bit programs compiled by the GCC or XLC compilers.

Build the executable program with relocation information. To do this, call the linker with the `--emit-relocs` (or `-q`) option. Alternatively, pass the `-Wl,--emit-relocs` (or `-Wl,-q`) options to the GCC or XLC compiler.

The SDK helps you build sample programs using a make script named *make.footer*. It compiles and links both the PPE and SPE parts of a program, and includes a predefined set of compiler and linker options. Typically, a user has a simple Makefile that begins with `include $(CELL_TOP)/buildutils/make.footer`. To preserve relocation information, add the following lines to the Makefile before the `include $(CELL_TOP)/buildutils/make.footer` line:

```
LDFLAGS_xlc += -Wl,-q
LDFLAGS_gcc += -Wl,-q
```

Alternatively, edit the make.footer file are add "`-Wl,-q`" to the definition of `_LDFLAGS`

Instrumentation and profiling

The fdprpro command creates an instrumented file and a profile file. The profile file is populated with profile information while the instrumented program runs with a specified workload.

The instrumented program requires a shared library named `libfsprinst32.so` for ELF32 programs, or `libfdprinst64.so` for ELF64 programs. These libraries are placed in the library search path directory during installation.

The default directory for the profile file is the directory containing the instrumented program. To specify a different directory, set the environment variable `FDPR_PROF_DIR` to the directory containing the profile file.

Optimizations

If you invoke fdprpro with the basic optimization flag `-O`, it performs code reordering optimization as well as optimization of branch prediction, branch folding, code alignment and removal of redundant `NOOP` instructions.

To specify higher levels of optimizations, pass one of the flags `-O2`, `-O3`, or `-O4` to the optimizer. Higher optimization levels perform more aggressive function inlining, DFA (data flow analysis) optimizations, data reordering, and code restructuring such as loop unrolling. These high level optimization flags work well for most applications. You can achieve optimal performance by selecting and testing specific optimizations for your program.

Instrumentation and optimization options

The fdprpro command accepts many options to control optimization. In our tests, the `-O3` option consistently gave good performance results. For complete details, see the fdprpro man page.

Profiling SPE executable files

About this task

When the optimizer processes PPE executables, it generates a profile file and an instrumented file. The profile file is filled with counts while the instrumented file runs. In contrast, when the optimizer processes SPE executables, the profile is generated when the instrumented executable runs. Running a PPE/SPE instrumented executable typically generates a number of profiles, one for each SPE image whose thread is executed. This type of profile accumulates the counts of all threads which execute the corresponding image. The SPE instrumented executable generates an SPE profile named `<spename>.mprof` in the output directory, where `<spename>` represents the name of the SPE thread. For more information, see "Processing PPE/SPE executable files."

If an old profile exists before instrumentation starts, fdprpro accumulates new data into it. In this way you can combine the profiles of multiple workloads. If you do not want to combine profiles, remove the old profile before starting the optimizer.

The instrumented file is 5% to 20% larger than the original file. Because of the limited local store size of the Cell/B.E. architecture, instrumentation might cause SPE memory overflow. If this happens, fdprpro issues an error message and exits. To avoid this problem, the user can use the `--ignore-function-list file` or `-ifl file` option. The file referenced by the `file` parameter contains names of the functions that should not be instrumented and optimized. This results in a reduced instrumented file size. Specify the same `-ifl` option in both the instrumentation and optimization phases.

Note: The fdprpro command uses lock files named `/tmp/fdpr_xflck`*xxxx* to synchronize multiple SPE threads updating a common profile file. A lock file is created and removed one or more times during an instrumented run. In rare cases, the file might still exist after instrumentation. It is advisable to remove the lock files periodically.

Processing PPE/SPE executable files

By default, fdprpro processes the executable file as a PPE executable or as an SPE executable, depending on its intended target (the intended target is specified inside the executable file). Two modes are available in order to fully process the PPE/SPE hybrid file: *integrated mode*, and *standalone mode*.

Integrated mode

About this task

The integrated mode of operation does not display the details of SPE processing. This interface is convenient for performing full PPE/SPE processing, but flexibility is reduced. To completely process a PPE/SPE file, run the fdprpro command with the -cell (or --cell-supervisor) command-line option. The following is an example.

```
$ fdprpro -cell -a instr myprog -o myprog.instr
```

To optimize the program myprog, type the following command.

```
$ fdprpro -cell -a opt[<opts> ...] myprog -f myprog.nprof -o myprog.fdpr
```

The option -spedir specifies the directory into which SPE files are extracted, where they are processed, and from where they are encapsulated back into the PPE file. If this option is not specified, a temporary directory is created in the /tmp directory and is deleted if fdprpro exits without error.

Standalone mode

About this task

In integrated mode, the same optimization options are used when processing the PPE file and when processing each of the SPE files. Full flexibility is available in standalone mode, where you can specify the explicit commands needed to extract the SPE files, process them, and then encapsulate and process the PPE file. The following list shows the details of this mode.

- Extraction

  SPE images are extracted from the input program and written as executable files in the specified directory. The following is an example.

  ```
  $ fdprpro -a extract -spedir mydir myprog
  ```
- SPE processing

  The SPE images are processed one by one. You should place all of the output files into a distinct directory by their original name. The following is an example.

  ```
  $ fdprpro -a <action> mydir/<spe1> [-f <prof1>] [<opts> ...] -o outdir/<spe1>
  $ fdprpro -a <action> mydir/<spe2> [-f <prof2>] [<opts> ...] -o outdir/<spe2>
  ...
  ```

  Select either instr or opt for action. Specify the profile file with the -f command line option. If you do not specify this option, the program searches for a default profile file named mydir/<spename>.mprof in the current directory.

  Note: The FDPR_PROF_DIR environment variable cannot be used for overriding the SPE profile directory. For more information, see "Instrumentation and profiling" on page 20
- Encapsulation and PPE processing

  The SPE files are encapsulated as a part of the PPE processing. The following is an example. The -spedir option specifies the output SPE directory.

  ```
  $ fdprpro -a <action> --encapsulate -spedir outdir [<opts> ...] myprog
  ```

Human-readable output

In addition to creating an optimized or instrumented program, fdprpro produces human-readable output. The following list details the possible output streams of fdprpro.

- Standard output. The output contains the sign-on message, progress information and the sign-off message. Progress information displays the passage of fdprpro through different phases of processing. The following is an example.

```
FDPR-Pro 5.4.0.10 for Linux (CELL)
fdprpro -a opt -O3 li.linux.gcc32.base -o 1.base
> reading_exe ...
> adjusting_exe ...
> analyzing ...
> building_program_infrastructure ...
...
> updating_executable ...
> writing_executable ...
bye.
```

Specify the `--quiet` option to suppress this output.

- Standard error. Warnings and errors messages are written to the standard error stream. fdprpro exits after the first error.
- Statistics file. If you specify the `--verbose <level>` option, fdprpro writes various statistics to a file. The default file name for the statistics file is `<output_file>.stat`. This file contains a list of tables in the form of `<attribute>` `<value>` pairs, one per line. You can control the output detail level by specifying the **level** parameter. The following is an excerpt from the statistics file corresponding to the above example.

```
options.group                   active_options
options.optimization            -bf -bp -dp -hr -hrf 0.10 -kr -las -lro
                                -lu 9 -isf 12 -nop -pr -RC -RD -rt 0.00
                                -si -tlo -vro
options.output                  -o 1.base
global.use_try_and_catch:       0
global.profile_info:            not_available

file.input:                     li.linux.gcc32.base
file.output:                    1.base
file.statistics:                1.base.stat
analysis.csects:                     347
analysis.functions:                  343
analysis.constants:                   13
analysis.basic_blocks:              5360
analysis.function_descriptors:         0
analysis.branch_tables:               10
analysis.branch_table_entries:       374
analysis.unknown_basic_units:         17
analysis.traceback_tables:             0
...
```

The options specified in the optimization group are those enabled by the `-O3` option.

Running fdprpro from the IDE

You can invoke fdprpro using the GUI of the Eclipse-based Cell/B.E. IDE. A special plugin is integrated to the IDE to enable this feature. See the IDE documentation for more detailed information.

Cross-development with FDPR-Pro

About this task

FDPR-Pro can be used also in cross-development environment available on Linux X86 systems. The same three-phase profile-driven optimization process is used: instrumentation, profile collection (training), and optimization. In addition, the fdprpro commands used during instrumentation and optimization are identical. The difference is in how profile is collected.

Profile collection in native development is achieved by running the instrumented file locally on the host and using the created profile when performing the optimization phase. However, the instrumented file (like the optimized file) can only be executed on a Cell BE-based system. Perform the following steps to collect the profile in a cross-development environment:

1. Pass the instrumented file, with its empty PPE profile (typically with an `.nprof` extension), and any input files needed for its execution, to a native Cell BE environment (or to the Cell BE full-system simulator). Verify that the native environment includes the shared libraries required for instrumentation: `/usr/lib/libfdprinst32.so` and `/usr/lib64/libfdprinst64.so`.
2. Execute the instrumented file with its workload. This fills the PPE profile and creates the SPE profile (with the `.mprof` extension).
3. Pass the generated profiles back to the cross-development environment where they will be used in the optimization phase.

Chapter 3. OProfile

OProfile is a tool for profiling user and kernel level code. It uses the hardware performance counters to sample the program counter every *N* events. You specify the value of *N* as part of the event specification. The system enforces a minimum value on *N* to ensure the system does not get completely swamped trying to capture a profile.

Make sure you select a large enough value of *N* to ensure the overhead of collecting the profile is not excessively high.

The `opreport` tool produces the output report. Reports can be generated based on the file names that correspond to the samples, symbol names or annotated source code listings.

How to use OProfile and the postprocessing tool is described in the user manual available at:

```
http://oprofile.sourceforge.net/doc/
```

The current Software Development Kit for Multicore Acceleration version of OProfile for Cell BE supports profiling on the POWER processor events and SPU cycle profiling. These events include cycles as well as the various processor, cache and memory events. It is possible to profile on up to four events simultaneously on the Cell BE system. There are restrictions on which of the PPU events can be measured simultaneously. When using PPU profiling, events must be within the same group due to restrictions in the underlying hardware support for the performance counters. You can use the `opcontrol –list-events` command to view the events and which group contains each event.

There is one set of performance counters for each node that are shared between the two CPUs on the node. For a given profile period, only half of the time is spent collecting data for the even CPUs and half of the time for the odd CPUs. You may need to allow more time to collect the profile data across all CPUs.

Note:

1. Before you issue an `opcontrol --start`, you should issue the following command:
   ```
   opcontrol --start-daemon
   ```
2. To produce a report with Linux kernel symbol information you should install the corresponding Kernel debuginfo RPM..

SPU profiling restrictions

When SPU cycle profiling is used, the `opcontrol` command is configured for separating the profile based on SPUs and on the library. This corresponds to the you specifying `–separate=CPU` and `–separate=lib`. The separate CPU is required because it is possible to have multiple SPU binary images embedded into the executable file or into a shared library. So for a given executable, the various SPUs may be running different SPU images.

With `–separate=CPU`, the image and corresponding symbols can be displayed for each SPU. The user can use the opreport `–merge` command to create a single report for all SPUs that shows the counts for each symbol in the various embedded SPU binaries. By default, opreport does not display the `app name` column when it reports samples for a single application, such as when it profiles a single SPU application. For opreport to attribute samples to a binary image, the opcontrol script defaults to using `–separate=lib` when profiling SPU applications so that the `image name` column is always displayed in the generated reports.

SPU report anomalies

The report file uses the term CPUs when the event is SPU_CYCLES. In this case, CPUs actually refer to the various SPUs in the system. For all other events, the CPU term refers to the virtual PPU processors.

With SPU profiling, opreport's `--long-filenames` option may not print the full path of the SPU binary image for which samples were collected. Short image names are used for SPU applications that employ the technique of embedding SPU images in another file (executable or shared library). The embedded SPU ELF data contains only the filename and no path information to the SPU binary file being embedded because this file may not exist or be accessible at runtime. You must have sufficient knowledge of the application's build process to be able to correlate the SPU binary image names found in the report to the application's source files.

Tip

Compile the application with `-g` and generate the OProfile report with `-g` to facilitate finding the right source file(s) to focus on.

Generally, when the report contains information about a single application, `opreport` does not include the report column for the application name. It is assumed that the performance analyst knows the name of the application being profiled.

Chapter 4. Cell-perf-counter tool

The cell-perf-counter (cpc) tool is used for setting up and using the hardware performance counters in the Cell/B.E. processor. These counters allow you to see how many times certain hardware events are occurring, which is useful if you are analyzing the performance of software running on a Cell Broadband Engine Architecture system. Hardware events are available from all of the logical units within the Cell/B.E. processor, including the PPE, SPEs, interface bus, and memory and I/O controllers. Four 32-bit counters, which can also be configured as pairs of 16-bit counters, are provided in the Cell/B.E. performance monitoring unit (PMU) for counting these events.

The cpc tool also makes use of the hardware sampling capabilities of the Cell/B.E. PMU. This feature allows the hardware to collect very precise counter data at programmable time intervals. The accumulated data can be used to monitor the changes in performance of the Cell/B.E. system over longer periods of time.

The cpc tool provides a variety of output formats for the counter data. Simple text output is shown in the terminal session, HTML output is available for viewing in a Web browser, and XML output can be generated for use by higher-level analysis tools such as the Visual Performance Analyzer (VPA).

You can find details in the documentation and manual pages included with the `cellperfctr-tools` package, which can found in the `/usr/share/doc/cellperfctr-<version>/ directory` after you have installed the package.

Chapter 5. Hybrid performance tools

Overview

An application running in a hybrid environment typically consists of a root application that runs on the host (for example, an AMD X86_64 processor) and application fragments that run on one or more accelerators (in this case a Cell/B.E. processor). The hybrid application itself is launched from the host system and by default the output from all parts of the application is returned to the host console.

There are a variety of performance and debug tools that either work on the AMD X86_64 processor, on the Cell/B.E., or on both. To use these tools on a hybrid application, follow this procedure:

- Launch a Cell/B.E. performance tool against an application fragment which runs on an arbitrary accelerator when the application itself is launched from the host.
- Launch a host performance tool and a Cell/B.E. performance tool for a hybrid application and coordinate the output.

The hybrid performance tooling works with DaCS for Hybrid and is able to solve these problems for the following tools:

- CPC
- FDPR-Pro
- OProfile
- PDT
- PDTR

DaCS for Hybrid provides a mechanism to allow environment variables to be exported from the host application process to the accelerator application fragment process. In addition, when you launch a program, you can specify a parent executable to be specified, which is called to do the final launch of the application fragment on the accelerator.

With these two capabilities, the hybrid tooling currently consists of Bash scripts. Typically, there is a script for the host and a script for the accelerator. The host script sets up the environment, including setting up environment variables to pass along to the accelerator process, and launches the root application. It also can coordinate the launch of the host based performance tools.

Anytime that DaCS for Hybrid needs to start an application fragment on an accelerator, it instead starts the corresponding accelerator script. This script does any required setup needed by the performance tool, launches the application, and does any required post processing when the application is finished.

Requirements

Your system must meet the following requirements to run the Cell/B.E. performance tools.

SSH usage

Some of the tools make use of SSH (Secure SHell) to launch applications on the accelerator. Ensure that SSH support between the host and the accelerator is configured so that scripts can ssh from the host to the accelerator without needing to supply a password.

Hybrid application

The scripts are designed to run against a hybrid application. The scripts depend on DaCS being used in the application to launch an application fragment on the accelerator.

Hybrid tools RPMs

There are two RPMs associated with the hybrid tools:
- One for the X86_64 host system
- One for the PPC64 accelerator system

The base RPM name is *cell-perf-hybrid-tools*.

Supporting performance tools

The cell-perf-hybrid-tools RPMs do not directly require other RPMs, however; the RPM's associated with the base performance tools you want to use must be installed. For example, if you want to run the script to launch CPC against your hybrid application, the CPC RPMs must be installed on the Cell/B.E. system(s) you are using for acceleration. See the *IBM Software Development Kit for Multicore Acceleration Installation Guide* for details about how to install the performance tools.

NFS

For tooling data output, create an NFS mount point that can be shared between the host and accelerator systems . This is mandatory for hybrid systems that do not have a local hard drive on the accelerator part of the node.

Setting up and configuring the performance tool scripts

This topic describes how to set up and configure the performance tool scripts.

By default the hybrid performance tooling scripts are installed in /usr/bin.

The hybrid performance tools use common setup scripts to set up environment information for the tools and applications.

The `perfToolHostSetup` script is sourced in the host portion of the tooling scripts. Within this script the `perfToolUsrEnv` script is sourced. This script is generally where environment settings that you want or need to change can be found. This is installed as a read-only file. If the defaults do not work for a given user, the user can copy this file and modify the values of the environment variables.

To make the `perfToolHostSetup` find your new `perfToolUsrEnv`, export *PERF_TOOLS_USR_ENV*. The environment variable should reference the full path and file name of the new `perfToolUsrEnv`. Type for example:

```
$ export PERF_TOOLS_USR_ENV=/home/johndoe/bin/myPerfToolUsrEnv
```

The current user environment variables are as follows:
- *SDK_ROOT* - points to the SDK install location. Only set this variable if the SDK is installed to other than the default location.
- *PERF_DATA_ROOT* - this is the base location for all the tool output. This needs to be the same on both the host and the accelerator. This is usually an NFS mounted file system. All output directories created by the tools have this as a base directory.

The default location is: /$SCRATCH/perfData

Where *$SCRATCH* points to a common NFS mount point which is shared between host and accelerator to output data.

