Software Development Kit for Multicore Acceleration
Version 3.1

Programmer's Guide

SC33-8325-03

Software Development Kit for Multicore Acceleration
Version 3.1

Programmer's Guide

SC33-8325-03

Note: Before using this information and the product it supports, read the general information in "Notices" on page 95.

Edition Notice

This edition applies to the 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 SC33-8325-02.

© Copyright International Business Machines Corporation 2006, 2008.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

© Copyright IBM Corp. 2006, 2008

Preface

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

- Chapter 1, "SDK technical overview," on page 1 describes the components of the SDK
- Chapter 2, "Programming with the SDK," on page 13 explains how to program applications for the Cell/B.E. platform
- Chapter 3, "Debugging Cell/B.E. applications," on page 23 describes how to debug your applications
- Chapter 4, "Debugging common Direct Memory Access (DMA) errors," on page 47
- Chapter 5, "Using the SPU GNU profiler," on page 63 describes how to sue the SPU GNU profiler tool
- Chapter 6, "Analyzing Cell/B.E. SPUs with kdump and crash," on page 67 describes a means of debugging kernel data related to SPUs through specific crash commands, by using a dumped kernel image.
- Chapter 7, "Using SPU code overlays," on page 71 describes how to use overlays

About this book

This book describes how to use the SDK to write applications. How to install SDK is described in a separate manual, *Software Development Kit for Multicore Acceleration Installation Guide*, and there is also a programming tutorial to help get you started.

What's new for SDK 3.1

This book includes information about the new functionality delivered with the SDK, and completely replaces the previous version of this book.

This new information includes:

- Information about sharing of MFC tag identifiers, sharing MFC tag masks, and interrupt safe MFC command requests, see "SDK programming policies and conventions" on page 18
- Chapter 4, "Debugging common Direct Memory Access (DMA) errors," on page 47
- GDB usability enhancements, see "Multi-location breakpoints" on page 37
- For the combined debugger, information about a new facility for symbol determination, see "Disambiguation of multiply-defined global symbols" on page 39
- There are now two versions of the gdbserver debugger, see "Debugging applications remotely" on page 43
- How to generate automatic overlays, see "Generating automatic overlay scripts" on page 89
- How to use the SPU GNU profiler, see Chapter 5, "Using the SPU GNU profiler," on page 63

© Copyright IBM Corp. 2006, 2008

- Performance-tool related information has been removed from this book and is now located in the *Performance Tools Guide*

Supported operating environments

This topic describes the SDK hardware and software requirements.

Supported hardware requirements

This topic describes the supported hardware requirements.

Cell/B.E. applications can be developed on the following platforms:

- x86
- x86-64
- 64-bit PowerPC® (PPC64)
- IBM BladeCenter QS21
- IBM BladeCenter QS22

Software requirements

This topic describes the SDK software requirements.

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.

Unsupported beta-level 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, work-arounds to problems are provided in the respective forums. The following libraries and utilities are provided on an ″as-is″ basis:

- ALF for Hybrid
- DaCS for Hybrid
- Hybrid Performance Tools
- 3D Fast Fourier Transform Library
- SPU Timing
- Security Toolkit Isolation, Crypto Library
- CPC RHEL user tool
- CPC Fedora user tool
- IBM BladeCenter® QS20

Getting support

The SDK is available through Passport Advantage® with full support at:
`http://www.ibm.com/software/passportadvantage`

You can locate documentation and other resources on the World Wide Web. Refer to the following Web sites:

- IBM BladeCenter systems, optional devices, services, and support information at

  `http://www.ibm.com/bladecenter/`

  For service information, select **Support**.
- developerWorks® Cell BE Resource Center at:

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

  To access the Cell BE forum on developerWorks, select **Community**.
- The Barcelona Supercomputing Center (BSC) Web site at

  `http://www.bsc.es/projects/deepcomputing/linuxoncell`
- There is also support for the Full-System Simulator and XL C/C++ Compiler through their individual alphaWorks® forums. If in doubt, start with the Cell BE architecture forum.
- The GNU Project debugger, GDB is supported through many different forums on the Web, but primarily at the GDB Web site

  `http://www.gnu.org/software/gdb/gdb.html`

This version of the SDK supersedes all previous versions of the SDK.

Related documentation

For a list of documentation referenced in this *Programmer's Guide*, see Appendix B. Related documentation.

Chapter 1. SDK technical overview

This section describes the contents of the SDK, where it is installed on the system, and how the various components work together.

It covers the following topics:

GNU tool chain

This topic provides an overview of the GNU tool chain.

The GNU tool chain contains the GNU Compiler Collection compilers for the C, C++, and Fortran programming languages (gcc, g++, and gfortran) for the PPU and the SPU. For the PPU it is a replacement for the native GCC compiler on PowerPC (PPC) platforms and it is a cross-compiler on X86. The GCC compiler for the PPU is the default and the Makefiles are configured to use it when building the libraries and samples.

The GCC compiler also contains a separate SPE cross-compiler that supports the standards defined in the following documents:

- *C/C++ Language Extensions for Cell Broadband Engine Architecture V2.6*. The GCC compiler shipped in SDK supports all language extension described in the specification except for the following:
  - The GCC compilers currently do not support alignment of stack variables greater than 16 bytes as described in section 1.3.1.
  - The GCC compilers currently do not support the optional alternate vector literal format specified in section 1.4.6.
  - The GCC compilers currently support mapping between SPU and VMX intrinsics as defined in section 5 only in C++ code.
  - The recommended vector `printf` format controls as specified in section 8.1.1 due to library restrictions.
  - The GCC compiler does not support the optional Altivec style of vector literal construction using parenthesis ("(" and ")"). The standard C method of array initialization using curly braces ("{" and "}") should be used.
  - The C99 complex math library as specified in section 8.1.1 due to library restrictions
  - The GCC compiler currently does not support the Vector Shift Right and Quadword Shift Right families of the SPU intrinsics (`spu_sr, spu_sra, spu srqw, spu_srqwbyte, spu_srqwbytebc`)
- *SPU Application Binary Interface (ABI) Specification V1.9*
- *SPU Instruction Set Architecture V1.2*

© Copyright IBM Corp. 2006, 2008

The associated assembler and linker additionally support the *SPU Assembly Language Specification V1.7*. The assembler and linker are common to both the GCC compiler and the "IBM XL C/C++ compiler."

GDB support is provided for both PPU and SPU debugging, and the debugger client can be in the same process or a remote process. GDB also supports combined (PPU and SPU) debugging.

On a non-PPC system, the install directory for the GNU tool chain is `/opt/cell/toolchain`. There is a single `bin` subdirectory, which contains both PPU and SPU tools.

On a PPC64 or an IBM BladeCenter QS21, or IBM BladeCenter QS22, both tool chains are installed into `/usr`. See "SDK directories" on page 13 for further information.

IBM XL C/C++ compiler

IBM XL C/C++ for Multicore Acceleration for Linux is an advanced, high-performance cross-compiler that is tuned for the CBEA.

The XL C/C++ compiler, which is hosted on an x86, IBM PowerPC technology-based system, or a IBM BladeCenter QS21, or IBM BladeCenter QS22, generates code for the PPU or SPU. The compiler requires the GCC toolchain for the CBEA, which provides tools for cross-assembling and cross-linking applications for both the PPE and SPE.

IBM XL C/C++ supports the revised 2003 International C++ Standard *ISO/IEC 14882:2003(E), Programming Languages -- C++* and the *ISO/IEC 9899:1999, Programming Languages -- C standard*, also known as C99. The compiler also supports:

- The C89 Standard and K & R style of programming
- Language extensions for vector programming
- Language extensions for SPU programming
- Numerous GCC C and C++ extensions to help users port their applications from GCC.

The XL C/C++ compiler available for the SDK supports the languages extensions as specified in the *IBM XL C/C++ Advanced Edition for Multicore Acceleration for Linux Language Reference*.

The XL compiler also contains a separate SPE cross-compiler that supports the standards defined in the following documents:

- *C/C++ Language Extensions for Cell Broadband Engine Architecture V2.6*. The XL compiler shipped in SDK supports all language extension described in the specification except for the following:
  - The XL compilers currently do not support the `__builtin_expect_call builtin` function call
  - The XL compilers currently support mapping between SPU and VMX intrinsics as defined in section 5 only in C++ code
  - The recommended vector printf format controls as specified in section 8.1.1 due to library restrictions
  - The C99 complex math library as specified in section 8.1.1 due to library restrictions

– The SPU XL compilers currently do not support the Vector Shift Right and Quadword Shift Right families of the SPU intrinsics (spu_sr, spu_sra, spu srqw, spu_srqwbyte, spu_srqwbytebc)

- *SPU Application Binary Interface (ABI) Specification Version 1.9*
- *SPU Instruction Set Architecture Version 1.2*

For information about the XL C/C++ compiler invocation commands and a complete list of options, refer to the *IBM XL C/C++ Advanced Edition for Multicore Acceleration for Linux Programming Guide*.

Program optimization is described in *IBM XL C/C++ Advanced Edition for Multicore Acceleration for Linux Programming Guide*.

The XL C/C++ for Multicore Acceleration for Linux compiler is installed into the`/opt/ibmcmp/xlc/cbe/<compiler version number>` directory. Documentation is located on the following Web site:

`http://publib.boulder.ibm.com/infocenter/cellcomp/v9v111/index.jsp`

Linux kernel

For the IBM IBM BladeCenter QS21 and IBM BladeCenter QS22, the kernel is installed into the `/boot` directory, `yaboot.conf` is modified and a reboot is required to activate this kernel.

The `cellsdk install` task is documented in the *SDK Installation Guide*.

Note: The `cellsdk uninstall` command does not automatically uninstall the kernel. This avoids leaving the system in an unusable state.

Libraries and frameworks

This topic provides a brief overview of the Cell/B.E. libraries.

The following libraries are described:

SPE Runtime Management Library Version 2.3

The SPE Runtime Management Library (libspe) constitutes the standardized low-level application programming interface (API) for application access to the Cell/B.E. SPEs.

This library provides an API to manage SPEs that is neutral with respect to the underlying operating system and its methods. Implementations of this library can provide additional functionality that allows for access to operating system or implementation-dependent aspects of SPE runtime management. These capabilities are not subject to standardization and their use may lead to non-portable code and dependencies on certain implemented versions of the library.

The `elfspe` is a PPE program 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.

For the IBM BladeCenter QS21 and IBM BladeCenter QS22, the SDK installs the `libspe` headers, libraries, and binaries into the `/usr` directory and the standalone SPE executive, `elfspe`, is registered with the kernel during boot by commands added to `/etc/rc.d/init.d` using the `binfmt_misc` facility.

For the simulator, the `libspe` and `elfspe` binaries and libraries are preinstalled in the same directories in the system root image and no further action is required at install time.

SIMD math libraries

The SIMD math library provides short vector versions of the math functions.

The traditional math functions are scalar instructions, and do not take advantage of the powerful Single Instruction, Multiple Data (SIMD) vector instructions available in both the PPU and SPU in the Cell/B.E. Architecture. SIMD instructions perform computations on short vectors of data in parallel, instead of on individual scalar data elements. They often provide significant increases in program speed because more computation can be done with fewer instructions.

While the SIMD math library provides short vector versions of math functions, the MASS library provides long vector versions. These vector versions conform as closely as possible to the specifications set out by the scalar standards.

The SIMD math library is provided by the SDK as both a linkable library archive and as a set of inline function headers. The names of the SIMD math functions are formed from the names of the scalar counterparts by appending a vector type suffix to the standard scalar function name. For example, the SIMD version of the absolute value function `abs()`, which acts on a vector of long integers, is called `absi4()`. Inline versions of functions are prefixed with the character "_" (underscore), so the inline version of `absi4()` is called `_absi4()`.

For more information about the SIMD math library, refer to *SIMD Math Library Specification for Cell Broadband Engine Architecture Version 1.1*.

Mathematical Acceleration Subsystem (MASS) libraries

The Mathematical Acceleration Subsystem (MASS) consists of libraries of mathematical intrinsic functions, which are tuned specifically for optimum performance on the Cell/B.E. processor.

Currently the 32-bit, 64-bit PPU, and SPU libraries are supported. These libraries:

- Include both scalar and vector functions
- Are thread-safe
- Support both 32- and 64-bit compilations
- Offer improved performance over the corresponding standard system library routines
- Are intended for use in applications where slight differences in accuracy or handling of exceptional values can be tolerated

You can find information about using these libraries on the MASS Web site:

`http://www.ibm.com/software/awdtools/mass`

ALF library

The ALF provides a programming environment for data and task parallel applications and libraries.

The ALF API provides library developers with a set of interfaces to simplify library development on heterogenous multi-core systems. Library developers can use the provided framework to offload computationally intensive work to the accelerators. More complex applications can be developed by combining the several function offload libraries. Application programmers can also choose to implement their applications directly to the ALF interface.

ALF supports the multiple-program-multiple-data (MPMD) programming module where multiple programs can be scheduled to run on multiple accelerator elements at the same time.

The ALF functionality includes:

- Data transfer management
- Parallel task management
- Double buffering
- Dynamic load balancing

With the provided platform-independent API, you can also create descriptions for multiple compute tasks and define their ordering information execution orders by defining task dependency. Task parallelism is accomplished by having tasks without direct or indirect dependencies between them. The ALF runtime provides an optimal parallel scheduling scheme for the tasks based on given dependencies.

From the application or library programmer's point of view, ALF consists of the following two runtime components:

- A host runtime library
- An accelerator runtime library

The host runtime library provides the host APIs to the application. The accelerator runtime library provides the APIs to the application's accelerator code, usually the computational kernel and helper routines. This division of labor enables programmers to specialize in different parts of a given parallel workload.

The runtime framework handles the underlying task management, data movement, and error handling, which means that the focus is on the kernel and the data partitioning, not the direct memory access (DMA) list creation or the lock management on the work queue.

The ALF APIs are platform-independent and their design is based on the fact that many applications targeted for Cell/B.E. or multi-core computing follow the general usage pattern of dividing a set of data into self-contained blocks, creating a list of data blocks to be computed on the SPE, and then managing the distribution of that data to the various SPE processes. This type of control and compute process usage scenario, along with the corresponding work queue definition, are the fundamental abstractions in ALF.

DaCS library

The DaCS library provides a set of services for handling process-to-process communication in a heterogeneous multi-core system.

In addition to the basic message passing service these include:

- Mailbox services
- Resource reservation
- Process and process group management
- Process and data synchronization
- Remote memory services
- Error handling

The DaCS services are implemented as a set of APIs providing an architecturally neutral layer for application developers. These APIs are available in both the C and Fortran programming languages. They structure the processing elements, referred to as DaCS Elements (DE), into a hierarchical topology. This includes general purpose elements, referred to as Host Elements (HE), and special processing elements, referred to as Accelerator Elements (AE). Host elements usually run a full operating system and submit work to the specialized processes which run in the Accelerator Elements.

Fast Fourier Transform library

The Fast Fourier Transform (FFT) library handles a wide range of FFTs.

It consists of the following:

- API for the following routines used in single precision:
  - FFT Real -> Complex 1D
  - FFT Complex-Complex 1D
  - FFT Complex -> Real 1D
  - FFT Complex-Complex 2D for frequencies (from 1000x1000 to 2500x2500)

  The implementation manages sizes up to 10000 and handles multiples of 2, 3, and 5 as well as powers of those factors, plus one arbitrary factor as well. User code running on the PPU makes use of the CBE FFT library by calling one of either 1D or 2D streaming functions.
- Power-of-two-only 2D FFT code for complex-to-complex single and double precision processing.

Both parts of the library run using a common interface that contains an initialization and termination step, and an execution step which can process "one-at-a-time" requests (streaming) or entire arrays of requests (batch).

Monte Carlo libraries

The Monte Carlo libraries are a Cell/B.E. implementation of Random Number Generator (RNG) algorithms and transforms. The objective of this library is to provide functions needed to perform Monte Carlo simulations.

The following RNG algorithms are implemented:

- Hardware-based
- Kirkpatrick-Stoll
- Mersenne Twister
- Sobol

The following transforms are provided:

- Box-Mueller
- Moro's Inversion

- Polar Method

Basic Linear Algebra Sublibrary

This topic provides a short overview of the Basic Linear Algebra Sublibrary (BLAS).

The BLAS library is based upon a published standard interface, see the BLAS Technical Forum Standard document available at

`http://www.netlib.org/blas/blast-forum/blas-report.pdf`

for commonly-used linear algebra operations in high-performance computing (HPC) and other scientific domains.

It is widely used as the basis for other high quality linear algebra software, for example LAPACK and ScaLAPACK. The Linpack (HPL) benchmark largely depends on a single BLAS routine (DGEMM) for good performance.

The BLAS APIs are available as standard ANSI C and standard FORTRAN 77/90 interfaces. BLAS implementations are also available in open-source (netlib.org). Based on their functionality, BLAS routines are categorized into the following three levels:

- Level 1 routines are for scalar and vector operations
- Level 2 routines are for matrix-vector operations
- Level 3 routines are for matrix-matrix operations

BLAS routines can have up to four versions – real single precision, real double precision, complex single precision and complex double precision, represented by prefixing S, D, C and Z respectively to the routine name.

The BLAS library in the SDK supports only real single precision (SP) and real double precision (DP) versions. All SP and DP routines in the three levels of standard BLAS are supported on the Power Processing Element (PPE). These are available as PPE APIs and conform to the standard BLAS interface. (Refer to `http://www.netlib.org/blas/blasqr.pdf`)

Some of these routines have been optimized using the Synergistic Processing Elements (SPEs) and these exhibit substantially better performance in comparison to the corresponding versions implemented solely on the PPE. An SPE interface in addition to the PPE interface is provided for some of these routines; however, the SPE interface does not conform to the standard BLAS interface and provides a restricted version of the standard BLAS interface.

LAPACK library

The LAPACK (Linear Algebra Package) library is based upon a published standard interface for commonly used linear algebra operations in high performance computing (HPC) and other scientific domains.

The LAPACK API is available with standard ANSI C and standard FORTRAN 77 interfaces. LAPACK implementations are also available as open source from http://netlib.org.

Each LAPACK routine has up to four versions, as detailed in the following table:

Table 1. LAPACK routine precision

Precision	Routine name prefix
Real single precision	S
real double precision	D
complex single precision	C
complex double precision	Z

The LAPACK library in the SDK supports only real double precision, hereafter referred to as *DP*. All DP routines are available as PPE APIs and conform to the standard LAPACK FORTRAN 77 interface.

The following routines have been optimized to use features of the Synergistic Processing Elements (SPEs):

- DGETRF - Compute the LU factorization of a general matrix
- DGETRI - Compute the inverse of a general matrix using the LU factorization
- DGEQRF - Compute the QR factorization of a general matrix
- DPOTRF - Compute the Cholesky factorization of a symmetric positive matrix
- DBDSQR - Compute the singular value decomposition of a real bi-diagonal matrix using the implicit zero-shift QR algorithm
- DSTEQR - Compute the singular value decomposition of a real symmetric tridiagonal matrix using the implicit QR algorithm

Code examples and example libraries

The example libraries package provides a set of optimized library routines that greatly reduce the development cost and enhance the performance of Cell/B.E. programs.

To demonstrate the versatility of the Cell/B.E. architecture, a variety of application-oriented libraries are included, such as:

- Fast Fourier Transform (FFT)
- Image processing
- Software managed cache
- Game math
- Matrix operation
- Multi-precision math
- Synchronization
- Vector

Additional examples and demos show how you can exploit the on-chip computational capacity.

Both the binary and the source code are shipped in separate RPMs. The RPM names are:

- cell-libs
- cell-examples
- cell-demos
- cell-tutorial

For each of these, there is one RPM that has the binaries - already built versions, that are installed into /opt/cell/sdk/usr, and for each of these, there is one RPM that has the source in a tar file. For example, cell-demos-source-3.0-1.rpm has demos_source.tar and this tar file contains all of the source code.

The default installation process installs the binaries and installs the source tar files. You need to decide into which directory you want to untar those files, either into /opt/cell/sdk/src, or into a 'sandbox' directory.

The libraries and examples RPMs have been partitioned into the following subdirectories.

Table 2. Subdirectories for the libraries and examples RPM

Subdirectory	Description
/opt/cell/sdk/buildutils	Contains a README and the make include files (make.env, make.header, make.footer) that define the SDK build environment.
/opt/cell/sdk/docs	Contains all documentation, including information about SDK libraries and tools.
/opt/cell/sdk/usr/bin /opt/cell/sdk/usr/spu/bin	Contains executable programs for that platform. On an x86 system, this includes the SPU Timing tool. On a PPC system, this also includes all of the prebuilt binaries for the SDK examples (if installed). In the SDK build environment (that is, with buildutils/make.footer) the $SDKBIN_<target> variables point to these directories.
/opt/cell/sdk/usr/include /opt/cell/sdk/usr/spu/include	Contains header files for the SDK libraries and examples on a PPC system. In the SDK build environment (that is, with the buildutils/make.footer) the $SDKINC_<target> variables point to these directories.
/opt/cell/sdk/usr/lib /opt/cell/sdk/usr/lib64 /opt/cell/sdk/usr/spu/lib	Contains library binary files for the SDK libraries on a PPC system. In the SDK build environment (that is, with the buildutils/make.footer) the $SDKLIB_<target> variables point to these directories.
/opt/cell/sdk/src	Contains the tar files for the libraries and examples (if installed). The tar files are unpacked into the subdirectories described in the following rows of this table. Each directory has a README that describes their contents and purpose.
/opt/cell/sdk/src/lib	Contains a series of libraries and reusable header files. Complete documentation for all library functions is in the /opt/cell/sdk/docs/lib/SDK_Example_Library_API_v3.1.pdf file.

Table 2. Subdirectories for the libraries and examples RPM (continued)

Subdirectory	Description
`/opt/cell/sdk/src/examples`	The examples directory contains examples of Cell/B.E. programming techniques. Each program shows a particular technique, or set of related techniques, in detail. You can review these programs when you want to perform a specific task, such as double-buffered DMA transfers to and from a program, performing local operations on an SPU, or provide access to main memory objects to SPU programs. Some subdirectories contain multiple programs. The sync subdirectory has examples of various synchronization techniques, including mutex operations and atomic operations. The spulet model is intended to encourage testing and refinement of programs that need to be ported to the SPUs; it also provides an easy way to build filters that take advantage of the huge computational capacity of the SPUs, while reading and writing standard input and output. Other samples worth noting are: • Overlay samples • SW managed cache samples
`/opt/cell/sdk/src/tutorial`	Contains tutorial code samples.
`/opt/cell/sdk/src/demos`	The demo directory provides a handful of examples that can be used to better understand the performance characteristics of the Cell/B.E. processor. There are sample programs, which contain insights into how real-world code should run. **Note:** Running these examples using the simulator takes much longer than on the native Cell/B.E.-based hardware. The performance characteristics in wall-clock time using the simulator are extremely inaccurate, especially when running on multiple SPUs. You need to examine the emulator CPU cycle counts instead. For example, the `matrix_mul` program lets you perform matrix multiplications on one or more SPUs. Matrix multiplication is a good example of a function which the SPUs can accelerate dramatically. Unlike some of the other example programs, these examples have been tuned to get the best performance. This makes them harder to read and understand, but it gives an idea for the type of performance code that you can write for the Cell/B.E. processor.
`/opt/cell/sdk/src/benchmarks`	The benchmarks directory contains sample benchmarks for various operations that are commonly performed in Cell/B.E. applications. The intent of these benchmarks is to guide you in the design, development, and performance analysis of applications for systems based on the Cell/B.E. processor. The benchmarks are provided in source form to allow you to understand in detail the actual operations that are performed in the benchmark. This also provides you with a basis for creating your own benchmark codes to characterize performance for operations that are not currently covered in the provided set of benchmarks.
`/opt/cell/sdk/prototype/src`	Contains the tar files for examples and demos for various prototype packages that ship with the SDK. Each has a README that describes their contents and purpose.

Table 2. Subdirectories for the libraries and examples RPM (continued)

Subdirectory	Description
`/opt/cell/sysroot`	Contains the header files and libraries used during cross-compiling and contains the compiled results of the libraries and examples on an x86 system. The compiled libraries and examples (everything under `/opt/cell/sysroot/opt/cell/sdk`) can be synched up with the simulator system root image by using the command: `/opt/cell/cellsdk_sync_simulator`.

Performance tools

Support libraries and utilities are provided by the SDK to help you with development and performance testing your Cell/B.E. applications.

For information about these libraries and tools, please refer to the *SDK Performance Guide* and the *SDK SPU Runtime Library Extensions Guide.*

IBM Eclipse IDE for the SDK

IBM Eclipse IDE for the SDK is built upon the Eclipse and C Development Tools (CDT) platform. It integrates the GNU tool chain, compilers, the Full-System Simulator, and other development components to provide a comprehensive, Eclipse-based development platform that simplifies development.

