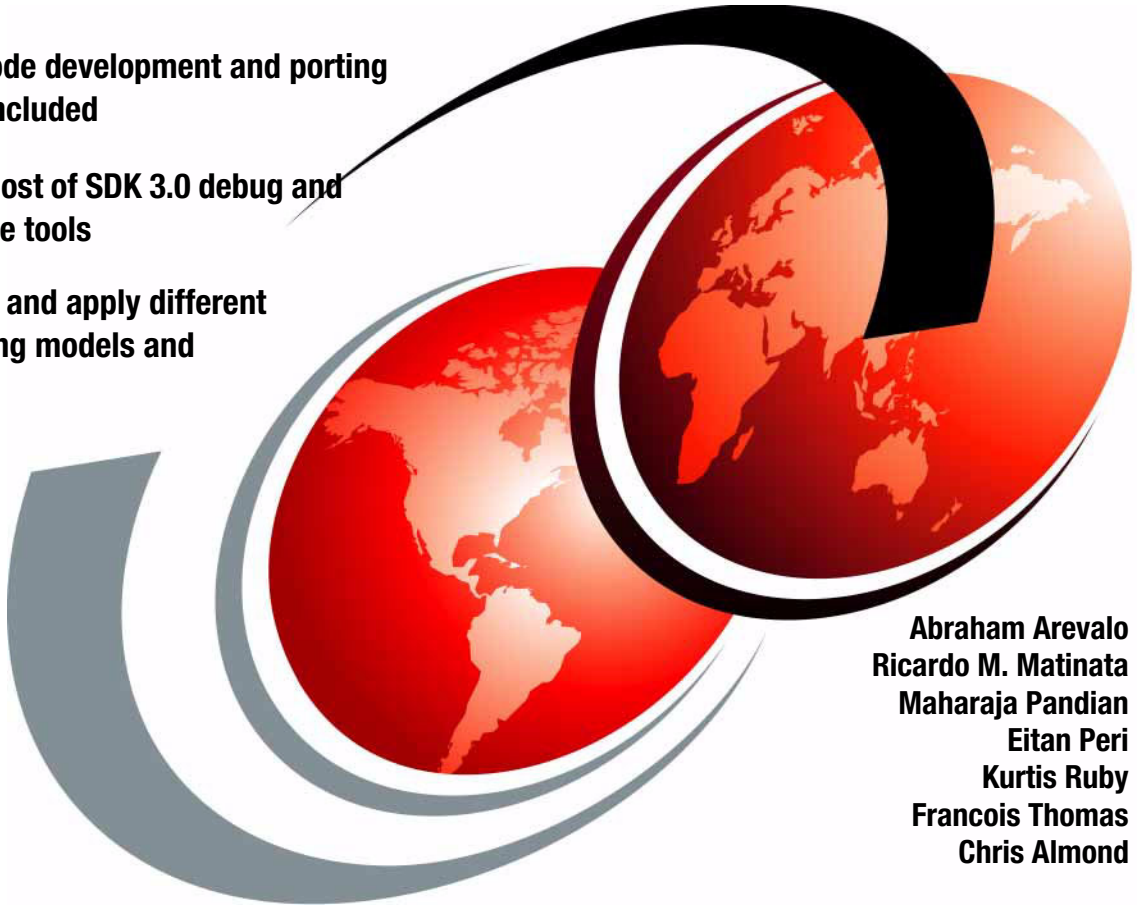
Draft Document for Review February 15, 2008 4:59 pm SG24-7575-00

Programming the Cell Broadband Engine

Examples and Best Practices

- Practical code development and porting examples included
- Make the most of SDK 3.0 debug and performance tools
- Understand and apply different programming models and strategies



Abraham Arevalo
Ricardo M. Matinata
Maharaja Pandian
Eitan Peri
Kurtis Ruby
Francois Thomas
Chris Almond

ibm.com/redbooks

Redbooks

International Technical Support Organization

Programming the Cell Broadband Engine: Examples and Best Practices

December 2007

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

First Edition (December 2007)

This edition applies to Version 3.0 of the IBM Cell Broadband Engine SDK, and the IBM BladeCenter QS-21 platform.

Contents

Preface

In this IBM® Redbooks publication we introduce, and show in detail, samples from real-world application development projects and provide tips and best practices for programming the Cell Broadband Engine™ applications. We provide an introduction to the Cell Broadband Engine platform and the describe the content and packaging of the IBM Software Development Kit (SDK) version 3.0 for Multicore Acceleration. This SDK provides all the tools and resources necessary to build applications that run IBM QS21 and QS20 Blade Servers.

There are chapters and sections in the Redbook that show in-depth and real-world useage of tools and resources found in the SDK.

In a chapter at the end of this book we provide some installation, configuration and administration tips and best practices for the IBM BladeCenter QS21. Discussion of supporting software provided by IBM Alphaworks is also provided.

This redbook was written for developers and programmers, customers, IBM Business Partners, and the IBM and Cell Broadband Engine community who need in depth technical understanding of how to develop applications using the Cell Broadband Engine SDK 3.0.

The team that wrote this book

This book was produced by a team of specialists from around the world working at the International Technical Support Organization, Austin Center.

Abraham Arevalo is a Software Engineer the Linux Technology Center's Test Department in Austin, Texas. He has worked on ensuring quality and functional performance of RHEL5.1 and Fedora 7 distributions on BladeCenter QS20 and QS21s. Additionally, Abraham has been involved on other Cell related projects including expanding Cell's presence on consumer electronics. He has prior experience working with hardware development mostly with System on Chip design.

Ricardo M. Matinata is an Advisory Software Engineer for the Linux Technology Center, in IBM Systems and Technology Group at IBM Brazil. He has over 10 years of experience in software research and development. He has been part of the global Cell BE SDK development team, in the area Toolchain (IDE), for almost two years. His areas of expertise include system software and application

development for both product and open source types of projects, Linux, programming models, development tools, debugging and networking.

Maharaja (Raj) Pandian is a High Performance Computing specialist working on scientific applications in the IBM WW Advanced Client Technology (A.C.T!) Center, Poughkeepsie, NY. He has twenty years of experience in high performance computing, software development, and market support. He holds a Ph.D. in Applied Mathematics from University of Texas, Arlington. His areas of expertise include parallel algorithms for distributed memory system and symmetric multiprocessor system, numerical methods for partial differential equations, performance optimization, and benchmarking. He has worked with engineering analysis ISV applications such as MSC/NASTRAN (Finite Element Analysis) and Fluent (Computational Fluid Dynamics) for several years. Also, he has worked with weather modeling applications on IBM AIX and Linux clusters. Currently, he is developing and benchmarking Financial Sector Services applications on the Cell BE.

Eitan Peri works in IBM Haifa Research Lab as the technical lead for Cell BE pre-sales activities in Israel. He holds a B.Sc. in Computer Engineering from Israel Institute of Technology (the Technion), and M.Sc. in Biomedical Engineering from Tel-Aviv University, where he specialized in brain imaging analysis. He has 9 years of experience in real time programming, chip design and chip verification. His areas of expertise include Cell BE programming and consulting, application parallelization and optimization, algorithm performance analysis, and medical imaging. He is currently working on projects focusing on porting applications to the Cell BE architecture within the health care, computer graphics, aerospace and defense industries.

Kurtis Ruby is a software consultant with IBM Lab Services at IBM Rochester, Minnesota. He has over twenty-five years of experience in various programming assignments in IBM. He holds a degree in Mathematics from Iowa State University. His expertise includes Cell Broadband Engine programming annd consulting.

Francois Thomas is an IBM Certified IT Specialist working on HPC pre-sales in the Deep Computing Europe organization in France. He has 18 years of experience in the field of scientific and technical computing. He holds a PhD in Physics from ENSAM/Paris VI University. His areas of expertise include application code tuning and parallelization as well as Linux clusters management. He works with weather forecast institutions in Europe and on enabling petroleum engineering ISV applications to the Linux on POWER platform.

Production of this IBM Redbook was managed by:

Chris Almond, an ITSO Project Leader and IT Architect based at the ITSO Center in Austin, Texas. In his current role, he specializes in managing content development projects focused on Linux®, AIX 5L systems engineering, and other innovation programs. He has a total of 17 years of IT industry experience, including the last eight with IBM.

Acknowledgements

This IBM Redbooks publication would not have been possible without the generous support and contributions provided many IBMers.

First, the authoring team would like to gratefully acknowledge the critical support and sponsorship for this project provided by the following IBMers:

- **Tanaz Sowdagar**, Marketing Manager, Systems and Technology Group
- **Jeffrey Scheel**, Blue Gene Software Program Manager and Software Architect, Systems and Technology Group
- **Daniel Brokenshire**, Senior Technical Staff Member and Software Architect, Quasar/Cell BE Software Development, Systems and Technology Group
- **Paula Richards**, Director, Global Engineering Solutions, Systems and Technology Group
- **Rebecca Austen**, Director, Systems Software, Systems and Technology Group

We would also like to thank the following IBMers for their significant input to this project during the development and technical review process:

- **Marina Biberstein**, Research Scientist, Haifa Reseach Lab, IBM Research
- **Michael Brutman**, Solutions Architect, Lab Services, IBM Systems and Technology Group
- **Dean Burdick**, Developer, Cell Software Development, IBM Systems and Technology Group
- **Catherine Crawford**, Senior Technical Staff Member and Chief Architect, Next Generation Systems Software, IBM Systems and Technology Group
- **Bruce D'Amora**, Research Scientist, Systems Engineering, IBM Research
- **Matthew Drahzal**, Software Architect, Deep Computing, IBM Systems and Technology Group
- **Matthias Fritsch**, Enterprise System Development, IBM Systems and Technology Group
- **Gad Haber**, Manager, Performance Analysis and Optimization Technology, Haifa Reseach Lab, IBM Research

- **Francesco Iorio**, Solutions Architect, Next Generation Computing, IBM Software Group
- **Kirk Jordan**, Solutions Executive, Deep Computing and Emerging HPC Technologies, IBM Systems and Technology Group
- **Melvin Kendrick**, Manager, Cell Ecosystem Technical Enablement, IBM Systems and Technology Group
- **Mark Mendell**, Team Lead, Cell BE Compiler Development, IBM Software Group
- **Michael P. Perrone**, Ph.D., Manager Cell Solutions, IBM Systems and Technology Group
- **Juan Jose Porta**, Executive Architect HPC & e-Science Platforms, IBM Systems and Technology Group
- **Uzi Shvadron**, Research Scientist, Cell BE Performance Tools, Haifa Reseach Lab, IBM Research
- **Van To**, Advisory Software Engineer, Cell BE & Next Generation Computing Systems, IBM Systems and Technology Group
- **Duc J Vianney**, Ph. D, Technical Education Lead, Cell BE Ecosystem & Solutions Enablement, IBM Systems and Technology Group
- **Brian Watt**, Systems Development, Quasar Design Center Development, IBM Systems and Technology Group
- **Ulrich Weigand**, Developer, Linux on Cell BE, IBM Systems and Technology Group
- **Cornell Wright**, Developer, Cell Software Development, IBM Systems and Technology Group

Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks® in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Notices

This information was developed for products and services offered in the U.S.A.

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

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

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

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

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

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

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.

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.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

Redbooks (logo)  ®	AIX®	PowerPC®
^®	BladeCenter®	POWER™
eServer™	IBM®	Redbooks®
xSeries®	PowerPC Architecture™	System x™

The following terms are trademarks of other companies:

Cell Broadband Engine and Cell/B.E. are trademarks of Sony Computer Entertainment, Inc., in the United States, other countries, or both and is used under license therefrom.