The following four optional environment variables allow you to identify the location of the host and accelerator pieces of your hybrid application so they can be added to the appropriate path.

These also allow you to point to alternative versions of your executable which you are using to work with the tools (for example, if you have compiled a version of your application with trace-enabled, when you run the trace launching tool you want to make sure the traced version of your executable is placed at the front of various paths).

HOST_APP_PATH	Path where the host application executable is located.
HOST_APP_LD_LIBRARY_PATH	Path for host application dependent shared libraries.
ACCEL_APP_PATH	Path where the accelerator application executable is located.
ACCEL_APP_LD_LIBRARY_PATH	Path for accelerator application dependent shared libraries.

General output directory

The top part of the directory structure which is common to all the tools is as follows:

```
$PERF_DATA_ROOT/userid/hostname
```

where:
- `userid` is the userid of the person running the host script command
- `hostname` is the hostname of the machine where the host script command was launched

The rest of the directory structure is unique for each tool.

Hybrid tools description

How to run each performance tool against a sample hybrid application.

Using the DaCS for Hybrid-x86 sample

Note: For each tool a walk-thru is provided, which shows you how to run the tool against a sample hybrid application. To run the sample application you must install the dacs-hybrid-examples-source-*.*.-* RPM on the host system. Follow the directions in the README and be able to run the sample_dacs_hybrid_1t_he application.

The sample comes with a bash script, ../dacs-hybrid-examples/dacs_hello/hybrid/bin/runsample.sh, which sets some environment variables and launches the sample_dacs_hybrid_1t_he program. You can use this script to run the sample directly. The hybrid performance tools MUST run against the host executable directly otherwise they do not function so do NOT use runsample.sh with the performance tools.

The simplest way to work around this problem is to export the following environment variable prior to running any of the performance tools against the sample. Make sure you export from the directory where the sample_dacs_hybrid_1t_he is located.

```
export ACCEL_PROG_PATH=`pwd`/accel
```

CPC hybrid support

The hybrid performance tools package includes the scripts cpch and cpca to assist in using CPC in a hybrid environment.

A hybrid program can be monitored using the cpch script. The cpca script is used internally by cpch and is not intended to be used by a user directly.

CPC usage

```
cpch [options] <application name> [<application parameters>]
```

Options:

--runid=ID	Optional. Name of the current run. If no runid is provided a timestamp is used.
--cell-event=EVT	Required. Can be specified multiple times depending on the capabilities of the Cell/B.E. PMU and OProfile. The event(s) to be monitored on the Cell/B.E. part of the application is (are) specified. For example: `--cell-event=C`
--cell-options=OPTS	Optional. Specifies any parameters for the CPC command. Multiple parameter values can be specified by enclosing them in quotes.
--html	Create HTML output.
--xml	Create XML output.
--help,-h	Print help information for this command.
--listenv,-l	Prints out trace environment variable information.

Example:

```
cpch --cell-event=C --cell-options="-i 32000000" my_application -xyz
```

CPC tool results

- Any application output from either the host or the accelerator parts of the application is normally routed back to the host console window. Any output generated by the CPC hybrid scripts is also displayed in the host console window.
- CPC output: The output from the CPC tool is in the following directory:

  ```
  <PERF_DATA_ROOT>/<username>/<hostname>/cpc/<runid>/cbe/<acceleratorhostname>
  ```

  where:

  PERF_DATA_ROOT
  is the environment variable set in the perfToolUsrEnv configuration file (see "Setting up and configuring the performance tool scripts" on page 30)

  username
  is the userid that called the `cpch` script

  hostname
  is the host system's name

  runid
  is either the runid supplied on the `cpch` invocation, or at date-timestamp taken at the time `cpch` is called

  accelerator hostname
  is the hostname for the accelerator where `cpc` is running.

  The basic output is in a file named cpc.out.

  If XML or HTML output is requested when you invoke cpch, cpc.html, or cpc.xml, or both is also in this directory.

CPC example

Take the sample hybrid application `sample_dacs_hybrid_1t` and run CPC against it.

1. Make sure that you can run the `sample_dacs_hybrid_1t` application. To do this, change to the `*/dacs-hybrid-examples/dacs_hello/bin` directory and type:

   ```
   ./runsample.sh
   ```

   Note: If you have previously exported *DACS_START_ENV_LIST* make sure you reset it before you run `./runsample.sh`:

   ```
   export DACS_START_ENV_LIST=
   ```

   The hybrid sample is now running.
2. Run the cpc tool against the Cell/B.E. part of the application. Export the following variable:

   ```
   export ACCEL_PROG_PATH=`pwd`/accel
   ```

   ```
   /usr/bin/cpch --runid=hybridSample_run1 --cell-event=C ./sample_dacs_hybrid_1t_he
   ```

   Given the following:

   ```
   PERF_DATA_ROOT = /myData
   Userid = johndoe
   Host System Name = myHost
   Accelerator System Name = myAcceleratorName
   ```

   The file output by `cpch` is in the following directory:

   ```
   /myData/perfData/johndoe/myHost/cpc/hybridSample_run1/cbe/myAcceleratorName
   ```

FDPR-Pro hybrid support

The hybrid performance tools include the scripts `fdprproh` and `fdprproa` to assist in using FDPR-Pro in a hybrid environment.

Use the `fdprproh` script to analyze and optimize a hybrid program. The `fdprproa` script is used internally by `fdprproh` and is not intended to be used by a user directly.

Note: The hybrid script, `fdprproh`, supports coordinating `fdrprpro` usage on the Cell/B.E. accelerator ().

FDPR-Pro usage

```
fdprproh [OPTION] ... <application name> [<application parameters>]
```

Note: If application arguments contain switches (for example `-p`), all the arguments must be placed in double quotes.

Options:

`-l, --listenv`	List the environment variables for the script processing.
`-h, --help`	Print help information for this command.
`-o, --optimization-options "<fdprpro optimization options>"`	Enclose the optimization options in double quotes.

Example:

```
fdprproh --optimization-options "-O3"  /myAppPath/myApp appArguments
```

Note: If `fdprpro` fails, the output directory contains files that can contain additional information, which can help you to determine the cause of the failure. For example, a log file is created, which logs every phase of the tool when it is running; instrumentation, executing the instrumented code, and optimizing the code.

FDPR-Pro tool results

- Any application output from either the host or the accelerator parts of the application is normally routed back to the host console window. Any output generated by the `fdprpro` hybrid scripts is also displayed in the host console window.
- FDPR-Pro output:

  All output from the three stages of the `fdprpro` tool is in the following directory:

  ```
  $PERF_DATA_ROOT/<userid>/<hostname>/fdprpro/<application name>/
  ```

  The final optimized version of the original executable file is named:

  ```
  <applicationName>.opt
  ```

FDPR-Pro example

This example takes the sample hybrid application `sample_dacs_hybrid_1t` and runs a basic `fdprproh` run against it.

1. Make sure that you can run the `sample_dacs_hybrid_1t` application. To do this, change to the */dacs-hybrid-examples/dacs_hello/bin directory and type:

```
./runsample.sh
```

Note: If you have previously exported DACS_START_ENV_LIST, make sure you reset it before you type `./runsample.sh`:

```
export DACS_START_ENV_LIST=
```

The sample provides instructions about how to build and run it.

Now that the hybrid sample is running, run the `fdprpro` tool against the Cell/B.E. part of the application.

2. **Modify the Makefiles:**`fdprpro` requires its executables to be built with relocation information. To do this, make the following modifications to the sample's Makefiles:
   a. In file: `*/dacs-hybrid-examples/dacs_hello/hybrid/ppu64/Makefile` on the `LDFLAGS` line, insert ″-Wl,-q″ after ″+=″ (do not add any spaces).
   b. To rebuild the files, type the following from the combined directory:
      ```
      make clean
      make
      ```
3. **Run:**
   ```
   cd bin
   ```

   Export the following variable:
   ```
   export ACCEL_PROG_PATH=`pwd`/accel

   /usr/bin/fdprproh –o "-O2" ./sample_dacs_hybrid_1t_he
   ```

   Given the following:
   ```
   PERF_DATA_ROOT       = /myData
   Userid               = johndoe
   Host System Name     = myHost
   PPC Executable Name  = myPPCExecName
   SPE Executable Name  = mySPEExecName
   ```

   The output files from the `fdprproh` run are saved in the following directory:
   ```
   /myData/perfData/johndoe/myHost/fdprpro/myPPCExecName
   ```

   If the run is successful, the following files are in the directory:

`myPPCExecName.instr`	The instrumented executable.
`myPPCExecName.instr.log`	`fdprpro` console output from the instrumentation step.
`myPPCExecName.instr.out`	Console output from running `myPPCExecName.instr.`
`myPPCExecName.nprof`	Tool output file for the PPC executable.
`mySPEExecName.mprof`	Tool output file for the SPE executable.
`myPPCExecName.log`	`fdprpro` console output from the optimization step.
`spe_dir`	SPE temporary directory. Empty unless there was a failure concerning the SPE part of the run.
`myPPCExecName.opt`	The optimized PPC with the embedded SPE executable file.

OProfile hybrid support

OProfile is available for hybrid. You can download it from:

```
http://oprofile.sourceforge.net/
```

Root access

Because the OProfile tool accesses hardware registers, it requires that the userid running this tool either has root authority or sudo authority to run OProfile. For more information about which events can be monitored for each type of CPU, the event names, and other command line options, refer to the OProfile documentation at:

```
http://oprofile.sourceforge.net/
```

OProfile usage

The Hybrid Tools includes the scripts `oprofileh`, `oprofilea`, and `oprofilerpt` to assist in using OProfile in a hybrid environment. A hybrid program can be profiled using the `oprofileh` script. The `oprofilerpt` script can be used at a later time to create a report from the profile data. The `oprofilea` script is used internally by `oprofileh` and is not intended to be called by a user directly. OProfile can be run on the host part of the application and/or the Cell/B.E. part of the application.

oprofileh usage:

```
oprofileh [options] <application name> [<application parameters>]
```

Note: If you want to pass parameters to your application you need to add a blank parameter (--) between the last `oprofileh` parameter and the program name, for example:

```
oprofileh --host-vm=--no-vmlinux --host-event=CPU_CLK_UNHALTED:1000000 --  myApp myAppParm
```

Parameters for oprofileh

`--runid=ID`	Optional. Name of the current run to refer to the data using the oprofilerpt command at a later time. If no runid is provided a timestamp is used.
`--host-vm=VM`	Optional: (`--no-vmlinux,--vmlinux=,...`) Common OProfile parameters that are provided before starting OProfile. These option are used on the host OProfile. Multiple parameter values may be specified by enclosing in quotes.
`--cell-vm=VM`	Optional: (`--no-vmlinux,--vmlinux=,...`) Common OProfile parameters that are provided prior to starting OProfile. These option are used on the Cell/B.E. OProfile. Multiple parameter values can be specified by enclosing in quotes.

`--host-event=EVT`	Optional. Can be specified multiple times depending on the capabilities of the host system's PMU and OProfile. The events to be monitored on the host part of the program are specified. For example: `--host-event= CPU_CLK_UNHALTED:500000`
`--cell-event=EVT`	Optional. Can be specified multiple times depending on the capabilities of the Cell/B.E. PMU and OProfile. The event to be monitored on the Cell/B.E. part of the program are specified. For example: `--host-event=SPU_CYCLES:500000`
`--host-options=OPTS`	Optional. Can specify any parameters for the OProfile host `opcontrol --start` command. Multiple parameter values can be specified by enclosing in quotes.
`--cell-options=OPTS`	Optional. Can specify any parameters for the OProfile Cell/B.E. `opcontrol --start` command. Multiple parameter values can be specified by enclosing in quotes.
`--help,-h`	Prints help information for this command.
`--listenv,-l`	Prints out trace variable information.

Example for oprofileh:

```
oprofileh --runid=Run7 --cell-vm=--no-vmlinux
          --cell-event=SPU_CYCLES:200000 my_application -lx
```

The `oprofilerpt` command is used after `oprofileh` to create the desired reports. It references the data using the required parameter *runid*. Reports are stored in the performance tools directory structure. The script prints the directory name where the reports are found as it runs the report commands.

oprofilerpt usage:

```
oprofilerpt --runid=<id> [options] opreport|opannotate|opgprof
```

Parameters for oprofilerpt:

`--runid=ID`	Name of the run that was specified or created when oprofileh was previously used to profile a program.
`--host`	Optional. Create a report on the host OProfile data.
`--cell`	Optional,. Create report(s) on the Cell/B.E. OProfile data.
`--host-options=OPT`	Optional. Specify any parameters for the host `opreport/opannotate/opgprof` command.
`--cell-options=OPT`	Optional. Specify any parameters for the Cell/B.E. `opreport/opannotate/opgprof` command. Multiple parameter values can be specified by enclosing in quotes.

`--delete,-d`	Delete the OProfile session data (saved under runid) after the report has been created.
`--print,-p`	Print a copy of the OProfile report.
`--help,-h`	Prints help information for this command.

Example for oprofilerpt:

```
oprofilerpt --runid=Run7 --cell --cell-options=--symbols
           --print opreport
```

OProfile tool results

- Any application output from either the host or the accelerator parts of the application is normally routed back to the host console window. Any output generated by the Oprofile hybrid scripts is also displayed in the host console window.
- OProfile output: the output from the OProfile tool is in the following directory:

  ```
  <PERF_DATA_ROOT>/<username>/<hostname>/oprofile/<runid>/<arch>/<accelerator hostname>
  ```

  where:

  PERF_DATA_ROOT
  : Is the environment variable set in *perfToolUsrEnv* configuration file (see "Setting up and configuring the performance tool scripts" on page 30).

  username
  : Is the userid that called the `oprofileh` script

  hostname
  : Is the host system's name

  runid
  : Is either the runid supplied on the `oprofileh` invocation, or a date-timestamp taken at the time `cpch` was called.

  arch
  : Is the hardware architecture (either x86_64 or cbe)

  accelerator hostname
  : Is the hostname for the accelerator where OProfile is running.

After `oprofileh` has run, `oprofilerpt` should be run to generate a report which is placed in the same directory.

Note: if you run `oprofileh` against both the host and the accelerator at the same time you get multiple <arch> directories.

OProfile example

Take the sample hybrid application `sample_dacs_hybrid_1t` and run OProfile against both the host and accelerator executables.

1. Make sure that you can run the `sample_dacs_hybrid_1t` application. To do this, change to the */dacs-hybrid-examples/dacs_hello/hybrid/bin directory and type:

   ```
   ./runsample.sh
   ```

   Note: If you have previously exported *DACS_START_ENV_LIST*, reset it before typing ./runsample.sh:

   ```
   export DACS_START_ENV_LIST=
   ```

The sample comes with instructions about how to build and run it.

Now that you have the hybrid sample running, run the `oprofileh` tool against the application.

2. Export the following variable:

```
export ACCEL_PROG_PATH=`pwd`/accel

/usr/bin/oprofileh
       --runid=hybridSample_run1
       --host-vm=--no-vmlinux
       --host-event=CPU_CLK_UNHALTED:1000000
       --cell-vm=--no-vmlinux
       -–cell-event=CYCLES:1000000 ./sample_dacs_hybrid_1t_he
```

   Given the following:

```
PERF_DATA_ROOT          = /myData
Userid                  = johndoe
Host System Name        = myHost
Accelerator System Name = myAcceleratorName
```

3. The output files from the `oprofileh` run are in the following directory:

```
/myData/perfData/johndoe/myHost/oprofile/hybridSample_run1/cbe/myAcceleratorName
/myData/perfData/johndoe/myHost/oprofile/hybridSample_run1/x86_64/myAcceleratorName
```

4. Run the OProfile report tool against the output:

```
/usr/bin/oprofilerpt --runid=hybridSample_run1
     --cell --cell-options=--symbols –d --print opreport
```

After running the `oprofilerpt` command each of the output directories contains the file opreport.out, one in the x86_64 directory and one in the cbe directory. This contains the reports from the OProfile runs.

PDT support for hybrid

For general information about how to use and configure PDT, see Chapter 1, “Cell Broadband Engine Performance Debugging Tool (PDT),” on page 1.

The scripts provided here in the hybrid tooling RPMs enable easy setup, coordination, and use of the trace facility in the hybrid environment.

These scripts assume you have enabled your application for trace (see the PDT documentation for details). The scripts also allow for transparent switching of the *LD_LIBRARY_PATH* to include the traced versions of the IBM provided shared libraries. This is useful if you are using the shared library versions of these libraries. If you are statically linking in libraries that are trace enabled you need to modify your make files accordingly (see the PDT documentation).

Environment variables are also provided so that the user can point to traced versions of users’ shared libraries and have them substituted when tracing the application (see the user environment variable descriptions above for *ACCEL_APP_PATH* and *ACCEL_APP_LD_LIBRARY_PATH*).

PDT usage

```
traceh [OPTION] ... <application name> [<application arguments>]
```

Note: If application arguments contain switches (for example, `-p` or `--myOpt`), all the arguments must be placed in double quotes.

Options:

`-h, --help`	Print command help.
`-l, --listenv`	Prints out trace environment variable information.
`--runid`	Specifies a prefix to prepend to the base trace directory associated with the trace of this application. Default is a date/time based directory name. **Note:** This is used to coordinate PDTR analysis.

Example:

```
traceh --runid myFastRun2 myHybridApp argument
```

Setting up PDT

PDT exists on both the host and the accelerator. It supports the ability to trace shipped libraries that have been enabled for trace. It also supports the ability for the user to add trace points to their code (see the PDT users guide for details).