The key features include the following:

- A C/C++ editor that supports syntax highlighting, a customizable template, and an outline window view for procedures, variables, declarations, and functions that appear in source code
- A visual interface for the PPE and SPE combined GDB (GNU debugger)
- Seamless integration of the simulator into Eclipse
- Automatic builder, performance tools, and several other enhancements
- Remote launching, running and debugging on a IBM BladeCenter QS21
- ALF source code templates for programming models within IDE
- An ALF Code Generator to produce an ALF template package with C source code and a `readme.txt` file
- A configuration option for both the Local Simulator and Remote Simulator target environments that allows you to choose between launching a simulation machine with the Cell/B.E. processor or an enhanced CBEA-compliant processor with a fully pipelined, double precision SPE processor
- Remote Cell/B.E. and simulator BladeCenter support
- SPU timing integration
- PDT integration
- Automatic makefile generation for both GCC and XLC projects

For information about how to install and remove the IBM Eclipse IDE for the SDK, see the *SDK Installation Guide*.

For information about using the IDE, an online tutorial is available. The IDE and related programs must be installed before you can access the tutorial. The tutorial is also available as a section in the *SDK IDE Tutorial and User's Guide*.

Overview of the hybrid programming environment

The Cell Broadband Engine Architecture (CBEA) is an example of a multi-core hybrid system on a chip.

That is to say, heterogeneous cores integrated on a single processor with an inherent memory hierarchy. Specifically, the synergistic processing elements (SPEs) can be thought of as computational accelerators for a more general purpose PPE core. These concepts of hybrid systems, memory hierarchies and accelerators can be extended more generally to coupled I/O devices, and examples of those systems exist today, for example, GPUs in PCIe slots for workstations and desktops. Similarly, the Cell/B.E. processors is being used in systems as an accelerator, where computationally intensive workloads well suited for the CBEA are off-loaded from a more standard processing node. There are many ways to move data and functions from a host processor to an accelerator processor and vice versa.

To provide a consistent methodology and set of application programming interfaces (APIs) for a variety of hybrid systems, including the Cell/B.E. SoC hybrid system, the SDK has implementations of the Cell/B.E. multi-core data communication and programming model libraries, Data and Communication Synchronization (DaCS) and Accelerated Library Framework (ALF), which can be used on x86/Linux host process systems with Cell/B.E.-based accelerators. A prototype implementation over sockets is provided so that you can gain experience with this programming style and focus on how to manage the distribution of processing and data decomposition. For example, in the case of hybrid programming when moving data point to point over a network, care must be taken to maximize the computational work done on accelerator nodes potentially with asynchronous or overlapping communication, given the potential cost in communicating input and results.

For more information about the DaCS programming APIs, refer to the *Data and Communication Synchronization Library Programmer's Guide and API Reference*.

For more information about the ALF programming APIs, refer to the *Accelerated Library Framework Programmer's Guide and API Reference*.

Chapter 2. Programming with the SDK

This section is a short introduction about programming with the SDK.

It covers the following topics:

- "SDK directories"
- "Specifying the processor architecture"
- "SPU stack analysis" on page 27
- "SDK programming examples and demos" on page 15
- "Using the huge translation lookaside buffer (TLB) to reserve memory" on page 20
- "Developing applications with the SDK" on page 17
- "Performance considerations" on page 20

Refer to the *Cell/B.E. Programming Tutorial*, and other documentation for more details.

SDK directories

Because of the cross-compile environment in the SDK, there are several different system root directories.

Table 3 describes these directories.

Table 3. System root directories

Directory name	Description
`Host`	The system root for the host system is "/". The SDK is installed relative to this host system root.
`GCC Toolchain`	The system root for the GCC tool chain depends on the host platform. For PPC platforms including the IBM BladeCenter QS21, this directory is the same as the host system root. For x86 and x86-64 systems this directory is `/opt/cell/sysroot`. The tool chain PPU header and library files are stored relative to the GCC Tool chain system root in directories such as `usr/include` and `usr/lib`. The tool chain SPU header and library files are stored relative to the GCC Toolchain system root in directories such as `usr/spu/include` and `usr/spu/lib`.
`Examples and Libraries`	The Examples and Libraries system root directory is `/opt/cell/sysroot`. When the samples and libraries are compiled and linked, the resulting header files, libraries and binaries are placed relative to this directory in directories such as `usr/include`, `usr/lib`, and `/opt/cell/sdk/usr/bin`. The libspe library is also installed into this system root.

Specifying the processor architecture

Many of the tools provided in SDK support multiple implementations of the CBEA.

© Copyright IBM Corp. 2006, 2008

These include the Cell/B.E. processor and the PowerXCell 8i processor. The PowerXCell 8i processor is a CBEA-compliant processor with a fully pipelined, enhanced double precision SPU.

The processor supports five optional instructions to the SPU Instruction Set Architecture. These include:

- DFCEQ
- DFCGT
- DFCMEQ
- DFCMEQ
- DFCMGT

Detailed documentation for these instructions is provided in version 1.2 (or later) of the *Synergistic Processor Unit Instruction Set Architecture specification*. The PowerXCell 8i processor also supports improved issue and latency for all double precision instructions.

The SDK compilers support compilation for either the Cell/B.E. processor or the PowerXCell 8i processor.

Table 4. spu-gcc compiler options

Options	Description
`-march=<cpu type>`	Generate machine code for the SPU architecture specified by the CPU type. Supported CPU types are either `cell` (default) or `celledp`, corresponding to the Cell/B.E. processor or PowerXCell 8i processor, respectively.
`-mtune=<cpu type>`	Schedule instructions according to the pipeline model of the specified CPU type. Supported CPU types are either `cell` (default) or `celledp`, corresponding to the Cell/B.E. processor or PowerXCell 8i processor, respectively.

Table 5. spu-xlc compiler options

Option	Description
`-qarch=<cpu type>`	Generate machine code for the SPU architecture specified by the CPU type. Supported CPU types are either `spu` (default) or `edp`, corresponding to the Cell/B.E. processor or PowerXCell 8i processor, respectively.
`-qtune=<cpu type>`	Schedule instructions according to the pipeline model of the specified CPU type. Supported CPU types are either `spu` (default) or `edp`, corresponding to the Cell/B.E. processor or PowerXCell 8i processor, respectively.

The static timing analysis tool, `spu_timing`, also supports multiple processor implementations. The command line option `–march=celledp` can be used to specify that the timing analysis be done corresponding to the PowerXCell 8i processors' enhanced pipeline model. If the architecture is unspecified or invoked with the command line option `–march=cell`, then analysis is done corresponding to the Cell/B.E. processor's pipeline model.

SDK programming examples and demos

Each of the examples and demos has an associated `README.txt` file. There is also a top-level `readme` in the `/opt/cell/sdk/src` directory, which introduces the structure of the example code source tree.

Almost all of the examples run both within the simulator and on the IBM BladeCenter QS21 and IBM BladeCenter QS22. Some examples include SPU-only programs that can be run on the simulator in standalone mode.

The source code, which is specific to a given Cell/B.E. processor unit type, is in the corresponding subdirectory within a given example's directory:

- `ppu` for code compiled to run on the PPE
- `ppu64` for code specifically compiled for 64-bit ABI on the PPE
- `spu` for code compiled to run on an SPE
- `spu_sim` for code compiled to run on an SPE under the system simulator in standalone environment

Overview of the build environment

In `/opt/cell/sdk/buildutils` there are some top level Makefiles that control the build environment for all of the examples.

Most of the directories in the libraries and examples contain a Makefile for that directory and everything below it. All of the examples have their own Makefile but the common definitions are in the top level Makefiles.

The build environment Makefiles are documented in `/opt/cell/sdk/buildutils/README_build_env.txt`.

Changing the build environment

Environment variables in the `/opt/cell/sdk/buildutils/make.* files` are used to determine which compiler is used to build the examples.

Note: These environment variables and scripts ONLY work for Makefile examples that use the make.footer provided by the SDK. Other Makefiles may not be affected by these actions.

The `/opt/cell/sdk/buildutils/cellsdk_select_compiler` script can be used to switch the compiler. The syntax of this command is:

```
.../cell/sdk:/opt/cell/sdk/buildutils/cellsdk_select_compiler -?
```

Usage: `cellsdk_select_compiler <gcc | gfortran | gnu | xlc | xlf | xl>`
where:

- `gcc`: set GNU gcc as default c/c++ compiler in make.footer (default)
- `gfortran`: set GNU gfortran as default fortran compiler in make.footer (default)
- `gnu`: set both GNU gcc and gfortran as default compilers in make.footer (default)
- `xlc`: set IBM xlc as default c/c++ compiler in make.footer
- `xlf`: set IBM xlf as default fortran compiler in make.footer
- `xl`: set IBM xlc and xlf as default compilers in make.footer

The default is `gcc` and `gfortran`.

After you have selected a particular compiler, that same compiler is used for all future builds, unless it is specifically overwritten by shell environment variables:

- For C/C++: SPU_COMPILER, PPU_COMPILER, PPU32_COMPILER, or PPU64_COMPILER
- For Fortran: FTN_SPU_COMPILER, FTN_PPU_COMPILER, FTN_PPU32_COMPILER, or FTN_PPU64_COMPILER

Building and running a specific program

You do not need to build all the example code at once, you can build each program separately. To start from scratch, issue a `make clean` using the `Makefile` in the `/opt/cell/sdk/src` directory or anywhere in the path to a specific library or sample.

If you have performed a `make clean` at the top level, you need to rebuild the include files and libraries first before you compile anything else. To do this run a `make` in the `src/include` and `src/lib` directories.

Note: From SDK 3.0 onwards, the make.footer include file for the Cell/B.E. example and demo programs is in the subdirectory buildutils under the main SDK directory `/opt/cell/sdk`. If you used the example make.footer from previous versions of the SDK, you may need to modify your Makefile to reference this new location.

Compiling and linking with the GNU tool chain

This release of the GNU tool chain includes a GCC compiler and utilities that optimize code for the Cell/B.E. processor.

These are:

- The `spu-gcc` compiler for creating an SPU binary
- The `ppu32-embedspu` tool which enables an SPU binary to be linked with a 32-bit PPU binary into a single 32-bit executable program
- The `ppu-gcc` compiler for compiling the 64-bit PPU binary and linking it with the SPU binary.
- The `ppu-embedspu` tool which enables an SPU binary to be linked with a 64-bit PPU binary into a single 64-bit executable program
- The `ppu32-gcc` compiler for compiling the 32-bit PPU binary and linking it with the SPU binary

The example below shows the steps required to create the executable program `simple` which contains SPU code, `simple_spu.c`, and PPU code, `simple.c`.

1. Compile and link the SPE executable.
   ```
   /usr/bin/spu-gcc  -g -o simple_spu simple_spu.c
   ```
2. Optionally run `embedspu` to wrap the SPU binary into a CESOF (CBE Embedded SPE Object Format) linkable file. This contains additional PPE symbol information.
   ```
   /usr/bin/ppu32-embedspu simple_spu simple_spu simple_spu-embed.o
   ```
3. Compile the PPE side and link it together with the embedded SPU binary.
   ```
   /usr/bin/ppu32-gcc -g -o simple simple.c simple_spu-embed.o  -lspe
   ```
4. Or, compile the PPE side and link it directly with the SPU binary. The linker will invoke `embedspu`, using the file name of the SPU binary as the name of the program handle struct.
   ```
   /usr/bin/ppu32-gcc -g -o simple simple.c simple_spu -lspe
   ```

Note:

1. This section only highlights 32-bit ABI compilation. To compile for 64-bit, use `ppu-gcc` (instead of `ppu32-gcc`) and use `ppu-embedspu` (instead of `ppu32-embedspu`).
2. You are strongly advised to use the `-g` switch as shown in the examples. This embeds extra debugging information into the code for later use by the GDB debuggers supplied with the SDK. See Chapter 3, "Debugging Cell/B.E. applications," on page 23 for more information.

Customizing the compiler

This topic provides information about the GCC compiler options.

For information about how to use the compiler, refer to the Redbook Programming the Cell Broadband Engine Architecture: Examples and Best Practices, Chapter 5.2 Compiling and building executables.

Table 6. GCC compiler options

Compiler option	Description
-mdouble=accurate | fast	When using -mdouble=fast (default), GCC will automatically generate double-precision fused multiply-and-add instructions. When using -mdouble=accurate, GCC will not do so.
-mfloat=accurate | fast	When using -mfloat=fast (default), GCC automatically generates single-precision fused multiply-and-add instructions. When using -mfloat=accurate, GCC will not do so. In addition, when using -mfloat=fast, GCC generates inline code for single-precision division and square root operations that is fast, but produces results that are not always fully accurate. When using -mfloat=accurate, GCC instead generates calls to library functions that produce fully accurate results for those operations.
-mstdmain	By default, GCC links against startup code that assumes the SPU-style main function interface (which has an unconventional parameter list). With -mstdmain, GCC will link your program against startup code that assumes a C99-style interface to main, including a local copy of argv strings.

Developing applications with the SDK

This topic describes some best practices in terms of developing applications using the SDK.

See also developerWorks articles about programming tips and best practices for writing Cell/B.E. applications at

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

SDK programming policies and conventions

To ensure interoperability between applications and the SDK software, policies and conventions have been established for the cooperative sharing of various Memory Flow Control (MFC) resources.

Sharing SPE tag identifiers

These include the sharing of MFC tag identifiers, sharing MFC tag masks, and interrupt safe MFC command requests.

The SPE MFC supports 32 tag groups. If multiple pieces of SW inadvertently utilize the same tag group, then extraneous ordering dependencies or ″wait for completions″ can result. Therefore, all SDK middleware software utilizes the tag manager services, `mfc_tag_reserve` and `mfc_multi_tag_reserve`, to cooperatively obtain tag identifiers for its use. See chapter 4 of the *C/C++ Language Extensions for the Cell Broadband Engine Architecture* specification for a description of the tag manager services.

Sharing MFC tag masks

There are two usage strategies for managing MFC tag masks – ″set on use″ and ″save and restore″. The ″set on use″ strategy dictates that every time software tests for tag group completion, it must set the tag mask. This strategy is generally more efficient because most applications are double buffered so the tag mask must be set anyway. The ″save and restore″ strategy dictates that software wishing to test tag group completion must first save the current tag mask, set the tag mask and test for tag group completion, and finally restore the original tag mask. This solution is robust at a minimal cost of the two instructions to save and restore the tag mask.

SDK libraries and middleware software utilize the ″save and restore″ strategy for its tag mask policy. This allows application developers to utilize either strategy when checking for DMA completion.

Interrupt safe MFC requests

There are several MFC instruction sequences that must not be interrupted by an SPE interrupting event handler that also issues the MFC instruction sequence. This is particularly common when micro-profiling application software using PDT. The critical MFC instructions sequences include:

- MFC command request (up to 6 channel writes) interrupted by a event handler that issues also MFC command.
- Test for DMA completion (which consists of a write to the tag status update channel followed by a read of the tag status channel) interrupted by a handler which also tests for DMA completion.
- Atomic command sequence (for example, GETLLAR, PUTLLC, followed by a read of the atomic command status channel) interrupted by a handler that also issues an atomic sequence.

Applications that perform MFC sequences from within an interrupting event handler should guard its critical sections using the `spu_mfcio.h` functions, `mfc_begin_critical_section` and `mfc_end_critical_section` as documented in chapter 4 of the *C/C++ Language Extensions for the Cell Broadband Engine Architecture* specification. The functions disable and restore interrupts, respectively so that all code executed between these functions is interrupt safe.

SDK libraries and middleware software are designed to support interrupt safe operation. All critical sections are guarded so that applications are free to safely issue MFC commands from within their interrupt event handler.

Managing a DMA list element crossing 4 GB boundary

This topic describes how to prevent an DMA list element error when constructing a DMA list on the SPE.

The CBEA specifies that the EAL (the 32-bit low-order effective address) for each list element in a DMA list must be in the 4 GB aligned area defined by the EAH (the 32-bit high-order effective address). Although each EAL starting address is in a single 4 GB area, a list element transfer may cross the 4 GB boundary.

However, in the Cell/B.E. and PowerXCell 8i processors, a DMA list element that crosses a 4 GB boundary results in a Class0 DMA Alignment Error exception. The Linux operating system makes no effort to detect or recover from this error. Therefore, having a list element crossing a 4 GB boundary in a DMA list results in a bus error at execution time.

Note: This error only occurs for 64-bit applications, 32-bit applications never encounter a 4 GB boundary crossing.

Programmers need to be aware of this limitation when constructing and executing DMA list on the SPE. If the DMA list does not cross a 4 GB boundary, no action is required. There are several strategies one can use to ensure that DMA list elements do not cross the 4 GB boundary. There are two distinct cases:

- Case #1: The DMA list crosses one or more 4 GB boundaries, but not within a list element. This requires you to break up the list into n+1 lists, where n is the number of boundary crossings. 2.
- Case #2: The DMA list crosses one or more 4 GB boundaries within a list element. To prevent this, you can use one of the following strategies:
  - Split the offending 4 GB crossing list elements into two elements and handle the new enlarged list as prescribed in case #1.
  - Remove the 4 GB list elements and issue them using a non-list DMA. The remaining list elements can be handled as prescribed in case #1.

The overhead for detecting list elements that cross the 4 GB boundary is nontrivial, especially if you need to check every list element in a DMA list (case #2). For applications that have control over how and where memory is allocated, you can allocate memory such that a 4 GB crossing never occurs by using the `mmap()` function to map allocations that are less than 4 GB to start at the beginning of a 4 GB region. For allocations that are larger than 4 GB, you can select the allocation address such that the crossing does not occur within a list element. For example, for internally allocated matrices, you can pad the allocation such that the crossing only occurs on a row or tile boundary so that case #2 never happens.

Here is an example of using the `mmap()` function to allocate memory. In this example, we are allocating 512 MB of system memory. We are using `mmap()` function to ensure that the 512 MB of memory does not straddle the 4 GB boundary.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/mman.h>
```

```
int main()
{
  void* ptr;
  size_t length;

  /* We want to allocate 512MB of memory */
  length = (512 * 1024 * 1024);

  /* Allocate memory for avoiding 4GB boundary crossing */
  ptr = mmap((void *)(0x110000000ULL), length, PROT_READ |
  PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
  if (ptr == MAP_FAILED) {
    perror ("Failed allocating memory needed\n");
  }

  printf ("beginning of allocation = %p, end of allocation = %p\n",
      ptr, (void*)((unsigned long long)ptr + length));

}
```

The output of this example shows that the allocated memory does not straddle the 4GB boundary.

```
$ gcc -m64 -o 4GB_example 4GB_example.c
$ ./4GB_example
beginning of allocation = 0x110000000, end of allocation = 0x130000000
```

Performance considerations

This topic describes performance considerations that you should take into account when you are developing applications:

It covers the following topics:

- "Using NUMA" on page 21
- "Preemptive context switching" on page 22

Using the huge translation lookaside buffer (TLB) to reserve memory

The SDK supports the huge translation lookaside buffer (TLB) file system, which allows you to reserve 16 MB huge pages of pinned, contiguous memory. This feature is particularly useful for some Cell/B.E. applications that operate on large data sets, such as the FFT16M workload sample.

To configure the IBM BladeCenter QS21 for 20 huge pages (320 MB), run the following commands:

```
mkdir -p /huge
echo 20 > /proc/sys/vm/nr_hugepages
mount -t hugetlbfs nodev /huge
```

If you have difficulties configuring adequate huge pages, it could be that the memory is fragmented and you need to reboot.

You can add the command sequence shown above to a startup initialization script, such as `/etc/rc.d/rc.sysinit`, so that the huge TLB file system is configured during the system boot.

To verify the large memory allocation, run the command `cat /proc/meminfo`. The output is similar to:

```
MemTotal:      1010168 kB
MemFree:        155276 kB
. . .

HugePages_Total:    20
HugePages_Free:     20
Hugepagesize:    16384 kB
```

Huge pages are allocated by invoking **mmap** of a /huge file of the specified size. For example, the following code sample allocates 32 MB of private huge paged memory :

```
int fmem;
  char *mem_file = "/huge/myfile.bin";

  fmem = open(mem_file, O_CREAT | O_RDWR, 0755)) == -1) {
  remove(mem_file);

  ptr = mmap(0, 0x2000000, PROT_READ | PROT_WRITE, MAP_PRIVATE, fmem, 0);
```

mmap succeeds even if there are insufficient huge pages to satisfy the request. On first access to a page that can not be backed by huge TLB file system, the application is ″killed″. That is, the process is terminated and the message ″killed″ is emitted. You must be ensure that the number of huge pages requested does not exceed the number available. Furthermore, on an IBM BladeCenter QS20, IBM BladeCenter QS21, and IBM BladeCenter QS22 the huge pages are equally distributed across both Non-Uniform Memory Architecture (NUMA) memory nodes. Applications that restrict memory allocation to a specific node find that the number of available huge pages for the specific node is half of what is reported in `/proc/meminfo`.

Using NUMA

This topic describes how to select an optimal NUMA policy.

The IBM BladeCenter QS20, IBM BladeCenter QS21 and IBM BladeCenter QS22 are both Non-Uniform Memory Architecture (NUMA) systems, which consist of two Cell/B.E. processors, each with its own system memory. The two processors are interconnected thru a FlexIO interface using the fully coherent BIF protocol. The bandwidth between processor elements or processor elements and memory is greater if accesses are local and do not have to communicate across the FlexIO. In addition, the access latency is slightly higher on node 1 (Cell BE 1) as compared to node 0 (Cell BE 0) regardless of whether they are local or non-local.

To maximize the performance of a single application, you can specify CPU and memory binding to either reduce FlexIO traffic or exploit the aggregated bandwidth of the memory available on both nodes. You can specify the Linux scheduling policy or memory placement either through application-specified NUMA policy library calls (man `numa(3)`) or using the `numactl` command (man `numactl(8)`).

For example, the following command invokes a program that allocates all CPUs on node 0 with a preferred memory allocation on node 0:

```
numactl --cpunodebind=0 --preferred=0 <program name>
```

Choosing an optimal NUMA policy depends upon the application's communication and data access patterns. However, you should consider the following general guidelines:

- Choose a NUMA policy compatible with typical system usage patterns. For example, if the multiple applications are expected to run simultaneously, do not bind all CPUs to a single node forcing an overcommit scenario that leaves one of the nodes idle. In this case, it is recommended that you do not constrain the Linux scheduler with any specific bindings.
- Consider the operating system services when you choose the NUMA policy. For example, if the application incorporates extensive GbE networking communications, the TCP stack will consume some PPU resources on node 0 for eth0. In this case, it may be advisable to bind the application to node 1.
- Avoid over committing CPU resources. Context switching of SPE threads is not instantaneous and the scheduler quanta for SPE's threads is relatively large. Scheduling overhead is minimized if you can avoid over-committing resources.
- Applications that are memory bandwidth-limited should consider allocating memory on both nodes and exploit the aggregated memory bandwidth. If possible, partition application data such that CPUs on node 0 primarily access memory on node 0 only. Likewise, CPUs on node 1 primarily access memory on node 1 only.

Preemptive context switching

The Linux operating system provides preemptive context switching of virtualized SPE contexts that each resemble the functionality provided by a physical SPE, but there are limitations to the degree to which the architected hardware interfaces can be used in a virtualized environment.

In particular, memory mapped I/O on the problem state register area and MFC proxy DMA access can only be used while the SPE context is both running in a thread and not preempted, otherwise the thread trying to perform these operations blocks until the conditions are met.

This can result in poor performance and deadlocks for programs that overcommit SPEs and rely on SPE thread communications and synchronization. In this case, you should avoid running more SPE threads then there are physical SPEs.

Chapter 3. Debugging Cell/B.E. applications

This section describes how to debug Cell/B.E. applications.

It describes the following:

Overview of GDB

GDB is the standard command-line debugger available as part of the GNU development environment.

GDB has been modified to allow debugging in a Cell/B.E. processor environment and this section describes how to debug Cell/B.E. software using the new and extended features of the GDBs which are supplied with SDK.

Debugging in a Cell/B.E. processor environment is different from debugging in a multithreaded environment, because threads can run either on the PPE or on the SPE.

There are three versions of GDB which can be installed on a IBM BladeCenter QS21:

- `gdb` which is installed with the Linux operating system for debugging PowerPC applications. You should NOT use this debugger for Cell/B.E. applications.
- `ppu-gdb` for debugging PPE code or for debugging combined PPE and SPE code. This is the combined debugger.
- `spu-gdb` for debugging SPE code only. This is the standalone debugger.

This section also describes how to run applications under `gdbserver`. The `gdbserver` program allows remote debugging.

GDB for SDK

The GDB program released with SDK replaces previous versions and contains the following enhancements:

- It is based on GDB 6.8
- It is able to handle both SPE and PPE architecture code within a single thread, see "Switching architectures within a single thread" on page 33
- When referring to a symbol defined both in PPE code and in one or more SPE contexts, GDB always resolves to the definition in the current context, see "Disambiguation of multiply-defined global symbols" on page 39

Compiling and linking applications

The linker embeds all the symbolic and additional information required for the SPE binary within the PPE binary so it is available for the debugger to access when the program runs. You should use the `-g` option when compiling both SPE and PPE code with GCC or XLC. W The `-g` option adds debugging information to the binary which then enables GDB to lookup symbols and show the symbolic

© Copyright IBM Corp. 2006, 2008

information. When you use the toplevel Makefiles of the SDK, you can specify the `-g` option on compilation commands by setting the `CC_OPT_LEVEL` makefile variable to `-g`.

When you use the top level Makefiles of the SDK, you can specify the `-g` option on compilation by setting the `CC_OPT_LEVEL` Makefile variable to `-g`.

For more information about compiling with GCC, see "Compiling and linking with the GNU tool chain" on page 16.

Debugging applications

This topic describes how to debug applications but assumes that you are familiar with the standard features of GDB.

The following topics are described:

- "Debugging PPE code"
- "Debugging SPE code"

Debugging PPE code

There are several ways to debug programs designed for the Cell/B.E. processor. If you have access to Cell/B.E. hardware, you can debug directly using `ppu-gdb` . You can also run the application under `ppu-gdb` inside the simulator. Alternatively, you can debug remotely as described in "Debugging applications remotely" on page 43.

Whichever method you choose, after you have started the application under `ppu-gdb`, you can use the standard GDB commands available to debug the application. The GDB manual is available at the GNU Web site

`http://www.gnu.org/software/gdb/gdb.html`

and there are many other resources available on the World Wide Web.

Debugging SPE code

Standalone SPE programs or spulets are self-contained applications that run entirely on the SPE. Use `spu-gdb` to launch and debug standalone SPE programs in the same way as you use `ppu-gdb` on PPE programs.

Note: You can use either `spu-gdb` or `ppu-gdb` to debug SPE only programs. In this section `spu-gdb` is used.

The examples in this section use a standalone SPE (spulet) program, `simple.c`, whose source code and Makefile are given below:

Source code:

```
#include <stdio.h>
#include <spu_intrinsics.h>

unsigned int
fibn(unsigned int n)
{
    if (n <= 2)
        return 1;
    return (fibn (n-1) + fibn (n-2));
}
```

```
int
main(int argc, char **argv)
{
    unsigned int c;
    c = fibn (8);
    printf ("c=%d\n", c);
    return 0;
}
```

Note: Recursive SPE programs are generally not recommended due to the limited size of local storage. An exception is made here because such a program can be used to illustrate the `backtrace` command of GDB.

Makefile:

```
simple: simple.c
        spu-gcc simple.c -g -o simple
```

Debugging source level code

Source-level debugging of SPE programs with `spu-gdb` is similar in nearly all aspects to source-level debugging for the PPE.

For example, you can:

- Set breakpoints on source lines
- Display variables by name
- Display a stack trace and single-step program execution

The following example illustrates the backtrace output for the `simple.c` standalone SPE program.