Snapshot, and the Network Appliance logo are trademarks or registered trademarks of Network Appliance, Inc. in the U.S. and other countries.

AMD, AMD Opteron, the AMD Arrow logo, and combinations thereof, are trademarks of Advanced Micro Devices, Inc.

Flex, and Portable Document Format (PDF) are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Java, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Excel, Fluent, Microsoft, Visual Basic, Visual C++, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel Centrino, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.

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

MultiCore Plus is a trademark of Mercury Computer Systems, Inc. in the United States, other countries, or both.

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

Part 1

Introduction to the Cell Broadband Engine

The Cell Broadband Engine (CELL BE) is a new class of mult-core processors being brought to the consumer and business market. It has a radically different design than those offered by other consumer and business chip makers in the global market. This radical departure warrants a brief discussion of the CELL BE hardware and software architecture.

There is also a brief discusson of the IBM Software Development Kit (SDK) for Multicore Acceleration from a content and packaging perspective. These discussions complement the in-depth content of the remaining chapters of this Redbook.

1

Cell Broadband Engine Overview

The Cell BE processor is the first implementation of a new multiprocessor family conforming to the Cell Broadband Engine Architecture (CBEA). The CBEA and the Cell BE processor are the result of a collaboration between Sony, Toshiba, and IBM known as STI, formally begun in early 2001. Although the Cell BE processor is initially intended for applications in media-rich consumer-electronics devices such as game consoles and high-definition televisions, the architecture has been designed to enable fundamental advances in processor performance. These advances are expected to support a broad range of applications in both commercial and scientific fields.

Figure 1-1 on page 4 shows a block diagram of the Cell BE processor hardware.

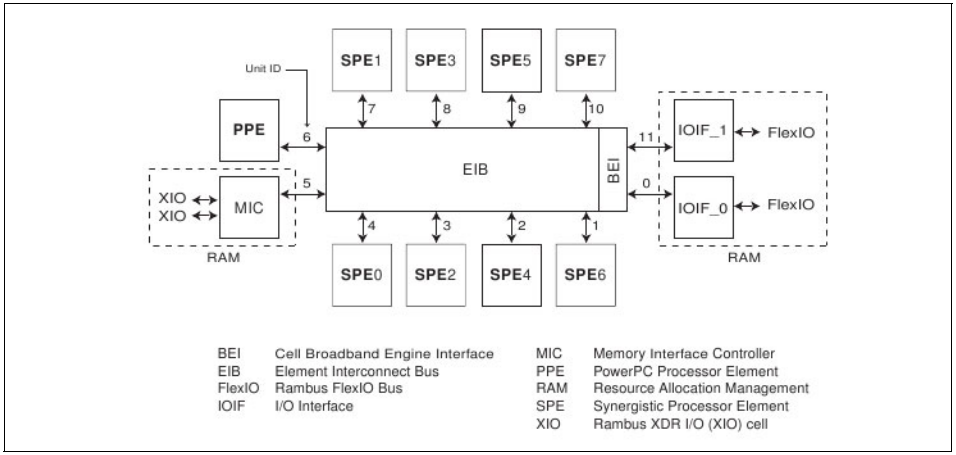


Figure 1-1 Cell Broadband Engine Overview

1.1 Motivation

The Cell Broadband Engine Architecture has been designed to support a very broad range ofapplications. The first implementation is a single-chip multiprocessor with nine processor elements operating on a shared memory model, as shown in Figure 1-1 on page 4. In this respect, the Cell BE processor extends current trends in PC and server processors. The most distinguishing feature of the Cell BE processor is that, although all processor elements can share or access all available memory, their function is specialized into two types: the Power Processor Element (PPE) and the Synergistic Processor Element (SPE). The Cell BE processor has one PPE and eight SPEs.

The first type of processor element, the PPE, contains a 64-bit PowerPC® Architecture™ core. It complies with the 64-bit PowerPC Architecture and can run 32-bit and 64-bit operating systems and applications. The second type of processor element, the SPE, is optimized for running compute-intensive SIMD applications; it is not optimized for running an operating system.

The SPEs are independent processor elements, each running their own individual application programs or threads. Each SPE has full access to shared memory, including the memory-mapped I/O space implemented by multiple DMA units. There is a mutual dependence between the PPE and the SPEs. The SPEs depend on the PPE to run the operating system, and, in many cases, the top-level thread control for an application. The PPE depends on the SPEs to provide the bulk of the application performance.

The SPEs are designed to be programmed in high-level languages, such as (but certainly not limited to) C/C++. They support a rich instruction set that includes extensive SIMD functionality. However, like conventional processors with SIMD extensions, use of SIMD data types is preferred, not mandatory. For programming convenience, the PPE also supports the standard PowerPC Architecture instructions and the vector/SIMD multimedia extensions.

To an application programmer, the Cell BE processor looks like a single core, dual threaded processor with 8 additional cores each having their own local store. The PPE is more adept than the SPEs at control-intensive tasks and quicker at task switching. The SPEs are more adept at compute-intensive tasks and slower than the PPE at task switching. However, either processor element is capable of both types of functions. This specialization is a significant factor accounting for the order-of-magnitude improvement in peak computational performance and chip-area-and-power efficiency that the Cell BE processor achieves over conventional PC processors.

The more significant difference between the SPE and PPE lies in how they access memory. The PPE accesses main storage (the effective-address space) with load and store instructions that move data between main storage and a private register file, the contents of which may be cached. PPE memory access is like that of a convential processor technology, which is found on convential machines. The SPEs, in contrast, access main storage with direct memory access (DMA) commands that move data and instructions between main storage and a private local memory, called a local store or local storage (LS). An SPE's instruction-fetches and load and store instructions access its private LS rather than shared main storage, and the LS has no associated cache. This 3-level organization of storage (register file, LS, main storage), with asynchronous DMA transfers between LS and main storage, is a radical break from conventional architecture and programming models, because it explicitly parallelizes computation with the transfers of data and instructions that feed computation and store the results of computation in main storage.

One of the motivations for this radical change is that memory latency, measured in processor cycles, has gone up several hundredfold from about the years 1980 to 2000. The result is that application performance is, in most cases, limited by memory latency rather than peak compute capability or peak bandwidth. When a sequential program on a conventional architecture performs a load instruction that misses in the caches, program execution can come to a halt for several hundred cycles (techniques such as hardware threading can attempt to hide these stalls, but it does not help single threaded applications). Compared to this penalty, the few cycles it takes to set up a DMA transfer for an SPE are a much better trade-off, especially considering the fact that each of the eight SPE's DMA controller can have up to 16 DMA transfer in flight simultaneously. Anticpating DMA needs efficently can provide "just in time delivery" of data which many

reduce this stall or eliminate them entirely. Conventional processors, even with deep and costly speculation, manage to get, at best, a handful of independent memory accesses in flight.

One of the SPE's DMA transfer methods supports a list (such as a scatter-gather list) of DMA transfers that is constructed in an SPE's local store, so that the SPE's DMA controller can process the list asynchronously while the SPE operates on previously transferred data. In several cases, this approach to accessing memory has led to application performance exceeding that of conventional processors by almost two orders of magnitude significantly more than one would expect from the peak performance ratio (approximately 10x) between the Cell BE processor and conventional PC processors. The DMA transfers can be set up and controlled by the SPE that is sourcing or receiving the data, or in some circumstances by the PPE or another SPE.

1.2 Scaling the three performance-limiting walls

The Cell Broadband Engine overcomes three important limiters of contemporary microprocessor performance: power use, memory use, and processor frequency.

1.2.1 Scaling the power-limitation wall

Increasingly, microprocessor performance is limited by achievable power dissipation rather than by the number of available integrated-circuit resources (transistors and wires).

Therefore, the only way to significantly increase the performance of microprocessors is to improve power efficiency at about the same rate as the performance increase.

One way to increase power efficiency is to differentiate between:

- processors optimized to run an operating system and control-intensive code, and
- processors optimized to run compute-intensive applications.

The Cell Broadband Engine does this by providing a general-purpose PPE to run the operating system and other control-plane code, and eight SPEs specialized for computing data-rich (data-plane) applications. The specialized SPEs are more compute efficient because they have simpler hardware implementations. The hardware does not devote transistors to branch prediction, out of order execution, speculative execution, shadow registers and register renaming,

extensive pipeline interlocks, etc. By weight, more of the transistors are used for computation than in conventional processor cores.

1.2.2 Scaling the memory-limitation wall

On multi-gigahertz symmetric multiprocessors (even those with integrated memory controllers) latency to DRAM memory is currently approaching 1,000 cycles.

As a result, program performance is dominated by the activity of moving data between main storage (the effective-address space that includes main memory) and the processor. Increasingly, compilers and even application writers must manage this movement of data explicitly, even though the hardware cache mechanisms are supposed to relieve them of this task.

The Cell Broadband Engine's SPEs use two mechanisms to deal with long main-memory latencies:

- 3-level memory structure (main storage, local stores in each SPE, and large register files in each SPE),
- asynchronous DMA transfers between main storage and local stores.

These features allow programmers to schedule simultaneous data and code transfers to cover long latencies effectively. Because of this organization, the Cell Broadband Engine can usefully support 128 simultaneous transfers between the eight SPE local stores and main storage. This surpasses the number of simultaneous transfers on conventional processors by a factor of almost twenty.

1.2.3 Scaling the frequency-limitation wall

Conventional processors require increasingly deeper instruction pipelines to achieve higher operating frequencies. This technique has reached a point of diminishing returns – and even negative returns if power is taken into account.

By specializing the PPE and the SPEs for control and compute-intensive tasks, respectively, the Cell Broadband Engine Architecture, on which the Cell Broadband Engine is based, allows both the PPE and the SPEs to be designed for high frequency without excessive overhead. The PPE achieves efficiency primarily by executing two threads simultaneously rather than by optimizing single-thread performance.

Each SPE achieves efficiency by using a large register file, which supports many simultaneous in-process instructions without the overhead of register-renaming or out-of-order processing. Each SPE also achieves efficiency by using

asynchronous DMA transfers, which support many concurrent memory operations without the overhead of speculation.

1.2.4 How the Cell Broadband Engine overcomes performance limitations

By optimizing control-plane and data-plane processors individually, the Cell Broadband Engine alleviates the problems posed by the power, memory, and frequency limitations.

The net result is a processor that, at the power budget of a conventional PC processor, can provide approximately ten-fold the peak performance of a conventional processor. Of course, actual application performance varies. Some applications may benefit little from the SPEs, whereas others show a performance increase well in excess of ten-fold. In general, compute-intensive applications that use 32-bit or smaller data formats (such as single-precision floating-point and integer) are excellent candidates for the Cell Broadband Engine.

1.3 Hardware Environment

In the following sections we describe the different components in the Cell BE hardware environment.

1.3.1 The Processor Elements

Figure 1-1 on page 4 shows a high-level block diagram of the Cell BE processor hardware. There is one PPE and there are eight identical SPEs. All processor elements are connected to each other and to the on-chip memory and I/O controllers by the memory-coherent element interconnect bus (EIB).

The PPE contains a 64-bit, dual-thread PowerPC Architecture RISC core and supports a PowerPC virtual-memory subsystem. It has 32 KB level-1 (L1) instruction and data caches and a 512 KB level-2 (L2) unified (instruction and data) cache. It is intended primarily for control processing, running operating systems, managing system resources, and managing SPE threads. It can run existing PowerPC Architecture software and is well-suited to executing system-control code. The instruction set for the PPE is an extended version of the PowerPC instruction set. It includes the vector/SIMD multimedia extensions and associated C/C++ intrinsic extensions.