Both PDT on the host as well as PDT on the accelerator require a configuration file to tell them which trace functions you want turned on for the run of your application. Use the following export statements from the host environment to point to the appropriate configuration files:

```
# Host PDT configuration file example:
export PDT_CONFIG_FILE=/usr/share/pdt/config/pdt_dacs_config_hybrid.xml

# Accelerator PDT configuration file example:
export DACS_START_ENV_LIST=
"PDT_CONFIG_FILE=/usr/share/pdt/config/pdt_dacs_config_cell.xml"
```

Note: The example shows an export statement split onto two lines because of the width of the printed page. The export statement is normally one continuous line.

Note: The example listed enables default tracing of all the DaCS code.

PDT tool results

- Any application output from either the host or the accelerator parts of the application is normally routed back to the host console window. Any output generated by the trace hybrid scripts are also displayed in the host console window.
- Trace/PDT Output: The output from the PDT tool is in the following directories:

  ```
  <PERF_DATA_ROOT>/<username>/<hostname>/trace/<runid>
  ```

where:

PERF_DATA_ROOT
: Is the environment variable set in perfToolUsrEnv configuration file (see "Setting up and configuring the performance tool scripts" on page 30).

username
: Is the userid that called the `traceh` script.

hostname
: Is the host system's name.

runid Is either the runid supplied on the `traceh` invocation, or at date-timestamp <YYYYmmddHHMMSS> taken at the time `traceh` was called.

Individual files generated by PDT also have the source hostname prepended to the front so that point of origin can be determined.

After `traceh` is run, pdtrh can be run to analyze the trace or print a readable text file output. VPA can also be used to visualize the results.

PDT Trace example

Take the sample hybrid application `sample_dacs_hybrid_lt` and run `traceh` against both the host and accelerator executables.

Required RPMs for Trace/PDT to function for DaCS:

Platform	Required RPMs
Hybrid-x86	pdt-*.*.-* pdt-devel-*.*.-* pdt-cross-devel-* pdt-cross-devel-*.*.-* pdtr-*.*.-* trace-*.*.-* trace-devel-*.*.-* trace-cross-devel-*.*.-* dacs-hybrid-trace-*.*.-* dacs-hybrid-trace-devel-*.*.-*
Cell/B.E.	pdt-*.*.-* pdtr-*.*.-* trace-*.*.-* dacs-trace-*.*.-* dacs-hybrid-trace-*.*.-*

1. Make sure that you can run the sample_dacs_hybrid_lt application by navigating to the */dacs-hybrid-examples/dacs_hello/hybrid/bin directory and invoking:

   ```
   ./runsample.sh
   ```

   Note: If you have previously exported *DACS_START_ENV_LIST* make sure you reset it before typing `./runsample.sh` as follows:

   ```
   export DACS_START_ENV_LIST=
   ./runsample.sh
   ```

   If it does not run, refer to the sample documentation on how to build and run it.
2. Export the following two variables:

   ```
   export ACCEL_PROG_PATH=`pwd`/accel
   ```
3. Because DaCS on the host as well as on the PPU ship shared libraries with tracing enabled, it is a matter of using those libraries instead of the normal libraries. The traceh and tracea scripts facilitate this by defining *TRACE* to enable it. Run the following export:

   ```
   export TRACE=1
   ```

   Then navigate to the ../hybrid directory and type:

   ```
   make clean
   make
   ```

4. You have now built the hybrid sample for trace. Next, run the traceh tool against the application. To do this, you first need to use environment variables to tell PDT on the host and accelerator where to find the config file you want it to use. For this example, tell it to use a config file which enables tracing of DaCS.
   a. For the host:
      ```
      export PDT_CONFIG_FILE=/usr/share/pdt/config/pdt_dacs_config_hybrid.xml
      ```
   b. For the accelerator:
      ```
      export DACS_START_ENV_LIST=
      "PDT_CONFIG_FILE=/usr/share/pdt/config/pdt_dacs_config_cell.xml"
      ```

      Note: The example shows an export statement split onto two lines because of the width of the printed page. The export statement is normally one continuous line.
5. Execute a run with tracing:
   ```
   cd bin
   /usr/bin/traceh --runid=hybridSample_trace ./sample_dacs_hybrid_1t_he
   ```

   Given the following:
   ```
   PERF_DATA_ROOT          = /myData
   Userid                  = johndoe
   Host System Name        = myHost
   Accelerator System Name = myAccelertorName
   ```

   Then the output files from the traceh run are in the following directory:
   ```
   /myData/perfData/johndoe/myHost/trace/hybridSample_trace
   ```
6. Run the pdtr tool against the output:
   ```
   /usr/bin/pdtrh --runid=hybridSample_trace
   ```

   Output from pdtr is in the same directory, and ends with a pep file extension.

PDTR support for Hybrid

PDTR is a command line tool that provides both viewing and postprocessing of PDT traces on the target machine.

Use of PDTR requires trace output files from PDT. After you have run the PDT application and it has generated trace output files, use PDTR to show the trace output and analysis.

For more information about using PDTR see the pdtr manual page or the Chapter 1, "Cell Broadband Engine Performance Debugging Tool (PDT)," on page 1.

PDTR usage

```
pdtrh --runid=ID [options]
```

Required parameters:

`--runid=ID`	Required. Must either match the runID given to traceh when generating the trace, or match the default date-timestamp assigned when traceh was run.
Options:	

`-h,--help`	Prints out trace environment variable information.
`-l,--listenv`	Specifies a prefix to be pre-pend to the base trace directory associated with the trace of this application. The default is a date/time-based directory name. **Note:** This is used to coordinate PDTR analysis.
`-o, --pdtr-options`	Options "<pdtr command line options>" **Note:** If more than one option is supplied it must be enclosed in double quotes.

Example:

If you create the trace with:

```
traceh --runid myFastRun2 myHybridApp argument
```

To produce a text-readable basic trace file, run:

```
pdtrh –runid myFastRun2
```

PDTR tool results

- Any application output from either the host or the accelerator parts of the application are normally routed back to the host console window. Any output generated by the trace hybrid scripts is also displayed in the host console window.
- Trace/PDT output: The output from the pdtrh tool is placed in the same directory as the source trace files generated by traceh/pdt and has a .pep extension.

Chapter 6. Performance tools example

The performance tools example is a practical ″hands-on″ example, which shows you how to use the performance tools, collect information, and access relevant visualization features.

FFT16M sample application

The target sample application for analysis is the FFT16M application that can be found in the Cell BE SDK demos bundle:

```
/opt/cell/sdk/src/demos/FFT16M
```

This application, which was hand-tuned, performs a 4-way SIMD single-precision complex FFT on an array of size 16,777,216 elements. The available command options are:

```
fft <ncycles> <printflag> [<log2_spus> <numa_flag> <largepage_flag>]
```

Preparing and building for profiling

You need to set up a ″sandbox″ styled project tree structure, so that you have more flexibility when you modify and generate files:

Before you begin

About this task

1. Copy the application from the SDK tree. To work on a ″sandbox″ tree you need your own copy of the project in an accessible location (for example your home directory):

   ```
   cp -R /opt/cell/sdk/demos/FFT16M ~/
   ```

2. Prepare the Makefile.
   a. Go to your recently created project structure and locate the following three Makefiles:

      ```
      ~/FFT16M/Makefile
      ~/FFT16M/ppu/Makefile
      ~/FFT16M/ppu/Makefile
      ```

   b. Modify the Makefiles to prevent them from trying to install executable files back to the SDK tree, and introduce the required compilation flags for profiling data. To do this:
      - Comment out the install directives in ~/FFT16M/ppu/Makefile
      - Introduce the `-g` and `-Wl,-q` compilation flags in order to preserve the relocation and the line number information in the final integrated executable

      The following is an example of how to change ~/FFT16M/ppu/Makefile for gcc.

      ```
      ##
      Target
      ########################################################################
      #
      PROGRAM_ppu= fft
      ########################################################################
      ##
      ```

```
Objects
######################################################################
#
IMPORTS = ../spu/fft_spu.a -lspe2 -lpthread -lm -lnuma
#INSTALL_DIR= $(EXP_SDKBIN)/demos
#INSTALL_FILES= $(PROGRAM_ppu)
LDFLAGS_gcc = -Wl,-q
CFLAGS_gcc = -g
######################################################################
##
buildutils/make.footer
######################################################################
#
```

Note:

- No further Makefile modifications, beyond these, are required
- There are specific changes depending whether you use gcc our xlc as the compiler

```
ifdef CELL_TOP
include $(CELL_TOP)/buildutils/make.footer
else
include ../../../../buildutils/make.footer
endif
Changing ~/FFT16M/ppu/Makefile for gcc
######################################################################
##
Target
######################################################################
#
PROGRAM_ppu= fft
######################################################################
##
Objects
######################################################################
#
PPU_COMPILER = xlc
IMPORTS = ../spu/fft_spu.a -lspe2 -lpthread -lm -lnuma
#INSTALL_DIR= $(EXP_SDKBIN)/demos
#INSTALL_FILES= $(PROGRAM_ppu)
LDFLAGS_xlc = -Wl,-q
CFLAGS_xlc = -g
######################################################################
##
buildutils/make.footer
######################################################################
#
ifdef CELL_TOP
include $(CELL_TOP)/buildutils/make.footer
else
include ../../../../buildutils/make.footer
endif
_ ~/FFT16M/spu/Makefile
```

Example: Changing ~/FFT16M/ppu/Makefile for gcc

```
######################################################################
##
Target
######################################################################
#
PROGRAM_ppu= fft
######################################################################
##
Objects
######################################################################
#
```

```
PPU_COMPILER = xlc
IMPORTS = ../spu/fft_spu.a -lspe2 -lpthread -lm -lnuma
#INSTALL_DIR= $(EXP_SDKBIN)/demos
#INSTALL_FILES= $(PROGRAM_ppu)
LDFLAGS_xlc = -Wl,-q
CFLAGS_xlc = -g
########################################################################
##
buildutils/make.footer
########################################################################
#
ifdef CELL_TOP
include $(CELL_TOP)/buildutils/make.footer
else
include ../../../../buildutils/make.footer
endif
_ ~/FFT16M/spu/Makefile
```

Introduce the `-g` and `-Wl,-q` compilation flags in order to preserve the relocation and the line number information in the final integrated executable file.

Example: Modifying ~/FFT16M/spu/Makefile for gcc

```
########################################################################
##
Target
########################################################################
#
PROGRAMS_spu:= fft_spu
LIBRARY_embed:= fft_spu.a
########################################################################
##
Local Defines
########################################################################
#
CFLAGS_gcc:= -g --param max-unroll-times=1 # needed to keep size of
program down
LDFLAGS_gcc = -Wl,-q -g
########################################################################
#
# buildutils/make.footer
########################################################################
#
ifdef CELL_TOP
include $(CELL_TOP)/buildutils/make.footer
else
include ../../../../buildutils/make.footer
endif
```

Example: Modifying ~/FFT16M/spu/Makefile for xlc

```
########################################################################
##
Target
########################################################################
#
SPU_COMPILER = xlc
PROGRAMS_spu:= fft_spu
LIBRARY_embed:= fft_spu.a
########################################################################
##
Local Defines
########################################################################
#
CFLAGS_xlc:= -g -qnounroll -O5
LDFLAGS_xlc:= -O5 -qflag=e:e -Wl,-q -g
########################################################################
#
```

```
# buildutils/make.footer
########################################################################
#
ifdef CELL_TOP
include $(CELL_TOP)/buildutils/make.footer
else
include ../../../../buildutils/make.footer
endif
```

3. Before the actual build, make sure you set the default compiler accordingly. To do this, issue the following command:

   ```
   /opt/cell/sdk/buildutils/cellsdk_select_compiler [gcc|xlc]
   ```

4. You are now ready for the build:

   ```
   cd ~/FFT16M ; CELL_TOP=/opt/cell/sdk make
   ```

Creating and working with profile data

After you have set up the project tree and have a successful build, you can collect and work with profile data.

About this task

The following steps describe how to:

1. Collect data with Cell-perf-counter tool (CPC).
2. Display the CPC report in the Visual Performance Analyzer (VPA).
3. Collect data with OProfile.
4. Display the OProfile report in VPA.
5. Use Feedback Directed Program Restructuring (FDPR-Pro) to gather frequency information.
6. Analyze and display FDPR-Pro frequency information in VPA.

Collecting data with CPC

The following procedure describes how to collect data with CPC.

1. Before collecting the application data, run a small test in order to verify that CPC is properly work. Type the following command to measure clock-cycles and branch instructions committed on both hardware threads for all processes on all CPUs for five seconds, and you should immediately see the following counter statistics:

   ```
   cpc --cpus all --time 5s --events C
   ```

2. Collect counter data for the FFT16M application. The following example counts PPC instructions committed in one event-set, and L1 cache load misses in a second event-set and writes the output in xml format (suitable for the counter analyzer) to the file fft_cpc.pmf:

   ```
   cd ~/FFT16M
   cpc --events C,2100,2119 --cpus all --xml fft_cpc.pmf \
      ./ppu/fft 40 1
   ```

3. This results in the following file:

   ~/FFT16M/fft_cpc.pmf

Example

A more useful way to use the CPC is to use the CPC **--interval** option to set a sampling interval. Setting a time interval in which events are counted can show the number of events occurring at different time units during the interval.

The following is an example of the CPC with the -interval option. Note the CPC is split onto two lines because of the width of the printed page. The CPC is normally one continuous line.

```
$ cpc -e 2100,2101,2106,2109 -e 2103,2104,2111,2119 -c all \
   --sampling-buffer-size 15 --interval 100000000 -X fft_cpc2.pmf \
   -t 10 ./fft 1 1 4 1 0
```

The output file fft_cpc2.pmf now includes counted events information for each of the following sampled events within the specified interval:

- Branch_Commit_t0
- Branch_Commit_t1
- Branch_Flush_t0
- Dispatch_Blocked_t0
- IERAT_Miss_t0
- IL1_Miss_Cycles_t0
- Instr_Flushed_t0
- PPC_Commit_t0

Displaying the CPC report in VPA

The generated counter information can now be visualized with the VPA tool.

Before you begin

You can download the VPA tool from

`http://www.alphaworks.ibm.com/tech/vpa/download`

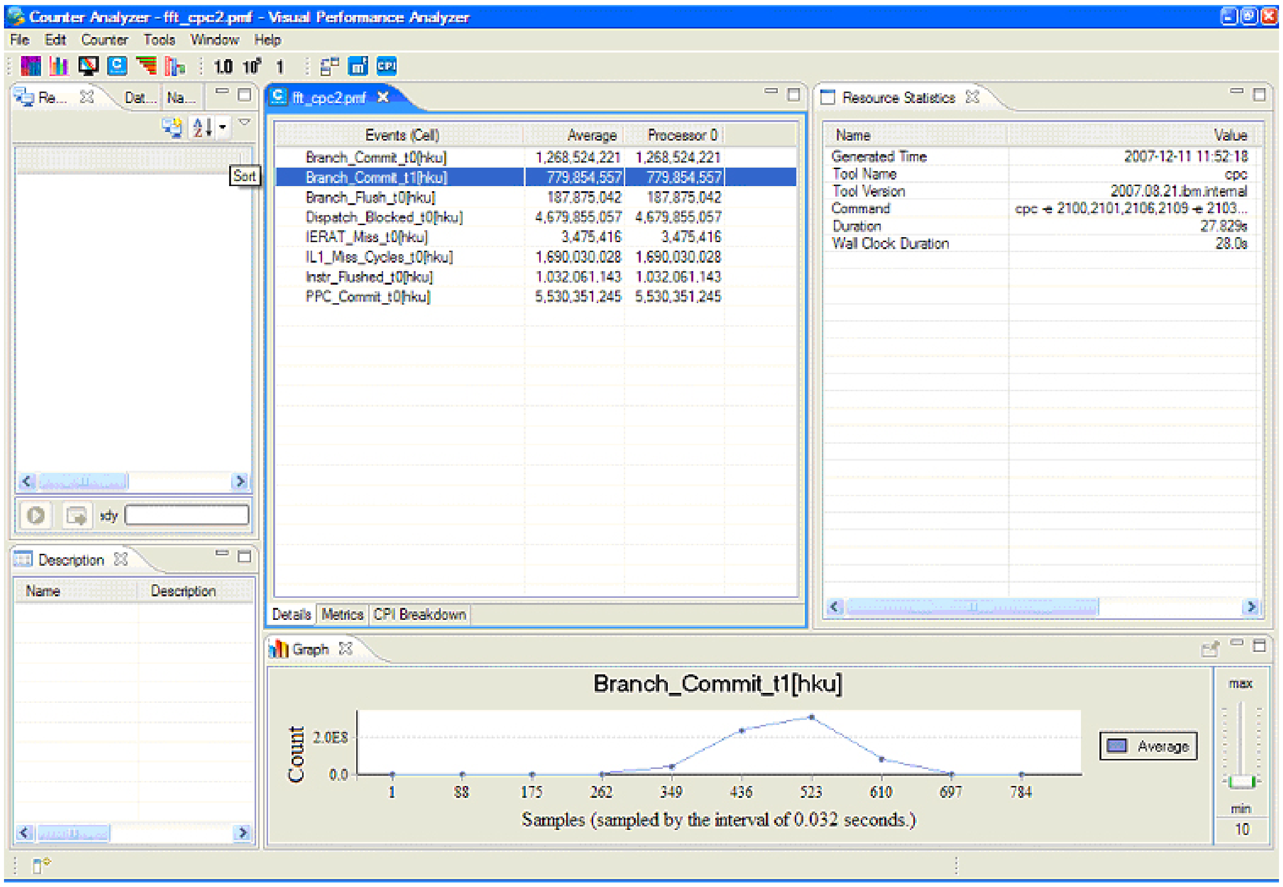
About this task

To display the counter information, do the following:

1. Open VPA and select **Tools** → **Counter Analyzer**.
2. Select **File** → **Open File**.
3. Locate the fft_cpc.pmf file and select it.