```
$ spu-gdb ./simple
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later $ spu-gdb ./simple
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=powerpc64-unknown-linux-gnu --target=spu"...
(gdb) break 8
Breakpoint 1 at 0x194: file simple.c, line 8.
(gdb) break 18
Breakpoint 2 at 0x230: file simple.c, line 18.
(gdb) run
Starting program: /home/brian_horton/tmp/s/simple

Breakpoint 1, fibn (n=2) at simple.c:8
8 return 1;
(gdb) backtrace
#0 fibn (n=2) at simple.c:8
#1 0x000001b0 in fibn (n=3) at simple.c:9
#2 0x000001b0 in fibn (n=4) at simple.c:9
#3 0x000001b0 in fibn (n=5) at simple.c:9
#4 0x000001b0 in fibn (n=6) at simple.c:9
#5 0x000001b0 in fibn (n=7) at simple.c:9
#6 0x000001b0 in fibn (n=8) at simple.c:9
#7 0x0000021c in main (argc=1, argv=0x3ffd0) at simple.c:17
(gdb) delete breakpoint 1
(gdb) continue
Continuing.

Breakpoint 2, main (argc=1, argv=0x3ffd0) at simple.c:18
```

```
18 printf ("c=%d\n", c);
(gdb) print c
$1 = 21
(gdb)
```

Debugging assembler level code

The spu-gdb program also supports many of the familiar techniques for debugging SPE programs at the assembler code level.

For example, you can:

- Display register values
- Examine the contents of memory (which for the SPE means local storage)
- Disassemble sections of the program
- Step through a program at the machine instruction level

The following example illustrates some of these facilities.

```
$ spu-gdb ./simple
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=powerpc64-unknown-linux-gnu --target=spu"...
(gdb) br 18
Breakpoint 1 at 0x230: file simple.c, line 18.
(gdb) r
Starting program: /home/brian_horton/tmp/s/simple

Breakpoint 1, main (argc=1, argv=0x3ffd0) at simple.c:18
18 printf ("c=%d\n", c);
(gdb) print c
$1 = 21
(gdb) x /8i $pc
0x230 <main+72>: ila $3,0x8e0 <_fini+32>
0x234 <main+76>: lqd $2,32($1) # 20
0x238 <main+80>: ori $4,$2,0
0x23c <main+84>: brsl $0,0x2d0 <printf> # 2d0
0x240 <main+88>: il $2,0
0x244 <main+92>: ori $3,$2,0
0x248 <main+96>: ai $1,$1,80 # 50
0x24c <main+100>: lqd $0,16($1)
(gdb) nexti
0x00000234 18 printf ("c=%d\n", c);
(gdb) nexti
0x00000238 18 printf ("c=%d\n", c);
(gdb) print $r4
$2 = {uint128 = 0x00000015000000150000001500000015, v2_int64 = {90194313237, 90194313237}, v4_int32 = {
21, 21, 21, 21}, v8_int16 = {0, 21, 0, 21, 0, 21, 0, 21}, v16_int8 = {0, 0, 0, 21, 0, 0, 0, 21, 0,
0, 0, 21, 0, 0, 0, 21}, v2_double = {4.4561911620646097e-313, 4.4561911620646097e-313},
v4_float = {2.94272678e-44, 2.94272678e-44, 2.94272678e-44, 2.94272678e-44}}
(gdb)
```

How spu-gdb manages SPE registers

Because each SPE register can hold multiple fixed or floating point values of several different sizes, spu-gdb treats each register as a data structure that can be accessed with multiple formats.

The spu-gdb ptype command, illustrated in the following example, shows the mapping used for SPE registers:

```
(gdb) ptype $r80
type = union __spu_builtin_type_vec128 {
   int128_t uint128;
   int64_t v2_int64[2];
```

```
   int32_t v4_int32[4];
   int16_t v8_int16[8];
   int8_t v16_int8[16];
   double v2_double[2];
   float v4_float[4];
}
```

To display or update a specific vector element in an SPE register, specify the appropriate field in the data structure, as shown in the following example:

```
(gdb)  p $r80.uint128
$1 = 0x00018ff000018ff000018ff000018ff0
(gdb)  set $r80.v4_int32[2]=0xbaadf00d
(gdb)  p $r80.uint128
$2 = 0x00018ff000018ff0baadf00d00018ff0
```

SPU stack analysis

SPU local store space is limited. Allocating too much stack space limits space available for code and data. Allocating too little stack space can cause runtime failures. To help you allocate stack space efficiently, the SPU linker provides an estimate of maximum stack usage when it is called with the option `--stack-analysis`.

The value returned by this command is not guaranteed to be accurate because the linker analysis does not include dynamic stack allocation such as that done by the `alloca` function. The linker also does not handle calls made by function pointers or recursion and other cycles in the call graph. However, even with these limitations, the estimate can still be useful. The linker provides detailed information on stack usage and calls in a linker map file, which can be enabled by passing the parameter `-Map <filename>` to the linker. This extra information combined with known program behavior can help you to improve on the linker's simple analysis.

For the following simple program, hello.c:

```
#include <stdio.h>
#include <unistd.h>

int foo (void)
{
   printf (" world\n");
   printf ("brk: %x\n", sbrk(0));
   (void) fgetc (stdin);
   return 0;
}

int main (void)
{
   printf ("Hello");
   return foo ();
}
```

The command `spu-gcc -o hello -O2 -Wl,--stack-analysis,-Map,hello.map hello.c` generates the following output:

```
$ spu-gcc -o hello -O2 -Wl,--stack-analysis,-Map,hello.map hello.c
Stack size for call graph root nodes.
_start: 0x120
_fini: 0x40
call___do_global_dtors_aux: 0x20
call_frame_dummy: 0x20
__sfp: 0x0
```

```
__check_init: 0x0
__cleanup: 0xd0
call___do_global_ctors_aux: 0x20
Maximum stack required is 0x120
```

This output shows that the main entry point `_start` will require `0x120` bytes of stack space below `__stack`. There are also a number of other root nodes that the linker fails to connect into the call graph. These are either functions called through function pointers, or unused functions. `_fini`, registered with `atexit()` and called from `exit`, is an example of the former. All other nodes here are unused.

The `hello.map` section for stack analysis shows:

```
my hello.map:

Stack size for functions. Annotations: '*' max stack, 't' tail call
_exit: 0x0 0x0
__call_exitprocs: 0xd0 0xd0
exit: 0x30 0x100
calls:
_exit
* __call_exitprocs
__sinit: 0x0 0x0
__send_to_ppe: 0x50 0x50
fgetc: 0x40 0x90
calls:
__sinit
* __send_to_ppe
__stack_reg_va: 0x0 0x0
printf: 0x0 0x50
calls:
* __send_to_ppe
__stack_reg_va
sbrk: 0x0 0x0
puts: 0x30 0x80
calls:
* __send_to_ppe
foo: 0x20 0xb0
calls:
* fgetc
printf
sbrk
puts
main: 0x20 0xb0
calls:
*t foo
printf
__register_exitproc: 0x0 0x0
atexit: 0x0 0x0
calls:
t __register_exitproc
_init: 0x0 0x0
__do_global_ctors_aux: 0x30 0x30
_init: 0x0 0x0
frame_dummy: 0x20 0x20
_init: 0x0 0x0
_init: 0x20 0x50
calls:
* __do_global_ctors_aux
frame_dummy
_start: 0x20 0x120
calls:
* exit
main
atexit
_init
```

```
_fini: 0x0 0x0
__do_global_dtors_aux: 0x20 0x20
_fini: 0x0 0x0
_fini: 0x20 0x40
calls:
* __do_global_dtors_aux
call___do_global_dtors_aux: 0x20 0x20
call_frame_dummy: 0x20 0x20
__sfp: 0x0 0x0
__check_init: 0x0 0x0
calls:
t __sinit
fclose: 0x40 0x90
calls:
__sinit
* __send_to_ppe
__cleanup: 0x40 0xd0
calls:
* fclose
call___do_global_ctors_aux: 0x20 0x20
```

This analysis shows that in the entry for the `main` function, `main` requires 0x20 bytes of stack. The total program requires a total of 0x120 bytes including all called functions. The function called from `main` that requires the maximum amount of stack space is `foo`, which `main` calls through the`tail` function call. Tail calls occur after the local stack for the caller is deallocated. Therefore, the maximum stack space allocated for `main` is the same as the maximum stack space allocated for `foo`. The `main` function also calls the `printf` function.

If you are uncertain whether the `_fini` function might require more stack space than `main`, trace down from the `_start` function to the `__call_exitprocs` function (where `_fini` is called) to find the stack requirement for that code path. The stack size is 0x20 (local stack for `_start`) plus 0x30 (local stack for `exit`) plus 0xD0 (local stack for `__call_exitprocs`) plus 0x40 (total stack for `_fini`), or 0x160 bytes. Therefore, the stack is sufficient for `_fini`.

If you pass the `--emit-stack-syms` option to the linker, it will save the stack sizing information in the executable for use by post-link tools such as FDPRPro. With this option specified, the linker creates symbols of the form `__stack_<function_name>` for global functions, and `__stack_<number>_<function_name>` for static functions. The value of these symbols is the total stack size requirement for the corresponding function.

You can link against these symbols. The following is an example.

```
extern void __stack__start;

   printf ("Total stack is %ld\n", (long) &__stack__start);
```

SPE stack debugging

The SPE stack shares local storage with the application's code and data. Because local storage is a limited resource and lacks hardware-enabled protection it is possible to overflow the stack and corrupt the program's code or data or both. This often results in hard to debug problems because the effects of the overflow are not likely to be observed immediately.

Overview of SPE stack debugging:

To understand how to debug stack overflows, it is important to understand how the SPE local storage is allocated and the stack is managed.

Note: For more information about SPE local storage allocation,see Figure 1.

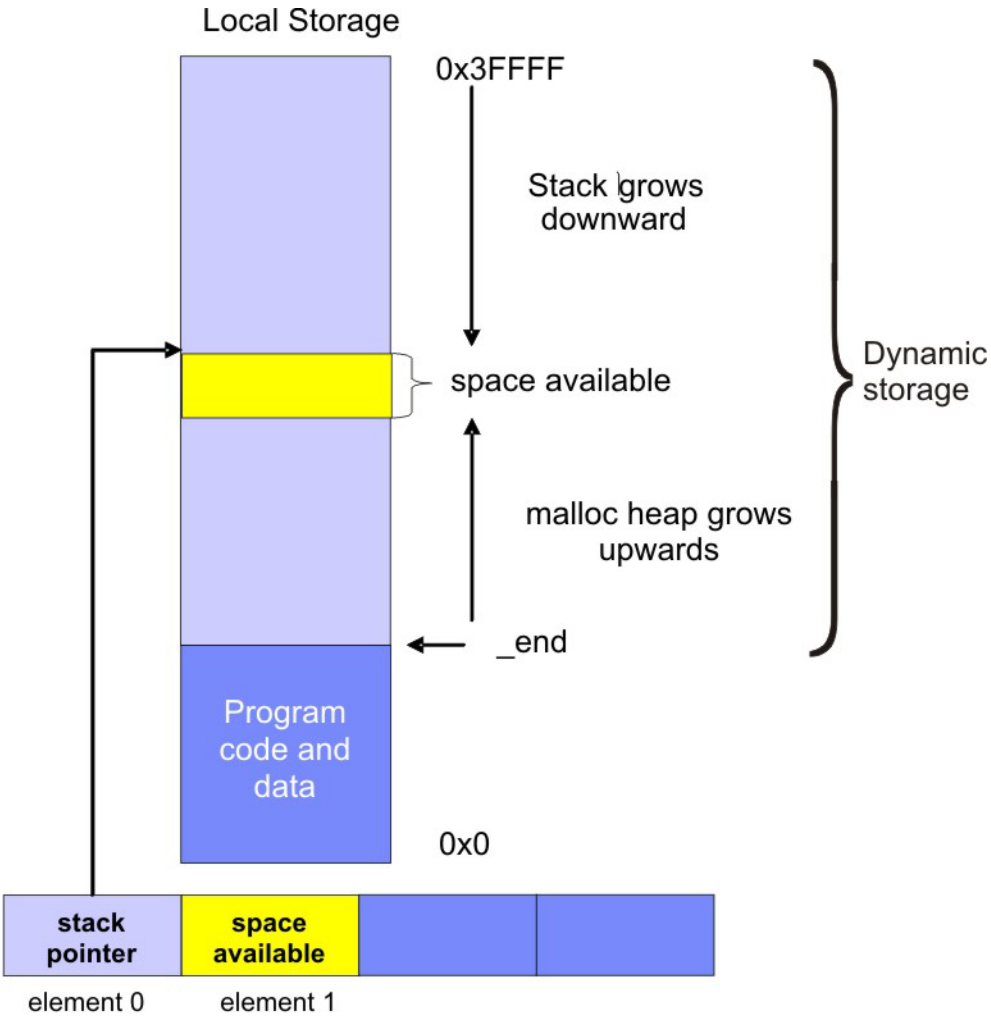


Figure 1. SPE local storage and stack anatomy

The area between the linker symbol that marks the end of the programs code and data sections, `_end`, and the top of local storage is dynamic storage. This dynamic storage is used for two purposes, the stack and the malloc heap. The stack grows downward (from high addressed memory to low addressed memory), and the malloc heap grows upward.

The C runtime startup code (crt0) initializes the stack pointer register (register 1) such that word element 0 contains the current stack pointer and word element 1 contains the number of dynamic storage bytes currently available. The stack pointer and space available is negatively adjusted when a stack frame is acquired and positively adjusted when a stack frame is released. The space available is negatively adjusted, up to the available space, whenever the malloc heap is enlarged.

Stack overflow checking:

During application development it is advisable that you use stack overflow checking and then disable it when the application is released.

Because the spu-gcc and spuxlc compilers do not by default emit code to detect stack overflow, you must include a compile line option:

- The spu-gcc compile line option is `-fstack-check`
- The spuxlc compile line option is `-qcheck=stack`

Stack checking introduces additional code to test for stack overflow. The additional code halts execution conditional on the available space going negative as a result of acquiring or enlarging a stack frame.

For a standalone SPU program, the occurrence of a halt results in a "`spe_run: Bad address`" message and exit code of `SPE_SPU_HALT (4)`.

For SPE contexts being run from a PPE program, a stack overflow results in a `stopinfo`, `stop_reason` of `SPE_RUNTIME_ERROR` with a `spe_runtime_error` equal to `SPE_SPU_HALT`. See the `spe_context_run` subroutine specification of the *SPE Runtime Management Library* for additional details.

Runtime checking space available

To check the amount of space available for both stack and memory anytime during execution, you only need to inspect the element 1 of the stack pointer register. You can use the following inline assembly to do this:

```
int space_available;
asm volatile ("rotqbyi %0, $1, 4"  : "=r" (space_available));
```

Note: It should be noted that the SPE application binary interface (ABI) allows a functions to access up to 2000 bytes beyond its stack frame without allocating a new stack frame. Therefore, it is possible that up to 2000 bytes of the available space can actually be used at the time the register is read. Applications that attempt to use the current available space for determining how much addition space the memory heap can be grown should account for this possibility as well as any additional stack use encountered by further increased call depth.

Stack management strategies:

To reduce the occurrence of stack overflows, you should adopt some stack management strategies.

You should consider the following strategies:

- Avoid or reduce memory heap allocations. Because most application's working data set exceeds the size of local storage, data must be sequenced into the local store in blocks. Preallocate block storage as global variables instead of using automatic or dynamic-allocated memory arrays.
- Avoid recursion. Either eliminate the recursion, or in the case of tail recursion, transform the recursion into a state array and optionally use a software managed cache to virtualize the state array.
- Free up local storage space to accommodate a larger stack by using overlays.

Debugging in the Cell/B.E. environment

To debug combined code, that is code containing both PPE and SPE code, you must use `ppu-gdb`.

Debugging multithreaded code

Typically a simple program contains only one thread. For example, a PPU ″hello world″ program is run in a process with a single thread and the GDB attaches to that single thread.

On many operating systems, a single program can have more than one thread. The `ppu-gdb` program allows you to debug programs with one or more threads. The debugger shows all threads while your program runs, but whenever the debugger runs a debugging command, the user interface shows the single thread involved. This thread is called the current thread. Debugging commands always show program information from the point of view of the current thread. For more information about GDB support for debugging multithreaded programs, see the sections 'Debugging programs with multiple threads' and 'Stopping and starting multi-thread programs' of the GDB User's Manual, available at

`http://www.gnu.org/software/gdb/gdb.html`

The `info threads` command displays the set of threads that are active for the program, and the `thread` command can be used to select the current thread for debugging.

Note: The source code for the program `simple.c` used in the examples below comes with the SDK and can be found at `/opt/cell/sdk/src/tutorial/simple` after extracting the `tutorial_source.tar` tar file in the `src` directory.

Debugging architecture

This topic provides an overview of debugging architecture.

On the Cell/B.E. processor, a thread can run on either the PPE or on an SPE at any given point in time. All threads, both the main thread of execution and secondary threads started using the pthread library, will start execution on the PPE. Execution can switch from the PPE to an SPE when a thread executes the **spe_context_run** function. See the libspe2 manual for details. Conversely, a thread currently executing on an SPE may switch to use the PPE when executing a library routine that is implemented via the PPE-assisted call mechanism See the Cell BE Linux Reference Implementation ABI document for details. When you choose a thread to debug, the debugger automatically detects the architecture the thread is currently running on. If the thread is currently running on the PPE, the debugger will use the PowerPC architecture. If the thread is currently running on an SPE, the debugger will use the SPE architecture. A thread that is currently executing code on an SPE may also be referred to as an *SPE thread*.

To see which architecture the debugger is using, use the `show architecture` command.

Example: show architecture

The example below shows the results of the `show architecture` command at two different breakpoints in a program. At breakpoint 1 the program is executing in the original PPE thread, where the `show architecture` command indicates that architecture is `powerpc:common`. The program then spawns an SPE thread which will execute the SPU code in `simple_spu.c`. When the debugger detects that the SPE thread has reached breakpoint 3, it switches to this thread and sets the architecture to `spu:256K` For more information about breakpoint 2, see "Setting pending breakpoints" on page 36.

```
[user@localhost simple]$ ppu-gdb ./simple
...
...
...
(gdb) break main
Breakpoint 1 at 0x1801654: file simple.c, line 23.
(gdb) run
Starting program: /home/user/md/simple/simple
[Thread debugging using libthread_db enabled]
[New Thread 4160655360 (LWP 2490)]
[Switching to Thread 4160655360 (LWP 2490)]

Breakpoint 1, main (argc=1, argv=0xfff7a9e4) at simple.c:23
23          int i,  status = 0;
(gdb) show architecture
The target architecture is set automatically (currently powerpc:common)
(gdb) break simple_spu.c:5
No source file named simple_spu.c.
Make breakpoint pending on future shared library load? (y or [n]) y

Breakpoint 2 (simple_spu.c:5) pending.
(gdb) continue
Continuing.
Breakpoint 3 at 0x158: file simple_spu.c, line 5.
Pending breakpoint "simple_spu.c:5" resolved
[New Thread 4160390384 (LWP 2495)]
[Switching to Thread 4160390384 (LWP 2495)]

Breakpoint 3, main (id=103079215104) at simple_spu.c:13
13      {
(gdb) show architecture
The target architecture is set automatically (currently spu:256K)
(gdb)
```

Switching architectures within a single thread

This topic describes the debugger *backtrace* command.

As described in "Debugging architecture" on page 32, any thread of a combined Cell/B.E. application is executing either on the PPE or an SPE at the time the debugger interrupted execution of the process currently being debugged. This determines the main architecture GDB will use when examining the thread. However, during the execution history of that thread, execution may have switched between architectures one or multiple times. When looking at the thread's stack *backtrace* (using the backtrace command), the debugger will reflect those switches. It will show stack frames belonging to both the PPE and SPE architectures.

Example: An SPE context is interrupted by the debugger while executing a PPE-assisted scanf call

```
(gdb) backtrace
#0 0x0ff1a8e8 in __read_nocancel () from /lib/libc.so.6
#1 0x0feb7e04 in _IO_new_file_underflow (fp=<value optimized out>) at fileops.c:590
#2 0x0feb82c0 in _IO_default_uflow (fp=<value optimized out>) at genops.c:435
#3 0x0feba518 in *__GI___uflow (fp=<value optimized out>) at genops.c:389
#4 0x0fe9b834 in _IO_vfscanf_internal (s=<value optimized out>, format=<value optimized out>,
argptr=<value optimized out>, errp=<value optimized out>) at vfscanf.c:542
#5 0x0fe9f858 in ___vfscanf (s=<value optimized out>, format=<value optimized out>,
argptr=<value optimized out>)
at vfscanf.c:2473
#6 0x0fe18688 in __do_vfscanf (stream=<value optimized out>, format=<value optimized out>,
vlist=<value optimized out>)
at default_c99_handler.c:284
#7 0x0fe1ab38 in default_c99_handler_vscanf (ls=<value optimized out>, opdata=<value optimized out>)
at default_c99_handler.c:1193
```

```
#8 0x0fe176b0 in default_c99_handler (base=<value optimized out>, offset=<value optimized out>)
at default_c99_handler.c:1990
#9 0x0fe1f1b8 in handle_library_callback (spe=<value optimized out>, callnum=<value optimized out>,
npc=<value optimized out>) at lib_builtin.c:152
#10 <cross-architecture call>
#11 0x0003fac4 in ?? ()
#12 0x00000360 in scanf (fmt=<value optimized out>) at ../../../../../../src/newlib/libc/machine/spu/scanf.c:74
#13 0x00000170 in main () at test.c:8
```

When you choose a particular stack frame to examine using the `frame`, `up`, or `down` commands, the debugger switches its notion of the current architecture as appropriate for the selected frame. For example, if you use the `info registers` command to look at the selected frame's register contents, the debugger shows the SPE register set if the selected frame belongs to an SPE context, and the PPE register set if the selected frame belongs to PPE code.

Example: continued

```
gdb) frame 7
#7 0x0fe1ab38 in default_c99_handler_vscanf (ls=<value optimized out>,
opdata=<value optimized out>)
at default_c99_handler.c:1193
1193 default_c99_handler.c: No such file or directory.
in default_c99_handler.c
(gdb) show architecture
The target architecture is set automatically (currently powerpc:common)
(gdb) info registers
r0 0x3 3
r1 0xfec2eda0 4274187680
r2 0xfff9ba0 268409760
r3 0x200 512
<...>

(gdb) frame 13
#13 0x00000170 in main () at test.c:8
8 scanf ("%d\n", &x);
(gdb) show architecture
The target architecture is set automatically (currently spu:256K)
(gdb) info registers
r0 {uint128 = 0x00000170000000000000000000000000, v2_int64 = {0x17000000000, 0x0},
v4_int32 = {0x170, 0x0, 0x0, 0x0}, v8_int16 = {0x0, 0x170, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
v16_int8 = {0x0, 0x0, 0x1, 0x70, 0x0 <repeats 12 times>}, v2_double = {0x0, 0x0},
v4_float = {0x0, 0x0, 0x0, 0x0}}
r1 {uint128 = 0x0003ffa00000000000000000000000, v2_int64 = {0x3ffa000000000, 0x0},
v4_int32 = {0x3ffa0, 0x0, 0x0, 0x0}, v8_int16 = {0x3, 0xffa0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
v16_int8 = {0x0, 0x3, 0xff, 0xa0, 0x0 <repeats 12 times>}, v2_double = {0x0, 0x0},
v4_float = {0x0, 0x0, 0x0, 0x0}}
<...>
```

Viewing symbolic and additional information

Compiling with the '-g' option adds debugging information to the binary that enables GDB to lookup symbols and show the symbolic information.

The debugger sees SPE executable programs as shared libraries. The `info sharedlibrary` command shows all the shared libraries including the SPE executables when running SPE threads.

Example: info sharedlibrary

The example below shows the results of the `info sharedlibrary` command at two breakpoints on one thread. At breakpoint 1, the thread is running on the PPE, at breakpoint 3 the thread is running on the SPE. For more information about breakpoint 2, see "Setting pending breakpoints" on page 36.