The eight identical SPEs are single-instruction, multiple-data (SIMD) processor elements that are optimized for data-rich operations allocated to them by the PPE. Each SPE contains a RISC core, 256 KB software-controlled LS for instructions and data, and a 128-bit, 128-entry unified register file. The SPEs support a special SIMD instruction set the Synergistic Processor Unit Instruction Set Architecture and a unique set of commands for managing DMA transfers and interprocessor messaging and control. SPE DMA transfers access main storage using PowerPC effective addresses. As in the PPE, SPE address translation is governed by PowerPC Architecture segment and page tables, which are loaded into the SPEs by privileged software running on the PPE. The SPEs are not intended to run an operating system.

An SPE controls DMA transfers and communicates with the system by means of channels that are implemented in and managed by the SPE's memory flow controller (MFC). The channels are unidirectional message-passing interfaces. The PPE and other devices in the system, including other SPEs, can also access this MFC state through the MFC's memory-mapped I/O (MMIO) registers and queues, which are visible to software in the main-storage address space.

1.3.2 The Element Interconnect Bus

The element interconnect bus (EIB) is the communication path for commands and data between all processor elements on the Cell BE processor and the on-chip controllers for memory and I/O. The EIB supports full memory-coherent and symmetric multiprocessor (SMP) operations. Thus, a Cell BE processor is designed to be ganged coherently with other Cell BE processors to produce a cluster.

The EIB consists of four 16-byte-wide data rings. Each ring transfers 128 bytes (one PPE cache line) at a time. Each processor element has one on-ramp and one off-ramp. Processor elements can drive and receive data simultaneously. Figure 1-1 on page 4 shows the unit ID numbers of each element and the order in which the elements are connected to the EIB. The connection order is important to programmers seeking to minimize the latency of transfers on the EIB: latency is a function of the number of connection hops, such that transfers between adjacent elements have the shortest latencies and transfers between elements separated by six hops have the longest latencies.

The EIB's internal maximum bandwidth is 96 bytes per processor-clock cycle. Multiple transfers can be in-process concurrently on each ring, including more than 100 outstanding DMA memory transfer requests between main storage and the SPEs in either direction. This requests also may include SPE memory to and from the I/O space. The EIB does not support any particular qualityof-service (QoS) behavior other than to guarantee forward progress. However, a resource allocation management (RAM) facility, shown in Figure 1-1 on page 4, resides in

the EIB. Privileged software can use it to regulate the rate at which resource requesters (the PPE, SPEs, and I/O devices) can use memory and I/O resources.

1.3.3 Memory Interface Controller

The on-chip memory interface controller (MIC) provides the interface between the EIB and physical memory. The IBM Bladecenter QS20 supports one or two Rambus extreme data rate (XDR) memory interfaces, which together support between 64 MB and 64 GB of XDR DRAM memory. The IBM Bladecenter QS21 uses normal DDR memory and additional hardware logic to implement the MIC.

Memory accesses on each interface are 1 to 8, 16, 32, 64, or 128 bytes, with coherent memoryordering. Up to 64 reads and 64 writes can be queued. The resource-allocation token manager provides feedback about queue levels.

The MIC has multiple software-controlled modes, including fast-path mode (for improved latency when command queues are empty), high-priority read (for prioritizing SPE reads in front of allother reads), early read (for starting a read before a previous write completes), speculative read, and slow mode (for power management). The MIC implements a closed-page controller (bank rows are closed after being read, written, or refreshed), memory initialization, and memory scrubbing.

The XDR DRAM memory is ECC-protected, with multi-bit error detection and optional single-bit error correction. It also supports write-masking, initial and periodic timing calibration. It also supports write-masking, initial and periodic timing calibration, dynamic width control, sub-page activation, dynamic clock gating, and 4, 8, or 16 banks.

1.3.4 Cell Broadband Engine Interface Unit

The on-chip Cell Broadband Engine interface (BEI) unit supports I/O interfacing. It includes a bus interface controller (BIC), I/O controller (IOC), and internal interrupt controller (IIC), as defined in the Cell Broadband Engine Architecture document. It manages data transfers between the EIB and I/O devices and provides I/O address translation and command processing.

The BEI supports two Rambus FlexIO interfaces. One of the two interfaces (IOIF1) supports only a noncoherent I/O interface (IOIF) protocol, which is suitable for I/O devices. The other interface (IOIF0, also called BIF/IOIF0) is software-selectable between the noncoherent IOIF protocol and the memory-coherent Cell Broadband Engine interface (BIF) protocol. The BIF protocol is the EIB's internal protocol. It can be used to coherently extend the

EIB, through IOIF0, to another memory-coherent device, that can be another Cell BE processor.

1.4 Programming Environment

In the following sections we provide an overview of the programming environment.

1.4.1 Instruction Sets

The instruction set for the PPE is an extended version of the PowerPC Architecture instruction set. The extensions consist of the vector/SIMD multimedia extensions, a few additions and changes to PowerPC Architecture instructions, and C/C++ intrinsics for the vector/SIMD multimedia extensions.

The instruction set for the SPEs is a new SIMD instruction set, the Synergistic Processor Unit Instruction Set Architecture, with accompanying C/C++ intrinsics, and a unique set of commands for managing DMA transfer, external events, interprocessor messaging, and other functions. The instruction set for the SPEs is similar to that of the PPE's vector/SIMD multimedia extensions, in the sense that they operate on SIMD vectors. However, the two vector instruction sets are distinct, and programs for the PPE and SPEs are often compiled by different compilers generating code streams for two entirely different instruction sets.

Although most coding for the Cell BE processor will probably be done in a high-level language like C or C++, an understanding of the PPE and SPE machine instructions adds considerably to a software developer's ability to produce efficient, optimized code. This is particularly true because most of the C/C++ intrinsics have a mnemonic that relates directly to the underlying assemblylanguage mnemonic.

1.4.2 Storage Domains and Interfaces

The Cell BE processor has two types of storage domains one main-storage domain and eight SPE local-storage (LS) domains, as shown in Figure 1-2 on page 13. In addition, each SPE has a channel interface for communication between its synergistic processor unit (SPU) and its MFC. The main-storage domain, which is the entire effective-address space, can be configured by PPE privileged software to be shared by all processor elements and memory-mapped devices in the system1. An MFC's state is accessed by its associated SPU through the channel interface, and this state can also be accessed by the PPE and other devices (including other SPEs) by means of the MFC's MMIO registers in the main-storage space. An SPU's LS can also be accessed by the PPE and other devices through the main-storage space in a non-coherent manner. The PPE accesses the mainstorage space through its PowerPC processor storage subsystem (PPSS).

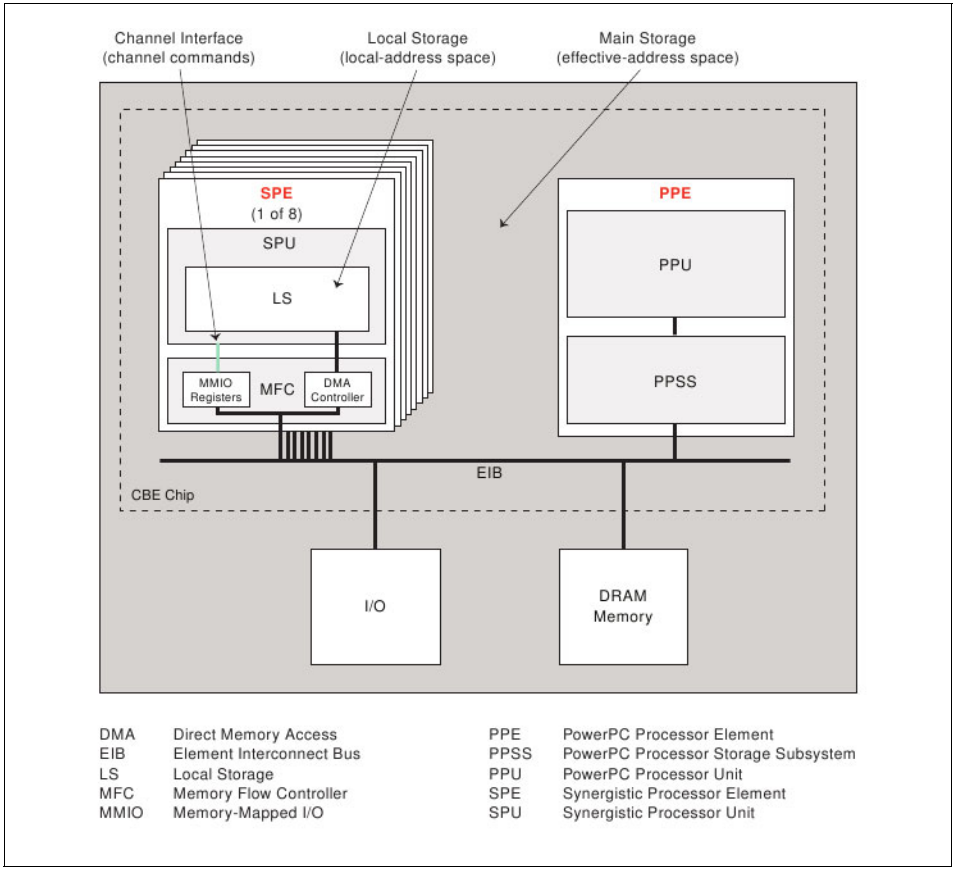


Figure 1-2 Storage and Domain Interfaces

The address-translation mechanisms used in the main-storage domain are described in Section 4 Virtual Storage Environment on page 77. The channel domain is described in Section 19 DMA Transfers and Interprocessor Communication on page 507. An SPE's SPU can fetch instructions only from its own LS, and load and store instructions executed by the SPU can only access the LS. SPU software uses LS addresses (not main storage effective addresses) to do this. Each SPE's MFC contains a DMA controller. DMA transfer requests contain both an LS address and an effective address, thereby facilitating transfers between the domains.

Data transfer between an SPE Local Store and Main Storage is performed by the Memory Flow Controller that is local to the SPE. Software running on an SPE

sends commands to its MFC using the private channel interface. The MFC can also be manipulated by remote SPEs, the PPE, or IO devices using memory mapped IO. Software running on the associated SPE interacts with its own MFC through its channel interface. The channels support enqueueing of DMA commands and other facilities, such as mailbox and signal-notification messages. Software running on the PPE or another SPE can interact with an MFC through MMIO registers, which are associated with the channels and visible in the mainstorage space.

Each MFC maintains and processes two independent command queues for DMA and other commands one queue for its associated SPU, and another queue for other devices accessing the SPE through the main-storage space. Each MFC can process multiple in-progress DMA commands. Each MFC can also autonomously manage a sequence of DMA transfers in response to a DMA list command from its associated SPU (but not from the PPE or other SPEs). Each DMA command is tagged with a tag group ID that allows software to check or wait on the completion of commands in a particular tag group.

The MFCs support naturally aligned DMA transfer sizes of 1, 2, 4, or 8 bytes, and multiples of 16 bytes, with a maximum transfer size of 16 KB per DMA transfer. DMA list commands can initiate up to 2048 such DMA transfers. Peak transfer performance is achieved if both the effective addresses and the LS addresses are 128-byte aligned and the size of the transfer is an even multiple of 128 bytes.

Each MFC has a synergistic memory management (SMM) unit that processes address-translation and access-permission information supplied by the PPE operating system. To process an effective address provided by a DMA command, the SMM uses essentially the same addresstranslation and protection mechanism used by the memory management unit (MMU) in the PPE's PowerPC processor storage subsystem (PPSS)2. Thus, DMA transfers are coherent with respect to system storage, and the attributes of system storage are governed by the page and segment tables of the PowerPC Architecture.