Example

The result is something similar to the following screen, which shows the collected counter information and a graph that displays the amount of events occurring at every time unit during the sampling interval:


Counter Analyzer - fft_cpc2.pmf - Visual Performance Analyzer
File Edit Counter Tools Window Help
fft_cpc2.pmf
Resource Statistics
Description
Details Metrics CPI Breakdown
Graph
Branch_Commit_t1[hku]
Count
2.0E8
0.0
1 88 175 262 349 436 523 610 697 784
Samples (sampled by the interval of 0.032 seconds.)
Average
max
min
10

Events (Cell)	Average	Processor 0
Branch_Commit_t0[hku]	1,268,524,221	1,268,524,221
Branch_Commit_t1[hku]	779,854,557	779,854,557
Branch_Flush_t0[hku]	187,875,042	187,875,042
Dispatch_Blocked_t0[hku]	4,679,855,057	4,679,855,057
IERAT_Miss_t0[hku]	3,475,416	3,475,416
IL1_Miss_Cycles_t0[hku]	1,690,030,028	1,690,030,028
Instr_Flushed_t0[hku]	1,032,061,143	1,032,061,143
PPC_Commit_t0[hku]	5,530,351,245	5,530,351,245

Name	Value
Generated Time	2007-12-11 11:52:18
Tool Name	cpc
Tool Version	2007.08.21.ibm.internal
Command	cpc -e 2100,2101,2106,2109 -e 2103...
Duration	27.829s
Wall Clock Duration	28.0s

Figure 2. Counter information displayed by the VPA

Collecting data with OProfile

The following steps generate appropriate profile information (suitable for the Profile Analyzer) for both PPU and SPU, from the FFT16M application:

Before you begin

About this task

Before you run Oprofile make sure you remove any previous profiling options and setup which can interfere with the profile generation process. To this, you should completely remove (as root) the file daemonrc located under `/root/.oprofile` as follows:

```
# rm /root/.oprofile/daemonrc
```

1. Initialize the OProfile environment for the SPU and run the fft workload to collect SPU average cycle events:

```
# As root
opcontrol --deinit
opcontrol --init
opcontrol --reset
opcontrol --separate=all --event=SPU_CYCLES:100000
opcontrol --start
# As regular user
```

```
fft 20 1
# As root
opcontrol --stop
opcontrol --dump
```

2. To generate the report, type the following:

   ```
   opreport -X -g -l -d -o fft.spu.opm
   ```

3. Repeat the steps for PPU. The following is an example of OProfile initialization and run for PPU profiling:

   ```
   # As root
   opcontrol --deinit
   opcontrol --init
   opcontrol --reset
   opcontrol --separate=all --event=CYCLES:100000
   opcontrol --start
   # As regular user
   fft 20 1
   # As root
   opcontrol --stop
   opcontrol --dump
   ```

4. To generate the report, type the following:

   ```
   opreport -X -g -l -d -o fft.ppu.opm
   ```

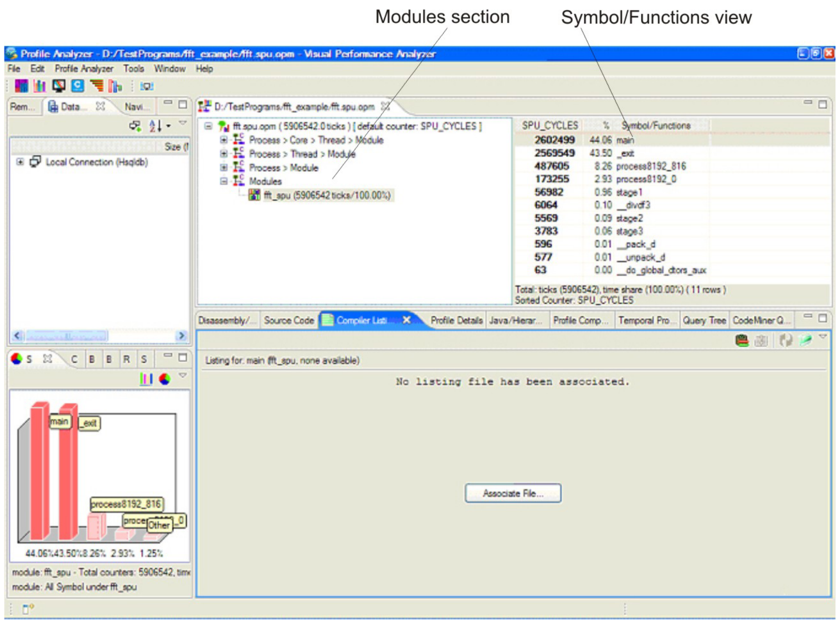
Displaying the OProfile report in VPA

Load the generated profile information with VPA and use the Profile Analyzer plugin to display the information.

About this task

To display the OProfile report in VPA, do the following:

1. Open VPA and select **Tools → Profile Analyzer**.
2. Choose **File → Open File**.
3. Locate the fft.spu.opm file and select it. The following screen is displayed:



Modules section
Symbol/Functions view
Profile Analyzer - D:/TestPrograms/fft_example/fft.spu.opm - Visual Performance Analyzer
File Edit Profile Analyzer Tools Window Help
Local Connection (Hsqldb)
D:/TestPrograms/fft_example/fft.spu.opm
Process > Core > Thread > Module
Process > Thread > Module
Process > Module
Modules
Disassembly/... Source Code Compiler List... Profile Details Java/Hierar... Profile Comp... Temporal Pro... Query Tree CodeMiner Q...
Associate File...
main
_exit
process8192_816
Other

fft.spu.opm (5906542.0 ticks) [default counter: SPU_CYCLES]
fft_spu (5906542 ticks/100.00%)

SPU_CYCLES	%	Symbol/Functions
2602499	44.06	main
2569549	43.50	_exit
487605	8.26	process8192_816
173255	2.93	process8192_0
56982	0.96	stage1
6064	0.10	__divdf3
5569	0.09	stage2
3783	0.06	stage3
596	0.01	__pack_d
577	0.01	__unpack_d
63	0.00	__do_global_dtors_aux

Total: ticks (5906542), time share (100.00%) (11 rows)
Sorted Counter: SPU_CYCLES

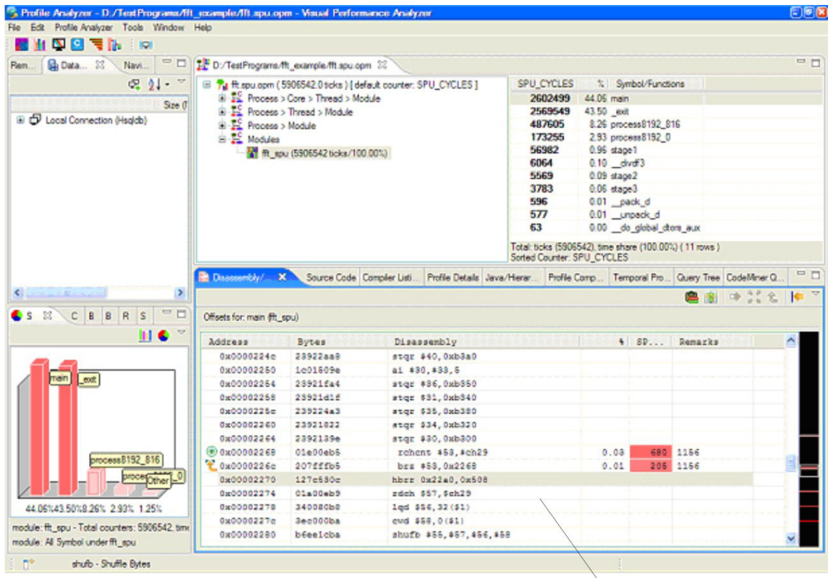
Listing for: main (fft_spu, none available)

```
No listing file has been associated.
```

44.06% 43.50% 8.26% 2.93% 1.25%
module: fft_spu - Total counters: 5906542, time
module: All Symbol under fft_spu

Figure 3. fft.spu.opm displayed in the Profile Analyzer

4. Examine the disassembly information by selecting the fft_spu entry contained inside the **Modules** section at the center of the screen (see Figure 3 on page 51), then double-click the **main** symbol in the**Symbol/Functions** view.
5. The result appears in the **Disassembly** view, as follows:


Profile Analyzer - D:/TestPrograms/fft_example/fft_spu.opm - Visual Performance Analyzer
File Edit Profile Analyzer Tools Window Help
D:/TestPrograms/fft_example/fft_spu.opm
fft_spu.opm (5906542.0 ticks) [default counter: SPU_CYCLES]
Process > Core > Thread > Module
Process > Thread > Module
Process > Module
Modules
fft_spu (5906542 ticks/100.00%)
Local Connection (Hsqldb)
Disassembly/...
Source Code
Compiler Listi...
Profile Details
Java/Hierar...
Profile Comp...
Temporal Pro...
Query Tree
CodeMiner Q...
main
_exit
process8192_816
Other
44.06%43.50%8.26% 2.93% 1.25%
module: fft_spu - Total counters: 5906542. time
module: All Symbol under fft_spu
shufb - Shuffle Bytes
Disassembly view

SPU_CYCLES	%	Symbol/Functions
2602499	44.06	main
2569549	43.50	_exit
487605	8.26	process8192_816
173255	2.93	process8192_0
56982	0.96	stage1
6064	0.10	__divdf3
5569	0.09	stage2
3783	0.06	stage3
596	0.01	__pack_d
577	0.01	__unpack_d
63	0.00	__do_global_dtors_aux

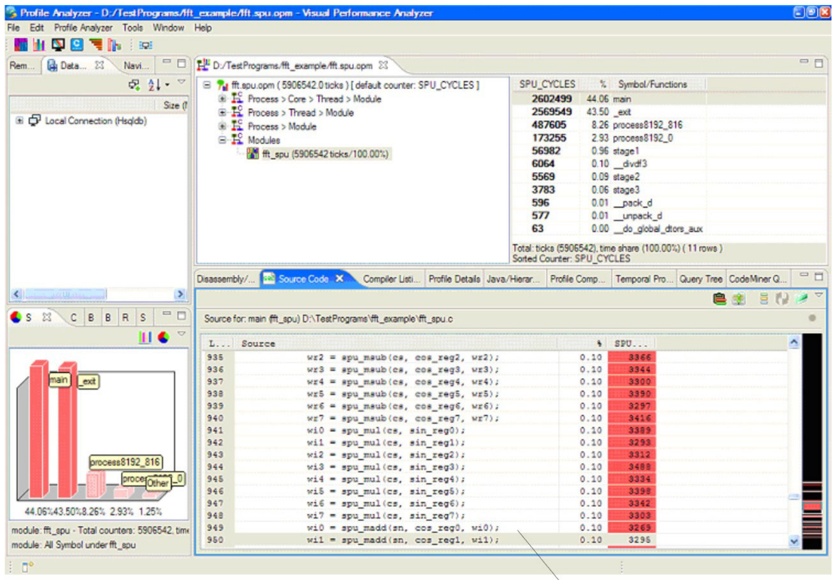
Total: ticks (5906542), time share (100.00%) (11 rows)
Sorted Counter: SPU_CYCLES

Offsets for: main (fft_spu)

```
Address      Bytes      Disassembly                    %     SP...  Remarks
0x0000224c   23922aa9   stqr $40,0xb3a0
0x00002250   1c01609e   ai $30,$33,5
0x00002254   23921fa4   stqr $36,0xb350
0x00002258   23921d1f   stqr $31,0xb340
0x0000225c   239224a3   stqr $35,0xb380
0x00002260   23921822   stqr $34,0xb320
0x00002264   2392139e   stqr $30,0xb300
0x00002268   01e00eb5    rchcnt $53,$ch29         0.03    680   1156
0x0000226c   207fffb5    brz $53,0x2268           0.01    205   1156
0x00002270   127c530c   hbrr 0x22a0,0x508
0x00002274   01a00eb9   rdch $57,$ch29
0x00002278   340080b8   lqd $56,32($1)
0x0000227c   3ec000ba   cwd $58,0($1)
0x00002280   b6ee1cba   shufb $55,$57,$56,$58
```

Figure 4. Disassembly view for fft_spu.opm

6. After you have double-clicked the symbol, the tool optionally asks you for that particular symbol's source code. If you have the source code, click the **Source Code** tab at the bottom center portion of the screen.


Profile Analyzer - D:/TestPrograms/fft_example/fft_spu.opm - Visual Performance Analyzer
File Edit Profile Analyzer Tools Window Help
D:/TestPrograms/fft_example/fft_spu.opm
fft_spu.opm (5906542.0 ticks) [default counter: SPU_CYCLES]
Process > Core > Thread > Module
Process > Thread > Module
Process > Module
Modules
fft_spu (5906542 ticks/100.00%)
Local Connection (Hsqldb)
Disassembly/...
Source Code
Compiler Listi...
Profile Details
Java/Hierar...
Profile Comp...
Temporal Pro...
Query Tree
CodeMiner Q...
main
_exit
process8192_816
Other
44.06%43.50%8.26% 2.93% 1.25%
module: fft_spu - Total counters: 5906542. time
module: All Symbol under fft_spu
Source Code view

SPU_CYCLES	%	Symbol/Functions
2602499	44.06	main
2569549	43.50	_exit
487605	8.26	process8192_816
173255	2.93	process8192_0
56982	0.96	stage1
6064	0.10	__divdf3
5569	0.09	stage2
3783	0.06	stage3
596	0.01	__pack_d
577	0.01	__unpack_d
63	0.00	__do_global_dtors_aux

Total: ticks (5906542), time share (100.00%) (11 rows)
Sorted Counter: SPU_CYCLES

Source for: main (fft_spu) D:\TestPrograms\fft_example\fft_spu.c

```
L...  Source                                          %     SPU...
935          wr2 = spu_msub(cs, cos_reg2, wr2);      0.10   3366
936          wr3 = spu_msub(cs, cos_reg3, wr3);      0.10   3344
937          wr4 = spu_msub(cs, cos_reg4, wr4);      0.10   3300
938          wr5 = spu_msub(cs, cos_reg5, wr5);      0.10   3390
939          wr6 = spu_msub(cs, cos_reg6, wr6);      0.10   3297
940          wr7 = spu_msub(cs, cos_reg7, wr7);      0.10   3416
941          wi0 = spu_mul(cs, sin_reg0);            0.10   3389
942          wi1 = spu_mul(cs, sin_reg1);            0.10   3293
943          wi2 = spu_mul(cs, sin_reg2);            0.10   3312
944          wi3 = spu_mul(cs, sin_reg3);            0.10   3488
945          wi4 = spu_mul(cs, sin_reg4);            0.10   3334
946          wi5 = spu_mul(cs, sin_reg5);            0.10   3398
947          wi6 = spu_mul(cs, sin_reg6);            0.10   3342
948          wi7 = spu_mul(cs, sin_reg7);            0.10   3303
949          wi0 = spu_madd(sn, cos_reg0, wi0);      0.10   3269
950          wi1 = spu_madd(sn, cos_reg1, wi1);      0.10   3295
```

Figure 5. Source view for fft_spu.opm

7. To load a listing file that was generated by the compiler, select the **Compiler Listing** tab and open the file. You can generate listing files at compile time using the GCC flags: **-Wa**, **-a**, **-ad** or with the **-qlist** IBM XLC compiler option.
8. Optionally, you can repeat the procedure to analyze the fft_ppu.opm profile results.

Using FDPR-Pro to gather frequency information

In addition to using FDPR-Pro to optimize applications, you can also use it in combination with the VPA Code Analyzer plugin to investigate application performance, while mapping back to the source code.