```
(gdb) break main
Breakpoint 1 at 0x1801654: file simple.c, line 23.
(gdb) r
Starting program: /home/user/md/simple/simple
[Thread debugging using libthread_db enabled]
[New Thread 4160655360 (LWP 2528)]
[Switching to Thread 4160655360 (LWP 2528)]

Breakpoint 1, main (argc=1, argv=0xffacb9e4) at simple.c:23
23          int i,  status = 0;
(gdb) info sharedlibrary
From        To          Syms Read   Shared Object Library
0x0ffc1980  0x0ffd9af0  Yes         /lib/ld.so.1
0x0fe14b40  0x0fe20a00  Yes         /usr/lib/libspe.so.1
0x0fe5d340  0x0ff78e30  Yes         /lib/libc.so.6
0x0fce47b0  0x0fcf1a40  Yes         /lib/libpthread.so.0
0x0f291cc0  0x0f2970e0  Yes         /lib/librt.so.1
(gdb) break simple_spu.c:5
No source file named simple_spu.c.
Make breakpoint pending on future shared library load? (y or [n]) y

Breakpoint 2 (simple_spu.c:5) pending.
(gdb) c
Continuing.
Breakpoint 3 at 0x158: file simple_spu.c, line 5.
Pending breakpoint "simple_spu.c:5" resolved
[New Thread 4160390384 (LWP 2531)]
[Switching to Thread 4160390384 (LWP 2531)]

Breakpoint 3, main (id=103079215104) at simple_spu.c:13
13      {
(gdb) info sharedlibrary
From        To          Syms Read   Shared Object Library
0x0ffc1980  0x0ffd9af0  Yes         /lib/ld.so.1
0x0fe14b40  0x0fe20a00  Yes         /usr/lib/libspe.so.1
0x0fe5d340  0x0ff78e30  Yes         /lib/libc.so.6
0x0fce47b0  0x0fcf1a40  Yes         /lib/libpthread.so.0
0x0f291cc0  0x0f2970e0  Yes         /lib/librt.so.1
0x00000028  0x00000860  Yes         simple_spu@0x1801d00 <6>
(gdb)
```

GDB creates a unique name for each shared library entry representing SPE code. That name consists of the SPE executable name, followed by the location in PPE memory where the SPE is mapped (or embedded into the PPE executable image), and the SPE ID of the SPE thread where the code is loaded.

Using scheduler-locking

Scheduler-locking is a feature of GDB that simplifies multithread debugging by enabling you to control the behavior of multiple threads when you single-step through a thread. By default scheduler-locking is off, and this is the recommended setting.

In the default mode where scheduler-locking is off, single-stepping through one particular thread does not stop other threads of the application from running, but allows them to continue to execute. This applies to both threads executing on the PPE and on the SPE. This may not always be what you expect or want when debugging multithreaded applications, because those threads executing in the background may affect global application state asynchronously in ways that can make it difficult to reliably reproduce the problem you are debugging. If this is a concern, you can turn scheduler-locking on. In that mode, all other threads remain stopped while you are debugging one particular thread. A third option is to set scheduler-locking to step, which stops other threads while you are single-stepping the current thread, but lets them execute while the current thread is freely running.

However, if scheduler-locking is turned on, there is the potential for deadlocking where one or more threads cannot continue to run. Consider, for example, an application consisting of multiple SPE threads that communicate with each other through a mailbox. If you single-step one thread across an instruction that reads from the mailbox, and that mailbox happens to be empty at the moment, this instruction (and thus the debugging session) will block until another thread writes a message to the mailbox. However, if scheduler-locking is on, that other thread will remain stopped by the debugger because you are single-stepping. In this situation none of the threads can continue, and the whole program stalls indefinitely. This situation cannot occur when scheduler-locking is off, because in that case all other threads continue to run while the first thread is single-stepped. You should ensure that you enable scheduler-locking only for applications where such deadlocks cannot occur.

There are situations where you can safely set scheduler-locking on, but you should do so only when you are sure there are no deadlocks.

The syntax of the command is:

```
set scheduler-locking <mode>
```

where `mode` has one of the following values:

- off
- on
- step

You can check the scheduler-locking mode with the following command:

```
show scheduler-locking
```

Using the combined debugger

Generally speaking, you can use the same procedures to debug code for Cell/B.E. as you would for PPC code.

However, some existing features of GDB and one new command can help you to debug in the Cell/B.E. processor multithreaded environment. These features are described in the following section.

Setting pending breakpoints

Breakpoints stop programs running when a certain location is reached. You set breakpoints with the `break` command, followed by the line number, function name, or exact address in the program.

You can use breakpoints for both PPE and SPE portions of the code. There are some instances, however, where GDB must defer insertion of a breakpoint because the code containing the breakpoint location has not yet been loaded into memory. This occurs when you wish to set the breakpoint for code that is dynamically loaded later in the program. If `ppu-gdb` cannot find the location of the breakpoint it sets the breakpoint to pending. When the code is loaded, the breakpoint is inserted and the pending breakpoint deleted.

You can use the `set breakpoint` command to control the behavior of GDB when it determines that the code for a breakpoint location is not loaded into memory. The syntax for this command is:

```
set breakpoint pending <on off auto>
```

where:

- on on specifies that GDB should set a pending breakpoint if the code for the breakpoint location is not loaded.
- off off specifies that GDB should not create pending breakpoints, and break commands for a breakpoint location that is not loaded result in an error.
- auto auto specifies that GDB should prompt the user to determine if a pending breakpoint should be set if the code for the breakpoint location is not loaded. This is the default behavior.

Example: Pending breakpoints

The example below shows the use of pending breakpoints. Breakpoint 1 is a standard breakpoint set for simple.c, line 23. When the breakpoint is reached, the program stops running for debugging. After set breakpoint pending is set to off, GDB cannot set breakpoint 2 (break simple_spu.c:5) and generates the message No source file named simple_spu.c. After set breakpoint pending is changed to auto, GDB sets a pending breakpoint for the location simple_spu.c:5. At the point where GDB can resolve the location, it sets the next breakpoint, breakpoint 3.

```
(gdb) break main
Breakpoint 1 at 0x1801654: file simple.c, line 23.
(gdb) r
Starting program: /home/user/md/simple/simple
[Thread debugging using libthread_db enabled]
[New Thread 4160655360 (LWP 2651)]
[Switching to Thread 4160655360 (LWP 2651)]

Breakpoint 1, main (argc=1, argv=0xff95f9e4) at simple.c:23
23          int i,  status = 0;
(gdb)  off
(gdb) break simple_spu.c:5
No source file named simple_spu.c.
(gdb) set breakpoint pending auto
(gdb) break simple_spu.c:5
No source file named simple_spu.c.
Make breakpoint pending on future shared library load? (y or [n]) y

Breakpoint 2 (simple_spu.c:5) pending.
(gdb) c
Continuing.
Breakpoint 3 at 0x158: file simple_spu.c, line 5.
Pending breakpoint "simple_spu.c:5" resolved
[New Thread 4160390384 (LWP 2656)]
[Switching to Thread 4160390384 (LWP 2656)]

Breakpoint 3, main (id=103079215104) at simple_spu.c:13
13      {
(gdb)
```

Note: The example above shows one of the ways to use pending breakpoints. For more information about other options, see the documentation available at

http://www.gnu.org/software/gdb/gdb.html

Multi-location breakpoints

GDB is capable of assigning one or more locations to each breakpoint. Many applications, especially Cell/B.E.-specific applications, consist of many congeneric threads (instances of the same source file).

As an example a user sets a breakpoint in a thread. If a new thread is created after that thread, the available breakpoint is already assigned to a new location in the

new thread. In previous version of GDB the user had to create a new breakpoint in the new thread, which appeared as a new and different breakpoint.

The following example illustrates the new behavior introduced with multi-location breakpoint support in GDB:

```
(gdb) info threads
[New Thread 0xf6b1f4b0 (LWP 12780)]
  4 Thread 0xf6b1f4b0 (LWP 12780)  0x0fc1ed78 in __lll_lock_wait () from /lib/libpthread.so.0
* 3 Thread 0xf755f4b0 (LWP 12778)  main (speid=268568208, argp=0, envp=0) at break_spu_bin.c:12
  2 Thread 0xf7faf4b0 (LWP 12777)  main (speid=268566536, argp=0, envp=0) at break_spu_bin.c:30
  1 Thread 0xfff2a40  (LWP 12774)  0x0fc14bd0 in __nptl_create_event () from /lib/libpthread.so.0
(gdb) br break_spu_bin.c:main
Breakpoint 3 at 0x5e0: file break_spu_bin.c, line 12. (2 locations)
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
3       breakpoint     keep y     <MULTIPLE>
3.1                         y     0x000005e0 in main at break_spu_bin.c:12
3.2                         y     0x000005e0 in main at break_spu_bin.c:12
(gdb)
```

GDB recognizes that `break_spu_bin.c:main` is found in two locations. On that account a multi-location breakpoint with two locations is created (3.1 and 3.2). The entry numbered 3 shows that this breakpoint is a multi-location breakpoint. Whenever a new thread from `break_spu_bin.c` is created, a new location is added to that breakpoint:

```
(gdb) continue
Continuing.

...

Hello World! from SPU
Temporary breakpoint 4 at 0x5e0: file break_spu_bin.c, line 12.
Breakpoint 3, main (speid=268568848, argp=0, envp=0) at break_spu_bin.c:12
12        i = 7;

(gdb) info breakpoints
Num     Type           Disp Enb Address    What
3       breakpoint     keep y     <MULTIPLE>
        breakpoint already hit 1 time
3.1                         y     0x000005e0 in main at break_spu_bin.c:12
3.2                         y     0x000005e0 in main at break_spu_bin.c:12
3.3                         y     0x000005e0 in main at break_spu_bin.c:12
```

After continuing the process, a new thread is created from `break_spu_bin.c` and with it a new location is added to breakpoint 3.

Using the set spu stop-on-load command

The `set spu stop-on-load` stops each thread before it starts running on the SPE.

While `set spu stop-on-load` is in effect, the debugger automatically sets a temporary breakpoint on the "main" function of each new SPE thread immediately after it is loaded. You can use the `set spu stop-on-load` command to do this instead of simply issuing a `break main` command, because the latter is always interpreted to set a breakpoint on the "main" function of the PPE executable.

Note: The `set spu stop-on-load` command has no effect in the SPU standalone debugger spu-gdb. To let an SPU standalone program proceed to its "main" function, you can use the `start` command in spu-gdb.

The syntax of the command is:

```
set spu stop-on-load <mode>
```

where mode is on or off.

To check the status of spu stop-on-load, use the show spu stop-on-load command.

Example: set spu stop-on-load on

```
(gdb) break main
Breakpoint 1 at 0x1801654: file simple.c, line 23.
(gdb) r
Starting program: /home/user/md/simple/simple
[Thread debugging using libthread_db enabled]
[New Thread 4160655360 (LWP 3009)]
[Switching to Thread 4160655360 (LWP 3009)]

Breakpoint 1, main (argc=1, argv=0xffc7c9e4) at simple.c:23
23          int i,  status = 0;
(gdb) show spu stop-on-load
Stopping for new SPE threads is off.
(gdb) set spu stop-on-load on
(gdb) c
Continuing.
Breakpoint 2 at 0x174: file simple_spu.c, line 16.
[New Thread 4160390384 (LWP 3013)]
Breakpoint 3 at 0x174: file simple_spu.c, line 16.
[Switching to Thread 4160390384 (LWP 3013)]
main (id=25272376) at simple_spu.c:16
16        for (i = 0, n = 0; i<5; i++) {
(gdb) info threads
* 2 Thread 4160390384 (LWP 3013)  main (id=25272376) at simple_spu.c:16
  1 Thread 4160655360 (LWP 3009)  0x0ff27428 in mmap () from /lib/libc.so.6
(gdb) c
Continuing.
Hello Cell (0x181a038) n=3
Hello Cell (0x181a038) n=6
Hello Cell (0x181a038) n=9
Hello Cell (0x181a038) n=12
Hello Cell (0x181a038) n=15
[Thread 4160390384 (LWP 3013) exited]
[New Thread 4151739632 (LWP 3015)]
[Switching to Thread 4151739632 (LWP 3015)]
main (id=25272840) at simple_spu.c:16
16        for (i = 0, n = 0; i<5; i++) {
(gdb) info threads
* 3 Thread 4151739632 (LWP 3015)  main (id=25272840) at simple_spu.c:16
  1 Thread 4160655360 (LWP 3009)  0x0fe14f38 in load_spe_elf (
    handle=0x181a3d8, ld_buffer=0xf6f29000, ld_info=0xffc7c230)
    at elf_loader.c:224
(gdb)
```

Disambiguation of multiply-defined global symbols

When debugging a combined Cell/B.E. application consisting of a PPE program and more or more SPE programs, it can happen that multiple definitions of a global function or variable with the same name exist.

For example, both the PPE and SPE programs define a global *main* function. If you run the same SPE executable simultaneously within multiple SPE contexts, all its global symbols show multiple instances of definition.

To catch the most common usage cases, GDB uses the following rules when it looks up a global symbol. If the command is issued while currently debugging PPE code, the debugger first attempts to look up a definition in the PPE executable. If none is found, the debugger searches all currently loaded SPE executables and uses the first definition of a symbol with the given name it finds. However, when referring to a global symbol from the command line while

currently debugging an SPE context, the debugger first attempts to look up a definition in that SPE context. If none is found there, the debugger continues to search the PPE executable and all other currently loaded SPE executables and uses the first matching definition.

Example:

```
(gdb) br foo2
Breakpoint 2 at 0x804853f:
file /home/deuling/gdb/dev/gdb/testsuite/gdb.base/solib-symbol-main.c, line 40.

(gdb) delete breakpoints
Delete all breakpoints? (y or n) y

(gdb) br foo
Breakpoint 3 at 0xb7ffd53f:
file /home/deuling/gdb/dev/gdb/testsuite/gdb.base/solib-symbol-lib.c, line 23.

(gdb) continue
Continuing.
Breakpoint 3, foo () at /home/deuling/gdb/dev/gdb/testsuite/gdb.base/solib-symbol-lib.c:23
23 printf ("foo in lib\n");

(gdb) br foo2
Breakpoint 4 at 0xb7ffd569:
file /home/deuling/gdb/dev/gdb/testsuite/gdb.base/solib-symbol-lib.c, line 30.

(gdb) PASS: gdb.base/solib-symbol.exp: foo2 in mdlib
```

In this example, `foo2` is in the main file one time and in the library the other time depending on where GDB currently stands.

For the current version of the SDK, the combined debugger ppu-gdb introduces a new facility for symbol determination. A debugged executable can contain symbols with identical names in different parts of the application. For example, if the symbol `foo` exists in the PPU-part and in the SPU-part of a combined binary, the command `set multiple-symbols` defaults to `all`, which means one breakpoint is set on each symbol found. If set to `ask` and a multiply-defined symbol such as the one previously described is found, you are given a choice as to which symbol to select, and multiple choices are possible. You can set `cancel` to cancel the symbol search.

Example: An example GDB session could look like this:

```
(gdb) set multiple-symbols ask

(gdb) br foo2
[0] cancel
[1] all
[2] foo2 at break.c:76
[3] foo2 at break_spu_bin.c:46
> 2
Breakpoint 3 at 0x100019fc: file break.c, line 76.
(gdb)
```

Note: The user chose to use the symbol `foo2` found in `break.c` for breakpoint creation.

```
(gdb) br foo2
[0] cancel
[1] all
[2] foo2 at break.c:76
[3] foo2 at break_spu_bin.c:46
> 3
Breakpoint 4 at 0x6e4: file break_spu_bin.c, line 46. (2 locations)
(gdb)
```

Note: The user chose symbol 3 which has two locations. Hence a multi-location breakpoint was created.

```
(gdb) br foo2
[0] cancel
[1] all
[2] foo2 at break.c:76
[3] foo2 at break_spu_bin.c:46
> 0
canceled
(gdb)
```

Note: The user cancelled the selection. No breakpoint is created.

```
(gdb) br foo2
[0] cancel
[1] all
[2] foo2 at break.c:76
[3] foo2 at break_spu_bin.c:46
> 1
Note: breakpoint 3 also set at pc 0x100019fc.
Breakpoint 5 at 0x100019fc: file break.c, line 76.
Note: breakpoint 4 also set at pc 0x6e4.
Note: breakpoint 4 also set at pc 0x6e4.
Breakpoint 6 at 0x6e4: file break_spu_bin.c, line 46. (2 locations)
warning: Multiple breakpoints were set.
Use the "delete" command to delete unwanted breakpoints.
(gdb)
```

Note: The user chose to create a breakpoint on all symbols found. Hence there were two breakpoints created. One multi-location breakpoint at `break_spu_bin.c` and a single location breakpoint at `break.c`.

```
gdb) br foo2
[0] cancel
[1] all
[2] foo2 at break.c:76
[3] foo2 at break_spu_bin.c:46
> 1 2
Note: breakpoint 3 also set at pc 0x100019fc.
Breakpoint 5 at 0x100019fc: file break.c, line 76.
Note: breakpoint 4 also set at pc 0x6e4.
Note: breakpoint 4 also set at pc 0x6e4.
Breakpoint 6 at 0x6e4: file break_spu_bin.c, line 46. (2 locations)
warning: Multiple breakpoints were set.
Use the "delete" command to delete unwanted breakpoints.
(gdb)
```

Note: The user chose to create a breakpoint on all symbols found. Hence there were two breakpoints created. One multi-location breakpoint at `break_spu_bin.c` and a single location breakpoint at `break.c`.

Multiple selections are possible. In this case selecting `all` is the same as selecting `1` and `2`.

New command reference

In addition to the `set spu stop-on-load` command, the `ppu-gdb` and `spu-gdb` programs offer an extended set of the standard GDB `info` commands.

These are:

- `info spu event`
- `info spu signal`
- `info spu mailbox`

- `info spu dma`
- `info spu proxydma`

If you are working in GDB, you can access help for these new commands. To access help, type `help info spu` followed by the `info spu` subcommand name. This displays full documentation. Command name abbreviations are allowed if unambiguous.

Note: For more information about the various output elements, refer to the *Cell Broadband Engine Architecture* document available at

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

info spu event

Displays SPE event facility status. The output is similar to:

```
(gdb) info spu event
Event Status 0x00000000
Event Mask   0x00000000
```

info spu signal

Displays SPE signal notification facility status. The output is similar to:

```
(gdb) info spu signal
Signal 1 not pending (Type Or)
Signal 2 control word 0x30000001 (Type Or)
```

info spu mailbox

Displays SPE mailbox facility status. Only pending entries are shown. Entries are displayed in the order of processing, that is, the first data element in the list is the element that is returned on the next read from the mailbox. The output is similar to:

```
(gdb) info spu mailbox
SPU Outbound Mailbox
0x00000000
SPU Outbound Interrupt Mailbox
0x00000000
SPU Inbound Mailbox
0x00000000
0x00000000
0x00000000
0x00000000
```

info spu dma

Displays MFC DMA status. For each pending DMA command, the opcode, tag, and class IDs are shown, followed by the current effective address, local store address, and transfer size (updated as the command is processed). For commands using a DMA list, the local store address and size of the list is shown. The ″E″ column indicates commands flagged as erroneous by the MFC. If multiple pending DMA operations are required to execute in a particular sequence due to the use of barrier, fence, or synchronization commands, they are listed in the order they will be executed. The output is similar to:

```
(gdb) info spu dma
Tag-Group Status  0x00000002
Tag-Group Mask    0x00000001 (no query pending)
Stall-and-Notify  0x00000000
Atomic Cmd Status 0x00000000

Opcode  Tag TId RId EA                 LSA     Size    LstAddr LstSize E
put     0   0   0   0x00000000f77d0080 0x01000 0x04000                 *
```

```
getb    0   0   0   0x00000000f7be8080 0x01000 0x04000
put     0   0   0   0x00000000f77d4080 0x05000 0x04000                 *
getb    1   0   0   0x00000000f7bec080 0x05000 0x04000
```

info spu proxydma

Displays MFC Proxy-DMA status. The output is similar to:

```
(gdb) info spu proxydma
Tag-Group Status  0x00000000
Tag-Group Mask    0x00000000 (no query pending)

Opcode  Tag TId RId EA                  LSA     Size    LstAddr LstSize E
getfs   0   0   0   0xc000000000379100 0x00e00 0x00000
get     0   0   0   0xd000000000243000 0x04000 0x00000
0       0   0   0                      0x00000 0x00000
0       0   0   0                      0x00000 0x00000
0       0   0   0                      0x00000 0x00000
0       0   0   0                      0x00000 0x00000
0       0   0   0                      0x00000 0x00000
0       0   0   0                      0x00000 0x00000
```

Debugging applications remotely

This topic describes how to use the two versions of gdbserver provided with the SDK.

These are:

- `spu-gdbserver` to run a stand-alone spulet. You must use `spu-gdb` on the client
- `ppu-gdbserver` to run a 32-bit or 64–bit PPE or combined executable. You must use `ppu-gdb` on the client.

Note: In the following section, gdbserver is used as the generic term for both versions. Similarly GDB is used to refer to the two different debuggers.

This section describes how to set up remote debugging for the Cell/B.E. processor and the simulator. It covers the following topics:

Overview of remote debugging

You can run an application under gdbserver to allow remote hardware and simulator-based debugging. The application gdbserver is a companion program to GDB that implements the GDB remote serial protocol. This is used to convert GDB into a client/server-style application, where gdbserver launches and controls the application on the target platform, and GDB connects to gdbserver to specify debugging commands.

The connection between GDB and gdbserver can either be through a traditional serial line or through TCP/IP. For example, you can run gdbserver on a IBM BladeCenter QS21 and GDB on an Intel® x86 platform, which then connects to the gdbserver using TCP/IP.

Remote debugging has significantly improved in this version of the SDK. Previous versions of the SDK came with two versions of gdbserver, one for 32-bit and one

for 64-bit PowerPC executables. The version of gdbserver shipped with this version of the SDK transparently supports both 32-bit and 64-bit executables in a single version, just like GDB does.

One cause of confusion when debugging remotely with former SDK versions depended on GDB, which needs exact copies of the target application binary as well as all shared libraries used by that application on the host system. If those copies are not available or if they do not match exactly, remote debugging fails. This situation occurs in certain situations, such as for example, when installing a service upgrade affecting system libraries.

The combined debugger now offers an improved user experience in remote debugging. GDB now automatically retrieves the actual versions of the libraries needed by the executable to be debugged from the host, via an extension to the remote debugging connection.

Using remote debugging

To use remote debugging, you need a version of the program for the target platform and network connectivity. The gdbserver program comes packaged with GDB and is installed with the SDK.

Note: IDEs such as Eclipse do not directly communicate with gdbserver. However, an IDE can communicate with GDB running on the same host which can then in turn communicate with gdbserver running on a remote machine.

When using gdbserver to debug applications on a remote target it is mandatory to provide the same set of libraries (like for example pthread library, C library, libspe, and so on) on both the host (where GDB runs) and the target (where gdbserver runs) system.

Note: To connect thru the network to the simulator, you must enable *bogusnet* support in the simulator. This creates a special Ethernet device that uses a ″call-thru″ interface to send and receive packets to the host system. See the simulator documentation for details about how to enable *bogusnet.*

Further information on the remote debugging of Cell Broadband Engine applications is available in the DeveloperWorks article at
`http://www-128.ibm.com/developerworks/power/library/pa-celldebug/`

Starting remote debugging

This topic describes how to start remote debugging.

To start a remote debugging session, do the following:

1. Use gdbserver to launch the application on the target platform (either the IBM BladeCenter QS21 or inside the Simulator). To do this enter:

   `<gdbserver version> [ip address] :<port> <application> [arg1 arg2 ...]`

   where

   - `<gdbserver version>` refers to the version of gdbserver appropriate for the program you wish to debug
   - `[ip address]` is optional. Default address is localhost.
   - `:<port>` specifies the TCP/IP port to be used for communication with `gdbserver`
   - `<application>` is the name of the program to be debugged

- [arg1 arg2 ...] are the command line arguments for the program

An example for ppu-gdbserver using port 2101 for the program myprog which requires no command line arguments would be:

```
ppu-gdbserver :2101 myprog
```

Note: If you use ppu-gdbserver as shown here then you must use ppu-gdb on the client.

2. Start GDB from the client system (if you are using the simulator this is the host system of the simulator).

   For the simulator this is:

   ```
   /opt/cell/toolchain/bin/ppu-gdb myprog
   ```

   For the IBM BladeCenter QS21 this is:

   ```
   /usr/bin/ppu-gdb myprog
   ```

   You should have the source and compiled executable version for myprog on the host system. If your program links to dynamic libraries, GDB attempts to locate these when it attaches to the program. If you are cross-debugging, you need to direct GDB to the correct versions of the libraries otherwise it tries to load the libraries from the host platform. The default path is /opt/cell/sysroot. For the SDK, issue the following GDB command to connect to the server hosting the correct version of the libraries:

   ```
   set solib-absolute-prefix
   ```

   Note: If you have not installed the libraries in the default directory you must indicate the path to them. Generally the lib/ and lib64/ directories are under /opt/cell/sysroot/.

3. At the GDB prompt, connect to the server with the command:

   ```
   target remote 172.20.0.2:2101
   ```

   where 172.20.0.2 is the IP address of the Cell system that is running gdbserver, and the :2101 parameter is the TCP/IP port parameter that was used start gdbserver. If you are running the client on the same machine then the IP address can be omitted. If you are using the simulator, the IP address is generally fixed at 172.20.0.2 To verify this, enter the ifconfig command in the console window of the simulator.

   If gdbserver is running on the simulator, you can use a symbolic host name for the simulator, for example:

   ```
   target remote systemsim:2101
   ```

   To do this, edit the host system's /etc/hosts as follows:

   ```
   # Do not remove the following line, or various programs
   # that require network functionality will fail.
   127.0.0.1       localhost.localdomain   localhost
   172.20.0.2      systemsim
   ```

The following shows an example of myprog

```
8   {
9                char *needle, *haystack;
10                int count = 0;
11
12                if (argc < 3) {
13                    return 0;
14                }
15
16                needle = argv[1];
17                haystack = argv[2];
18
```

```
B+>19                while (*haystack != '\0')   20                    {
   21                        int i = 0;
   22                        while (needle[i] != '\0' && haystack[i] != '\0' && needle[i])
   23                            i++
   24                        }
   25                        if (needle[i] == '\0') {
   26                            count++;
   27                        }
   28                        haystack++;
   29                    }
   30
   31                return count;
   32    }
remote Thread 42000 In: main                 Line: 19   PC 0x18004c8
Type "how copying" to see conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=powerpc64-linux"....
(gdb) target remote 172.20.0.2:2101
Remote debugging using 172.20.0.2:2101
0xf7ff80f0 in ?? ()
(gdb) b 19
Breakpoint 1 at 0x18004c8: file myprog.c, line 19.
(gdb) c
Continuing.

Breakpoint 1, main (argc=3, argv=0xffab6b74) at myprog.c:19
(gdb)
```

Chapter 4. Debugging common Direct Memory Access (DMA) errors

This topic describes possible DMA errors and uses examples to describe how to debug these errors.

The Cell/B.E. is a multiprocessor system on a chip that is not a traditional shared-memory multiprocessor. The system consists of a PowerPC Processing Element (PPE) which accesses main storage, and eight Synergistic Processing Elements (SPEs) which access their own private local storage. To access main storage, the SPEs utilize direct memory access (DMA) commands which transfer data between main storage and their private local memory. This distributed storage organization enables high performance, but it requires the SPE programmer to explicitly handle DMA transfers between main storage and local storage. Errors during these DMA transfers can be difficult to detect and debug. This article provides techniques for handling common problems with SPE-initiated DMA transfers.

DMA errors

All DMA transactions on the Cell/B.E. must follow certain guidelines.

Due to the dynamic nature of DMA processing, a command parameter that does not adhere to the guidelines may not cause an error during compilation. Instead, during runtime, the MFC command queue processing will be suspended and an interrupt will be raised to the PPE. The application is usually terminated with either a Bus Error (SIGBUS) or a Segmentation Fault (SEGSEGV). Partial DMA transfer may be performed before the Memory Flow Controller (MFC) encounters an invalid parameter in a DMA command and raises an interrupt to the PPE.

Bus errors

The following table shows common DMA command errors which are detected by hardware and cause bus errors.

Table 7. Common DMA command errors that cause bus error

Error type	Description
DMA size errors	
Bad transfer size	Transfer size is not 0, 1, 2, 4, or 8 bytes or a multiple of 16 bytes
Transfer size is too big	Transfer size that is greater than 16 KB
List transfer size is too big	List element with size that is greater than 16 KB
DMA Alignment Errors	
Local storage address alignment	Target and source addresses are not naturally aligned for sizes less than 16 bytes or are not aligned on 16-byte boundary for sizes >= 16 bytes
Main storage address alignment	
List address alignment	DMA list is not stored in SPE local store on an 8-byte boundary
Tag ID Errors	

© Copyright IBM Corp. 2006, 2008

Table 7. Common DMA command errors that cause bus error (continued)

Error type	Description
Invalid Tag ID	Tag id is not between 0 and 31 inclusive
Other Errors	
List element crosses a 4 GB boundary	For 64-bit applications, list elements cannot cross a 4 GB boundary. 32-bit application does not have to worry about this.

Segmentation Violations

The following table shows common DMA command errors which are not detected by the hardware and cause a Segmentation violation. Segmentation faults that occur on the SPE are always caused by DMA transfers to and/or from bad effective addresses.

Table 8. Common DMA command errors that cause a segmentation violation

Error type	Description
Invalid target effective address	A DMA PUT command with a target effective address which can not be accessed, for example, an address of storage that was not allocated
Invalid source effective address	A DMA GET command with a source effective address which can not be accessed, for example, address of storage that was not allocated

Using ppu-gdb to debug DMA errors

This section provides information about how to debug Cell/B.E. applications using the ppu-gdb and spu-gdb debuggers.

GDB command "info spu dma"

This command is one of the extended commands that GDB offers to help debugging Cell/B.E. applications. It displays MFC DMA status. For each pending DMA command, the opcode, tag, and class IDs are shown, followed by the current effective address, local store address, and transfer size (updated as the command is processed). For commands using a DMA list, the local store address and size of the list is shown. The "E" column indicates commands flagged as erroneous by the MFC. The output is similar to:

```
(gdb) info spu dma
Tag-Group Status  0x00000002
Tag-Group Mask    0x00000001 ('any' query pending)
Stall-and-Notify  0x00000000
Atomic Cmd Status 0x00000000

Opcode  Tag TId RId EA                 LSA     Size    LstAddr LstSize E
put     0   0   0   0x00000000f77d0080 0x01000 0x04000                 *
getb    0   0   0   0x00000000f7be8080 0x01000 0x04000
put     0   0   0   0x00000000f77d4080 0x05000 0x04000                 *
getb    1   0   0   0x00000000f7bec080 0x05000 0x04000
```

The following table provides detailed description for each of the fields displayed using the **info spu dma** command:

Table 9. Description of fields

Field	Description
Tag-Group Status	Displays the current tag-group status.
Tag-Group Mask	Displays the tag-group query mask (MFC_WrTagMask channel). The statement in parenthesis indicates whether a tag status query is currently pending or not. This should read either ″no query pending″, ″'any' query pending″, or ″'all' query pending″.
Stall-and-Notify	Displays the content of the MFC Read List Stall-and-Notify Tag Status (MFC_RdListStallStat) channel. List elements for a list command contian a stall-and-notify flag. If the flag is set on a list element, the MFC stops executing the list command after completing the transfer requested by this element. The flag sets the bit corresponding to the tag group of the list command in this channel.
Atomic Cmd Status	Displays the content of the MFC Read Atomic Command Status channel (MFC_RdAtomicStat). This channel contains the status of the last-completed immediate atomic update DMA command.
Opcode	Displays the opcode for the DMA transfer. If the opcode is invalid, the actual input value is displayed and the ″E″ bit is marked.
Tag	Displays the tag group id for the DMA transfer. This field is a 5-bit field, which means that it does not show values outside the 0-31 range. If the tag id is invalid, the ″E″ bit is marked.
TId	Displays the Transfer Class ID. Generally set to 0.
Rid	Displays the Replacement Class ID. Generally set to 0.
EA	Displays the effective address of the DMA transfer. If an effective address is misaligned, this field will show the actual misaligned address and the ″E″ bit will be marked.
LSA	Displays the local store address of the DMA transfer. If a local store address is misaligned, this field will not show the actual misaligned address since the hardware does not retain the four least significant bits of the local store address. However, the ″E″ bit will be marked.
Size	Displays the size of the DMA transfer. If the command is one of the DMA list commands, this field displays the size of the current DMA list entry. If a given size is invalid (see Table 7 on page 47), this field will not necessarily show the invalid size. Instead it shows the number of bytes yet to be transferred when the interrupt is raised.
LstAddr	Displays the local store address of the DMA list if the command is one of the DMA list commands. If partial transfers have been done, this field displays the local store address of the current DMA list entry.
LstSize	Displays the size (in bytes) of the DMA list if the command is one of the DMA list commands. If partial transfers have been done, this field displays the size (in bytes) of the list entries that have not been processed. Note that the size of each list entry is 8 bytes.
E	Displays a ″*″ if there is an error in the DMA transfer that has been detected by the hardware.

Examples

This section shows how to debug some common DMA errors on Cell/B.E with example programs that intentionally contain errors.

Some of the examples are a little bit contrived since the errors are fairly easy to spot. However, the general approach used to debug DMA problems on Cell/B.E. should still apply to most ″real-world″ scenarios.

The examples are modified versions of an example that does single buffering using a DMA list command. It gathers data from main storage to SPE local storage, processes the data, and scatters the data back to main storage using DMA lists. The complete listing for the modified examples can be found in the directory:

`/opt/cell/sdk/src/examples/dma_errors/no_error`

Unaligned effective address

The first version of the example, *unaligned_ea_error*, contains a DMA transfer with an unaligned effective address.

A complete listing of the code for this example can be found in the directory:

`/opt/cell/sdk/src/examples/dma_errors/unaligned_ea_error`

When the example is compiled and run, the output should be:

Listing 1

```
$ ./dma_error
Bus error
```

A bus error can be caused by many different error conditions as shown in the table above, but which one? The first step is to recompile the program with the `-g` and `-O0` flags so the program can be debugged more easily using the combined gdb debugger. Next run the newly recompiled program using ppu-gdb, and you should get the following output:

Listing 2

```
$ ppu-gdb dma_error
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "powerpc64-linux"...
(gdb) r
Starting program: /opt/cell/sdk/src/examples/dma_errors/unaligned_ea_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x400012af230 (LWP 16671)]

Program received signal SIGBUS, Bus error.
[Switching to Thread 0x400012af230 (LWP 16671)]
0x000004ac in main (speid=268566544, argp=268566656, envp=0) at dma_error_spu.c:147
147       mfc_read_tag_status_all ();
```

The output shows that the program stopped in the SPU thread. To verify the error occurred in the SPU run the command `show architecture` to show the current architecture:

Listing 3

```
(gdb) show architecture
The target architecture is set automatically (currently spu:256K)
```

In *Listing 2* the architecture is `spu:256K` showing that it is definitely the SPU. Now looking more closely at the SPU program in *Listing 1* the gdb output shows the program error occurred within a DMA transfer when it stopped on the `mfc_read_tag_status_all` line.

Use the `gdb info spu dma` command to examine all in-flight DMA transfers.

Listing 4

```
(gdb) info spu dma
Tag-Group Status  0x00000000
Tag-Group Mask    0x00000001 (undefined query type)
Stall-and-Notify  0x00000000
Atomic Cmd Status 0x00000000

Opcode  Tag TId RId EA                 LSA     Size    LstAddr LstSize E
get     0   0   0   0x0000000010020088 0x21380 0x00020                 *
```

In *Listing 4* programmers can scan the shown parameters for effective addresses that are not aligned properly. The local store addresses cannot be examined for alignment errors using the same method since the hardware does not retain the four least significant bits of the local store addresses. The alignment rules and guidelines for DMA commands on the Cell/B.E. are as follow:

- Source and destination addresses must have the same 4 least significant bits
- For transfer sizes less than 16 bytes, address must be naturally aligned (bits 28 through 31 must provide natural alignment based on the transfer size)
- For transfer sizes of 16 bytes or greater, address must be aligned to at least a 16-byte boundary. In hexadecimal, the address must end with a '0'.
- Peak performance is achieved when both source and destination addresses are aligned on a 128-byte boundary (bits 25 through 31 are '0'). In hexadecimal, the address must end with a '80' or '00'.

Listing 4 shows a DMA with effective address of 0x10020088. This address is not aligned to a 16-byte boundary and thus caused the observed bus error. At this point, it might be useful to step through the program to detect the actual DMA that causes the Bus error and examine the effective address in that DMA. From *Listing 1*, we know that the SPE program stopped at line 147 in the SPE dma_error_spu.c file. We can try to set a break point at line 140 and start stepping through the program.

Listing 5

```
$ ppu-gdb dma_error
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "powerpc64-linux"...
(gdb) break dma_error_spu.c:140
No source file named dma_error_spu.c.
Make breakpoint pending on future shared library load? (y or [n]) y

Breakpoint 1 (dma_error_spu.c:140) pending.
(gdb) r
Starting program: /opt/cell/sdk/src/examples/dma_errors/unaligned_ea_error/dma_error
[Thread debugging using libthread_db enabled]
```

```
[New Thread 0x400012af230 (LWP 18864)]
[Switching to Thread 0x400012af230 (LWP 18864)]

Breakpoint 1, main (speid=268566544, argp=268566656, envp=0) at dma_error_spu.c:143
143       mfc_get (&control_block, argp + 8, sizeof (control_block_t), tag, 0, 0);
gdb) print /x argp
$2 = 0x10020080
(gdb) c
Continuing.

Program received signal SIGBUS, Bus error.
0x00000528 in main (speid=268566544, argp=268566656, envp=0) at dma_error_spu.c:147
147       mfc_read_tag_status_all ();
```

From *Listing 5*, we can see that the bus error probably happened because of the DMA transfer (`mfc_get`) on line 143 in the dma_error_spu.c file. Examine the effective address value (argp + 8) by printing out the `argp` symbol. Since `argp` has a value of 0x10020080, `argp + 8` is definitely a misaligned effective address for the DMA command. Changing the effective address parameter to just `argp` should fix the bus error.

Tag ID errors

All DMA commands except `getllar`, `putllc`, and `putlluc` can be tagged with a 5-bit tag group ID (which defines up to 32 IDs). Programs can use this identifier to check for, or wait on the completion of all queued DMA commands in one or more tag groups. Valid tag group IDs can be any value between 0 and 31 inclusive. Tag group IDs which are not in this range trigger the MFC unit to raise an interrupt to the PPE resulting in the program getting a ″Bus error″.

Use of the tag manager to reserve and release tag IDs is encouraged to ensure valid DMA tag Ids and to facilitate cooperative use of tag IDs among multiple software components. More information about the tag manager can be found in the *C/C++ Language Extensions for Cell Broadband Engine Architecture*.

The following example contains a DMA transfer with a bad tag group ID, and shows how to walk through a gdb debugging session to detect this problem.

Complete listing of the code for this example can be found in the directory:
`/opt/cell/sdk/src/examples/dma_errors/bad_tag_id_error`

The **info spu dma** command can be used to detect whether a DMA error has occurred. The DMA transfers shown with the ″E″ bit marked need to be examined more closely. The field `Tag` in the **info spu dma** output is a 5-bit field, meaning this field does not show when a tag ID is out of range. The recommended way to detect an invalid tag group ID is to examine the inputs to the DMA command immediately preceding the command which issues the DMA request. Examining the inputs requires setting a breakpoint in the SPU source code and printing out the values of the DMA parameters. The following listing shows an example gdb session:

Listing 1

```
$ ppu-gdb dma_error
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
```

```
and "show warranty" for details.
This GDB was configured as "powerpc64-linux"...
(gdb) break dma_error_spu.c:145
No source file named dma_error_spu.c.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (dma_error_spu.c:145) pending.
(gdb) r
Starting program: /opt/cell/sdk/src/examples/dma_errors/bad_tag_id_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x400012af230 (LWP 8766)]
[Switching to Thread 0x400012af230 (LWP 8766)]

Breakpoint 1, main (speid=268566544, argp=268566656, envp=0) at dma_error_spu.c:146
146       mfc_get (&control_block, argp, sizeof (control_block_t), tag, 0, 0);

(gdb) print tag
$4 = 32
```

In *Listing 1* a breakpoint is set to stop at line 145 in the SPU source code, which is before the line where the first DMA transfer is issued. The program is then run using the `r` command. When the breakpoint is hit, the tag ID is displayed using the `print` *`tag`* command. Note *`tag`* is the name of the variable containing the tag ID in this example. The tag ID is 32, which is out of the acceptable range of 0 to 31 inclusive. Changing the tag ID to 0, and recompiling the program enables it to run to completion.

Transfer size errors

The another version of the example, `bad_dma_size_error`, contains a DMA transfer with an illegal size.

The control block data structure, `control_block_t`, defined in `dma_error.h` is not padded to be a multiple of 16 bytes. The size of the control block data structure is only 24 bytes.

The complete listing of the code for this example can be found in:

```
/opt/cell/sdk/src/examples/dma_errors/bad_dma_size_error
```

Use the gdb **info spu dma** command to examine all in-flight DMA transfers.

Listing 1

```
$ ppu-gdb dma_error
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "powerpc64-linux"...
(gdb) r
Starting program: /opt/cell/sdk/src/examples/dma_errors/bad_dma_size_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x400012af230 (LWP 12328)]

Program received signal SIGBUS, Bus error.
[Switching to Thread 0x400012af230 (LWP 12328)]
0x000004f0 in main (speid=268566544, argp=268566656, envp=0) at dma_error_spu.c:147
147       mfc_read_tag_status_all ();
(gdb) info spu dma
Tag-Group Status  0x00000000
Tag-Group Mask    0x00000001 (undefined query type)
Stall-and-Notify  0x00000000
```

```
Atomic Cmd Status 0x00000000

Opcode  Tag TId RId EA                 LSA     Size    LstAddr LstSize E
get     0   0   0   0x0000000010020080 0x21280 0x00008                 *
```

Reviewing the output in *Listing 1* shows that there is one pending DMA transfer. The ″E″ bit is on indicating there is a DMA transfer error that has been detected by the hardware. Use the Bus Error table above and check each condition. Following the steps shown in the unaligned effective address example, we can se see that the EA is properly aligned since the low effective address ends in 0x80.

The size of the DMA transfer, however, is reported as 8 (*Listing 1*). At first, this appears to be valid, but it is not necessarily the size specified in the DMA command. What is reported here is actually the number of bytes yet to be transferred when the interrupt is raised.

Use the `info symbol ...` command to determine the symbol closest to the LSA specified in the output from *Listing 1*.

Listing 2

```
(gdb) info symbol 0x21280
control_block in section .bss
(gdb) print &control_block
$1 = (control_block_t *) 0x21280
```

Listing 2 the gdb output shows that the LSA of 0x21280 is the local address of the variable control_block in this DMA transfer. We can use gdb to take a closer look at the DMA transfer that fetches the content of the control block from main memory. Because *Listing 2* shows that the program breaks around line 147. Set a breakpoint to stop at line 140 and step through the program.

Listing 3

```
$ ppu-gdb dma_error
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "powerpc64-linux"...
(gdb) break dma_error_spu.c:140
No source file named dma_error_spu.c.
Make breakpoint pending on future shared library load? (y or [n]) y

Breakpoint 1 (dma_error_spu.c:140) pending.
(gdb) r
Starting program: /opt/cell/sdk/src/examples/dma_errors/bad_dma_size_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x400012af230 (LWP 12169)]
[Switching to Thread 0x400012af230 (LWP 12169)]

Breakpoint 1, main (speid=268566544, argp=268566656, envp=0) at dma_error_spu.c:143
143       mfc_get (&control_block, argp, sizeof (control_block_t), tag, 0, 0);
(gdb) print tag
$1 = 0
(gdb) print /x &control_block
$3 = 0x21280
```

The output in *Listing 3* shows that the tag id is 0 which is in the right range (between 0 and 31). The target local storage address (address of the control_block variable) is properly aligned since it ends in 0x80.

Next, query for the size of this data structure.

Listing 4

```
(gdb) print sizeof(control_block)
$1 = 24
```

In *Listing 4* since the size of the control block is 24, one of two things has happened. Either the size of the DMA request was set to 24 bytes, which is an error since 24 is not a multiple of 16, or the DMA request size was set to 32 bytes, which is an error because the block is only 24 bytes in size.

In this example, the request size was set to 24. To fix this error, pad the data structure to be a multiple of 16 byte adding two integers at the end of the control block structure. The new data structure in **dma_error.h** becomes:

Listing 5

```
typedef struct _control_block
{
  unsigned long long in_addr;   //beginning address of the input array
  unsigned long long out_addr;  //beginning address of the output array
  unsigned int       num_elements_per_spe; //number of elmts assigned to this spe
  unsigned int       id;        //spe id
  unsigned int       pad[2];    //pad this data structure to be multiple of 16
} control_block_t;
```

The DMA transfer size should now be 32 bytes. After the code is recompiled with these modifications the example runs successfully.

Setting a breakpoint to detect errors in the DMA command can only be done easily for programs with few DMA transfers. For DMA transfers in loops with thousands or more iterations, this technique is not very practical since it requires the programmer to look at the parameters for thousands or more DMA transfers. In these cases, the programmer is recommended to learn as much from the output of **info spu dma** as possible. If errors are not readily detected from such output, the programmer can use gdb to look at the code listing before and after the Bus error to determine the DMA transfer that caused the Bus error. Once the DMA command that caused the Bus error is detected, one can look at the inputs for the specific DMA command to find the invalid parameter. The following example shows a DMA list transfer with one of the list elements containing an invalid transfer size. The invalid DMA list command is in a loop with 32 iterations.

The complete listing of the code for this example can be found in:

```
/opt/cell/sdk/src/examples/dma_errors/bad_dma_size_loop_error
```

Listing 6

```
$ ppu-gdb dma_error
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "powerpc64-linux"...
(gdb) r
Starting program: /opt/cell/sdk/src/examples/dma_errors/bad_dma_size_loop_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x400012af230 (LWP 11930)]

Program received signal SIGBUS, Bus error.
```

```
[Switching to Thread 0x400012af230 (LWP 11930)]
0x000008a4 in main (speid=268566544, argp=268566656, envp=0) at dma_error_spu.c:210
210         mfc_read_tag_status_all ();
(gdb) info spu dma
Tag-Group Status  0x00000001
Tag-Group Mask    0x00000001 ('all' query pending)
Stall-and-Notify  0x00000000
Atomic Cmd Status 0x00000000

Opcode  Tag TId RId EA                 LSA     Size    LstAddr LstSize E
getl    0   0   0   0x000004000044f080 0x20500 0x00001 0x21618 0x00008 *
```

Listing 6 shows that the program stopped at line 210 in the dma_error_spu.c file. The output of the **info spu dma** command indicates that the Bus error was caused by a DMA list transfer. Looking at the parameters in the command, we can see that the size displayed is 1, which is a suspicious value for this parameter. Furthermore, the list size (LstSize) parameter is shown as 0x00008, which means the offending DMA list element is probably at the end of the list. We can now take a closer look at the code to determine the actual invalid DMA command.

Listing 7