1.4.3 Bit Ordering and Numbering

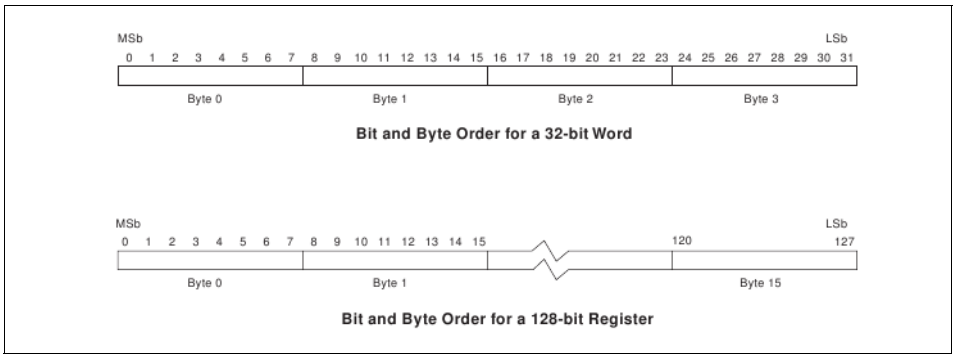


Figure 1-3 Big-Endian Byte and Bit Ordering

Storage of data and instructions in the Cell BE processor uses big-endian ordering, which has the following characteristics:

- Most-significant byte stored at the lowest address, and least-significant byte stored at the highest address.
- Bit numbering within a byte goes from most-significant bit (bit 0) to least-significant bit (bit n).

This differs from some other big-endian processors.

A summary of the byte-ordering and bit-ordering in memory and the bit-numbering conventions is shown in Figure 1-3 on page 15.

Neither the PPE nor the SPEs, including their MFCs, support little-endian byte-ordering. The MFC's DMA transfers are simply byte moves, without regard to the numeric significance of any byte. Thus, the big-endian or little-endian issue becomes irrelevant to the actual movement of a block of data. The byte-order mapping only becomes significant when data is loaded or interpreted by a processor element or an MFC.

1.4.4 Runtime Environment

The PPE runs PowerPC Architecture applications and operating systems, which can include both PowerPC Architecture instructions and vector/SIMD multimedia extension instructions. To use all of the Cell BE processor's features, the PPE requires an operating system that supports these features, such as multiprocessing with the SPEs, access to the PPE vector/SIMD multimedia extension operations, the Cell BE interrupt controller, and all the other functions provided by the Cell BE processor.

The PPE runs PowerPC Architecture applications and operating systems, which can include both PowerPC Architecture instructions and vector/SIMD multimedia extension instructions. To use all of the Cell BE processor's features, the PPE requires an operating system that supports these features, such as multiprocessing with the SPEs, access to the PPE vector/SIMD multimedia extension operations, the Cell BE interrupt controller, and all the other functions provided by the Cell BE processor.

A main thread running on the PPE can interact directly with an SPE thread through the SPE's LS. It can interact indirectly through the main-storage space. A thread can poll or sleep, waiting for SPE threads. The PPE thread can also communicate through mailbox and signal events implemented in the hardware.

The operating system defines the mechanism and policy for selecting an available SPE to schedule an SPU thread to run on. It must prioritize among all the Cell BE applications in the system, and it must schedule SPE execution independently from regular main threads. The operating system is also responsible for runtime loading, passing parameters to SPE programs, notification of SPE events and errors, and debugger support.

2

IBM SDK for Multicore Acceleration

This chapter provides a description of the software tools and libraries that are found in the Cell Broadband Engine SDK. This chapter includes a brief discussion of the following topics

- Compilers
- IBM Full System Simulator
- CELL BE Libraries
- Code examples and example libraries
- Performance tools
- IBM Eclipse IDE for the SDK
- Hybrid-x86 programming model

2.1 Compilers

There are a number of IBM supplied compilers as part of the IBM SDK for Multicore Acceleration. This section breifly describes the IBM product compilers and open source compilers in the SDK.

2.1.1 GNU Toolchain

The GNU toolchain, including compilers, the assembler, the linker, and miscellaneous tools, is available for both the PPU and SPU instruction set architectures. On the PPU it replaces the native GNU toolchain (which is generic for PowerPC architectures) with a version that is tuned for the Cell PPU processor core. The GNU compilers are the default compilers for the SDK.

The GNU toolchains run natively on Cell BE hardware, or as cross compilers on PowerPC or x86 machines.

2.1.2 IBM XLC/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 an IBM BladeCenter QS21, 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.

Note: The IBM XLC/C++ compiler that comes with SDK 3 is an OpenMP directed single source compiler that supports automatic program partitioning, data virtualization, code overlay, and more. This version of the compiler is in beta mode and users should not base production applications on this compiler.

2.1.3 GNU ADA Compiler

The GNU toolchain also contains an implementation of the GNU ADA compiler. This compiler comes in a navitve Cell BE and an x86 cross-compiler. This initial version of this compiler supports code generation for the PPU processor.

2.1.4 IBM XL Fortran for Multicore Acceleration for Linux

IBM XL Fortran for Multicore Acceleration for Linux is the latest addition to the IBM XL family of compilers. It adopts proven high-performance compiler technologies used in its compiler family predecessors, and adds new features tailored to exploit the unique performance capabilities of processors compliant with the new Cell Broadband Engine architecture. This version of XL Fortran is a cross-compiler. First, you compile your applications on an IBM System p compilation host running Red Hat Enterprise Linux 5.1 (RHEL 5.1). Then you move the executable application produced by the compiler onto a Cell BE system

also running the RHEL 5.1 Linux distribution. The Cell B.E. system will be the execution host where you will actually run your compiled application.

2.2 IBM Full System Simulator

The IBM Full-System Simulator (referred to as the simulator in this document) is a software application that emulates the behavior of a full system that contains a Cell BE processor. You can start a Linux operating system on the simulator and simulating a two chip Cell/B.E. environement and run applications on the simulated operating system. The simulator also supports the loading and running of statically-linked executable programs and standalone tests without an underlying operating system. There are other functions like debug output not available on hardware.

The simulator infrastructure is designed for modeling processor and system-level architecture at levels of abstraction, which vary from functional to performance simulation models with a number of hybrid fidelity points in between:

Functional-only simulation: Models the program-visible effects of instructions without modeling the time it takes to run these instructions. Functional-only simulation assumes that each instruction can be run in a constant number of cycles. Memory accesses are synchronous and are also performed in a constant number of cycles.

This simulation model is useful for software development and debugging when a precise measure of execution time is not significant. Functional simulation proceeds much more rapidly than performance simulation, and so is also useful for fast-forwarding to a specific point of interest.

Performance simulation: For system and application performance analysis, the simulator provides performance simulation (also referred to as timing simulation). A performance simulation model represents internal policies and mechanisms for system components, such as arbiters, queues, and pipelines.

Operation latencies are modeled dynamically to account for both processing time and resource constraints. Performance simulation models have been correlated against hardware or other references to acceptable levels of tolerance.

The simulator for the Cell BE processor provides a cycle-accurate SPU core model that can be used for performance analysis of computationally-intense applications

2.2.1 System root image for Simulator

The system root image for the simulator is a file that contains a disk image of Fedora files, libraries, and binaries that can be used within the system simulator. This disk image file is preloaded with a full range of Fedora utilities and also includes all of the Cell BE Linux support libraries.

2.3 Linux Kernel

For the IBM BladeCenter QS21, 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.

2.4 Cell BE Libraries

In the following sections we describe various programming libraries.

2.4.1 SPE Runtime Management Library

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

2.4.2 SIMD Math Library

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 BE 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.

2.4.3 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 BE processor. Currently the 32-bit, 64-bit PPU, and SPU libraries are supported.

These libraries:

- Include both scalar and vector functions v Are thread-safe v 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

2.4.4 Basic Linear Algebra Subprograms (BLAS)

The BLAS (Basic Linear Algebra Subprograms) library is based upon a published standard interface 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 API is available as standard ANSI C and standard FORTRAN 77/90 interfaces. BLAS implementations are also available in open-source (netlib.org).

The BLAS library in the IBM SDK for Multicore Acceleration supports only real single precision and real double precision versions (hereafter referred to as SP and DP respectively). 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.

Some of theses routines have been optimized using the Synergistic Processing Elements (SPEs) and these show a marked increase in performance in comparison to the corresponding versions implemented solely on the PPE. These optimized routines have an SPE interface in addition to the PPE interface; however, the SPE interface does not conform to the standard BLAS interface and provides a restricted version of the standard BLAS interface. The following routines have been optimized to use the SPEs; moreover, the single precision versions of these routines have been further optimized for maximum performance using various features of the SPE.

2.4.5 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 v Dynamic load balancing

2.4.6 Data Communication and Synchronization library (DaCS)

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 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.

2.5 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 BE programs.

To demonstrate the versatility of the Cell BE 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.

2.6 Performance Tools

The Cell/B.E. SDK supports many of the traditional Linux based performance tools available. The performance tools (such as gprof) pertain specifically to the PPE execution environement and do not support the SPE environment. The following tools are special tools that support the PPE and/or SPE environment.

2.6.1 SPU Timing Tool

The SPU static timing tool, spu_timing, annotates an SPU assembly file with scheduling, timing, and instruction issue estimates assuming a straight, linear execution of the program which is useful for analyzing basic code blocks. The tool generates a textual output of the execution pipeline of the SPE instruction stream from this input assembly file. The output generated can show pipeline stalls, which can be explained by looking at the subsequent instructions. Data dependencies are pipeline hazards can be readily identified using this tool. Lastly, it should be noted that this is a static analysis tool. It does not idenitify branch behavior or memory transfer delays.

2.6.2 OProfile

OProfile is a Linux tool that exists on other architectures besides the Cell/B.E., and that it has been extended to support the unique hardware on the PPU and SPUs. It is a sampling based tool that does not require special source compile flags to produce useful data reports.

The opreport tool produces the output report. Reports can be generated based on the file names that correspond to the samples, symbol names or annotated source code listings (special source compiler flags are required in this case).

2.6.3 Cell-perf-counter tool

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

2.6.4 Performance Debug Tool (PDT)

The Cell BE PDT is to provide programmers with a means of analyzing the execution of such a system and tracking problems in order to optimize execution time and utilization of resources.

The PDT addresses performance debugging of one Cell BE board with two PPEs that share the main memory, run under the same (Linux) operating system, and share up to 16 SPEs. The PDT also enables event tracing on the Hybrid-x86.

2.6.5 Feedback Directed Program Restructuring (FDPR-Pro)

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

2.6.6 Visual Performance Analyzer (VPA)

Visual Performance Analyzer (VPA) is an Eclipse-based performance visualization toolkit. It consists of six major components::

- Profile Analyzer
- Code Analyzer
- Pipeline Analyzer
- Counter Analyzer
- Trace Analyzer
- Control Flow Analyzer

Profile Analyzer provides a powerful set of graphical and text-based views that allow users to narrow down performance problems to a particular process, thread, module, symbol, offset, instruction, or source line. Profile Analyzer supports time-based system profiles (Tprofs) collected from a number of IBM platforms and Linux profile tool oprofile. The Cell BE is now a fully supported environment for VPA.

2.7 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) v Seamless integration of the simulator into Eclipse
- Automatic builder, performance tools, and several other enhancements v Remote launching, running and debugging on a BladeCenter QS21 v 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 BE processor or an enhanced

CBEA-compliant processor with a fully pipelined, double precision SPE processor

- Remote Cell BE and simulator BladeCenter support
- SPU timing integration v Automatic makefile generation for both GCC and XLC projects

2.8 Hybrid-x86 programming model

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 BE 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 potentially many ways to move data and functions from a host processor to an accelerator processor and vice versa.