About this task

Initially, you need to set up FDPR-Pro to collect the profiling data as follows:

1. Clean up old profile information and create a temporary working directory for FDPR-Pro:
   ```
   cd ~/FFT16M/ppu ; rm -f *.mprof *.nprof ; mkdir sputmp
   ```
2. Configure the fft executable file with the following command:
   ```
   fdprpro fft -cell -spedir sputmp -a instr
   ```
   This results in the following output:
   ```
   FDPR-Pro Version 5.4.0.16 for Linux (CELL)
   fdprpro ./fft -cell -spedir sputmp -a instr
   > spe_extraction -> ./fft ...
   ...
   > processing_spe_file -> sputmp/fft_spu ...
   ...
   > reading_exe ...
   > adjusting_exe ...
   ...
   > analyzing ...
   > building_program_infrastructure ...
   @Warning: Relocations based on section .data -- section may not be
   reordered
   > building_profiling_cfg ...
   > spe_encapsulation -> sputmp/out ...
   >> processing_spe -> sputmp/out/fft_spu ...
   > instrumentation ...
   >> throw_&_catch_fixer ...
   >> adding_universal_stubs ...
   >> running_markers_and_instrumenters ...
   >> linker_stub_fixer ...
   >> dynamic_entries_table_sections_bus_fixer ...
   >> writing_profile_template -> fft.nprof ...
   > symbol_fixer ...
   > updating_executable ...
   > writing_executable -> fft.instr ...
   bye.
   ```
3. Run the generated profile:
   ```
   ./fft.instr 20 1
   ```
4. The following two files are created:
   - ~/FFT16M/ppu/fft.nprof # PPU profile information
   - ~/FFT16M/ppu/fft_spu.mprof # SPU profile information

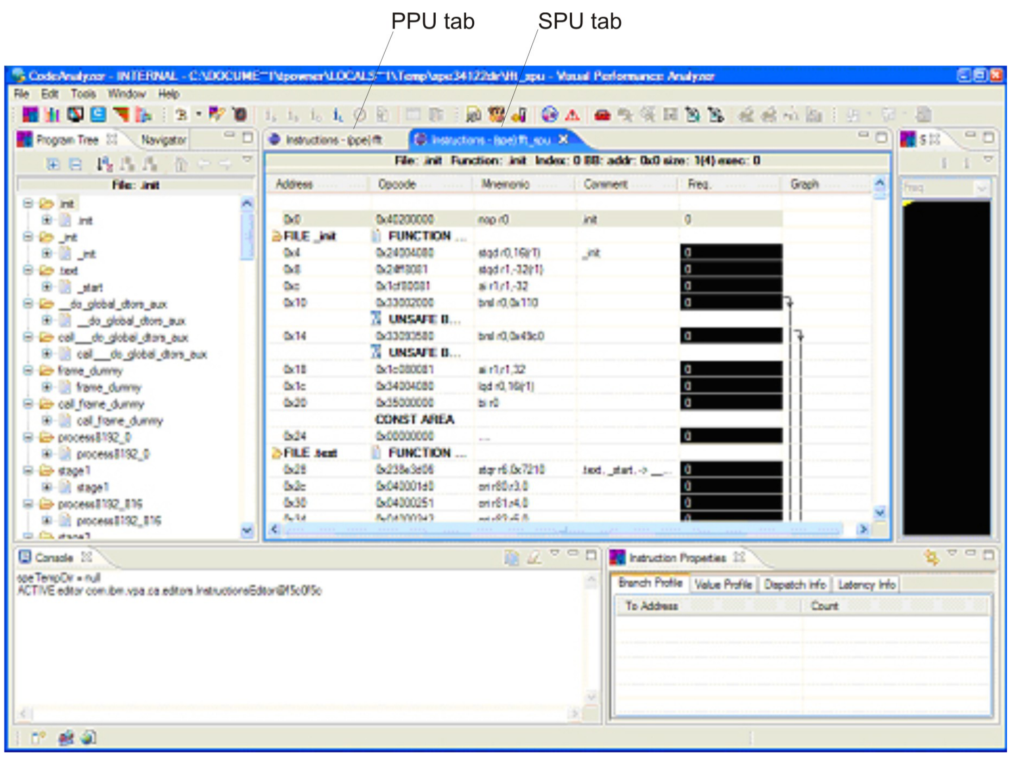
Analyzing and displaying FDPR-Pro frequency information in VPA

The VPA Code Analyzer plugin imports information from the FDPR-Pro, and displays it.

About this task

To import and display the information, do the following:

1. With VPA open, select **Tools → Code Analyzer**.
2. Go to **Tools → Code Analyzer → Analyze Executable** and locate the original fft executable file. Two editor tabs are displayed in the center view of the screen, one for PPU and one for SPU:


PPU tab
SPU tab
CodeAnalyzer - INTERNAL - C:\DOCUME~1\powner\LOCALS~1\Temp\spe34122dir\fft_spu - Visual Performance Analyzer
File Edit Tools Window Help
Program Tree
Navigator
Instructions - (ppe) fft
Instructions - (spe) fft_spu
Console
Instruction Properties
Branch Profile
Value Profile
Dispatch Info
Latency Info
To Address
Count

File: .init Function: .init Index: 0 BB: addr: 0x0 size: 1(4) exec: 0

Address	Opcode	Mnemonic	Comment	Freq.	Graph
0x0	0x40200000	nop r0	.init	0	
FILE _init	FUNCTION ...				
0x4	0x24004080	stqd r0,16(r1)	_init	0	
0x8	0x24ff8081	stqd r1,-32(r1)		0	
0xc	0x1cf80081	ai r1,r1,-32		0	
0x10	0x33002000	brsl r0,0x110		0	
	UNSAFE B...				
0x14	0x33093580	brsl r0,0x49c0		0	
	UNSAFE B...				
0x18	0x1c080081	ai r1,r1,32		0	
0x1c	0x34004080	lqd r0,16(r1)		0	
0x20	0x35000000	bi r0		0	
	CONST AREA				
0x24	0x00000000	...		0	
FILE .text	FUNCTION ...				
0x28	0x238e3d06	stqr r6,0x7210	.text, _start, -> __...	0	
0x2c	0x040001d0	ori r80,r3,0		0	
0x30	0x04000251	ori r81,r4,0		0	

```
speTempDir = null
ACTIVE editor com.ibm.vpa.ca.editors.InstructionsEditor@f5c0f5c
```

Figure 6. SPU and PPU editor tabs opened in the Code Analyzer

3. Associate the PPU profile information. To do this, select the PPU editor tab view and click **File → Code Analyzer → Add Profile Info**, then locate the fft.nprof file.

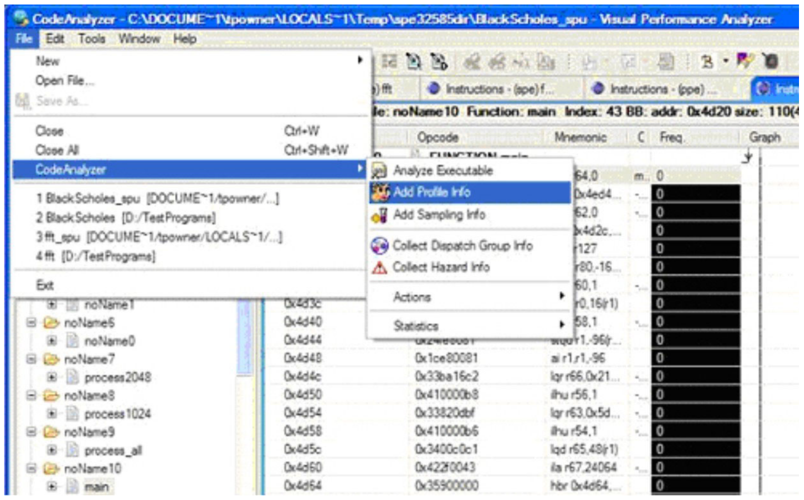


Figure 7. Adding profile information

4. Repeat the same procedure for the SPU part. To do this, select the SPU editor tab, and click **File → Code Analyzer → Add Profile Info**, then locate the fft_spu.mprof file. After you load the profile information, the instructions in both editors tabs are displayed in red, which indicates that these instructions are very frequently executed (refer the color-coded execution frequency scale displayed at the bottom of the screen).

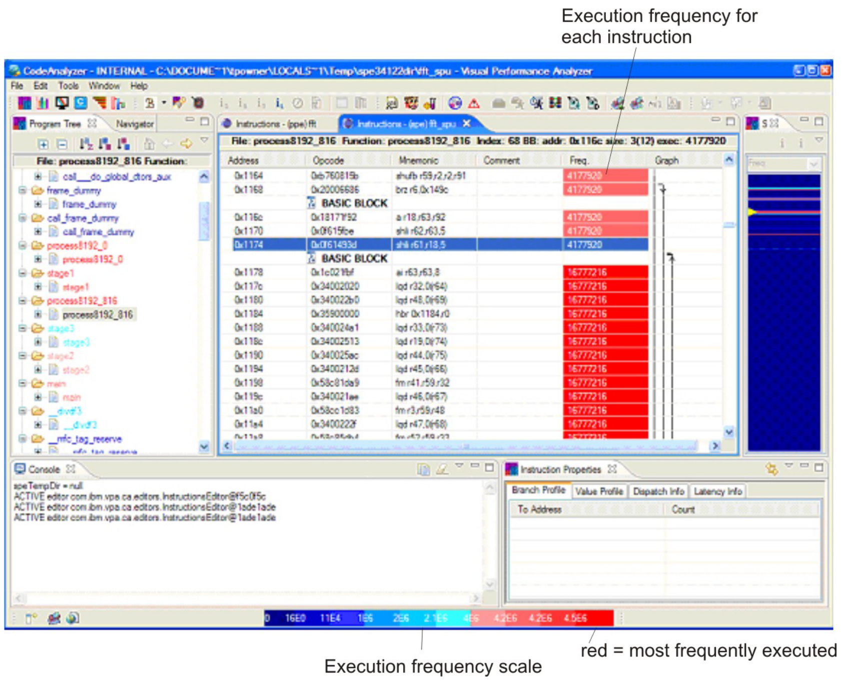
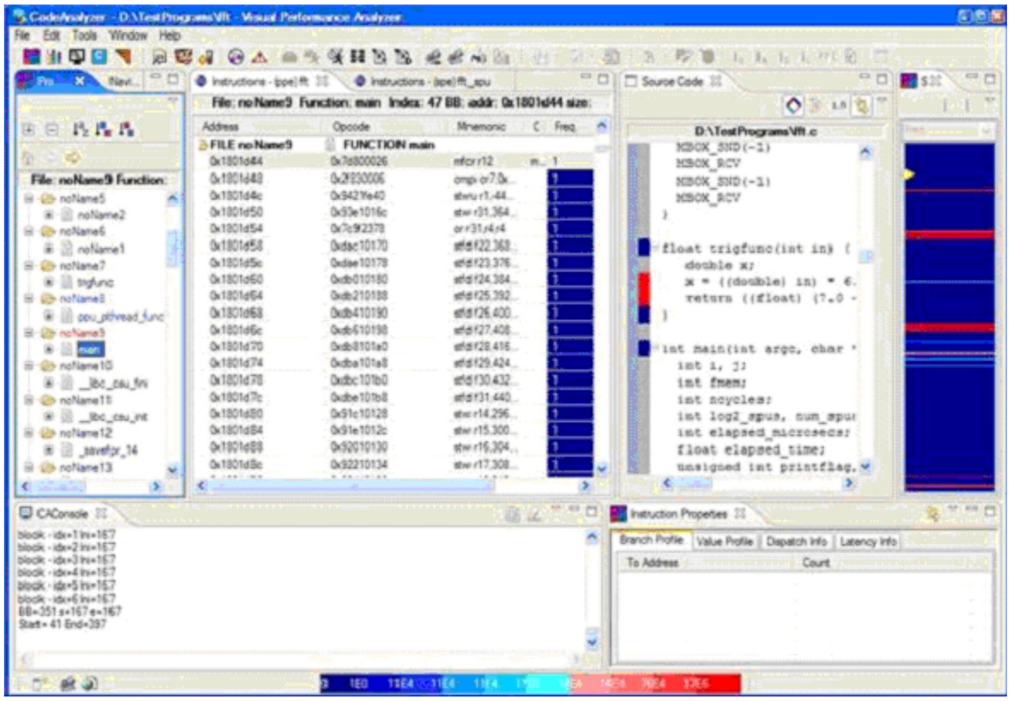

Execution frequency for each instruction
CodeAnalyzer - INTERNAL - C:\DOCUME~1\tpowner\LOCALS~1\Temp\spe34122dir\fft_spu - Visual Performance Analyzer
Program Tree
Navigator
Console
Instruction Properties
Branch Profile
Value Profile
Dispatch Info
Latency Info
BASIC BLOCK
Execution frequency scale
red = most frequently executed

Figure 8. Code Analyzer showing execution rates

5. You can also associate the source code by selecting symbols in the **Program Tree**. To do this, right click and select **Open Source Code**, then locate the source code. The **Source Code** tab displays rates of execution per line of source code in the center of the screen, see XREF. Click the **Link with Table button** at the top of the displayed source file, as follows:



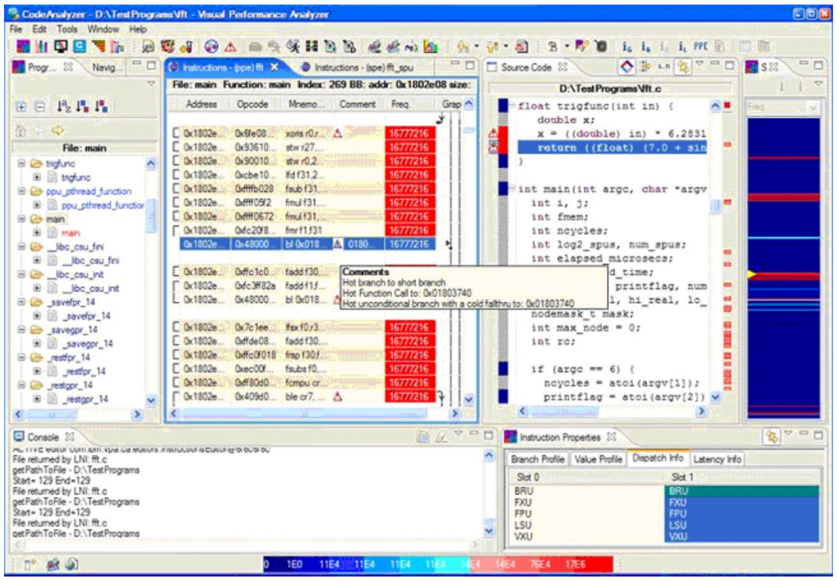
D:\TestPrograms\fft.c

```
  MBOX_SND(-1)
  MBOX_RCV
  MBOX_SND(-1)
  MBOX_RCV
}

float trigfunc(int in) {
   double x;
   x = ((double) in) * 6.
   return ((float) (7.0 -
}

int main(int argc, char *
  int i, j;
  int fmem;
  int ncycles;
  int log2_spus, num_spu
  int elapsed_microsecs;
  float elapsed_time;
  unsigned int printflag,
```

```
block - idx=1 ins=167
block - idx=2 ins=167
block - idx=3 ins=167
block - idx=4 ins=167
block - idx=5 ins=167
block - idx=6 ins=167
BB=351 s=167 e=167
Start= 41 End=397
```

Figure 9. Code Analyzer with source code tab

6. Calculate dispatch grouping boundaries for both fft PPE and fft SPU tabs. To do this, select each tab and click **Collect display information about dispatch groups**. You can also simultaneously click **Collect hazard info** to collect comments about performance bottlenecks above source lines that apply.



D:\TestPrograms\fft.c

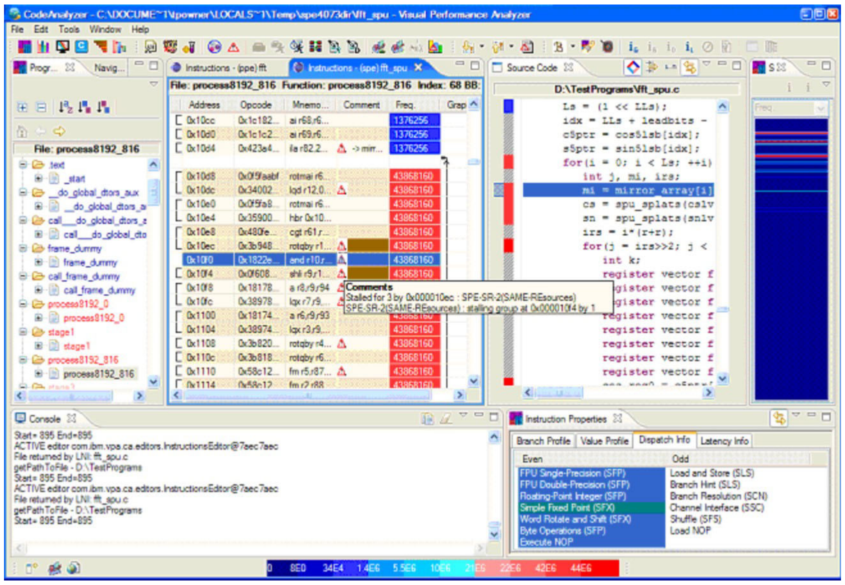
```
float trigfunc(int in) {
   double x;
   x = ((double) in) * 6.2831
   return ((float) (7.0 + sin
}

int main(int argc, char *argv
  int i, j;
  int fmem;
  int ncycles;
  int log2_spus, num_spus;
  int elapsed_microsecs;
                 d_time;
                  printflag, num
                 l, hi_real, lo_
  nodemask_t mask;
  int max_node = 0;
  int rc;

  if (argc == 6) {
   ncycles = atoi(argv[1]);
   printflag = atoi(argv[2])
```

Comments
Hot branch to short branch
Hot Function Call to: 0x01803740
Hot unconditional branch with a cold fallthru to: 0x01803740

```
File returned by LNI: fft.c
getPathToFile - D:\TestPrograms
Start= 129 End=129
File returned by LNI: fft.c
getPathToFile - D:\TestPrograms
Start= 129 End=129
File returned by LNI: fft.c
getPathToFile - D:\TestPrograms
```

Slot 0	Slot 1
BRU	BRU
FXU	FXU
FPU	FPU
LSU	LSU
VXU	VXU

Figure 10. The Collect Dispatch Grouping and the Performance Hazard buttons in the Code Analyzer for PPU instructions



CodeAnalyzer - C:\DOCUME~1\powner\LOCALS~1\Temp\spe4073dir\fft_spu - Visual Performance Analyzer

File: process8192_816 Function: process8192_816 Index: 68 BB:

Address	Opcode	Mnemo...	Comment	Freq.
0x10cc	0x1c182...	ai r68,r6...		1376256
0x10d0	0x1c1c2...	ai r69,r6...		1376256
0x10d4	0x423a4...	ila r82,2...	-> mirr...	1376256
0x10d8	0x0f5faabf	rotmai r6...		43868160
0x10dc	0x34002...	lqd r12,0...		43868160
0x10e0	0x0f5fa8...	rotmai r6...		43868160
0x10e4	0x35900...	hbr 0x10...		43868160
0x10e8	0x480fe...	cgt r61,r...		43868160
0x10ec	0x3b948...	rotqby r1...		43868160
0x10f0	0x1822e...	and r10,r...		43868160
0x10f4	0x0f608...	shli r9,r1...		43868160
0x10f8	0x18178...	a r8,r9,r94		
0x10fc	0x38978...	lqx r7,r9,...		
0x1100	0x18174...	a r6,r9,r93		
0x1104	0x38974...	lqx r3,r9,...		43868160
0x1108	0x3b820...	rotqby r4...		43868160
0x110c	0x3b818...	rotqby r6...		43868160
0x1110	0x58c12...	fm r5,r87...		43868160
0x1114	0x58c12...	fm r2,r88		43868160