```
(gdb) list
205         /* issue a DMA get list command to gather the NUM_LIST_ELEMENT chunks of data from system memory */
206         mfc_getl (local_buffer, in_addr, dma_list, NUM_LIST_ELEMENTS * sizeof(mfc_list_element_t), tag, 0, 0);
207
208        /* wait for the DMA get list command to complete */
209         mfc_write_tag_mask (1 << tag);
210         mfc_read_tag_status_all ();
211
212         /* invoke process_data to work on the EA, the tag-id and the transfer size as
            potential invalid DMA parameters (Listing 17).  The data that's just been moved
            into local store*/
213         process_data_simd (local_buffer, CHUNK_SIZE * NUM_LIST_ELEMENTS);
214
```

Listing 7 shows the source code listing centered around line 210, which is the stopping point of the SPU program. The DMA list command on line 206 seems to be the DMA that caused the Bus Error. Examining the dma_list parameter shows the following:

Listing 8

```
(gdb) print i
$1 = 992
(gdb) print dma_list
$2 = {{notify = 0, reserved = 0, size = 4096, eal = 4391040}, {notify = 0, reserved = 0, size = 4096,
    eal = 4395136}, {notify = 0, reserved = 0, size = 4096, eal = 4399232}, {notify = 0, reserved = 0,
    size = 4096, eal = 4403328}, {notify = 0, reserved = 0, size = 4096, eal = 4407424}, {notify = 0,
    reserved = 0, size = 4096, eal = 4411520}, {notify = 0, reserved = 0, size = 4096, eal = 4415616}, {
    notify = 0, reserved = 0, size = 4096, eal = 4419712}, {notify = 0, reserved = 0, size = 4096,
    eal = 4423808}, {notify = 0, reserved = 0, size = 4096, eal = 4427904}, {notify = 0, reserved = 0,
    size = 4096, eal = 4432000}, {notify = 0, reserved = 0, size = 4096, eal = 4436096}, {notify = 0,
    reserved = 0, size = 4096, eal = 4440192}, {notify = 0, reserved = 0, size = 4096, eal = 4444288}, {
    notify = 0, reserved = 0, size = 4096, eal = 4448384}, {notify = 0, reserved = 0, size = 4096,
    eal = 4452480}, {notify = 0, reserved = 0, size = 4096, eal = 4456576}, {notify = 0, reserved = 0,
    size = 4096, eal = 4460672}, {notify = 0, reserved = 0, size = 4096, eal = 4464768}, {notify = 0,
    reserved = 0, size = 4096, eal = 4468864}, {notify = 0, reserved = 0, size = 4096, eal = 4472960}, {
    notify = 0, reserved = 0, size = 4096, eal = 4477056}, {notify = 0, reserved = 0, size = 4096,
    eal = 4481152}, {notify = 0, reserved = 0, size = 4096, eal = 4485248}, {notify = 0, reserved = 0,
    size = 4096, eal = 4489344}, {notify = 0, reserved = 0, size = 4096, eal = 4493440}, {notify = 0,
    reserved = 0, size = 4096, eal = 4497536}, {notify = 0, reserved = 0, size = 4096, eal = 4501632}, {
    notify = 0, reserved = 0, size = 4096, eal = 4505728}, {notify = 0, reserved = 0, size = 4096,
    eal = 4509824}, {notify = 0, reserved = 0, size = 4096, eal = 4513920}, {notify = 0, reserved = 0,
    size = 4097, eal = 4518016}}
```

Listing 8 shows the content of the `dma_list` parameter. A close look at the content of the list elements at the end of the list confirms our suspicion. The last list

element has a size of 4097, which is not a valid value for a DMA transfer. Changing the size of the last list element to a multiple of 16 bytes should fix the bus error.

Unaligned local store address

Unaligned local store address can be detected by using gdb in a similar manner to the techniques in the previous example.

The following example contains a DMA transfer with a misaligned SPE local store address. Complete listing of the code for this example can be found in the directory:

```
/opt/cell/sdk/src/examples/dma_errors/unaligned_lsa_error
```

Once the program execution breaks due to a bus error, the command **info spu dma** can be used to determine if any of the DMA commands has an error. The DMA transfers shown with the ″E″ bit turned on need to be examined more closely.

The following gdb output shows the result of running the example and the command **info spu dma**:

Listing 1

```
(gdb dma_error
#2  <cross-architecture call>
#3  0x000000800bdeff00 in .syscall () from /lib64/libc.so.6
#4  0x000004000003567c in ._base_spe_context_run () from /usr/lib64/libspe2.so.2
#5  0x0000040000029e24 in .spe_context_run () from /usr/lib64/libspe2.so.2
#6  0x0000000010000c1c in ppu_pthread_function (argp=0xfffffc8ea38) at dma_error.c:69
#7  0x000000800c00bcf0 in .start_thread () from /lib64/libpthread.so.0
#8  0x000000800bdf49fc in .__clone () from /lib64/libc.so.6
```

From *Listing 1*, it is not apparent which of the four DMA parameters (tag id, effective address, local store address, and size) misbehave.

Listing 2

```
(gdb) r
Starting program: /opt/cell/sdk/src/examples/dma_errors/unaligned_lsa_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x400012af230 (LWP 12581)]
[Switching to Thread 0x400012af230 (LWP 12581)]

Breakpoint 1, main (speid=268566544, argp=268566656, envp=0) at dma_error_spu.c:140
140       if (tag == MFC_TAG_INVALID)
(gdb) step 1
148       mfc_get (&control_block_data[4], argp, sizeof (control_block_t), tag, 0, 0);
(gdb) print tag
$1 = 0
(gdb) print sizeof(control_block_t)
$2 = 32
(gdb) print /x &control_block_data[4]
$4 = 0x21304
```

Following the steps shown in the previous examples, we can quickly eliminate the EA, the tag-id and the transfer size as potential invalid DMA parameters (*Listing 2*). The local store address (*&control_block_data[4]*), however, ends with 0x04. This address is not aligned on a 16-byte boundary. This is the cause of the bus error.

Segmentation faults

Segmentation faults can be caused by DMA transfers to or from bad effective addresses.

Debugging segmentation faults caused by invalid effective addresses is similar to the techniques used to debug segmentation faults in other types of code.

The following example demonstrates a gdb debugging session to resolve a segmentation violation resulting from a bad effective address in a DMA list element.

Complete listing of the code for this example can be found in the directory:
`/opt/cell/sdk/src/examples/dma_errors/bad_eal_in_dma_list_error`

The example session within gdb begins as follows:

Listing 1

```
$ ppu-gdb dma_error
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "powerpc64-linux"...
(gdb) r
Starting program: /opt/cell/sdk/src/examples/dma_errors/bad_eal_in_dma_list_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x40000aef230 (LWP 11714)]

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x40000aef230 (LWP 11714)]
0x000006d4 in main (speid=268566544, argp=268566656, envp=0) at dma_error_spu.c:181
181         mfc_read_tag_status_all ();
```

The program is loaded and started with the `r` command. When the segmentation fault occurs gdb stops execution. Next a standard `backtrace` command is issued using the `bt` command to display the stack frames which led to the segmentation fault:

Listing 2

```
(gdb) bt
#0  0x000006d4 in main (speid=268566544, argp=268566656, envp=0) at dma_error_spu.c:181
#1  0x0000008c in _start () from dma_error_spu@0x10001780 <5>
#2  <cross-architecture call>
#3  0x000000800bdeff00 in .syscall () from /lib64/libc.so.6
#4  0x000004000003567c in ._base_spe_context_run () from /usr/lib64/libspe2.so.2
#5  0x0000040000029e24 in .spe_context_run () from /usr/lib64/libspe2.so.2
#6  0x0000000010000c1c in ppu_pthread_function (argp=0xfffffc8ea38) at dma_error.c:69
#7  0x000000800c00bcf0 in .start_thread () from /lib64/libpthread.so.0
#8  0x000000800bdf49fc in .__clone () from /lib64/libc.so.6
```

Listing 2 does not show any obvious problems, so the **info spu dma** command is issued to see if there is any error in the DMA commands:

Listing 3

```
(gdb) info spu dma
Tag-Group Status  0x00000001
Tag-Group Mask    0x00000001 (undefined query type)
Stall-and-Notify  0x00000000
Atomic Cmd Status 0x00000000

Opcode  Tag TId RId EA                 LSA     Size    LstAddr LstSize E
getl    0   0   0   0x0000040000000000 0x09300 0x01000 0x11360 0x00040
```

There is only one outstanding DMA list transfer shown in the output, and the ″E″ bit is not turned on. This indicates all the input parameters to the DMA command

are valid, and checks for transfer size, alignment, and list element crossing 4 GB boundary problems are not needed. The size of this DMA list transfer (LstSize) is shown as 0x40 bytes (64 bytes). Examining the source code shows the DMA list has a total size of 128 bytes (16 entries * 8 bytes/entries). These two facts mean half of the DMA list has been processed successfully and the problem occurs in the second half of the DMA list. Using this clue a closer look at the DMA list is done:

Listing 4

```
(gdb) print /x dma_list
$1 = {{notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0x50080},
{notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0x51080},
{notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0x52080},
{notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0x53080},
{notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0x54080},
{notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0x55080},
{notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0x56080},
{notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0x57080},
{notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0x0},
{notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0x59080},
{notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0x5a080},
{notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0x5b080},
{notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0x5c080},
{notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0x5d080},
{notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0x5e080},
{notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0x5f080}}
```

The gdb output shows the EAL of the 9th DMA list entry as 0. This caused the segmentation violation. Looking through the code it reveals that the **fill_dma_list** function (lines 112 to 115) sets the EAL of the 9th entry to 0. Removing this part of the code, recompiling the program, and the program runs successfully.

The example here is a little contrived since it purposely sets the EAL of a specific DMA list element to an invalid value; however, the general approach used to debug a segmentation fault should still apply to most ″real-world″ scenarios.

DMA list element crossing 4 GB boundary

The CBEA specifies that the EAL (the 32-bit low-order effective address) for each list element in a DMA list must be in the 4 GB aligned area defined by the EAH (the 32-bit high-order effective address). Although each EAL starting address is in a single 4 GB area, a list element transfer may cross the 4 GB boundary.

However, in the Cell/B.E. and PowerXCell 8i processors, a DMA list element that crosses a 4 GB boundary will result in a Class0 DMA Alignment Error exception. The Linux operating system makes no effort to detect or recover from this error. Therefore, having a list element crossing a 4 GB boundary in a DMA list results in a bus error at execution time.

A complete listing of the code for this example can be found in the directory:

```
/opt/cell/sdk/src/examples/dma_errors/4GB_crossing_error
```

In the following example, the code is run under gdb. After the program execution breaks due to the ″Bus error″, the **info spu dma** command is issued. The output from gdb is presented as follows:

Listing 1

```
$ ppu-gdb dma_error
GNU gdb 6.8.50.20080526-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "powerpc64-linux"...
(gdb) r
Starting program: /opt/cell/sdk/src/examples/dma_errors/4GB_crossing_error/dma_error
[Thread debugging using libthread_db enabled]
[New Thread 0x40000e9f230 (LWP 865)]

Program received signal SIGBUS, Bus error.
[Switching to Thread 0x40000e9f230 (LWP 865)]
0x00000694 in main (speid=268566544, argp=268566656, envp=0) at dma_error_spu.c:176
176         mfc_read_tag_status_all ();
(gdb) info spu dma
Tag-Group Status  0x00000001
Tag-Group Mask    0x00000001 (undefined query type)
Stall-and-Notify  0x00000000
Atomic Cmd Status 0x00000000

Opcode  Tag TId RId EA                 LSA     Size    LstAddr LstSize E
getl    0   0   0   0x00000001fffff080 0x08280 0x01000 0x092d8 0x00008 *
(gdb) info symbol 0x08280
local_buffer + 28672 in section .bss
(gdb) info symbol 0x092d8
dma_list + 56 in section .bss
```

The output shows an error in the DMA get list command. Looking up the symbols associated with the LSA and the *LstAddr* using the **info symbol** command, shows partial transfers have been completed and the LSA and *LstAddr* are not the same as the original LSA and *LstAddr* in the MFC command.

The next step is to take a closer look at the actual DMA list, as follows:

Listing 2

```
(gdb) print /x dma_list
$1 = {{notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0xffff8080}, {notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0xffff9080}, {
    notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0xffffa080}, {notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0xffffb080}, {
    notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0xffffc080}, {notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0xffffd080}, {
    notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0xffffe080}, {notify = 0x0, reserved = 0x0, size = 0x1000, eal = 0xfffff080}}
```

Listing 2 shows the content of the current DMA list. The last DMA entry has an EAL of 0xfffff080 and a size of 0x1000. This means the list entry actually crosses the 4 GB boundary. Changing either the allocation of the original input data or issuing another individual DMA transfer for the last DMA list entry should fix the problem. For more information explaining the limitations of DMA list elements crossing 4 GB boundaries, refer to “Managing a DMA list element crossing 4 GB boundary” on page 19

DMA race conditions

Debugging race conditions caused by DMA transfers on Cell/B.E. is a difficult task. The SDK provides some utilities to aid in finding them.

The race check library provides a set of routines, which support the software detection of frequently encountered race conditions involving local store data transfers and SPE local storage accesses. These race conditions occur as a result of the indeterminate ordering of the transactions performed on the local store

memory. For complete documentation on how to use the race check library to detect race conditions, refer to the *Cell Broadband Engine SDK Example Libraries* version 3.1.

An example demonstrating the use of the race check library can be found in the directory:

```
/opt/cell/sdk/src/examples/race_check
```

Programmers can also use the IBM Full System Simulator for the Cell Broadband Engine

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

to detect potential race conditions in SPU programs.

Chapter 5. Using the SPU GNU profiler

You use profiling to find out which parts of your program are traversed the most. Profiling helps to identify the hot spots and can be used to optimize the application performance.

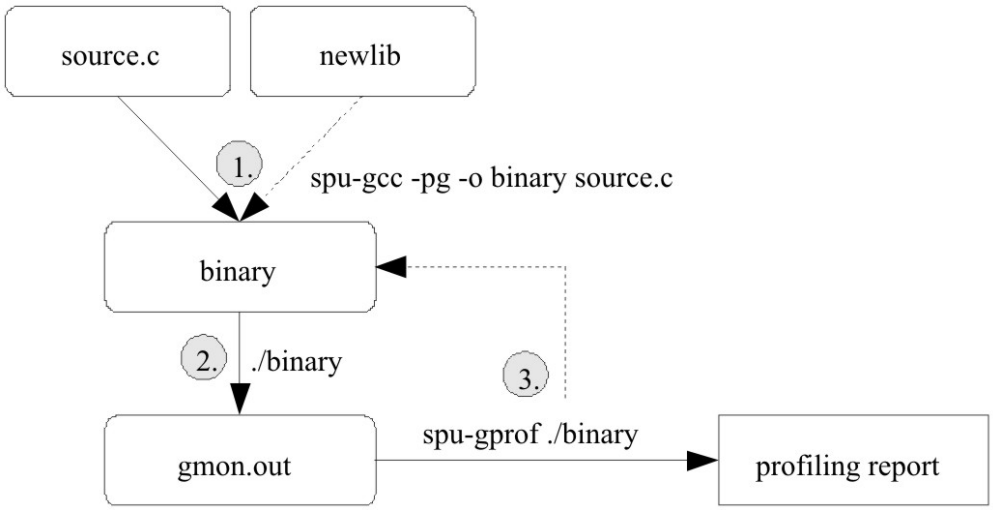
Several forms of output are available from the analysis from the gprof info page, for more information, refer to:

`http://sourceware.org/binutils/docs/gprof/Introduction.html#I`

The call graph shows, for each function, which functions called it, which other functions it called, and how many times. It also estimates how much time was spent in the subroutines of each function. This can suggest places where you can try to eliminate function calls that are very time-consuming.

Usage and toolchain flow

The following diagram shows the toolchain flow for SPU-Gprof usage:



- If the code gets compiled and linked with the `-pg` flag it is enriched by the compiler to collect profiling data. The linker automatically links a certain gcrt[1,2].o to the binary that manages profiling during runtime.
- When the binary is executed profiling data is being collected. The gmon.out file is written before the program exits.
- The spu-gprof tool reads the gmon.out and the binary files to generate the profiling report.

How the SPU GNU profiler works

The gcrt[1,2] calls a function that initializes the profiler. It sets up the timer and data structures needed for profiling. It also registers the a cleanup function at exit that writes the gmon.out file.

© Copyright IBM Corp. 2006, 2008

The flat profile is based on a histogram that is created using a sampling mechanism. The sampling collects the content of the program count register using SPU timers. The Histogram data is directly written into the gmon.out file.

For the callgraph profile, the profiler needs to know what functions are called, from where, and how often. The compiler generates a particular prologue that causes every function to call _mcount which is implemented in newlib. Its major task is to extract the address of the function just entered and the address of the caller of that function and then call the ordinary C function __mcount_internal that handles the call graph and counts the calls. The collected data is stored in the PPU memory using the EA cache management and gets written into the gmon.out file when the program ends.

A simple spulet test

The following shows a simple spulet test:

1. Create SPU code:
   ```
   echo 'int main(){ return 0;}' > test.c
   ```
2. Compile and link the code using the `-pg` switch:
   ```
   spu-gcc -pg test.c -o test
   ```
3. Run the spulet (this collects profiling data and create the gmon.out file):
   ```
   ./test
   ```
4. Run the profiler to view the results:
   ```
   spu-gprof -b ./test
   ```

   The result looks similar to this:
   ```
   Flat profile:

   Each sample counts as 0.01 seconds.
   no time accumulated

     %   cumulative   self              self     total
    time   seconds   seconds    calls  Ts/call  Ts/call  name
     0.00      0.00     0.00        1     0.00     0.00  main

                          Call graph
   granularity: each sample hit covers 4 byte(s) no time propagated

   index % time    self  children    called     name
                   0.00    0.00       1/1       _start [70]
   [1]      0.0    0.00    0.00       1         main [1]
   -----------------------------------------------------

   Index by function name

      [1] main
   ```

Limitations

- The support for binaries that are using overlays is not implemented. That would require major changes to the gprof tool of the binutils package.
- Since profiling increases the size of the binary there is a small window where it could not fit into the local store.
- SPE programs that install their own first level interrupt handlers are not supported. This is because the profiler uses the SPU Timer Library that installs its own interrupt handler. The limitation is inherited from the SPU Timer Library. The SPE can still use interrupts by using the infrastructure provided by

the SPU Timer Library. For more information about the SPU Timer Library, refer the *SDK SPU Runtime Library Extensions Programmer's Guide and API Reference*.

- The profiler needs one timer for its sampling mechanism there are only SPU_TIMER_NTIMERS -1 left for the users code.
- The sampling statistic could be influenced if user code suspends the profilers sampling mechanism i.e. using **spu_clock_stop()**.
- Gprof does not support profiling of multiple threads:
  - It's not possible to profile multiple SPE programs that run at the same time
  - If the PPE binary is being profiled using Gprof, the SPE program cannot be profiled
- The profiler uses the software managed cache for its profiling data (see section 3). That might have implications to programs that already use the software managed cache. The user should employ existing compiler options like -mcache-size= to tune cache usage by the profiling infrastructure.
- The newlibs profiler currently expects 32 bit PPU addresses. If you are embedding an SPU binary into a PPU binary you need to compile, embed, and link using the -m32 switch.

Further information

- http://www.gnu.org/software/binutils/
- http://sourceware.org/binutils/docs/gprof/index.html
- There are man and info pages for gprof

Chapter 6. Analyzing Cell/B.E. SPUs with kdump and crash

The SDK provides a means of debugging kernel data related to SPUs through specific crash commands, by using a dumped kernel image.

This functionality is based on the use of **kdump**, and the documentation can be found in the `Documentation/kdump/kdump.txt file` from the Linux kernel sources.

The solution is composed of two environments:

- The production system, which runs the kernel where problems can occur
- The analysis system, where the information (dump file) captured by the production system is analyzed and the possible problems are identified

Installation requirements

This topic describes the supported hardware and software.

The production system must be Cell/B.E. hardware. Otherwise, SPU-specific data that is used by crash is not available in the dump file. The analysis system can be any PowerPC hardware, either 32 or 64-bit.

To make use of the SPU crash commands, you need the following SDK packages:

- **Production system:**
  - `kexec-tools` – tool used for dump capture kernel warm boot
  - `kernel` – kernel image that starts dump capture kernel when a crash event occurs
  - `kernel-kdump` – dump capture kernel image
  - `busybox-kdump` – customized `initrd` for booting the dump capture kernel which uses the minimum of memory:
- **Analysis system**
  - `crash` – crash tool for analyzing dump files
  - `crash-spu-commands` – SPU-specific commands for crash tool
  - `kernel-debuginfo` – provides a kernel image with debug information

The `kernel-debuginfo` package is available for for PPC64 architecture. If the analysis system is a 32-bit system, it must be installed in a PPC64 or Cell/B.E. BladeCenter and the `/usr/lib/debug/lib/modules-<version>-<release>/vmlinux` file must be copied to the analysis system.

You must install all these packages manually. The package `crash-spu-commands` installs `crash` if necessary. The following is an example of how to install the analysis system packages using `yum`:

```
yum install crash-spu-commands kernel-debuginfo
```

For the production system, you can use a package manager such as `yum` to install the packages `kernel`, `kexec-tools`, `kernel-kdump` and `busybox-kdump` as follows:

```
yum install kernel kexec-tools kernel-kdump busybox-kdump
```

© Copyright IBM Corp. 2006, 2008

busybox-kdump is an optional package, which is recommended for the production system because it provides a custom initrd that allows to boot a dump capture kernel using the minimum amount of memory required to save a dump file.

Production system

The production system runs the kernel image included with the SDK kernel package.

The following additional steps are required to configure the production system:

1. Reserve at boot-time the memory necessary for booting the second (dump capture) kernel, that is provided by kernel-kdump package
2. Load the dump capture kernel into the reserved memory

The optional package busybox-kdump provides a custom initrd that runs with 48 MB of reserved memory instead of 128 MB, the minimum amount of reserved memory if default initrd is used. This value can be greater than 200 MB, it depends on how many system services are initialized by the system.

To reserve the memory, add the crashkernel=<X>@32M parameter to the kernel boot, where <X> is the number in megabytes to be reserved. In yaboot.conf, the ″append″ line for the busybox case looks like this:

```
append="console=hvc0 root=LABEL=/ crashkernel=48M@32M"
```

After the system has started with the crashkernel parameter, you need to load the dump capture kernel image to the reserved memory. To do this, use the kexec command from the kexec-tools package, as follows:

```
# Using busybox (crashkernel=44M@32M):
kexec -p /boot/vmlinux-2.6.22.5-bsckdump \
--initrd=/usr/share/busybox/root_initrd/bin/busybox

# Using default initrd (crashkernel=xM@32M, where x >= 128):
kexec -p /boot/vmlinux-2.6.22.5-bsckdump \
--initrd=/boot/initrd-2.6.22.5-bsckdump.img

# Do not use --append with maxcpus parameter,
# as it is known not to work with Cell/B.E. Architecture.
# Maybe it can be used to change the root partition
# where dump kernel will boot, through root parameter.
```

After running the above command, any future kernel panic event automatically triggers the boot of the dump capture kernel. You can also trigger the dump capture kernel through the use of the ″Magic SysRq key″ functionality (press Alt+SysRq+C, or echo ’c’ to /proc/sysrq-trigger). You might want to do this to capture kernel dump data in the event of a system hang.

After the dump capture kernel has booted, the only task you need to do is to copy the dump file from /proc/vmcore to a persistent storage media. To avoid problems during the dump capture, it is recommended that you define a place to save the dump, which has a size which is about the amount of memory:

```
cp /proc/vmcore <vmcore_path>
```

Analysis system

The SPU-commands extension for crash provides commands that format and show data about the state of SPUs at the time of the system crash or hang.

Two parameters are necessary to run crash successfully, these are the production system kernel image compiled with debug info and the kernel dump file. The first is provided by `kernel-debuginfo` package, in the `/usr/lib/debug/lib/modules/<version>-<release>/` directory. The dump file is provided by the dump capture kernel through `/proc/vmcore` (see previous section).

The order in which the parameters are invoked is not important. For example:

```
crash /usr/lib/debug/boot/vmlinux-<version>-<release>
<vmcore_path>
```

If the above files are consistent with each other (`<vmcore_path>` is generated by a version `<version>-<release>` kernel), and a crash prompt is provided.

First of all, it is necessary to load the spufs module symbols. This is done by the following command:

```
crash> mod -s spufs
```

To use crash SPU-specific commands, use the extend command to load the `spu.so` file:

Note: It is located in the `lib64` directory.