In order to provide a consistent methodology and set of application programming interfaces (APIs) for a variety of hybrid systems, including the Cell BE SoC hybrid system, the SDK has implementations of the Cell BE multi-core data communication and programming model libraries, Data and Communication Synchronization and Accelerated Library Framework, which can be used on x86/Linux host process systems with Cell BE-based accelerators. The current implementation is over TCP/IP 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.

Part 2

Programming Environment

In this part of the book we provide in depth coverage of various programming methods, tools, strategies, and adaptions to different computing workloads.

3

Enabling applications on the Cell BE

This chapter describes the process of enabling an existing application on Cell BE hardware. The aim is to provide guidance for choosing the best programming model and framework for a given application. This is valuable information for people actually developing applications and also to IT specialists who will have to manage a Cell BE application enablement project.

We also include the case of a completely new application, being written from scratch. This can be viewed as a special case of an application enablement, where the starting point is not actual code, but only algorithms with no initial data layout decisions. In a sense, this is an easier case as the options are completely open and not biased by the current state of the code.

This chapter tries to answer a few questions:

- Should I enable this application on Cell BE hardware? Is it a good fit?
- If the answer to this question is yes, then which parallel programming model should I use? The Cell BE, with its heterogenous design and software controlled memory hierarchy offers new parallel programming paradigms to complement the well established ones.
- Which Cell BE programming framework will best support the programming model that was chosen for the application under study ?

Once these questions have been answered, it is time to do the hard work, actually making the necessary code changes to exploit the Cell BE architecture. We describe this process and show that it can be iterative and incremental. These are two interesting features as we can use a step by step approach, inserting checkpoints during the course of the porting project to track the progress.

Finally, we present a few scenarios and make a first attempt at creating a set of design patterns for Cell BE programming.

This chapter contains seven parts. We first define the concepts and terminology, introducing:

- the computational kernels frequently found in applications,
- the distinctive features of the Cell BE, which are covered in great details in “Cell BE programming” on page 75
- the parallel programming models,
- the Cell BE programming frameworks, described in “Cell BE programming” on page 75 and Chapter 7, “Programming in distributed environments” on page 439

Next we describe the relationship between computational kernels and the Cell BE features on one hand and between parallel programming models and Cell BE programming frameworks on the other hand.

We give examples of some of the most common parallel programming models and contrast them in terms of control parallelism and data transfers. We make a first attempt at presenting some design patterns for Cell BE programming following a formalism used in other areas of computer sciences.

3.1 Concepts and terminology

Figure 3-1 on page 32 shows the concepts and how they are related. As the figure shows, we describe an application has having one or more computational kernels, *and* one or more potential parallel programming models. The computational kernels exercise or stress one or more of the Cell BE features (the Q1 connection). The different Cell BE features can either strongly or weakly support the different parallel programming model choices (the Q2 connection). And the chosen parallel programming model can be implemented on the Cell BE using various programming frameworks (the Q3 connection).

To answer questions Q1 and Q2 the programmer needs to be able to match the characteristics of the computational kernel and parallel programming model to the strengths of the Cell BE. There are many programming frameworks available for the Cell BE. Which one is best suited to implement the parallel programming model that is chosen for the application? We provide advice to the programmer for question Q3 in section 3.4, “Which Cell BE programming framework to use ?” on page 60.

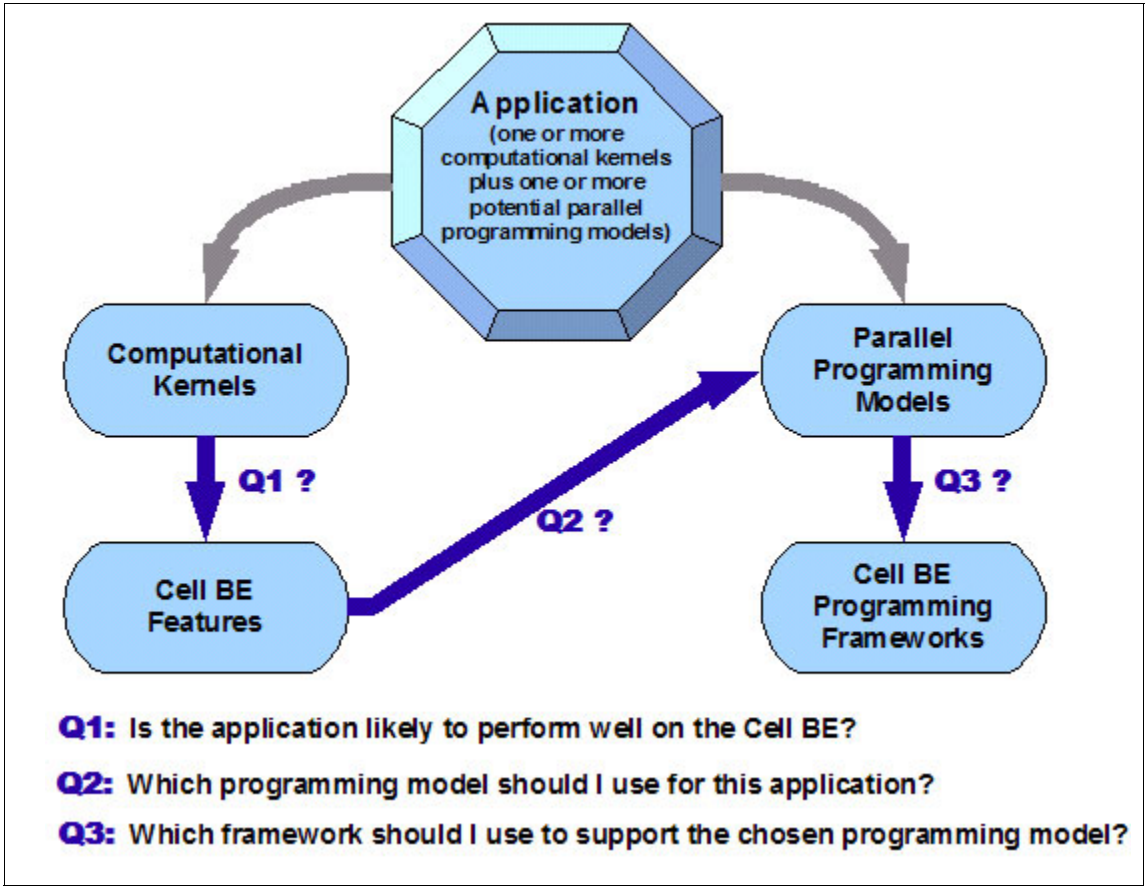


Figure 3-1 Overview of programming considerations and relations

3.1.1 The computation kernels

A study by David Patterson et al [1], complementing earlier work from Phillip Colella [8], establishes that the computational characteristics and data movement patterns of all applications in scientific computing, embedded computing, desktop and server computing can be captured by no more than thirteen different kernels[1] : the “13 dwarfs” as they are named in this paper. This work is based on a careful examination of the most popular benchmark suites:

- EEMBC[2] for the embedded computing,
- SPEC[3] int and fp for the desktop and server computing,

[1] Intel® [9] also classifies applications in three categories named RMS for Recognition, Mining and Synthesis to direct its research in computer architecture.
[2] EEMBC : Embedded Microprocessor Benchmark Consortium.

- HPCC[4]and NAS[5] parallel benchmarks for scientific computing,

as well as input from other domains : machine learning, database, computer graphics and games. The first 7 dwarfs are the ones initially found by Phillip Colella. The 6 remaining ones were identified by Patterson et al. The intent of the paper is to help the parallel computing research community, in the academia and the industry, by providing a limited set of patterns against which new ideas for hardware and software can be evaluated.

We describe the “13 dwarfs” in Table 3-1, with some example applications or benchmarks. This table is adapted from [1].

Table 3-1 The 13 dwarfs, description and examples

Dwarf name	Description	Example, application, benchmark
Dense matrices	BLAS, matrix-matrix operations	HPCC:HPL, ScaLAPACK, NAS:LU
Sparse matrices	Matrix-vector operations with sparse matrices	SuperLU, SpMV, NAS:CG
Spectral methods	FFT transforms	HPCC:FFT, NAS:FT, FFTW
N-body methods	Interactions between particles, external, near and far	NAMD, GROMACS
Structured grids	Regular grids, can be automatically refined	WRF, Cactus, NAS:MG
Unstructrured grids	Irregular grids, finite elements and nodes	ABAQUS, FIDAP (Fluent™)
Map-reduce	Independant data sets, simple reduction at the end	Monte-Carlo, NAS:EP, Ray tracing
Combinatorial logic	Logical functions on large data sets, encryption	AES, DES
Graph traversal	Decision tree, searching	XML parsing, Quicksort
Dynamic programming	Hidden Markov models, sequence alignment	BLAST

[3] SPEC : Standard Performance Evaluation Consortium.
[4] HPCC : High Performance Computing Challenge benchmarks.
[5] NAS : NASA Advanced Supercomputing benchmarks

Dwarf name	Description	Example, application, benchmark
Back-track and Branch+Bound	Constraint optimization	Simplex algorithm
Graphical models	Hidden Markov models, Bayesian networks	HMMER, bioinformatics, genomics
Finite state machine	XML transformation, Huffman decoding	SPECInt:gcc

During the course of writing [1], IBM provided an additional classification for the 13 dwarfs by evaluating which factor was often limiting its performance, whether it be the CPU, the memory latency or the memory bandwidth. Here is this table, extracted from [1].

Table 3-2 Performance bottlenecks of the 13 dwarfs

Dwarf	Performance bottleneck (CPU, memory bandwidth, memory latency
Dense matrices	CPU limited
Sparse matrices	CPU limited 50%, bandwidth limited 50%
Spectral methods	Memory latency limited
N-body methods	CPU limited
Structured grids	Memory bandwidth limited
Unstructrured grids	Memory latency limited
Map-reduce	(unknown)[a]
Combinatorial logic	Memory bandwidth limited for CRC, CPU limited for cryptography
Graph traversal	Memory latency limited
Dynamic programming	Memory latency limited
Back-track and Branch+Bound	(unknown)
Graphical models	(unknown)
Finite state machine	(unknown)

a. Every method will ultimately have a performance bottleneck of some kind. At the time of writing, specific performance bottlenecks for these “unknown” computational kernel types as applied to the Cell BE platform are well understood yet.

The Cell BE brings improvements in all three directions : 9 cores on a chip represent a lot of CPU power, the XDR memory subsystem is extremely capable and the software managed memory hierarchy (DMA) is a new way of dealing with the memory latency problem, quoted in this paper as the most critical one.

3.1.2 Important Cell BE features

Here we consider some features that are meaningful from an application programmer’s point of view. Referring to Table 3-3 below, “Not so good” here only means that the feature is probably going to incur more pain for the programmer or that a code exercising this feature a lot will not perform as fast as expected. The difference between “Good” or “Not so good” may also be related to the relative performance advantage of the Cell BE over contemporary processors from IBM or others. This analysis is based on current hardware implementation at the time this book was written. This table is likely to change with future products.

Most of the items below are described in great details in “Cell BE programming” on page 75.

Table 3-3 The important Cell BE features as seen from a programmer’s perspective

Good	Not so good
Large register file	
DMA (memory latency hiding) [a]	DMA latency
EIB bandwidth [b]	
Memory performance	Memory size
SIMD [c]	Scalar performance (Scalar on Vector)
Local Store (latency/bandwidth)	Local Store (limited size)
8 SPE per processor (high level of achievable parallelism)	PPE performance
NUMA (good scaling)	SMP scaling
	Branching
Single precision floating point	Double precision floating point[d]

a. See paragraph “Data transfer” on page 109

b. See paragraph “Direct problem state access and LS to LS transfer” on page 143
c. See paragraph “SIMD programming” on page 253
d. This is expected to improve with the introduction of the Bladecenter QS-22

In general, the percentage of peak performance that can be achieved on the Cell BE can be higher than for most general purpose processors thanks to the large register file, the short Local Store latency, the software managed memory hierarchy, the very high EIB bandwidth and the very capable memory subsystem.