Comments
Stalled for 3 by 0x000010ec : SPE-SR-2(SAME-REsources)
SPE-SR-2(SAME-REsources) : stalling group at 0x000010f4 by 1

Source Code — D:\TestPrograms\fft_spu.c

```
Ls = (1 << LLs);
idx = LLs + leadbits -
c5ptr = cos5lsb[idx];
s5ptr = sin5lsb[idx];
for(i = 0; i < Ls; ++i)
  int j, mi, irs;
  mi = mirror_array[i]
  cs = spu_splats(cslv
  sn = spu_splats(snlv
  irs = i*(r+r);
  for(j = irs>>2; j <
    int k;
    register vector f
    register vector f
    register vector f
    register vector f
    register vector f
    register vector f
    register vector f
    register vector f
```

Console

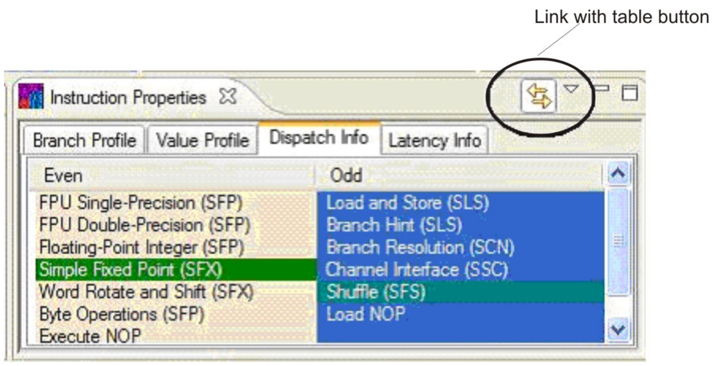
```
Start= 895 End=895
ACTIVE editor com.ibm.vpa.ca.editors.InstructionsEditor@7aec7aec
File returned by LNI: fft_spu.c
getPathToFile - D:\TestPrograms
Start= 895 End=895
ACTIVE editor com.ibm.vpa.ca.editors.InstructionsEditor@7aec7aec
File returned by LNI: fft_spu.c
getPathToFile - D:\TestPrograms
Start= 895 End=895
```

Instruction Properties — Branch Profile | Value Profile | Dispatch Info | Latency Info

Even	Odd
FPU Single-Precision (SFP)	Load and Store (SLS)
FPU Double-Precision (SFP)	Branch Hint (SLS)
Floating-Point Integer (SFP)	Branch Resolution (SCN)
Simple Fixed Point (SFX)	Channel Interface (SSC)
Word Rotate and Shift (SFX)	Shuffle (SFS)
Byte Operations (SFP)	Load NOP
Execute NOP	

Figure 11. Hazards Info commented source code for SPU instructions

7. Display pipeline population for each dispatch group. To do this, select the **Dispatch Info** tab (inside the **Instruction Properties tab**), then click the **Link with table** button.

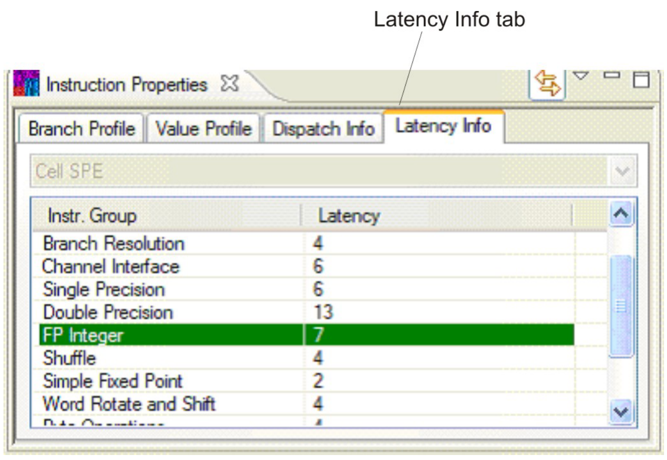


Instruction Properties — Branch Profile | Value Profile | Dispatch Info | Latency Info

Even	Odd
FPU Single-Precision (SFP)	Load and Store (SLS)
FPU Double-Precision (SFP)	Branch Hint (SLS)
Floating-Point Integer (SFP)	Branch Resolution (SCN)
Simple Fixed Point (SFX)	Channel Interface (SSC)
Word Rotate and Shift (SFX)	Shuffle (SFS)
Byte Operations (SFP)	Load NOP
Execute NOP	

Figure 12. The Dispatch info tab with the Link with Table option selected

The **Latency Info** tab displays latencies for each selected instruction, see Figure 13 on page 58.

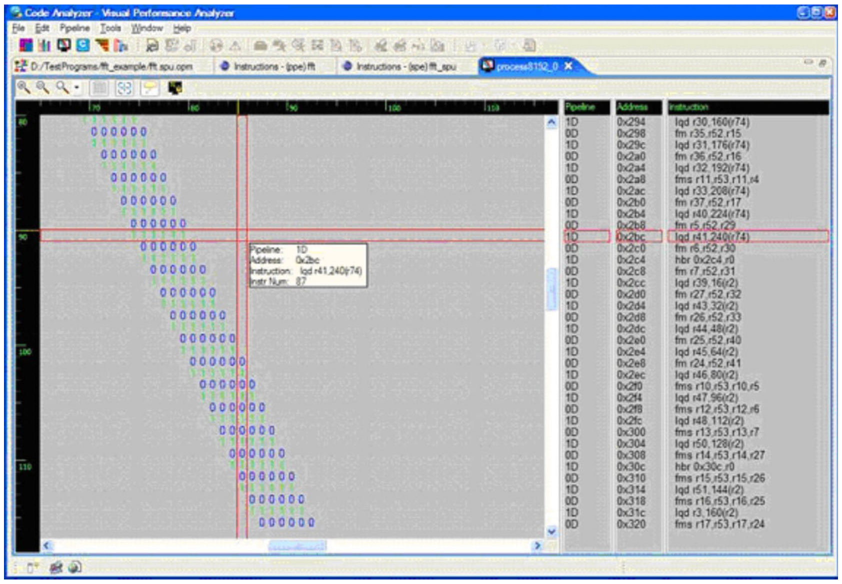


Instr. Group	Latency
Branch Resolution	4
Channel Interface	6
Single Precision	6
Double Precision	13
FP Integer	7
Shuffle	4
Simple Fixed Point	2
Word Rotate and Shift	4

Figure 13. Latency Info view

8. You can also use the Code Analyzer to inspect SPU Timing information at the pipeline level, with detailed stages of the Cell BE pipeline population. To do this, select the fft SPU editor tab, locate the desired symbol on the Program Tree, right-click and select **Show SPU-Timing**.

   The full pipeline layout is displayed as follows:

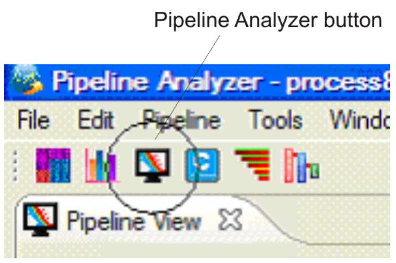

Code Analyzer - Visual Performance Analyzer
File Edit Pipeline Tools Window Help
D:/TestPrograms/fft_example/fft.spu.cpm
Instructions - (ppe) fft
Instructions - (spe) fft_spu
process8192_0

```
Pipeline:     1D
Address:      0x2bc
Instruction:  lqd r41,240(r74)
Instr Num:    87
```

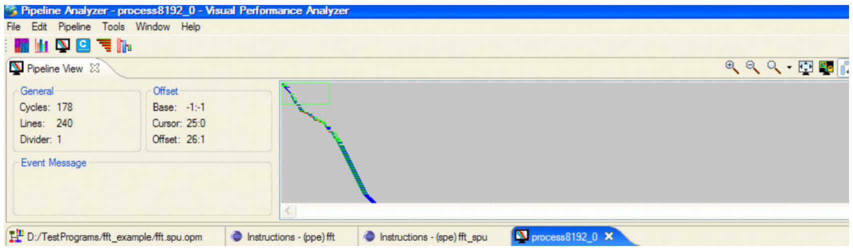
Pipeline	Address	Instruction
1D	0x294	lqd r30,160(r74)
0D	0x298	fm r35,r52,r15
1D	0x29c	lqd r31,176(r74)
0D	0x2a0	fm r36,r52,r16
1D	0x2a4	lqd r32,192(r74)
0D	0x2a8	fms r11,r53,r11,r4
1D	0x2ac	lqd r33,208(r74)
0D	0x2b0	fm r37,r52,r17
1D	0x2b4	lqd r40,224(r74)
0D	0x2b8	fm r5,r52,r29
1D	0x2bc	lqd r41,240(r74)
0D	0x2c0	fm r6,r52,r30
1D	0x2c4	hbr 0x2c4,r0
0D	0x2c8	fm r7,r52,r31
1D	0x2cc	lqd r39,16(r2)
0D	0x2d0	fm r27,r52,r32
1D	0x2d4	lqd r43,32(r2)
0D	0x2d8	fm r26,r52,r33
1D	0x2dc	lqd r44,48(r2)
0D	0x2e0	fm r25,r52,r40
1D	0x2e4	lqd r45,64(r2)
0D	0x2e8	fm r24,r52,r41
1D	0x2ec	lqd r46,80(r2)
0D	0x2f0	fms r10,r53,r10,r5
1D	0x2f4	lqd r47,96(r2)
0D	0x2f8	fms r12,r53,r12,r6
1D	0x2fc	lqd r48,112(r2)
0D	0x300	fms r13,r53,r13,r7
1D	0x304	lqd r50,128(r2)
0D	0x308	fms r14,r53,r14,r27
1D	0x30c	hbr 0x30c,r0
0D	0x310	fms r15,r53,r15,r26
1D	0x314	lqd r51,144(r2)
0D	0x318	fms r16,r53,r16,r25
1D	0x31c	lqd r3,160(r2)
0D	0x320	fms r17,r53,r17,r24

Figure 14. Cell Pipeline tab

9. Click the **Pipeline Analyzer** icon to navigate in the trace.



The following navigation view is displayed.


Pipeline Analyzer - process8192_0 - Visual Performance Analyzer
File
Edit
Pipeline
Tools
Window
Help
Pipeline View

General
Cycles: 178
Lines: 240
Divider: 1

Offset
Base: -1:-1
Cursor: 25:0
Offset: 26:1

Event Message

D:/TestPrograms/fft_example/fft.spu.opm
Instructions - (ppe) fft
Instructions - (spe) fft_spu
process8192_0

Figure 15. Navigation view

Creating and working with trace data

The PDT tool produces tracing data, which can be viewed and analyzed in the Trace Analyzer tool.

About this task

The following topics describe how to:

1. Create and work with trace data.
2. Import PDT data into VPA.

Collecting trace data with PDT

To properly collect trace data, you need to recompile the fft application according to the required PDT procedures:

1. Prepare the spu Makefile according to PDT requirements, depending on the compiler of your choice:

   Example: Modifying ~/FFT16M/spu/Makefile for gcc compiler

```
########################################################################
##
Target
########################################################################
#
PROGRAMS_spu:= fft_spu
LIBRARY_embed:= fft_spu.a
########################################################################
##
Local Defines
########################################################################
#
CFLAGS_gcc:= -g --param max-unroll-times=1 -Wall -Dmain=_pdt_main
-Dexit=_pdt_exit -DMFCIO_TRACE -DLIBSYNC_TRACE
LDFLAGS_gcc = -Wl,-q -g -L/usr/spu/lib/trace
INCLUDE = -I/usr/spu/include/trace
IMPORTS = -ltrace
########################################################################
#
```

```
# buildutils/make.footer
########################################################################
#
ifdef CELL_TOP
include $(CELL_TOP)/buildutils/make.footer
else
include ../../../../buildutils/make.footer
endif
Example 6-9 Modifying ~/FFT16M/spu/Makefile for xlc compiler
########################################################################
##
Target
########################################################################
#
SPU_COMPILER = xlc
PROGRAMS_spu:= fft_spu
LIBRARY_embed:= fft_spu.a
########################################################################
##
Local Defines
########################################################################
#
CFLAGS_xlc:= -g -qnounroll -O5
CPP_FLAGS_xlc := -I/usr/spu/include/trace -Dmain=_pdt_main
-Dexit=_pdt_exit -DMFCIO_TRACE -DLIBSYNC_TRACE
LDFLAGS_xlc:= -O5 -qflag=e:e -Wl,-q -g -L/usr/spu/lib/trace -ltrace
########################################################################
#
# buildutils/make.footer
########################################################################
#
ifdef CELL_TOP
include $(CELL_TOP)/buildutils/make.footer
else
include ../../../../buildutils/make.footer
endif
```

Modifying ~/FFT16M/spu/Makefile for xlc compiler

```
########################################################################
##
Target
########################################################################
#
SPU_COMPILER = xlc
PROGRAMS_spu:= fft_spu
LIBRARY_embed:= fft_spu.a
########################################################################
##
Local Defines
########################################################################
#
CFLAGS_xlc:= -g -qnounroll -O5
CPP_FLAGS_xlc := -I/usr/spu/include/trace -Dmain=_pdt_main
-Dexit=_pdt_exit -DMFCIO_TRACE -DLIBSYNC_TRACE
LDFLAGS_xlc:= -O5 -qflag=e:e -Wl,-q -g -L/usr/spu/lib/trace -ltrace
########################################################################
#
# buildutils/make.footer
########################################################################
#
ifdef CELL_TOP
include $(CELL_TOP)/buildutils/make.footer
else
include ../../../../buildutils/make.footer
endif
```

2. Rebuild the fft application:

```
cd ~/FFT16M ; CELL_TOP=/opt/cell/sdk make
```

3. Set up a configuration file with only the relevant stalls (mailboxes and read tag status for SPE):
   a. Copy the default xml to the place where the FFT runs, so that you can modify it.
      ```
      cp /usr/share/pdt/config/pdt_cbe_configuration.xml ~/FFT16M
      ```
   b. Open the copied file for editing. At the first line, change the application name value to `fft`.
   c. Search for <*configuration name="SPE"*> and below that line you find the MFCIO group tag. Set it to `active="false"`. Then delete the SPE_MFC group. This should be sufficient to trace only the *stalls* in the SPE.
4. Prepare the environment. To do this, set the following variables:
   ```
   export LD_LIBRARY_PATH=/usr/lib/trace
   export PDT_KERNEL_MODULE=/usr/lib/modules/pdt.ko
   export PDT_CONFIG_FILE=~/FFT16M/pdt_cbe_configuration.xml
   ```
5. Run the fft application at least three times to get better sampling:
   ```
   cd ~/FFT16M/ppu ; ./fft 1 1 4 1 0
   ```
   The following trace files should be available after you have run the application:
   - .pex,
   - .map
   - .trace
6. It is recommended that you use the PDTR tool to generate a textual summary report. To do this, type the following:
   ```
   $ /usr/bin/pdtr -trc <generated pdt trace file without its suffix>
   ```
   This produces a summary reports file with the suffix .pep.

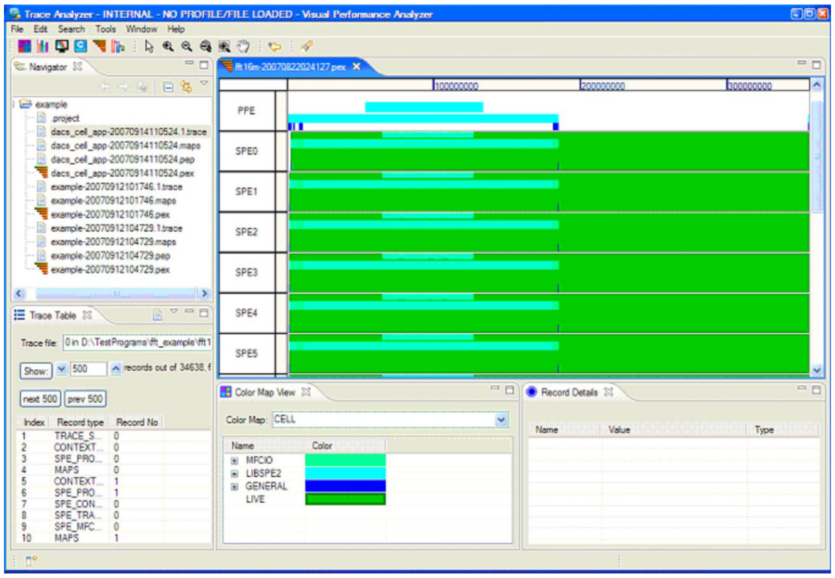
Importing PDT data into VPA

The VPA Trace Analyzer plugin displays the application's stages of execution. It works with data generated from the PDT tool, more specifically it reads information available in the generated .pex file.