```
crash> extend /usr/lib/crash/extensions/spu.so
```

When you load the extension, three SPU-specific commands are made available:

- `spus`
- `spurq`
- `spuctx`

These commands are described in the following paragraphs.

You can use the command `spus` to see which SPE contexts were running at the time of the system crash as shown in the following example:

```
crash> spus
NODE 0:
ID        SPUADDR      SPUSTATUS       CTXADDR       CTXSTATE    PID
 0   c000000001fac880   RUNNING   c00000003dcbdd80   RUNNABLE    1524
 1   c000000001faca80   RUNNING   c00000003bf34e00   RUNNABLE    1528
 2   c000000001facc80   RUNNING   c00000003bf30e00   RUNNABLE    1525
 3   c000000001face80   RUNNING   c000000039421d00   RUNNABLE    1533
 4   c00000003ee29080   RUNNING   c00000003dec3e80   RUNNABLE    1534
 5   c00000003ee28e80   RUNNING   c00000003bf32e00   RUNNABLE    1526
 6   c00000003ee28c80   STOPPED   c000000039e5e700    SAVED      1522
 7   c00000003ee2e080   RUNNING   c00000003dec4e80   RUNNABLE    1538

NODE 1:
ID        SPUADDR      SPUSTATUS       CTXADDR       CTXSTATE    PID
 8   c00000003ee2de80   RUNNING   c00000003dcbed80   RUNNABLE    1529
 9   c00000003ee2dc80   RUNNING   c00000003bf39e00   RUNNABLE    1535
10   c00000003ee2da80   RUNNING   c00000003bf3be00   RUNNABLE    1521
11   c000000001fad080   RUNNING   c000000039420d00   RUNNABLE    1532
12   c000000001fad280   RUNNING   c00000003bf3ee00   RUNNABLE    1536
13   c000000001fad480   RUNNING   c00000003dec2e80   RUNNABLE    1539
14   c000000001fad680   RUNNING   c00000003bf3ce00   RUNNABLE    1537
15   c000000001fad880   RUNNING   c00000003dec6e80   RUNNABLE    1540
```

The command `spuctx` shows context information. The command syntax is:

```
spuctx [ID | PID | ADDR
```

For example:

```
crash> spuctx c00000003dcbdd80 1529

Dumping context fields for spu_context c00000003dcbdd80:
  state                   = 0
  prio                    = 120
  local_store             = 0xc000000039055840
  rq                      = 0xc00000003dcbe720
  node                    = 0
  number                  = 0
  pid                     = 1524
  name                    = spe
  slb_replace             = 0x0
  mm                      = 0xc0000000005dd700
  timestamp               = 0x10000566f
  class_0_pending         = 0
  problem                 = 0xd000080080210000
  priv2                   = 0xd000080080230000
  flags                   = 0x0
  saved_mfc_sr1_RW        = 0x3b
  saved_mfc_dar           = 0xd000000000093000
  saved_mfc_dsisr         = 0x0
  saved_spu_runcntl_RW    = 0x1
  saved_spu_status_R      = 0x1
  saved_spu_npc_RW        = 0x0

Dumping context fields for spu_context c00000003dcbed80:
  state                   = 0
  prio                    = 120
  local_store             = 0xc00000003905a840
  rq                      = 0xc00000003dcbf720
  node                    = 1
  number                  = 8
  pid                     = 1529
  name                    = spe
  slb_replace             = 0x0
  mm                      = 0xc0000000005d1300
  timestamp               = 0x10000566f
  class_0_pending         = 0
  problem                 = 0xd000080080690000
  priv2                   = 0xd0000800806b0000
  flags                   = 0x0
  saved_mfc_sr1_RW        = 0x3b
  saved_mfc_dar           = 0xd0000000000f3000
  saved_mfc_dsisr         = 0x0
  saved_spu_runcntl_RW    = 0x1
  saved_spu_status_R      = 0x1
  saved_spu_npc_RW        = 0x0
```

You can use the command `spurq` to visualize all the SPU contexts that were on the SPU run-queue

```
crash> spurq
PRIO[120]:
c000000000fd7380
c00000003bf31e00
c0000000039422d00
c00000000181eb80
```

Chapter 7. Using SPU code overlays

This section describes how to use the overlay facility to overcome the physical limitations on code and data size in the SPU.

What are overlays

Optimally a complete SPU program is loaded into the local storage of the SPU before it is executed. This is the most efficient method of execution. However, when the sum of the code and data lengths of the program exceeds the local storage size it is necessary to use overlays. (For the IBM BladeCenter QS21 and IBM BladeCenter QS22 the storage size is 256 KB.) Overlays may be used in other circumstances; for example performance might be improved if the size of data areas can be increased by moving rarely used functions to overlays.

An overlay is a program segment which is not loaded into SPU local storage before the main program begins to execute, but is instead left in Cell main storage until it is required. When the SPU program calls code in an overlay segment, this segment is transferred to local storage where it can be executed. This transfer will usually overwrite another overlay segment which is not immediately required by the program.

In an overlay structure the local storage is divided into a root segment, which is always in storage, and one or more overlay regions, where overlay segments are loaded when needed. Any given overlay segment will always be loaded into the same region. A region may contain more than one overlay segment, but a segment will never cross a region boundary.

(A segment is the smallest unit which can be loaded as one logical entity during execution. Segments contain program sections such as functions and data areas.)

The overlay feature is supported for Cell SPU programming (but not for PPU programming) on a native IBM BladeCenter QS21 and IBM BladeCenter QS22 or on the simulator hosted on an x86 or PowerPC machine.

How overlays work

The code size problem can be addressed through the generation of overlays by the linker. Two or more code segments can be mapped to the same physical address in local storage. The linker also generates call stubs and associated tables for overlay management. Instructions to call functions in overlay segments are replaced by branches to these call stubs, which load the function code to be called, if necessary, and then branch to the function.

In most cases all that is needed to convert an ordinary program to an overlay program is the addition of a linker script to structure the module. In the script you specify which segments of the program can be overlaid. The linker then prepares the required segments so that they may be loaded when needed during execution of the program, and also adds supporting code from the overlay manager library.

At execution time when a call is made from an executing segment to another segment the system determines from the overlay tables whether the requested

© Copyright IBM Corp. 2006, 2008

segment is already in local storage. If not this segment is loaded dynamically (this is carried out by a DMA command), and may overlay another segment which had been loaded previously.

Restrictions on the use of overlays

When using overlays you must consider the scope of data very carefully. It is a widespread practice to group together code sections and the data sections used by them. If these are located in an overlay region the data can only be used transiently - overlay sections are not 'swapped out' (written back to Cell BE main storage) as on other platforms but are replaced entirely by other overlays.

Ideally all data sections are kept in the root segment which is never overlaid. If the data size is too large for this then sections for transient data may be included in overlay regions, but the implications of this must be carefully considered.

Planning to use overlays

The overlay structure should be considered at the program planning stage, as soon as code sizes can be estimated. The planning needs to include the number of overlay regions that are required; the number of segments which will be overlaid into each region; and the number of functions within each segment. At this stage it is better to overestimate the number of segments required than to underestimate them. It is easier to combine segments later than to break up oversize segments after they are coded.

Overview

The structure of an overlay SPU program module depends on the relationships between the segments within the module.

Two segments which do not have to be in storage at the same time may share an address range. These segments can be assigned the same load addresses, as they are loaded only when called. For example, segments that handle error conditions or unusual data are used infrequently and need not occupy storage until they are required.

Program sections which are required at any time are grouped into a special segment called the root segment. This segment remains in storage throughout the execution of an program.

Some overlay segments may be called by several other overlay segments. This can be optimized by placing the called and calling segments in separate regions.

To design an overlay structure you should start by identifying the code sections or stubs which receive control at the beginning of execution, and also any code sections which should always remain in storage. These together form the root segment. The rest of the structure is developed by checking the links between the remaining sections and analyzing these to determine which sections can share the same local storage locations at different times during execution.

Sizing

Because the minimum indivisible code unit is at the function level, the minimum size of the overlay region is the size of the largest overlaid function. If this function is so large that the generated SPU program does not fit in local storage then a

warning is issued by the linker. The user must address this problem by splitting the function into two or more smaller functions.

Scaling considerations

Even with overlays there are limits on how large an SPE executable can become. An infrastructure of manager code, tables, and stubs is required to support overlays and this infrastructure itself cannot reside in an overlay. For a program with *s* overlay segments in *r* regions, making cross-segment calls to *f* functions, this infrastructure requires the following amounts of local storage:

- manager: about 400 bytes
- tables: *s* * 16 + *r* * 4 bytes
- stubs: *f* * 16 bytes.

This allows a maximum available code size of about 512 megabytes, split into 4096 overlay sections of 128 kilobytes each. (This assumes a single entry point into each section and no global data segment or stack.)

Except for the local storage memory requirements described above, this design does not impose any limitations on the numbers of overlay segments or regions supported.

Overlay tree structure example

Suppose that a program contains seven sections which are labelled SA through SG, and that the total length of these exceeds the amount of local storage available. Before the program is restructured it must be analyzed to find the optimum overlay design.

The relationship between segments can be shown with a tree structure. This graphically shows how segments can use local storage at different times. It does not imply the order of execution (although the root segment is always the first to receive control). Figure 2 shows the tree structure for this program. The structure includes five segments:

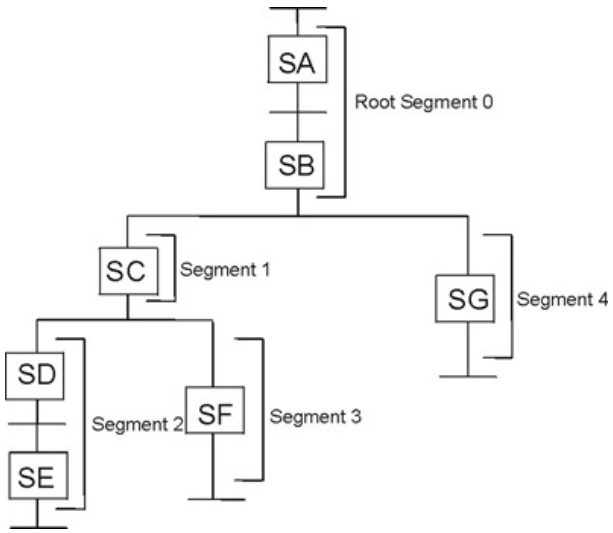


Figure 2. Overlay tree structure

The position of the segments in an overlay tree structure does not imply the sequence in which the segments are executed; in particular sections in the root

segment may be called from any segment. A segment can be loaded and overlaid as many times as the logic of the program requires.

Length of an overlay program

For purposes of illustration, assume the sections in the example program have the following lengths:

Table 10. example program lengths

Section	Length (in bytes)
SA	30,000
SB	20,000
SC	60,000
SD	40,000
SE	30,000
SF	60,000
SG	80,000

If the program did not use overlays it would require 320 KB of local storage; the sum of all sections. With overlays, however, the storage needed for the program is the sum of all overlay regions, where the size of each region is the size of its largest segment. In this structure the maximum is formed by segments 0, 4, and 2; these being the largest segments in regions 0, 1, and 2. The sum of the regions is then 200 KB, as shown in Figure 3.

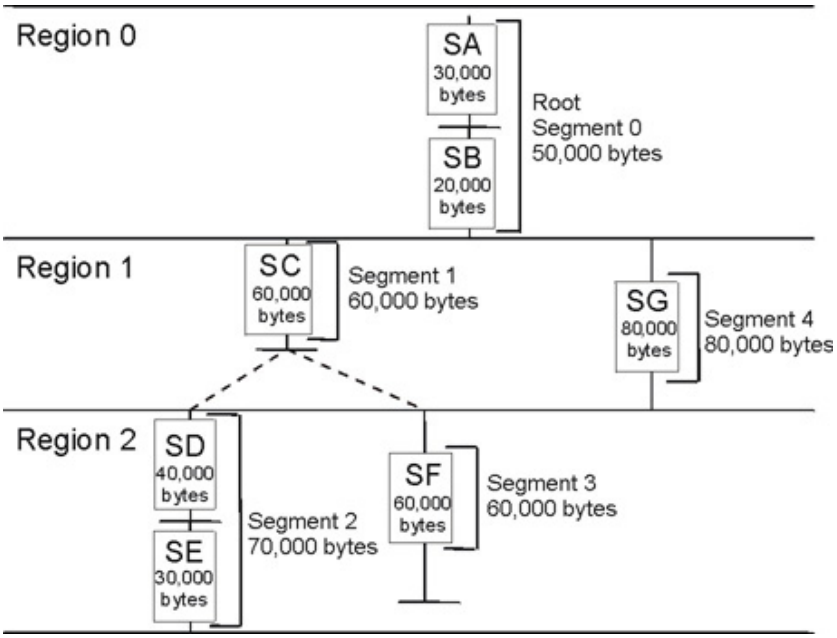


Figure 3. Length of an overlay module

Note: The sum of all regions is not the minimum requirement for an overlay program. When a program uses overlays, extra programming and tables are used and their storage requirements must also be considered. The storage required by these is described in "Scaling considerations" on page 73.

Segment origin

The linker typically assigns the origin of the root segment (the origin of the program) to address 0x80. The relative origin of each segment is determined by the length of all previously defined regions. For example, the origin of segments 2 and 3 is equal to the root origin plus 80 KB (the length of region 1 and segment 4) plus 50 KB (the length of the root segment), or 0x80 plus 130 KB. The origins of all the segments are as follows:

Table 11. Segment origins

Segment	Origin
0	0x80 + 0
1	0x80 + 50,000
2	0x80 + 130,000
3	0x80 + 130,000
4	0x80 + 50,000

The segment origin is also called the load point, because it is the relative location where the segment is loaded. Figure 4 shows the segment origin for each segment and the way storage is used by the example program. The vertical bars indicate segment origin; two segments with the same origin can use the same storage area. This figure also shows that the longest path is that for segments 0, 4, and 2.

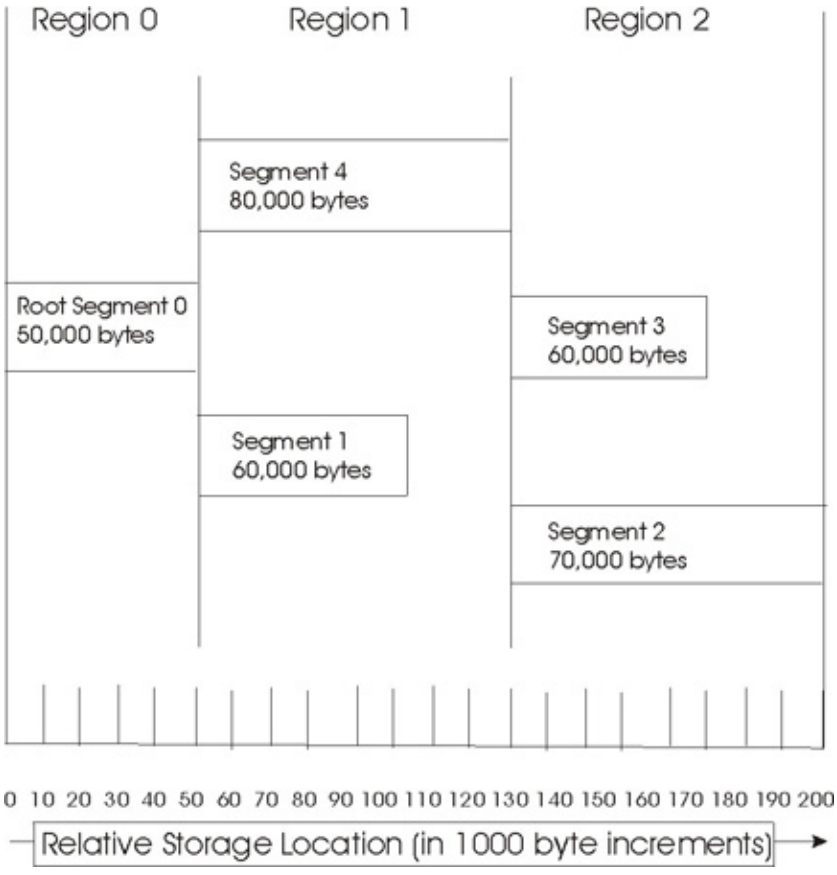


Figure 4. Segment origin and use of storage

Overlay processing

The overlay processing is initiated when a section in local storage calls a section not in storage.

The function which determines when an overlay is to occur is the overlay manager. This checks which segment the called section is in and, if necessary, loads the segment. When a segment is loaded it overlays any segment in storage with the same relative origin. No overlay occurs if one sectioncalls another section which is in a segment already in storage (in another region or in the root segment).

The overlay manager uses special stubs and tables to determine when an overlay is necessary. These stubs and tables are generated by the linker and are part of the output program module. The special stubs are used for each inter-segment call. The tables generated are the overlay segment table and the overlay region table. Figure 5 shows the location of the call stubs and the segment and region tables in the root segment in the example program.

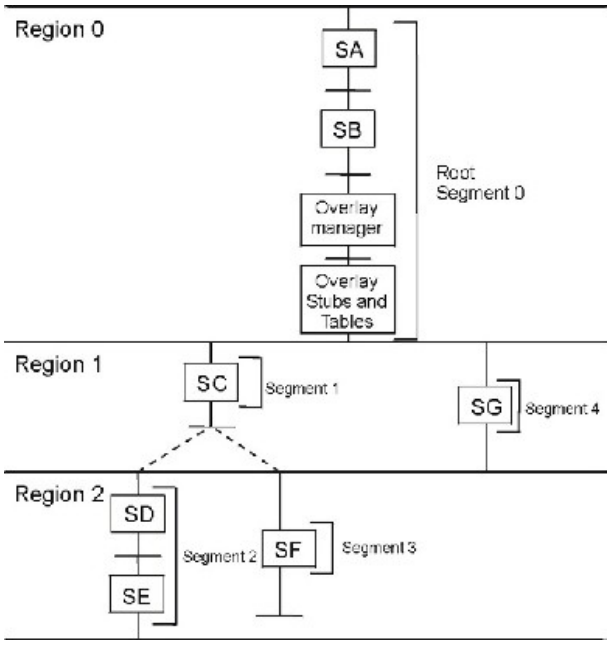


Figure 5. Location of stubs and tables in an overlay program

The size of these tables must be considered when planning the use of local storage.

Call stubs

Stubs are needed to transfer control between overlay segments. Stubs are short code sequences that specify the segment and destination address via reserved registers, and branch to the overlay manager. Any function in an overlay segment that has its address taken, and a function or case table destination in an overlay segment called from the root segment, requires one stub in the root segment. A function or case table destination in an overlay segment, called from another overlay segment, but not having its address taken, requires one stub in each caller segment. No stub is needed for control transfers within a segment.

Segment and region tables

Each overlay program contains one overlay segment table and one overlay region table. These tables are in the root segment. The segment table contains static (read-only) information about the relationship of the segments and regions in the

program. During execution the region table contains dynamic (read-write) control information such as which segments are loaded into each region.

Overlay graph structure example

If the same section is used by several segments it is usually desirable to place that section in the root segment. However, the root segment can get so large that the benefits of overlay are lost. If some of the sections in the root segment could overlay each other then the program might be described as an overlay graph structure (as opposed to an overlay tree structure) and it should use multiple regions.

With multiple regions each segment has access to both the root segment and other overlay segments in other regions. Therefore regions are independent of each other.

Figure 6 shows the relationship between the sections in the example program and two new sections: SH and SI. The two new sections are each used by two other sections in different segments. Placing SH and SI in the root segment makes the root segment larger than necessary, because SH and SI can overlay each other. The two sections cannot be duplicated in two paths.

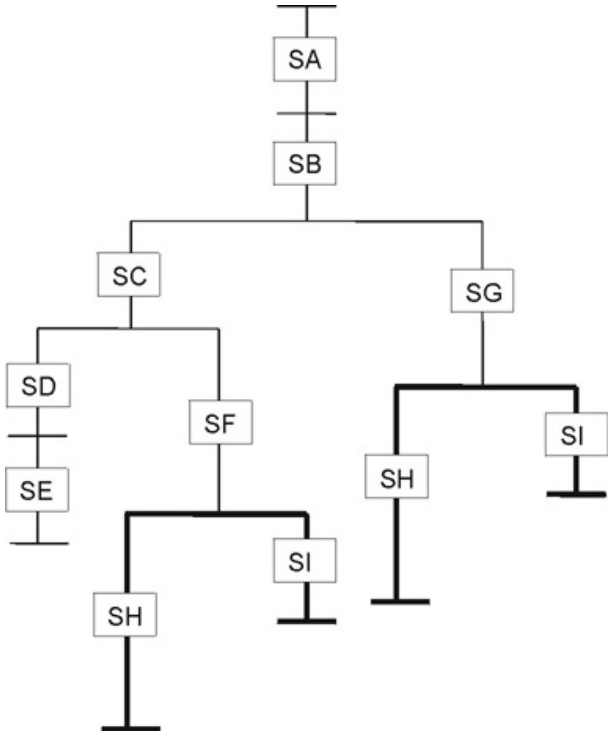


Figure 6. Overlay graph structure

However, if the two sections are placed in another region they can be in local storage when needed, regardless of the segments executed in the other regions. Figure 7 on page 78 shows the sections in a four-region structure. Either segment in region 3 can be in local storage regardless of the segments being executed in regions 0, 1, or 2. Segments in region 3 can cause segments in region 0, 1 or 2 to be loaded without being overlaid themselves.

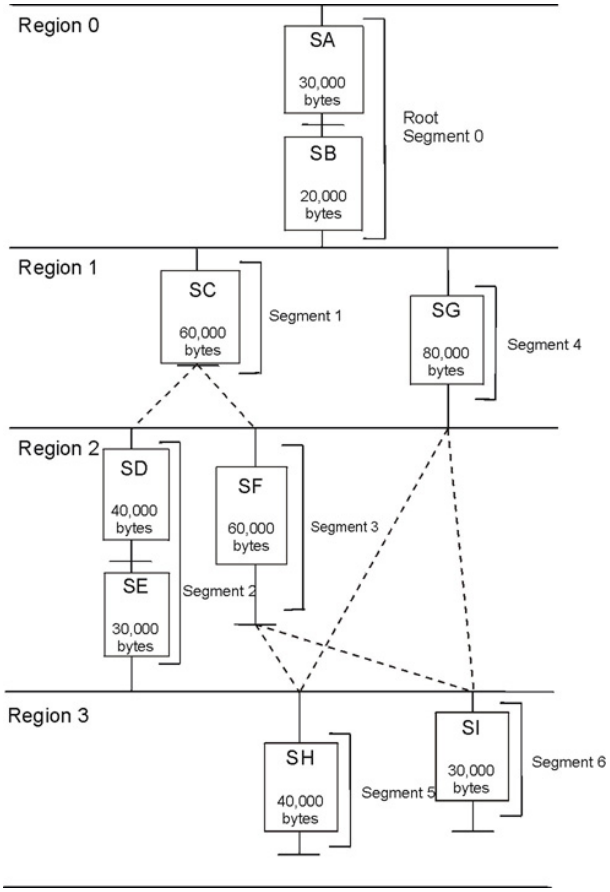


Figure 7. Overlay graph using multiple regions

The relative origin of region 3 is determined by the length of the preceding regions (200 KB). Region 3, therefore, begins at the origin plus 200 KB.

The local storage required for the program is determined by adding the lengths of the longest segment in each region. In Figure 7 if SH is 40 KB and SI is 30 KB the storage required is 240 KB plus the storage required by the overlay manager, its call stubs and its overlay tables. Figure 8 on page 79 shows the segment origin for each segment and the way storage is used by the example program.

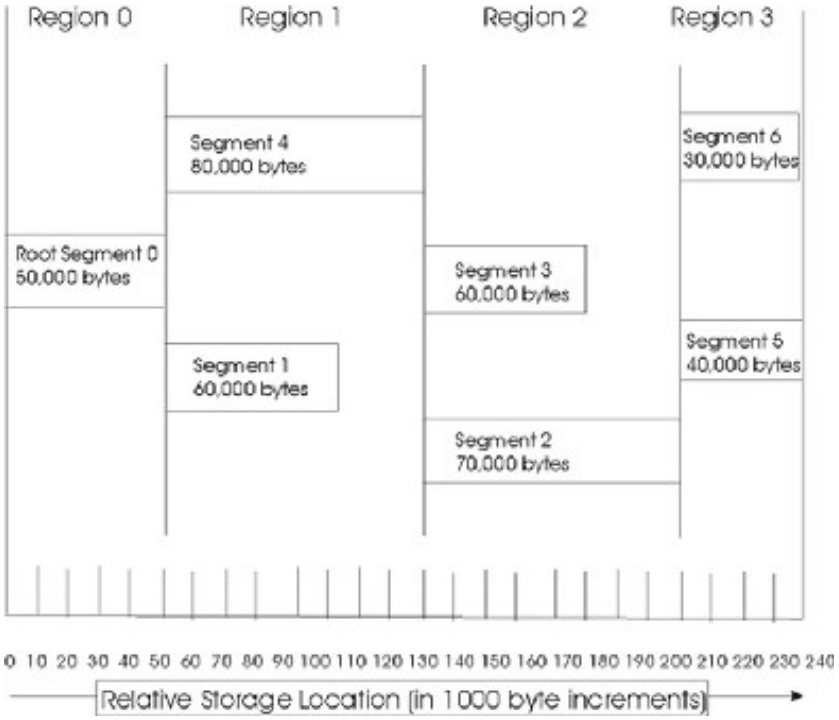


Figure 8. Overlay graph segment origin and use of storage

Specification of an SPU overlay program

Once you have designed an overlay structure, the program must be arranged into that structure. You must indicate to the linker the relative positions of the segments, the regions, and the sections in each segment, by using `OVERLAY` statements. Positioning is accomplished as follows:

Regions
: Are defined by each `OVERLAY` statement. Each `OVERLAY` statement begins a new region.

Segments
: Are defined within an `OVERLAY` statement. Each segment statement within an overlay statement defines a new segment. In addition, it provides a means to equate each load point with a unique symbolic name.

Sections
: Are positioned in the segment specified by the segment statement with which they are associated.

The input sequence of control statements and sections should reflect the sequence of the segments in the overlay structure (for example the graph in Figure 7 on page 78), region by region, from top to bottom and from left to right. This sequence is illustrated in later examples.

The origin of every region is specified with an `OVERLAY` statement. Each `OVERLAY` statement defines a load point at the end of the previous region. That load point is logically assigned a relative address at the quadword boundary that follows the last byte of the largest segment in the preceding region. Subsequent segments defined in the same region have their origin at the same load point.

In the example overlay tree program, two load points are assigned to the origins of the two OVERLAY statements and their regions, as shown in Figure 2 on page 73. Segments 1 and 4 are at the first load point; segments 2 and 3 are at the second load point.

The following sequence of linker script statements results in the structure in Figure 3 on page 74.