3.1.3 The parallel programming models

The parallel programming models abstract the hardware for the application programmers. The purpose is to offer an idealized view of the current system architectures, a view onto which applications can be mapped. This is pictured below.

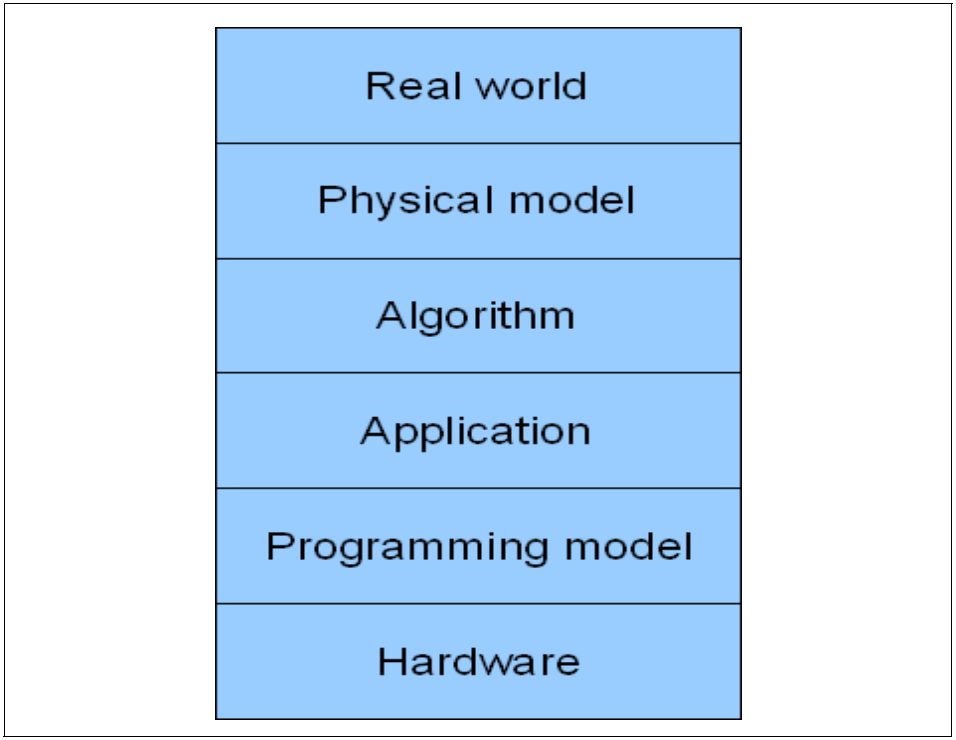


Figure 3-2 The programming model

A parallel application tries to bind multiple resources for its own use : memory and processors. The purpose is either to speed up the whole computation (more processors) or to treat bigger problems (more memory). The work of parallelizing an application involves:

- distributing the work across processors
- distributing the data if the memory is distributed
- synchronizing the sub-tasks, possibly through a shared data access if the memory is shared
- communicating the data if the memory is distributed

Let's take a look at the options for each of these components.

Work distribution

The first task is to find concurrency in the application, try to expose multiple independent tasks and group them together inside execution threads.The options are:

- independent tasks operating on largely independent data
- domain decomposition, where the whole data can be split in multiple sub-domains, each of which being assigned to a single task
- streaming, where the same piece of data undergoes successive transformations, each of which being performed by a single task, all tasks being arranged in a string and passing data in a producer-consumer mode. The amount of concurrency here is the number of different steps in the computation.

Now, each parallel task can perform the work itself or it can call other processing resources for assistance, this process being completely transparent to the rest of the participating tasks. We find the following techniques:

- function offload, where the compute intensive part of the job is being offloaded to a supposedly faster processor
- accelerator mode, a variant of the previous technique, where multiple processors can be called to collectively speed up the computation

Data distribution

The data model is a very important part of the parallelization work. Currently, the choices are between:

- shared memory, every execution thread has direct access to other threads' memory contents.

- distributed memory, it is the opposite, each execution thread can only access its own memory space. Specialized functions are required to import other threads' data into its own memory space.
- PGAS, Partitioned Global Address Space, where each piece of data must be explicitly declared as either shared or local but within a unified address space.

Task synchronization

Sometimes, during the program execution, the parallel tasks will need to synchronize. This can be realized though :

- messages or other asynchonous events emitted by other tasks
- locks or other synchronization primitives, for accessing a queue for example
- though transactional memory mechanisms

Data communication

In the case where the data is distributed, tasks will exchange information through one of these two mechanisms:

- message passing, using send and receive primitives
- remote direct memory access (rDMA), sometimes defined as one-sided communication

The programming models can further be classified according to the way each of these tasks is being taken care of : explicitely by the programmer of implicitely by the underlying runtime environment. The Table 3-4 lists a few common parallel programming models and shows how they can be described according to what was exposed above.

Table 3-4 A few parallel programming models

Programming model[a]	Task distribution	Data distribution	Task synchonization	Data communication
MPI	explicit	explicit	messages	messages, can do rDMA too
pthreads	explicit	n/a	mutexes, condition variables	n/a
OpenMP	implicit (directives)t	n/a	implicit (directives)	n/a
UPC, CAF (PGAS)	explicit	explicit	implicit	implicit
X10 (PGAS)	explicit	explicit	future, clocks	implicit

Programming model[a]	Task distribution	Data distribution	Task synchonization	Data communication
StreamIt	explicit	explicit	explicit	implicit

a. Not all of these programming models are available for appropriate for the Cell BE platform

Programming model composition

An application can make use of multiple programming models. This will incur additional efforts but may be dictated by the hardware on which it is to be run. For example, the MPI + OpenMP combination is quite common today for HPC applications as it matches the Beowulf[6] type of clusters, interconnecting small SMP nodes (4, 8 way) with a high speed interconnection network

In this discussion about the parallel programming models, we have ignored the instruction level (multiple execution units) and word level (SIMD) parallellisms. They are to be considedred too of course to maximize the application performance but usually do not interfere with the high level tasks of data and work distribution.

3.1.4 The Cell BE programming frameworks

The Cell BE supports a wide of range of programming frameworks, from the most basic ones, close to the hardware, to the most abstract ones.

At the lowest level, the Cell BE chip appears to the programmer as a distributed memory cluster of 8+1 computational cores, with a ultra high speed interconnect and a remote DMA engine on every core. On a single blade server, two Cell BE chips can be viewed as either a single 16+2 cores compute resource (SMP mode) or a NUMA machine with 2 NUMA nodes.

Multiple blade servers can then be gathered in a distributed cluster, a la Beowulf, using a high speed interconnect network like 10G Ethernet or Infiniband. Such a cluster is not any different, as far as programming is concerned, from a cluster of Intel, AMD™ or POWER™ based SMP servers. Very likely the programming model will be based on distributed memory programming using MPI as the communication layer across blades.

An alternative arrangement is to have Cell BE blade servers serve only as accelerator nodes for other systems. In this configuration, the application logic is not managed at the Cell BE level but at the accelerated system level and the dialog we are interested in is between the Cell BE and the system it provides acceleration for.

[6] Clusters based on commodity hardware. See http://www.beowulf.org/overview/index.html

The frameworks that are part of the IBM SDK for Multicore Acceleration are described in greater details in paragraph “Frameworks and domain-specific libraries” on page 283 and “Hybrid Programming Models in SDK 3.0” on page 440. We only give here a brief overview.

libspe2, newlib

This is the lowest possible level for application programmers. The libspe2 and newlib libraries let programmers deal with each feature of the Cell BE architecture with full control. If we recap the main libspe2/newlib features, we find:

- SPE context management for creating, running, scheduling and deleting SPE contexts
- DMA primitives for accessing remote memory locations from the PPE and the SPEs
- mailboxes, signal, events and synchronization functions for PPE-SPE and SPE-SPE dialogs and control
- PPE assisted calls, a mechanism to have the PPE service requests from the SPEs.

Important: Libspe2 is a framework for obtaining access to the SPEs. Mailbox, DMAs, signals, etc., are much faster when using direct problem state. High performance programs should avoid making frequent libspe calls since they often utilize kernel services. As such, it is best to use libspe2 to get parallel tasks started, then use the MFC HW facilities for application task management, communications, and synchronization.

Using these libraries, any kind of parallel programming model can be implemented. libspe2 is described in great details in “Task parallelism and PPE programming” on page 78.

Software cache

Software cache can help implement a shared memory parallel programming model when the data that the SPEs reference cannot be easily predicted. See “Automatic software caching on SPE” on page 155 for more details.

DaCS, Data Communication and Synchronization

DaCS provides services to multi-tier applications using a hiearchy of processing elements. A DaCS program can be either a Host Elements (HE) or an Accelerator Element (AE) or both if multiple levels of acceleration are needed. An AE can only communicate within its HE’s realm. A HE need not be of the same type as its HEs. This is the hybrid model. DaCS will take care of the necessary

byte swapping if the data flows from a little endian machine to the big endian Cell BE.

The typical DaCS services are :

- resource and process management, where a HE manipulates its AEs,
- group management, for defining groups within which synchronization events like barriers can happen,
- message passing, using send and receive primitives,
- mailboxes,
- remote DMA operations,
- process synchronization using barriers
- data synchronization using mutexes to protect memory accesses

The DaCS services are implemented as an API for the Cell BE only version and are complemented by a run time daemon for the hybrid case. For a complete discussion, see “DaCS - Data Communication and Synchronization” on page 284 and “Hybrid DaCS” on page 443.

MPI

MPI is not part of the IBM SDK for Multicore Acceleration but any implementation for Linux on POWER will be able to run on the Cell BE, leveraging the PPE only. The most common implementations are :

- MPICH/MPICH2, from Argonne National Laboratory,
- MVAPICH/MVAPICH2, from Ohio State University
- OpenMPI, from a large consortium involving, amongst others, IBM and Los Alamos National Laboratory

There is no difference from a functional point of view between these MPI implementations running on Cell BE and running on other platforms. MPI is obviously a very widespread standard for writing distributed memory applications. MPI, as opposed to DaCS, treats all tasks as equal and lets the programmer decide if, later on, some tasks are to play a particular role in the computation. MPI is implemented as an API and a runtime environment to support all sorts of interconnexion mechanisms between the MPI tasks : shared memory, sockets for TCP/IP networks or OpenIB (OFED) for Infiniband networks.

ALF, Accelerated Library Framework

An ALF program uses multiple ALF accelerator tasks to perform the compute intensive part of the work. The general idea is to have the host program split the work into multiple independent pieces, the so-called work blocks, described by a

computuation kernel, the input data they need as well as the output data they produce. On the accelerator side, the programmer only has to code the computational kernel, unwrap the input data and pack the output data when the kernel has finished processing. In between, the runtime system is responsible for managing the work blocks queue on the accelerated side and giving control to the computational kernel upon receiving a new work block on the accelerator side.

ALF imposes a clear separation between the application logic and control running on the host task from the computational kernels that run on the accelerator nodes, acting as service providers which are fed with input data and echo back output data. The ALF runtime provides the following services “for free” from the application programmer perspective :

- work blocks queue management,
- load balancing between accelerators,
- transparent DMA transfers, exploiting the data transfer list used to describe the input and output data.