About this task

Do the following to display the data in the Trace Analyzer:

1. With VPA open, select **Tools → Trace Analyzer**.
2. Go to **File → Open File** and locate the .pex file, generated in "Collecting trace data with PDT" on page 59. The following screen is displayed:



```
example
  .project
  dacs_cell_app-20070914110524.1.trace
  dacs_cell_app-20070914110524.maps
  dacs_cell_app-20070914110524.pep
  dacs_cell_app-20070914110524.pex
  example-20070912101746.1.trace
  example-20070912101746.maps
  example-20070912101746.pex
  example-20070912104729.1.trace
  example-20070912104729.maps
  example-20070912104729.pep
  example-20070912104729.pex
```

Trace file: 0 in D:\TestPrograms\fft_example\fft1

Show: 500 records out of 34638, f

next 500 prev 500

Index	Record type	Record No
1	TRACE_S...	0
2	CONTEXT...	0
3	SPE_PRO...	0
4	MAPS	0
5	CONTEXT...	1
6	SPE_PRO...	1
7	SPE_CON...	0
8	SPE_TRA...	0
9	SPE_MFC...	0
10	MAPS	1

Color Map: CELL

Name	Color
⊞ MFCIO	🟩
⊞ LIBSPE2	🟦
⊞ GENERAL	🟦
LIVE	🟩

Name	Value	Type

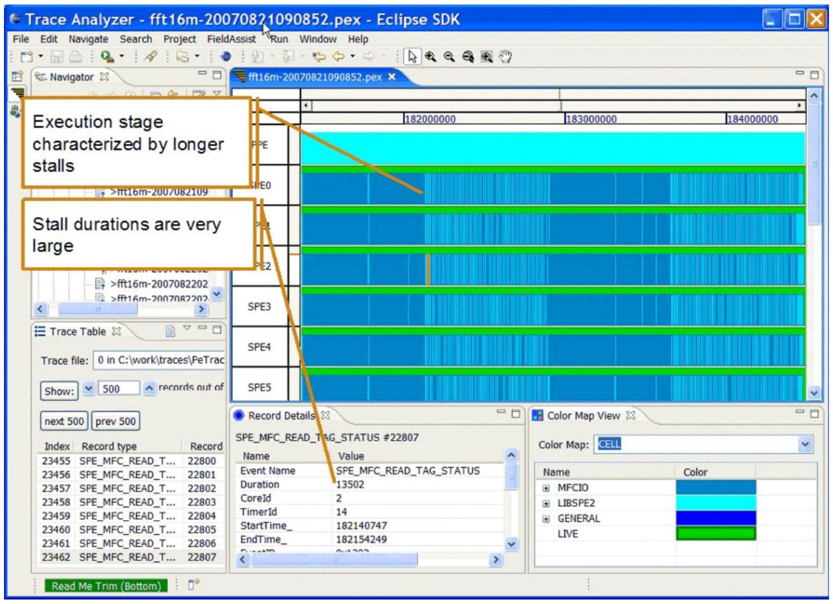
Figure 16. Trace Analyzer screen

Note:

- The default PDT_CONFIG_FILE for the SDK establishes the trace files prefix as "test". If you have not modified the file, look for the trace files, which have "test" as the prefix.
- Remember to unset LD_LIBRARY_PATH environment variable, before you run the original (non-PDT) binary later.

Figure 16 corresponds to the FFT16M application run with 16 SPEs and no huge pages. A less intensive blue has been selected for the MFC_IO group, and you now see the difference between the borders and the internals of the interval. The color map has been used to change the color of **read_in_mbox** to be red rather than its group's default blue. You see a huge stall in the middle. This is where the benchmark driver verifies the result of the test run to make sure the benchmark computes correctly. The timed run is the thin blue strip after the stall.

3. Zoom into this area, which is all that interests you in this benchmark.


Trace Analyzer - fft16m-20070821090852.pex - Eclipse SDK
File Edit Navigate Search Project FieldAssist Run Window Help
Navigator
fft16m-20070821090852.pex
Execution stage characterized by longer stalls
Stall durations are very large
182000000
183000000
184000000
PPE
SPE0
SPE1
SPE2
SPE3
SPE4
SPE5
Read Me Trim (Bottom)

Trace Table

Trace file: 0 in C:\work\traces\PeTrac

Show: 500 records out of

next 500 prev 500

Index	Record type	Record
23455	SPE_MFC_READ_T...	22800
23456	SPE_MFC_READ_T...	22801
23457	SPE_MFC_READ_T...	22802
23458	SPE_MFC_READ_T...	22803
23459	SPE_MFC_READ_T...	22804
23460	SPE_MFC_READ_T...	22805
23461	SPE_MFC_READ_T...	22806
23462	SPE_MFC_READ_T...	22807

Record Details

SPE_MFC_READ_TAG_STATUS #22807

Name	Value
Event Name	SPE_MFC_READ_TAG_STATUS
Duration	13502
CoreId	2
TimerId	14
StartTime_	182140747
EndTime_	182154249

Color Map View

Color Map: CELL

Name	Color
MFCIO	🟦
LIBSPE2	🟦
GENERAL	🟦
LIVE	🟩

Figure 17. Zoomed trace view

Figure 17 shows how the mailboxes (red bars) break the execution into six stages. Different stages have different behavior, for example, the third and sixth stages are much longer than the rest and have a lot of massive stalls. The Trace Analyzer allows you to click a stall to select it and to obtain further details (as shown in Figure 17 by the yellow highlight). The selection marker rulers on the left and top show the location of the selected item (and can be used to get back to it if you scroll away). The data collected by the PDT for the selected stall is shown in the record details window. In this example the stall is huge; almost 12K ticks.

You can now check Cell BE performance tips for a possible cause of the stall, and see that TLB misses is a possible cause, and huge pages are a possible fix.

Example

This is an example of how trace visualization allows you to discover a significant amount of information regarding the potential application problems.

What to do next

It is possible to observe how well balanced the application is by looking at how it executes and the start/stop time for each SPU. Because the Trace Analyzer breaks down the causes of stalls in the code by type, you can identify synchronization problems.

Appendix A. PDT troubleshooting

This section describes known issues that you may encounter and suggested solutions.

- Missing/wrong PDT_CONFIG_FILE environment variable at runtime

  Symptoms: when running the user application (with PDT enabled) the following message appears: ″(PDT) ERROR: Environment variable PDT_CONFIG_FILE was not set.″

  Solution: Set PDT_CONFIG_FILE to the right PDT configuration file.
- Missing/wrong LD_LIBRARY_PATH environment variable at runtime

  Symptoms: when running the user application (with PDT enabled) one of the following happen: a. Bus error: likely when a SPU starts running (when spe_context_run is called). b. Message: ″error while loading shared libraries: libtrace.so.3: cannot open shared object file: No such file or directory″.

  Solution: In both cases, the LD_LIBRARY_PATH is not set, incorrectly set (wrong path or 32/64 path error), or the paths' order is wrong where the PDT library path appears (/usr/lib[64]/trace) after another path so another library occlude the PDT library.
- Missing context switch notifications in trace file

  Symptoms: No context switch notifications records exist in the output trace files

  Solution: For RHEL 5.2, verify that the PDT kernel module is installed. For Fedora 9, ensure the you are using kernel version 2.6.25 or later.
- Config XML file errors

  Symptoms: the following messages are related:
  1. ″(PDT) ERROR: Invalid group name GROUP″
  2. ″(PDT) ERROR: Invalid group id GROUP_ID, for group : GROUP″
  3. ″(PDT) ERROR: Invalid subgroup name: SUBGROUP on group GROUP″
  4. ″(PDT) ERROR: Invalid event name: EVENT on subgroup SUBGROUP and group GROUP″
  5. ″(PDT) ERROR: Invalid event id EVENT_ID for event EVENT of subgroup SUBGROUP and group GROUP″
  6. ″(PDT) ERROR: The file FILE was not found″
  7. ″(PDT) ERROR: Invalid file FILE.″
  8. ″(PDT) ERROR: FILE is not a file.″
  9. ″(PDT) ERROR: Processor PROCESSOR does not appear in the configuration″
  10. ″(PDT) ERROR: Unknown processor type PROCESSOR″

  Solution: for 1, 2, 3, 4, and 5, the respective throttling sections of the config file should be checked. Problems 6, 7, and 8 mean that the value of the PDT_CONFIG_FILE environment variable is wrong (e.g. the file doesn't exists, it is not an XML config file, the file is corrupted, the value is a directory instead a config file, etc.). Finally, 9 and 10 indicate that the ″<configuration name=″PROCESSOR″>″ tag in the config file is missing or an unknown processor for PDT.
- 32/64 bit compilation and/or linking errors

  Symptoms: Cannot compile/link the PPU program with PDT libraries.

Solution: Re-compilation of the PPE code is needed when user events or dynamic control were added to the code, and re-linking is needed when compilation is needed or the SPU code is embedded in the PPE executable. Make sure that ″-I[/opt/cell/sysroot]/usr/include/trace″ flag is used for compilation and ″-ltrace″ and ″-L[/opt/cell/sysroot]/usr/lib[64]/trace″ are used for linkage.

- The program terminates with bus error when starting the SPE program run.

  Symptoms: The program terminates with bus error or segmentation fault when starting the SPE program run.

  Solution: First, check that the LD_LIBRARY_PATH is defined correctly (see problem ″Missing/wrong LD_LIBRARY_PATH environment variable at runtime″). Check that the SPE was compiled with PDT, specially look for the -Dmain=_pdt_main flag. If this is not the problem, recompile the SPE code with -Os (optimization for shorter code), in order to verify that PDT code in the SPE executable does not grow the executable to a size bigger than the 256K size allowed.
- Irrelevant context switch notifications

  Symptoms: In the output trace files, there are some context switch notifications with an unknown thread ID value.

  Solution: Do not relate to this context switches. Also, make sure that the SPEs are running before trying to access their problem state (e.g. send a mailbox, etc). In addition, only one user can use PDT at a time.
- Implicit declarations when compiling SPE or PPE

  Symptoms: ″warning: implicit declaration of function ′trace_XXXXX′ ″.

  Solution: The trace_* functions were added to the code but their respective ″#include″ are missing. (It is possible that the name of the function is wrong, too).
- Static event recording throttling are not working, the events are or are not recorded according to the user setup in the configuration file.

  Symptoms: Undesired events are recorded, or, desired events are not recorded.

  Solution: Check the XML configuration file pointed by the PDT_CONFIG_FILE environment variable.
- spu_mfcio events are not recorded.

  Symptoms: spu_mfcio (SPU) events are not recorded in the trace file.

  Solution: Make sure that the flag ″-DMFCIO_TRACE″ is in the compilation of ALL the SPE files. Also make sure that the flag ″-I[/opt/cell/sysroot]/usr/spu/include/trace″ appears as the first include flag (″-I″) in the compilation command.
- Large unexpected intervals in almost all the SPEs and PPE simultaneously

  Symptoms: The trace shows large intervals in many SPEs and PPE at the same time.

  Solution: If the intervals durations are intersected by a ″DAEMON interval″, then PDT was halted for some milliseconds by the OS. Do not infer that these large intervals are problems in your code.
- The output trace files are not in the right directory or are not found in the expected directory.

  Symptoms: No trace files in the expected directory.

  Solution: Check if the PDT_TRACE_OUTPUT is defined and see if the files were not written there. If the environment variable wasn't defined, check the output directory that is defined in the configuration file and see if the files were not written there. Check that the directory exists and that write permission is granted for the current user. Note that if both are not defined the trace output is directed to the current directory.

- Large amount of trace files are written to the disk and the program fails (although the program runs without PDT enabled).

  Symptoms: Many trace files are written to the disk and the program fails with error.

  Solution: Check space in the hard disk. Try using static or dynamic throttling to decrease the amount of events written to the trace files.

- PPE linking problem with -ltrace

  Symptoms: When linking the PPE code,″-ltrace″ and ″-L/usr/lib[64]/trace″ should be added, and then the message ″undefined reference to \`_Unwind_GetIPInfo@GCC_4.2.0′″ shows up.

  Solution: add also ″-lstdc++″ flag to the PPE linkage

- Opteron produces ″Illegal instruction″ when PDT is enabled

  Symptoms: The above is generated by PDT

  Solution: The problem is with the RDTSCP assembly instruction used by PDT. RDTSCP is a feature that is not found in all AMD processors. It was introduced in AMD′s NPT Family 0Fh processors. Make sure you are using one of those.

- PDTR problem with stripped executables

  Symptoms: In the pdtr output (.pep output file), instruction and/or data addresses are not mapped to symbolic names (or shows unknown(), no_map(), unknown-no_symbol_map()), and/or WARNINGS from pdtr tool indicating ″no symbol map″).

  Solution: ignore or rebuild the executable and do not strip

- Issues related to the PDT kernel module (RHEL 5.2 only):

  – Message ″insmod: error inserting ′pdt.ko′: -1 File exists″

    Symptom: The kernel is trying to be loaded by two processes (two persons trying to run a program with PDT at the same time).

    Solution: Do not run two processes with PDT at the same time.

  – Message ″insmod: can′t read /usr/lib/modules/pdt.ko′: No such file or directory″

    Symptom: The kernel cannot be loaded because it was not installed or the kernel module env variable is wrong. Note that the path in the message is the current module path that is trying to be loaded.

    Solution: Install the PDT kernel module if missing or set the PDT_KERNEL_MODULE to the actual directory.

  – Message ″ERROR: Module PDT does not exist in /proc/modules″

    Symptom: The kernel can not be unloaded because it was not loaded.

    Solution: Install the PDT kernel module if missing or set the PDT_KERNEL_MODULE to the actual directory.

  – Message ″There will be no error message (the first unloader will succeed)″

    Symptom: The kernel is being unloaded twice (two processes trying to unload the module, one owns it and the other tries to unload it).

    Solution: Do not run two processes with PDT at the same time.

Appendix B. Related documentation

This topic helps you find related information.

Document location

Links to documentation for the SDK are provided on the IBM developerWorks Web site located at:

`http://www.ibm.com/developerworks/power/cell/`

Click the **Docs** tab.

The following documents are available, organized by category:

Architecture

- *Cell Broadband Engine Architecture*
- *Cell Broadband Engine Registers*
- *SPU Instruction Set Architecture*

Standards

- *C/C++ Language Extensions for Cell Broadband Engine Architecture*
- *Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification*
- *SIMD Math Library Specification for Cell Broadband Engine Architecture*
- *SPU Application Binary Interface Specification*
- *SPU Assembly Language Specification*

Programming

- *Cell Broadband Engine Programmer's Guide*
- *Cell Broadband Engine Programming Handbook*
- *Cell Broadband Engine Programming Tutorial*

Library

- *Accelerated Library Framework for Cell Broadband Engine Programmer's Guide and API Reference*
- *Basic Linear Algebra Subprograms Programmer's Guide and API Reference*
- *Data Communication and Synchronization for Cell Broadband Engine Programmer's Guide and API Reference*
- *Example Library API Reference*
- *Fast Fourier Transform Library Programmer's Guide and API Reference*
- *LAPACK (Linear Algebra Package) Programmer's Guide and API Reference*
- *Mathematical Acceleration Subsystem (MASS)*
- *Monte Carlo Library Programmer's Guide and API Reference*
- *SDK 3.0 SIMD Math Library API Reference*
- *SPE Runtime Management Library*
- *SPE Runtime Management Library Version 1 to Version 2 Migration Guide*
- *SPU Runtime Extensions Library Programmer's Guide and API Reference*

- *Three dimensional FFT Prototype Library Programmer's Guide and API Reference*

Installation

- *SDK for Multicore Acceleration Version 3.1 Installation Guide*

Tools

- *Getting Started - XL C/C++ for Multicore Acceleration for Linux*
- *Compiler Reference - XL C/C++ for Multicore Acceleration for Linux*
- *Language Reference - XL C/C++ for Multicore Acceleration for Linux*
- *Programming Guide - XL C/C++ for Multicore Acceleration for Linux*
- *Installation Guide - XL C/C++ for Multicore Acceleration for Linux*
- *Getting Started - XL Fortran for Multicore Acceleration for Linux*
- *Compiler Reference - XL Fortran for Multicore Acceleration for Linux*
- *Language Reference - XL Fortran for Multicore Acceleration for Linux*
- *Optimization and Programming Guide - XL Fortran for Multicore Acceleration for Linux*
- *Installation Guide - XL Fortran for Multicore Acceleration for Linux*
- *Performance Analysis with the IBM Full-System Simulator*
- *IBM Full-System Simulator User's Guide*
- *IBM Visual Performance Analyzer User's Guide*

IBM PowerPC Base

- *IBM PowerPC Architecture Book*
  - *Book I: PowerPC User Instruction Set Architecture*
  - *Book II: PowerPC Virtual Environment Architecture*
  - *Book III: PowerPC Operating Environment Architecture*
- *IBM PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*

Appendix C. Accessibility features

Accessibility features help users who have a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

The following list includes the major accessibility features:
- Keyboard-only operation
- Interfaces that are commonly used by screen readers
- Keys that are tactilely discernible and do not activate just by touching them
- Industry-standard devices for ports and connectors
- The attachment of alternative input and output devices

IBM and accessibility

See the IBM Accessibility Center at http://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 U.S.A.

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

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

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

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

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

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

Trademarks

XDR is a trademark of Rambus Inc. in the United States and other countries.

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

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

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 works 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.

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.

Glossary

ABI. Application Binary Interface. This is the standard that a program follows to ensure that code generated by different compilers (and perhaps linking with various, third-party libraries) run correctly on Cell/B.E.. The ABI defines data types, register use, calling conventions and object formats.

ALF. Accelerated Library Framework. This an API that provides a set of services to help programmers solving data parallel problems on a hybrid system. ALF supports the multiple-program-multiple-data (MPMD) programming style where multiple programs can be scheduled to run on multiple accelerator elements at the same time. ALF offers programmers an interface to partition data across a set of parallel processes without requiring architecturally-dependent code.

API. Application Program Interface.

atomic operation. A set of operations, such as read-write, that are performed as an uninterrupted unit.

Auto-SIMDize. To automatically transform scaler code to vector code.