```
OVERLAY {
 .segment1 {./sc.o(.text)}
 .segment4 {./sg.o(.text)}
}
OVERLAY {
 .segment2 {./sd.o(.text) ./se.o(.text)}
 .segment3 {./sf.o(.text)}
}
```

Note: By implication sections SA and SB are associated with the root segment because they are not specified in the OVERLAY statements.

In the example overlay graph program, as shown in Figure 6 on page 77, one more load point is assigned to the origin of the last OVERLAY statement and its region. Segments 5 and 6 are at the third load point.

The following linker script statements add to the sequence for the overlay tree program creating the structure shown in Figure 7 on page 78:

```
.
.
.
OVERLAY {
 .segment5 {./si.o(.text)}
 .segment6 {./sh.o(.text)}
}
```

Coding for overlays

Migration/Co-Existence/Binary-Compatibility Considerations

This feature will work with both IPA and non-IPA code, though the partitioning algorithm will generate better overlays with IPA code.

Compiler options (spuxlc and GCC)

This section describes which compiler options you must use when you construct overlays.

XLC compiler options

Note: Not applicable for the GCC.

Table 12. Compiler options

Option	Description
-qipa=overlay	Specifies that the compiler should automatically create code overlays. The -qipa=partition={small | medium | large} option is used to control the size of the overlay buffer. The overlay buffer will be placed after the text segment of the linker script.

Table 12. Compiler options (continued)

Option	Description
-qipa=nooverlay	Specifies that the compiler should not automatically create code overlays. This is the default behavior for the dual source compiler.
-qipa=overlayproc=<names_list>	Specifies a comma-separated list of functions that should be in the same overlay. Multiple overlayproc suboptions may be present to specify multiple overlay groups. If a procedure is listed in multiple groups, it will be cloned for each group referencing it. C++ function names must be mangled.
-qipa=nooverlayproc= <names_list>	Specifies a comma-separated list of functions that should not be overlaid. These are always be resident in the local store. C++ function names must be mangled.

Examples:

```
# Compile and link without overlays.
spuxlc foo.c bar.c
spuxlc foo.c bar.c -qipa=nooverlay

# Compile and link with automatic overlays.
spuxlc foo.c bar.c -qipa=overlay

# Compile and link with automatic overlays and ensure that foo and bar are
# in the same overlay. The main function is always resident.
spuxlc foo.c bar.c -qipa=overlay:overlayproc=foo,bar:nooverlayproc=main

# Compile and link with automatic overlays and a custom linker script.
spuxlc foo.c bar.c -qipa=overlay -Wl,-Tmyldscript
```

GCC compiler options

The following are the GCC compiler options.

Table 13. GCC compiler options

Option	Description
-ffunction-sections	Place each function or data item into its own section in the output file. The name of the function or the name of the data item determines the section's name in the output file.
-fpartition-functions-into-sections=<bytes>	Partition a function into sections according to a threshold which indicates the maximum number of bytes each section can contain. This option can be used when a single function's body can not fully fit into the local store and thus using the above -ffunction-sections option will not be sufficient to construct the overlaid program.

Examples:

```
# Compile and link without overlays
spu-gcc bar.c

# Compile and link with custom linker script.
spu-gcc bar.c -ffunction-sections -Wl,-T,linker.script
spu-gcc bar.c -fpartition-functions-into-sections=1000 -Wl,-T,linker.script

# Compile and link with automatic overlays support.
spu-gcc bar.c -ffunction-sections -Wl,--auto-overlay
spu-gcc bar.c -fpartition-functions-into-sections=1000 -Wl,--auto-overlay
```

SDK overlay examples

Three examples are considered:

1. a very simple overlay program: "Simple overlay example";
2. the example used in the overview above: "Overview overlay example" on page 85;
3. and a ″large matrix″ example: "Large matrix overlay example" on page 86.

These examples can be found in the cell-examples RPMs included in the SDK.

Simple overlay example

This example consists of a single PPU program named `driver` which creates an SPU thread and launches an embedded SPU main program named `spu_main`. The SPU program calls four functions: `o1_test1`, `o1_test2`, `o2_test1`, and `o2_test2`. The first two functions are defined in a single compilation unit, `olay1/test.c`, and the second two functions are similarly defined in `olay2/test.c`. See the calling diagram in Figure 9. Upon completion of the SPU thread the driver returns a value from the SPU program to the PPU program.

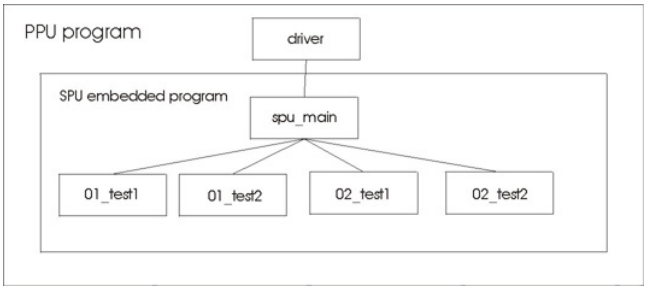


Figure 9. Simple overlay program call graph

The SPU program is organized as an overlay program with two regions and three segments. The first region is the non-overlay region containing the root segment (segment 0). This root segment contains the `spu_main` function along with overlay support programming and tables (not shown). The second region is an overlay region and contains segments 1 and 2. In segment 1 are the code sections of functions `o1_test1` and `o1_test2`, and in segment 2 are the code sections of functions `o2_test1` and `o2_test2`, as shown in Figure 10 on page 83.

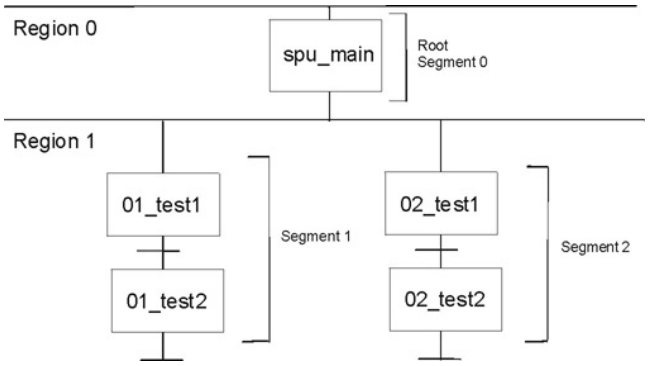


Figure 10. Simple overlay program regions, segments and sections

Combining these figures yields the following diagram showing the structure of the SPU program.

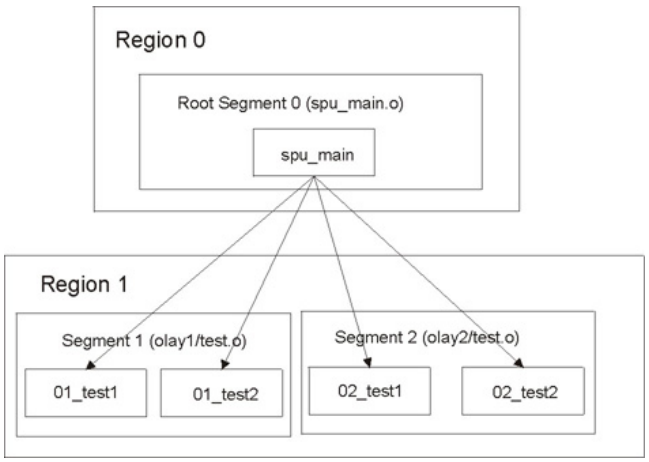


Figure 11. Simple overlay program logical structure

The physical view of this example (Figure 12 on page 84) shows one region containing the non-overlay root segment, and a second region containing one of two overlay segments. Because the functions in these two overlay segments are quite similar their lengths happen to be the same.

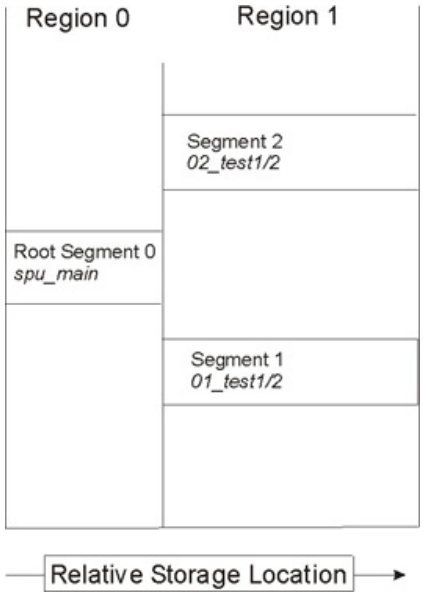


Figure 12. Simple overlay program physical structure

The `spu_main` program calls its sub-functions multiple times. Specifically the `spu_main` program first calls two functions, `o1_test1` and `o1_test2`, passing in an integer value (101 and 102 respectively) and upon return it expects an integer result (1 and 2 respectively). Next `spu_main` calls the two other functions, `o2_test1` and `o2_test2` passing in an integer value (201 and 202 respectively) and upon return it expects an integer result (11 and 12 respectively). Finally `spu_main` calls again the first two functions, `o1_test1` and `o1_test2` passing in an integer value (301 and 302 respectively) and upon return it expects an integer result (1 and 2 respectively). Between each pair of calls, the overlay manager loads the appropriate segment into the appropriate region. In this case, for the first pair it loads segment 1 into region 1 then for the second pair it loads segment 2 into region 1, and for the last pair it reloads segment 1 back into region 1. See Figure 13 on page 85.

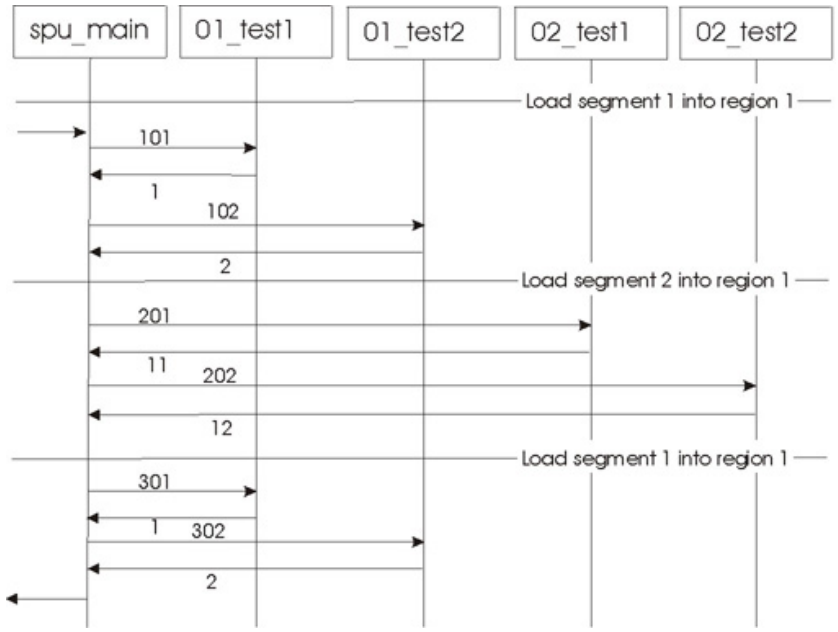


Figure 13. Example overlay program interaction diagram

The linker flags used are:

```
LDFLAGS = -Wl,-T,linker.script
```

The linker script is:

Note: To simplify the linker scripts only the affected statements are shown in this and the following examples.

```
SECTIONS
{OVERLAY :
{.segment1 {olay1/*.o(.text)}
.segment2 {olay2/*.o(.text)}
}
}
INSERT AFTER .text;
```

Overview overlay example

The overview overlay program is an adaptation of the program described in "Overlay graph structure example" on page 77. The structure is the same as that shown in Figure 7 on page 78 but the sizes of each segment are different. Each function is defined in its own compilation unit; a distinct file with a name the same as the function name.

The example consists of a single SPU `main` program. The `main` program calls the `SA` function which in turn calls the `SB` function. These three functions are all located in the root segment (segment 0) and cannot be overlaid.

The `SB` function calls the `SC` and `SG` functions. These are in two segments which are both located in region 1 and overlay each other.

`SC` calls `SD` and `SF`. `SD` in turn calls `SE`. The `SD` and `SE` functions are in segment 2 and the `SF` function is in segment 3. These two segments are both located in region 2 and overlay each other.

The SF and SG functions call the SH and SI functions. SI is in segment 6, and SH is in segment 5. These two segments are both located in region 3 and overlay each other.

The physical view of this example (Figure 8 on page 79) shows the four regions; one region containing a single non-overlay root segment and three regions containing six overlay segments.

The linker flags used are:

```
LDFLAGS = -Wl,-T,linker.script
```

The linker script is:

```
SECTIONS
{OVERLAY :
{
.segment1 {sc.o(.text)}
.segment4 {sg.o(.text) }
}
OVERLAY :
{
.segment2 {sd.o(.text) se.o(.text)}
.segment3 {sf.o(.text)}
}
OVERLAY :
{
.segment5 {sh.o(.text)}
.segment6 {si.o(.text)}
}
}
INSERT AFTER .text;
```

Large matrix overlay example

The large matrix overlay program consists of a single monolithic non-overlay SPU standalone program. This new example takes the existing program and converts it to an overlay program by providing a linker script. No changes (such as re-compilation) are made to the current library or to the test case code.

The updated example consists of a single standalone SPU program, large_matrix, which calls test functions test_index_max_abs_vec and test_solve_linear_system amongst others. These functions are defined in the single compilation unit large_matrix.c. A simplified structure is shown in Figure 14 on page 87 (some functions, and some calls within a region, have been omitted for clarity). If the test completes successfully the function returns a zero value; in other cases it returns a non-zero value.

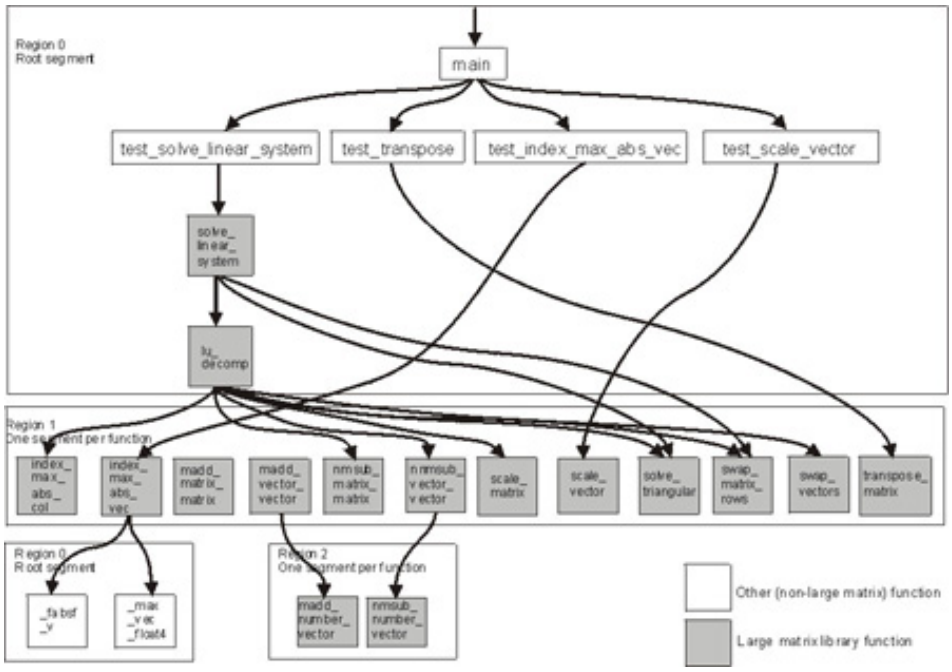


Figure 14. Large matrix overlay program call graph

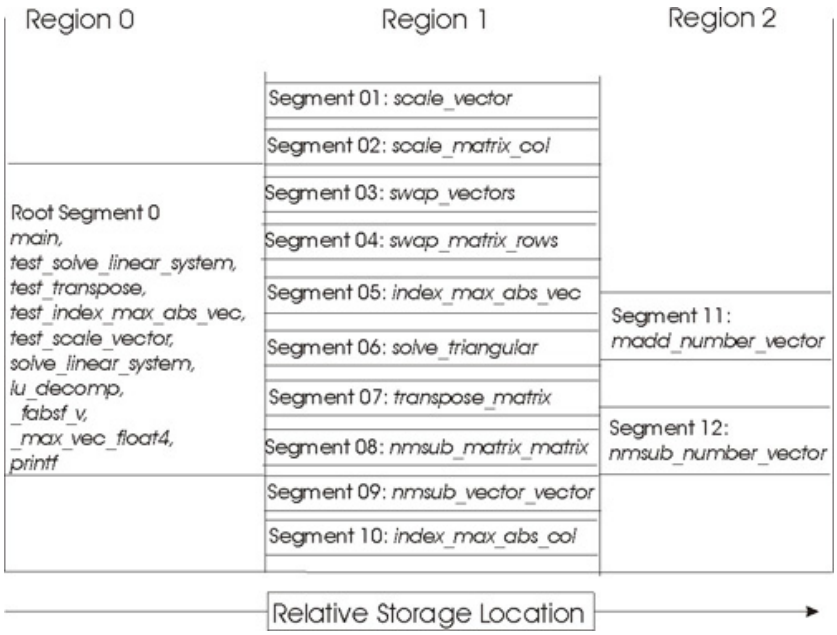


Figure 15. Large matrix program physical structure

The physical view of this example in Figure 15 shows three regions; one containing a single non-overlay root segment, and two containing twelve overlay segments.

This assumes the archive library directory, `/opt/cell/sdk/usr/lib`, and the archive library, `liblarge_matrix.a`, are specified to the SPU linker.

The linker flags used are:

```
LDFLAGS = -Wl,-T,linker.script
```

Note: this is a subset of all the functions in the `large_matrix` library. Only those needed by the test case driver, `large_matrix.c`, are used in this example.

The linker script is:

```
SECTIONS
{OVERLAY :
{
.segment01 {scale_vector.o*(.text)}
.segment02 {scale_matrix_col.o*(.text)}
.segment03 {swap_vectors.o*(.text)}
.segment04 {swap_matrix_rows.o*(.text)}
.segment05 {index_max_abs_vec.o*(.text)}
.segment06 {solve_triangular.o*(.text)}
.segment07 {transpose_matrix.o*(.text)}
.segment08 {nmsub_matrix_matrix.o*(.text)}
.segment09 {nmsub_vector_vector.o*(.text)}
.segment10 {index_max_abs_col.o*(.text)}
}
OVERLAY :
{
.segment11 {madd_number_vector.o*(.text)}
.segment12 {nmsub_number_vector.o*(.text)}
}
}INSERT AFTER .text;
```

Using the GNU SPU linker for overlays

The GNU SPU linker takes object files, object libraries, linker scripts, and command line options as its inputs and produces a fully or partially linked object file as its output. It is natural to control generation of overlays via a linker script as this allows maximum flexibility in specifying overlay regions and in mapping input files and functions to overlay segments. The linker has been enhanced so that one or more overlay regions may be created by simply inserting multiple `OVERLAY` statements in a standard script; no modification of the subsequent output section specifications, such as setting the load address, is necessary. (It is also possible to generate overlay regions without using `OVERLAY` statements by defining loadable output sections with overlapping virtual memory address (VMA) ranges.)

On detection of overlays the linker automatically generates the data structures used to manage them, and scans all non-debug relocations for calls to addresses which map to overlay segments. Any such call, apart from those used in branch instructions within the same section, causes the linker to generate an overlay call stub for that address and to remap the call to branch to that stub. At execution time these stubs call an overlay manager function which loads the overlay segment into storage, if necessary, before branching to the final destination.

If the linker command option: `–extra-overlay-stubs` is specified then the linker generates call stubs for all calls within an overlay segment, even if the target does not lie within an overlay segment (for example if it is in the root segment). Note that a non-branch instruction referencing a function symbol in the same section will also cause a stub to be generated; this ensures that function addresses which escape via pointers are always remapped to a stub as well.

The management data structures generated include two overlay tables in a `.ovtab` section. The first of these is a table with one entry per overlay segment. This table is read-write to the overlay manager. The low bit of size is set when a segment is loaded, and cleared when a segment is evicted. It has the format:

```
struct {
  u32 vma;    // SPU local store address that the section is loaded to.
  u32 size;   // Size of the overlay in bytes.
  u32 offset; // Offset in SPE executable where the section can be found.
  u32 buf;    // One-origin index into the _ovly_buf_table.
} _ovly_table[];
```

The second table has one entry per overlay region. This table is read-write to the overlay manager, and changes to reflect the current overlay mapping state. The format is:

```
struct {
  u32 mapped; // One-origin index into _ovly_table for the
              // currently loaded overlay. 0 if none.
} _ovly_buf_table[];
```

Note: These tables and the overlay manager itself must reside in the root (non-overlay) segment.

Whenever the overlay manager loads an segment into a region it updates the `mapped` field in the `_ovly_buf_table` entry for the region with the index of the segment entry in the `_ovly_buf` table.

The overlay manager may be provided by the user as a library containing the entries `__ovly_load` and `__ovly_return`. (It is an error for the user to provide `__ovly_return` without also providing `__ovly_load`.) If these entries are not provided the linker will use a built-in overlay manager.

Generating automatic overlay scripts

If given the *--auto-overlay* command option, the GNU SPU linker generates an overlay script automatically when a program does not fit in local store. The overlays so generated use a simple single region overlay buffer, but the linker does attempt to group input files and sections intelligently by considering the program call graph.

Better grouping is possible if you compile using the compiler's *-ffunction-sections* option. If you specify an output file for the generated script, for example, *--auto-overlay=link.script*, then the linker generates the script and then exits, unless you also specify *--auto-relink*, in which case the linker re-invokes itself using the generated script. If you do not specify an output file for the generated script, *--auto-relink* is assumed.

The linker allocates as large an overlay buffer as possible after allowing for data, stack and certain program text that must be in non-overlay memory, such as the overlay manager. Stack size is estimated (as for *--stack-analysis*) and a zero heap assumed. You can modify this behavior with a number of options:

--fixed-space=bytes
: This option can be used to increase the size for non-overlay code and data. If the value given exceeds the minimum needed, then the linker places functions that are called from many places (typically library functions), into this area.

--reserved-space=bytes
: This option specifies the size needed for stack and heap, overriding the linker's estimate. Do not forget to allocate space for access below the stack pointer. The current built-in overlay manager uses 64 bytes below sp. Other code may use up to 2000 bytes according to the SPU ABI.

--extra-stack-space=bytes
: This option specifies the space allocated for access below the stack pointer, to be added to the linker's stack estimate.

--overlay-rodata
: This option specifies that a function's read-only data should be placed along with the function's code in overlays. This is a dangerous option since this data may be accessed via pointers passed to other functions.

Appendix A. 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*

© Copyright IBM Corp. 2006, 2008

- *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 B. 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.

© Copyright IBM Corp. 2006, 2008

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.

© Copyright IBM Corp. 2006, 2008

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy,

modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A complete and current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc., in the United States, other countries, or both and is used under license therefrom.

Intel, MMX, and Pentium® are trademarks of Intel Corporation in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

UNIX® is a registered trademark of The Open Group in the United States and other countries.

Red Hat, the Red Hat “Shadow Man” logo, and all Red Hat-based trademarks and logos are trademarks or registered trademarks of Red Hat, Inc., in the United States and other countries.

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 the manufacturer.

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 the manufacturer.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any data, software or other intellectual property contained therein.

The manufacturer 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 the manufacturer, 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.

THE MANUFACTURER MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THESE 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 the Cell BE. 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 Bladecenter and Linux on cell.

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.

© Copyright IBM Corp. 2006, 2008

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

Special characters

Numerics

A

B

C

D

E

F

G

H

I

K

L

© Copyright IBM Corp. 2006, 2008

M

N

O

P

R

S

T

U

V

X

Printed in USA

SC33-8325-03