The table below summarizes the various duties :

Table 3-5 Work separation with ALF

Who	**does what**
Host code writer	program flow logic manage accelerators work blocks creation, input and output data specified as a series of address-type-length entries manage communication and synchronization with peer host tasks
Accelerator code writer	computational kernel
ALF runtime	schedule work blocks to accelerators data transfer

ALF also offers more sophisticated mechanisms for managing multiple computational kernels, express dependencies or tune further the data movement. Just like DaCS, ALF can operate inside a Cell BE server or in hybrid mode.

ALF is described in greater details in “ALF - Accelerated Library Framework” on page 291 and “Hybrid ALF” on page 456.

DAV, IBM Dynamic Application Virtualization

Using DAV, an IBM offering available from Alphaworks, an application can benefit from Cell BE acceleration without any source code changes. The original, untouched application, is only directed to use a stub library that is dynamically loaded and offloads the compute intense functions to a Cell BE. IBM DAV currently supports C/C++ or Visual Basic® Applications (like Excel® 2007 spreadsheets) running under the Microsoft® Windows® operating system. IBM DAV comes with tools to generate ad-hoc stub libraries based on the prototypes of the offloaded functions on the client side and similar information on the server side (the Cell BE system) where the actual functions are implemented. For the main appplication, the Cell BE is completely hidden. Of course, the actual implementation of the function on the Cell BE will use the existing programming frameworks to maximize the application perfornance.

See 7.1.3, “DAV - Dynamic Application Virtualization” on page 468 for a more complete description.

Workload specific libraries

The IBM SDK for Multicore Acceleration contains a few workload specialized libraries. These are the BLAS library for linear algebra, libFFT for fast Fourier transforms in 1D and 2D and libmc for random number generations.

OpenMP

The IBM SDK for Multicore Acceleration contains a technology preview of the XL C/C++ single source compiler. Using this compiler completely hides the Cell BE to the application programmer who can continue using OpenMP : a familiar shared memory parallel programming model. The compiler runtime library takes care of spawning threads of execution on the SPEs and manages the PPE threads to SPE threads data movement and synchronization.

There are other groups or companies working on providing programming frameworks for the Cell BE. They are briefly discussed here.

Mercury Computer Systems

Mercury has two main offerings for the Cell BE:

- MCF, the MultiCore Framework which implements the manager/worker model with an input/output tile management akin to the ALF work blocks
- Multicore Plus™ SDK, which bundles MCF with additional signal processing and FFT libraries (SAL, VSIPL), a trace library (TATL) and Open Source tools for MPI communications (OpenMPI) and debugging (gdb)

PeakStream

The PeakStream™ Platform offers an API, a generalize Array type and a Virtual Machine environment that abstracts the programmer from the real hardware. Data is moved back and forth between the application and the Virtual Machine that accesses the Cell BE resources using an I/O interface. All the work in the Virtual Machine is asynchronous from the main application perspective which can keep on doing work before reading the data from the Virtual Machine. The PeakStream Platform currently runs on the Cell BE, GPUs and traditional homogeneous multi-core CPUs.

Code Sourcery

CodeSourcery offers Sourcery VSIPL++™, a C++ implementation of the open standard VSIPL++ library used in signal and image processing. The programmer is freed from accessing the low level mechanisms of the Cell BE. This is taken care of by the CodeSourcery runtime library.

The VSIPL (Vector Signal and Image Processing Library) contains routines for :

- linear algebra for real and complex values
- random numbers
- signal and image processing (FFT, convolutions, filters)

Code Sourcery also runs on GPU and multi-core general purpose CPU.

Gedae

Gedae tries to automate the software development by using a model-driven approach. The algorithm is captured in a flow diagram that is then used by the multiprocessor compiler to generate a code that will match both the targer architecture and the data movements required by the application.

RapidMind

RapidMind works with standard C++ language constructs and augments the language using specialized macro language functions. The whole integration proceeds in three steps :

- replace float or int arrays by RapidMind equivalent types (Array, Value),
- capture the computations enclosed between the RapidMind keywords Program BEGIN and END and convert them into object modules
- stream the recorded computations to the underlying harware using platform specific constructs (Cell BE, CPU or GPU) when the modules are invoked

There are also research groups working on implementing other frameworks onto the Cell BE. Worth noting are the efforts of the Barcelona Supercomputing teams

with CellSs (Cell Superscalar) and derivatives like SMPSs (SMP Superscalar) and from Los Alamos National Laboratory with CellFS, based on concepts taken from the Plan9 operating system.

In Figure 3-3 on page 45 we plot these frameworks on a scale ranging from the closest to the hardware to the most abstract.

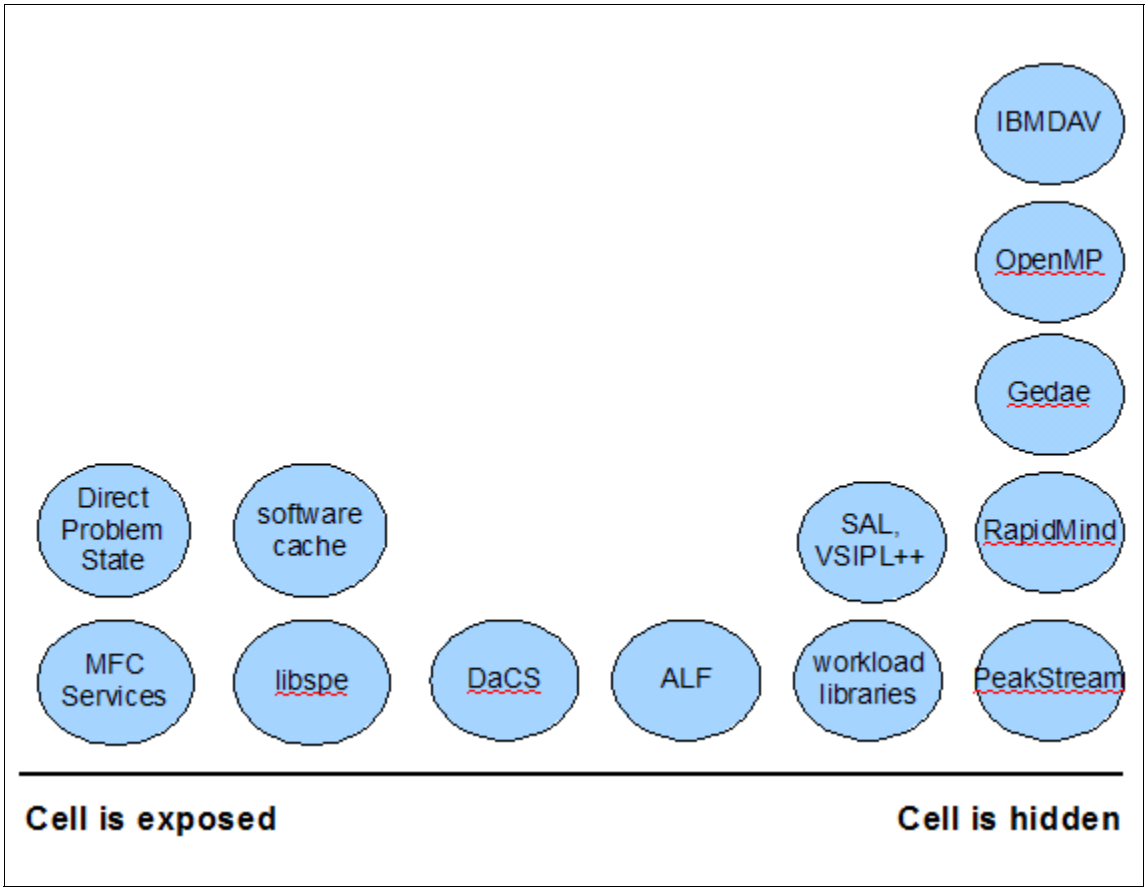


Figure 3-3 Relative positioning of the Cell programming frameworks

IBM DAV - Dynamic Application Virtualization is particular here. On the accelerated program side (the client side in DAV terminology), the Cell BE is completely hidden using the stub DLL mechanism. On the accelerator side (the server side for DAV), any Cell BE programming model can be used to implement the functions which have been offloaded from the client application.

3.2 Does the Cell BE fit the application requirements?

We will use the decision tree on Figure 3-4 on page 46 to answer this question.

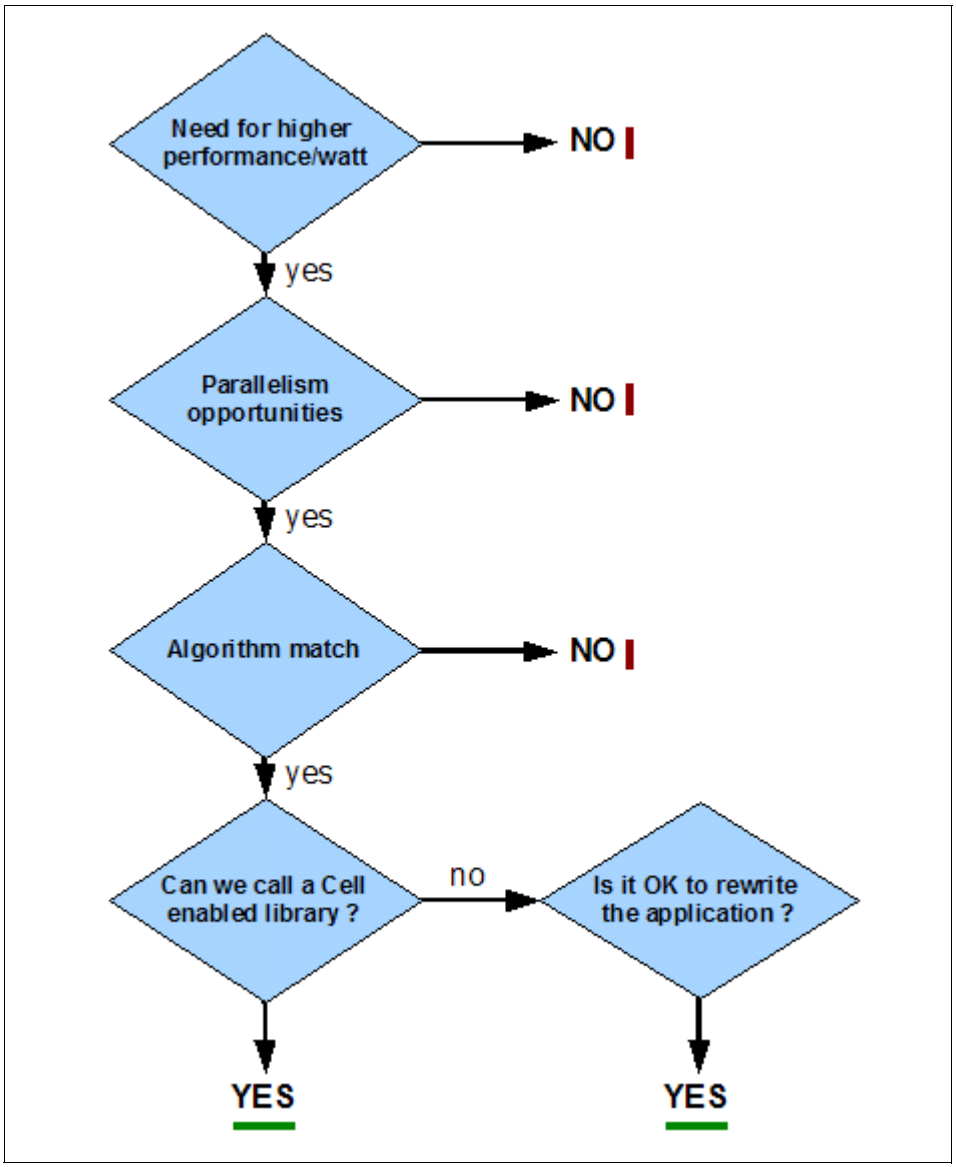


Figure 3-4 Is the Cell BE a good fit for this application

3.2.1 Higher performance/watt

The main driver for enabling applications on the Cell BE is the need for a higher level of performance per watt. This is a concern shared by many customers as is reported by the IDC study referenced in [15]. Customers may be willing to :

- lower their electricity bills,
- overcome computer rooms limits in space, power and cooling,
- adopt a green strategy for their IT : a green ITtude,
- allow for more computing power for a given space and electrical power budget as is often the case in embedded computing.