Barcelona Supercomputing Center. Spanish National Supercomputing Center, supporting IBM BladeCenter servers and Linux on Cell/B.E.

BE. Broadband Engine.

Broadband Engine. See *CBEA*.

BSC. See *Barcelona Supercomputing Center*.

C++. C++ is an object-orientated programming language, derived from C.

cache. High-speed memory close to a processor. A cache usually contains recently-accessed data or instructions, but certain cache-control instructions can lock, evict, or otherwise modify the caching of data or instructions.

call stub. A small piece of code used as a link to other code which is not immediately accessible.

Cell BE processor. The Cell BE processor is a multi-core broadband processor based on IBM's Power Architecture.

CBEA. Cell Broadband Engine Architecture. A new architecture that extends the 64-bit PowerPC Architecture. The CBEA and the Cell Broadband Engine are the result of a collaboration between Sony, Toshiba, and IBM, known as STI, formally started in early 2001.

Cell Broadband Engine processor. See *Cell BE*.

code section. A self-contained area of code, in particular one which may be used in an overlay segment.

coherence. Refers to memory and cache coherence. The correct ordering of stores to a memory address, and the enforcement of any required cache writebacks during accesses to that memory address. Cache coherence is implemented by a hardware snoop (or inquire) method, which compares the memory addresses of a load request with all cached copies of the data at that address. If a cache contains a modified copy of the requested data, the modified data is written back to memory before the pending load request is serviced.

compiler. A programme that translates a high-level programming language, such as C++, into executable code.

computational kernel. Part of the accelerator code that does stateless computation task on one piece of input data and generates corresponding output results.

compute task. An accelerator execution image that consists of a compute kernel linked with the accelerated library framework accelerator runtime library.

CPC. A tool for setting up and using the hardware performance counters in the Cell BE processor.

CPI. Cycles per instruction. Average number of clock cycles taken to perform one CPU instruction.

CPL. Common Public License.

cycle. Unless otherwise specified, one tick of the PPE clock.

Cycle-accurate simulation. See *Performance simulation*.

DaCS. The Data Communication and Synchronization (DaCS) library provides functions that focus on process management, data movement, data synchronization, process synchronization, and error handling for processes within a hybrid system.

DaCS Element. A general or special purpose processing element in a topology. This refers specifically to the physical unit in the topology. A DE can serve as a Host or an Accelerator.

DE. See DaCS element.

DMA. Direct Memory Access. A technique for using a special-purpose controller to generate the source and destination addresses for a memory or I/O transfer.

DMA command. A type of MFC command that transfers or controls the transfer of a memory location containing data or instructions. See *MFC*.

DMA list. A sequence of transfer elements (or list entries) that, together with an initiating DMA-list command, specify a sequence of DMA transfers between a single area of LS and discontinuous areas in main storage. Such lists are stored in an SPE's LS, and the sequence of transfers is initiated with a DMA-list command such as `getl` or `putl`. DMA-list commands can only be issued by programs running on an SPE, but the PPE or other devices can create and store the lists in an SPE's LS. DMA lists can be used to implement scatter-gather functions between main storage and the LS.

dual-issue. Issuing two instructions at once, under certain conditions. See *fetch group*.

EA. See *Effective address*.

ECC. Error-Correcting Code.

effective address. An address generated or used by a program to reference memory. A memory-management unit translates an effective address (EA) to a virtual address (VA), which it then translates to a real address (RA) that accesses real (physical) memory. The maximum size of the effective address space is 2^{64} bytes.

ELF. Executable and Linking Format. The standard object format for many UNIX operating systems, including Linux. Originally defined by AT®&T and placed in public domain. Compilers generate ELF files. Linkers link to files with ELF files in libraries. Systems run ELF files.

elfspe. The SPE that allows an SPE program to run directly from a Linux command prompt without needing a PPE application to create an SPE thread and wait for it to complete.

ext3. Extended file system 3. One of the file system options available for Linux partitions.

FDPR-Pro. Feedback Directed Program Restructuring. A feedback-based post-link optimization tool.

Fedora. Fedora is an operating system built from open source and free software. Fedora is free for anyone to use, modify, or distribute. For more information about Fedora and the Fedora Project, see the following Web site: http://fedoraproject.org/.

fence. An option for a barrier ordering command that causes the processor to wait for completion of all MFC commands before starting any commands queued after the fence command. It does not apply to these immediate commands: `getllar`, `putllc`, and `putlluc`.

FFT. Fast Fourier Transform.

firmware. A set of instructions contained in ROM usually used to enable peripheral devices at boot.

FSF. Free Software Foundation. Organization promoting the use of open-source software such as Linux.

FSS. IBM Full-System Simulator. IBM's tool which simulates the cell processor environment on other host computers.

GCC. GNU C compiler

GDB. GNU application debugger. A modified version of `gdb`, `ppu-gdb`, can be used to debug a Cell Broadband Engine program. The PPE component runs first and uses system calls, hidden by the SPU programming library, to move the SPU component of the Cell Broadband Engine program into the local store of the SPU and start it running. A modified version of gdb, spu-gdb, can be used to debug code executing on SPEs.

GNU. GNU is Not Unix. A project to develop free Unix-like operating systems such as Linux.

GPL. GNU General Public License. Guarantees freedom to share, change and distribute free software.

graph structure. A program design in which each child segment is linked to one or more parent segments.

group. A group construct specifies a collection of DaCS DEs and processes in a system.

guarded. Prevented from responding to speculative loads and instruction fetches. The operating system typically implements guarding, for example, on all I/O devices.

GUI. Graphical User Interface. User interface for interacting with a computer which employs graphical images and widgets in addition to text to represent the information and actions available to the user. Usually the actions are performed through direct manipulation of the graphical elements.

handle. A handle is an abstraction of a data object; usually a pointer to a structure.

host. A general purpose processing element in a hybrid system. A host can have multiple accelerators attached to it. This is often referred to as the master node in a cluster collective.

HTTP. Hypertext Transfer Protocol. A method used to transfer or convey information on the World Wide Web.

Hybrid. A module comprised of two Cell BE cards connected via an AMD Opteron processor.

IDE. Integrated Development Environment. Integrates the Cell/B.E. GNU tool chain, compilers, the Full-System Simulator, and other development components to provide a comprehensive, Eclipse-based development platform that simplifies Cell/B.E. development.

IDL. Interface definition language. Not the same as CORBA IDL

ILAR. IBM International License Agreement for early release of programs.

initrd. A command file read at boot

interrupt. A change in machine state in response to an exception. See *exception*.

intrinsic. A C-language command, in the form of a function call, that is a convenient substitute for one or more inline assembly-language instructions. Intrinsics make the underlying ISA accessible from the C and C++ programming languages.

ISO image. Commonly a disk image which can be burnt to CD. Technically it is a disk image of and ISO 9660 file system.

K&R programming. A reference to a well-known book on programming written by Dennis Kernighan and Brian Ritchie.

kernel. The core of an operating which provides services for other parts of the operating system and provides multitasking. In Linux or UNIX operating system, the kernel can easily be rebuilt to incorporate enhancements which then become operating-system wide.

L1. Level-1 cache memory. The closest cache to a processor, measured in access time.

L2. Level-2 cache memory. The second-closest cache to a processor, measured in access time. A L2 cache is typically larger than a L1 cache.

LA. Local address. A local store address of a DMA list. It is used as a parameter in a *MFC* command.

latency. The time between when a function (or instruction) is called and when it returns. Programmers often optimize code so that functions return as quickly as possible; this is referred to as the low-latency approach to optimization. Low-latency designs often leave the processor data-starved, and performance can suffer.

LGPL. Lesser General Public License. Similar to the *GPL*, but does less to protect the user's freedom.

libspe. A SPU-thread runtime management library.

list element. Same as transfer element. See *DMA list*.

lnop. A NOP (no-operation instruction) in a SPU's odd pipeline. It can be inserted in code to align for dual issue of subsequent instructions.

loop unrolling. A programming optimization that increases the step of a loop, and duplicates the expressions within a loop to reflect the increase in the step. This can improve instruction scheduling and memory access time.

LS. See *local store*.

LSA. Local Store Address. An address in the local store of a SPU through which programs running in the SPU, and DMA transfers managed by the MFC, access the local store.

main memory. See *main storage*.

main storage. The effective-address (EA) space. It consists physically of real memory (whatever is external to the memory-interface controller, including both volatile and nonvolatile memory), SPU LSs, memory-mapped registers and arrays, memory-mapped I/O devices (all I/O is memory-mapped), and pages of virtual memory that reside on disk. It does not include caches or execution-unit register files. See also *local store*.

Makefile. A descriptive file used by the `make`command in which the user specifies: (a) target program or library, (b) rules about how the target is to be built, (c) dependencies which, if updated, require that the target be rebuilt.

mailbox. A queue in a SPE's MFC for exchanging 32-bit messages between the SPE and the PPE or other devices. Two mailboxes (the SPU Write Outbound Mailbox and SPU Write Outbound Interrupt Mailbox) are provided for sending messages from the SPE. One mailbox (the SPU Read Inbound Mailbox) is provided for sending messages to the SPE.

main thread. The main thread of the application. In many cases, Cell BE architecture programs are multi-threaded using multiple SPEs running concurrently. A typical scenario is that the application consists of a main thread that creates as many SPE threads as needed and the application organizes them.

Mambo. Pre-release name of the IBM Full-System Simulator, see *FSS*

MASS. MASS and MASS/V libraries contain optimized scalar and vector math library operations.

MFC. Memory Flow Controller. Part of an SPE which provides two main functions: it moves data via DMA between the SPE's local store (LS) and main storage, and it synchronizes the SPU with the rest of the processing units in the system.

MFC proxy commands. *MFC* commands issued using the *MMIO* interface.

MPMD. Multiple Program Multiple Data. Parallel programming model with several distinct executable programs operating on different sets of data.

MT. See *multithreading*.

multithreading. Simultaneous execution of more than one program thread. It is implemented by sharing one software process and one set of execution resources but duplicating the architectural state (registers, program counter, flags and associated items) of each thread.

NaN. Not-a-Number. A special string of bits encoded according to the IEEE 754 Floating-Point Standard. A NaN is the proper result for certain arithmetic operations; for example, zero divided by zero = NaN. There are two types of NaNs, quiet NaNs and signaling NaNs. Signaling NaNs raise a floating-point exception when they are generated.

netboot. Command to boot a device from another on the same network. Requires a TFTP server.

node. A node is a functional unit in the system topology, consisting of one host together with all the accelerators connected as children in the topology (this includes any children of accelerators).

NUMA. Non-uniform memory access. In a multiprocessing system such as the Cell/B.E., memory is configured so that it can be shared locally, thus giving performance benefits.

Oprofile. A tool for profiling user and kernel level code. It uses the hardware performance counters to sample the program counter every N events.

overlay region. An area of storage, with a fixed address range, into which overlay segments are loaded. A region only contains one segment at any time.

overlay. Code that is dynamically loaded and executed by a running SPU program.

page table. A table that maps virtual addresses (VAs) to real addresses (RA) and contains related protection parameters and other information about memory locations.

parent. The parent of a DE is the DE that resides immediately above it in the topology tree.

PDF. Portable document format.

Performance simulation. Simulation by the IBM Full System Simulator for the Cell Broadband Engine in which both the functional behavior of operations and the time required to perform the operations is simulated. Also called cycle-accurate simulation.

PERL. Practical extraction and reporting language. A scripting programming language.

pipelining. A technique that breaks operations, such as instruction processing or bus transactions, into smaller stages so that a subsequent stage in the pipeline can begin before the previous stage has completed.

plugin. Code that is dynamically loaded and executed by running an SPU program. Plugins facilitate code overlays.

PPC-64. 64 bit implementation of the *PowerPC Architecture*.

PPC. See *Power PC*.

PPE. PowerPC Processor Element. The general-purpose processor in the Cell.

PPSS. PowerPC Processor Storage Subsystem. Part of the *PPE*. It operates at half the frequency of the *PPU* and includes an L2 cache and a Bus Interface Unit (BIU).

PPU. PowerPC Processor Unit. The part of the *PPE* that includes the execution units, memory-management unit, and L1 cache.

program section. See *code section*.

proxy. Allows many network devices to connect to the internet using a single IP address. Usually a single server, often acting as a firewall, connects to the internet behind which other network devices connect using the IP address of that server.

region. See *overlay region*.

root segment. Code that is always in storage when a SPU program runs. The root segment contains overlay control sections and may also contain code sections and data areas.

RPM. Originally an acronym for Red Hat Package Manager, and RPM file is a packaging format for one or more files used by many Linux systems when installing software programs.

Sandbox. Safe place for running programs or script without affecting other users or programs.

SDK. Software development toolkit for Multicore Acceleration. A complete package of tools for application development.

section. See *code section*.

segment. See *overlay segment* and *root segment*.

SFP. SPU Floating-Point Unit. This handles single-precision and double-precision floating-point operations.

signal. Information sent on a signal-notification channel. These channels are inbound registers (to a

SPE). They can be used by the PPE or other processor to send information to a SPE. Each SPE has two 32-bit signal-notification registers, each of which has a corresponding memory-mapped I/O (MMIO) register into which the signal-notification data is written by the sending processor. Unlike mailboxes, they can be configured for either one-to-one or many-to-one signalling. These signals are unrelated to UNIX signals. See *channel* and *mailbox*.

signal notification. See *signal*.

SIMD. Single Instruction Multiple Data. Processing in which a single instruction operates on multiple data elements that make up a vector data-type. Also known as vector processing. This style of programming implements data-level parallelism.

SIMDize. To transform scaler code to vector code.

SMP. Symmetric Multiprocessing. This is a multiprocessor computer architecture where two or more identical processors are connected to a single shared main memory.

SPE. Synergistic Processor Element. Extends the PowerPC 64 architecture by acting as cooperative offload processors (synergistic processors), with the direct memory access (DMA) and synchronization mechanisms to communicate with them (memory flow control), and with enhancements for real-time management. There are 8 SPEs on each cell processor.

SPE thread. A thread scheduled and run on a SPE. A program has one or more SPE threads. Each such thread has its own SPU local store (LS), 128 x 128-bit register file, program counter, and MFC Command Queues, and it can communicate with other execution units (or with effective-address memory through the MFC channel interface).

specific intrinsic. A type of C and C++ language extension that maps one-to-one with a single SPU assembly instruction. All SPU specific intrinsics are named by prefacing the SPU assembly instruction with `si_`.

splat. To replicate, as when a single scalar value is replicated across all elements of an SIMD vector.

SPMD. Single Program Multiple Data. A common style of parallel computing. All processes use the same program, but each has its own data.

SPU. Synergistic Processor Unit. The part of an SPE that executes instructions from its local store (LS).

spulet. 1) A standalone SPU program that is managed by a PPE executive. 2) A programming model that allows legacy C programs to be compiled and run on an SPE directly from the Linux command prompt.

stub. See *methodstub*.

synchronization. The order in which storage accesses are performed.

System X. This is a project-neutral description of the supervising system for a node.

tag group. A group of DMA commands. Each DMA command is tagged with a 5-bit tag group identifier. Software can use this identifier to check or wait on the completion of all queued commands in one or more tag groups. All DMA commands except `getllar`, `putllc`, and `putlluc` are associated with a tag group.

Tcl. Tool Command Language. An interpreted script language used to develop GUIs, application prototypes, Common Gateway Interface (CGI) scripts, and other scripts. Used as the command language for the Full System Simulator.

TFTP. Trivial File Transfer Protocol. Similar to, but simpler than the Transfer Protocol (FTP) but less capable. Uses UDP as its transport mechanism.

thread. A sequence of instructions executed within the global context (shared memory space and other global resources) of a process that has created (spawned) the thread. Multiple threads (including multiple instances of the same sequence of instructions) can run simultaneously if each thread has its own architectural state (registers, program counter, flags, and other program-visible state). Each SPE can support only a single thread at any one time. Multiple SPEs can simultaneously support multiple threads. The PPE supports two threads at any one time, without the need for software to create the threads. It does this by duplicating the architectural state. A thread is typically created by the `pthreads` library.

TLB. Translation Lookaside Buffer. An on-chip cache that translates virtual addresses (VAs) to real addresses (RAs). A TLB caches page-table entries for the most recently accessed pages, thereby eliminating the necessity to access the page table from memory during load/store operations.

tree structure. A program design in which each child segment is linked to a single parent segment.

TS. The transfer size parameter in an *MFC* command.

UDP. User Datagram Protocol. Transports data as a connectionless protocol, i.e. without acknowledgement or receipt. Fast but fragile.

user mode. The mode in which *problem state* software runs.

vector. An instruction operand containing a set of data elements packed into a one-dimensional array. The elements can be fixed-point or floating-point values. Most Vector/SIMD Multimedia Extension and SPU SIMD instructions operate on vector operands. Vectors are also called SIMD operands or packed operands.

virtual memory. The address space created using the memory management facilities of a processor.

virtual storage. See *virtual memory*.

VMA. Virtual memory address. See *virtual memory*.

work block. A basic unit of data to be managed by the framework. It consists of one piece of the partitioned data, the corresponding output buffer, and related parameters. A work block is associated with a task. A task can have as many work blocks as necessary.

workload. A set of code samples in the SDK that characterizes the performance of the architecture, algorithms, libraries, tools, and compilers.

work queue. An internal data structure of the accelerated library framework that holds the lists of work blocks to be processed by the active instances of the compute task.

X86. Generic name for Intel-based processors.

XDR. Rambus Extreme Data Rate DRAM memory technology.

XLC. The IBM optimizing C/C++ compiler.

yaboot. Linux utility which is a boot loader for PowerPC-based hardware.

Index

Printed in USA

SC34-2565-00