The design choices for the Cell BE exactly match these new requirements with a power efficiency (expressed in peak Gflops per Watt) that is over two times better than conventional general purpose processors.

3.2.2 Opportunities for parallelism

The Cell BE offers parallelism at four levels:

- across multiple System x™ servers in a hybrid environment. This is expressed using MPI at the cluster level or some sort of grid computing middleware.
- across multiple Cell BE chips/servers. Here we use MPI communication between the Cell BE servers in the case of a homogeneous cluster of standalone Cell BE servers or possibly ALF or DaCS for hybrid clusters.
- across multiple SPE inside the Cell BE chip/server, using libspe2, ALF, DaCS or a single source compiler.
- at the word level with SIMD instructions on each SPE, using SIMD intrinsics or the auto-SIMDization capabilities of the compilers.

The more parallel processing opportunities the application can leverage the better.

3.2.3 Algorithm match

Here, we are looking for a match between the main computational kernels of the application and the Cell BE strengths as listed on Table 3-3. As we have seen in 3.1.1, “The computation kernels” on page 32, most applications can be characterized by a composition of the 13 “dwarfs” of Patterson et al [1]. It is therefore important to know which kernels a given application is built with. This is

usually very easy to do as it is related to the numerical methods used in the applications.

In a paper by Williams et al. [2], the authors have studied how the Cell BE performs on four of the 13 “dwarfs” : denses matrices algebra, sparse matrices algebra, spectral methods and structures grids. They compared the performance of these kernels on a Cell BE with what they obtained on a superscalar processor (AMD Opteron™), a VLIW processor (Intel Itanium2™) and a vector processor (Cray X1E™). The results were found very interesting for the Cell BE, a very good result as these kernels are extremely common in many HPC applications.

Other authors have reported successes for graphical models (bioinformatics, HMMer [16]), dynamic programming (genomics, BLAST [17]), unstructured grids (Finite Element Solvers [18], combinatorial logic (AES, DES [19]).

The map-reduce dwarf is embarrassingly parallel and is therefore a perfect fit for the Cell BE. Examples can be found in ray-tracing or Monte-Carlo simulations.

The graph traversal dwarf is a more difficult target for the Cell BE due to random memory accesses although some new sorting algorithms (AA-sort in [5]) have been shown to exploit the Cell BE architecture.

The N-Body simulation does not seem yet ready for Cell BE exploitation although research efforts are providing good early results [20].

The table summarizes the results of these studies. We present for each of the “13 dwarfs”, its Cell BE affinity (from 1, poor to 5 excellent), and the Cell BE features that are of most value for each kernel.

The algorithm match also depends on the data types that are being used. The current Cell BE has a single precision floating point affinity. There will be much larger memory and the enhanced double precision floating point capabilities in later versions of Cell BE.

Table 3-6 tThe 13 dwarfs from Patterson et al. and their Cell BE affinity

Dwarf name	**Cell BE affinity 1, poor to 5, excellent**	**Main Cell BE features**
Dense matrices	5	8 SPE per Cell BE SIMD large register file for deep unrolling fused multiply-add
Sparse matrices	4	8 SPE per Cell BE memory latency hiding with DMA high memory sustainable load

Dwarf name	Cell BE affinity 1, poor to 5, excellent	Main Cell BE features
Spectral methods	5	8 SPE per Cell BE large register file 6 cycles Local Store latency memory latency hiding with DMA
N-body methods	?	?
Structured grids	5	8 SPE per Cell BE SIMD high memory bandwidth memory latency hiding with DMA
Unstructrured grids	3	8 SPE per Cell BE high memory thruput
Map-reduce	5	8 SPE per Cell BE
Combinatorial logic	4	large register file
Graph traversal	2	memory latency hiding
Dynamic programming	4	SIMD
Back-track and Branch+Bound	?	?
Graphical models	5	8 SPE per Cell BE SIMD
Finite state machine	?	?

As can be derived from the above table, the Cell BE is a good match for many of the common computational kernels. This is the result of the design decisions that were made to address the main bottlenecks : memory latency and throughput as well as a very high computational density with 8 SPE per Cell BE each with a very large register file and a extremely low local store latency (6 cycles compared to 15 for current general purpose processors from Intel or AMD).

3.2.4 Ready to make the effort?

The Cell BE may be easy on the electricity bill but can be hard on the programmer. Enabling an application on the Cell BE may result in very

substantial algorithmic and coding efforts. But the results are usually worth the efforts.

What are the alternatives ? The parallelization effort may have already been done using OpenMP at the process level. In this case, using the prototype of the XLC single source compiler might be the only viable alternative. Despite a very high usability, these compilers are still far from providing the level of performance that can be attained with native SPE programming. The portability of the code is maintained and for some customers this might be a key requirement.

For new developments, it might be a good idea to use the higher level of abstraction provided by the likes of Peakstream, Rapidmind or Streamit[7]. The portability of the code is maintained between Cell BE, GPU and general multi-core processors. But the application is tied to the development environment, a different form of lock-in.

In the long run, new standardized languages may emerge. Projects like X10[8] from IBM or Chapel[9] from Cray may become the preferred language for writing applications to run on massively multi-core systems. Adopting new languages has historically been a very slow process and even if we get a new language, that still does not help the millions of lines of code written in C/C++ and Fortran. Standard API for the host-accelerator model may be closer. ALF is a good candidate. The very fast adoption of MPI in the mid 90s has proved that an API can be just what we need to enable a wide range of applications.

But can we wait for these languages and standards to emerge? If the answer is no and the decision has been taken to enable the application on Cell BE now, here is a list of things to consider and possible workarounds when problems are encountered.

Table 3-7 Things to consider when enabling an application on Cell BE

Topic	Potential problem	Workaround
Source code changes	Portability concerns Limit the scope of code changes	The Cell BE API are standard C. Approaches like host-accelerator can limit the amount of source code changes
Operating systems	Windows applications	Cell BE runs Linux only. If the appplicaition runs on Windows, we may want to use IBM DAV to offload only the computational part to the Cell BE

[7] http://www.cag.csail.mit.edu/streamit
[8] http://domino.research.ibm.com/comm/research_projects.nsf/pages/x10.index.html
[9] http://chapel.cs.wahsington.edu

Topic	Potential problem	Workaround
Languages	C/C++ fully supported Fortran and ADA supported Other languages not supported	Rewrite the compute intensive part in C Use some sort of offloading for Java™ or VBA applications running on Windows with IBM DAV
Libraries	Not many libraries supported yet Little ISV support	Use the workload libraries provided by the IBM SDK for Multicore Acceleration
Data types	8, 16, 32bit data well supported 64bit float point supported	Full speed double precision support soon to appear
Memory requirements	Maximum is 2GB per blade server	Use more smaller MPI tasks, on an IBM Blade Server use a single MPI task with 16 SPE rather than 2 MPI tasks with 8 SPE. (This is subject to change as much larger memory configuration per blade is due in future product releases.)
Memory requirements	LS size is 256k	Large functions will need to be split Will have to use overlay Limit recursion (stack space)
I/O	I/O intensive tasks	Cell BE does not help I/O bound workloads.

3.3 Which parallel programming model ?

Large homogeneous compute clusters can be built by collecting standalone Cell BE blade servers with an Infiniband interconnect. At this cluster level, the usual distributed memory programming models such as MPI can be used. An application that is already programmed using MPI is a very good start as we only need to add Cell BE parallelism incrementally to fully exploit the Cell BE potential.

Hybrid clusters are becoming increasingly popular as a means of building very powerful configurations by incrementally upgrading existing clusters built with off the shelf components. In this model, the MPI parallelism at the cluster level is maintained but each task is now accelerated by one or more Cell BE.

We will first describe the parallel programming models found in the literature and then focus on the Cell BE chip or board level parallelism and the host-accelerator model.

3.3.1 Parallel programming models basics

Mattson et al. in [4] define a taxonomy of parallel programming models. First they define four “spaces” that the application programmer has to visit. They are described in table 3-8 on 52.

Table 3-8 Four design spaces from Mattson et al.

Space	Description
Finding concurrency	Find parallel tasks Group and order them
Algorithm structure	Organize the tasks in processes
Supporting structure	Code structures for tasks and data
Implementation mechanisms	Low level mechanisms for managing and synchronizing execution threads as well as data communication

The first space is very much application dependant. The implementation mechanisms are described more detail in “Cell BE programming” on page 75 and Chapter 7, “Programming in distributed environments” on page 439.

In the algorithm space, Mattson et al. propose to look at three different ways of decomposing the work, each with two modalities. This leads to six major algorithm structures described in the table 3-9 on 52.

Table 3-9 lAlgorithm structures

Organization principle	Organization sub-type	Algorithm stucture
By tasks	Linear	Task parallelism
	Recursive	Divide and conquer
By data decomposition	Linear	Geometric decomposition
	Recursive	Tree
By data flow	Regular	Pipeline
	Irregular	Event-based coordination

Task parallelism occurs when multiple independant tasks can be scheduled in parallel. Divide and conquer is applied when a problem can be recursively treated by solving smaller sub-problems. Geometric decomposition is very common when the we try to solve a partial differential equation which has been discretized on a 2D or 3D grid and grid regions are assigned to processors.

As for the supporting structures, they identified four structures for organizing tasks and three for organizing data. They are given side by side in table 3-10 on 53.

Table 3-10 Supporting structures for code and data

Code structures	Data structures
SPMD	Shared data
Master/Worker	Shared queue
Loop parallelism	Distributed array
Fork/Join	

SPMD is the Single Program Multiple Data code structure well know to MPI programmer. Although MPI does not impose the use of SPMD, this is a very frequent consruct. Master/worker is sometimes called “bag of tasks” when a master task distributes work elements independant of each other to a pool of workers. Loop parallelism is a low level structure where the iterations of a loop are shared between execution threads. Fork/Join is a model where a master execution threads calls (fork) multiple paralel execution threads and wait for their completion (join) before continuing with the sequential execution.

Shared data refers to the constructs necessary to share data between execution threads. Shared queue is the coordination among tasks to process a queue of work items. Distributed arrray addresses the decomposition of multi-dimensional arrays into smaller sub-arrays that are spread across multiple execution units.

We will now look at how these map to the Cell BE and what needs to be looked at to figure out the best parallel programming model for the application. There are forces which are specific to the Cell BE that will influence the choice. They are given here in no particular order of importance.

Table 3-11 Cell BE specific “forces”

Force
Heterogenous PPE/SPE
Distributed memory between SPE, shared memory view possible by memory mapping the Local Store

Force
SIMD
PPE slow compared to SPE
Software managed memory hierarchy
Limited size of the LS
Dynamic code loading (overlay)
High bandwidth on the EIB
Coherent shared memory
Large SPE context, startup time

3.3.2 Chip or board level parallelism

The Cell BE is a heterogenous, multi-core, distributed memory processor. It offers many opportunities for parallel processing at the chip or board level. On Figure 3-5 on page 55, we show the reach of multiple programming models. The models can sometimes be classified as PPE-centric or SPE-centric. Although this is a somewhat artificial distinction, it indicates that the application control is either run more on the PPE side or on the SPE side.

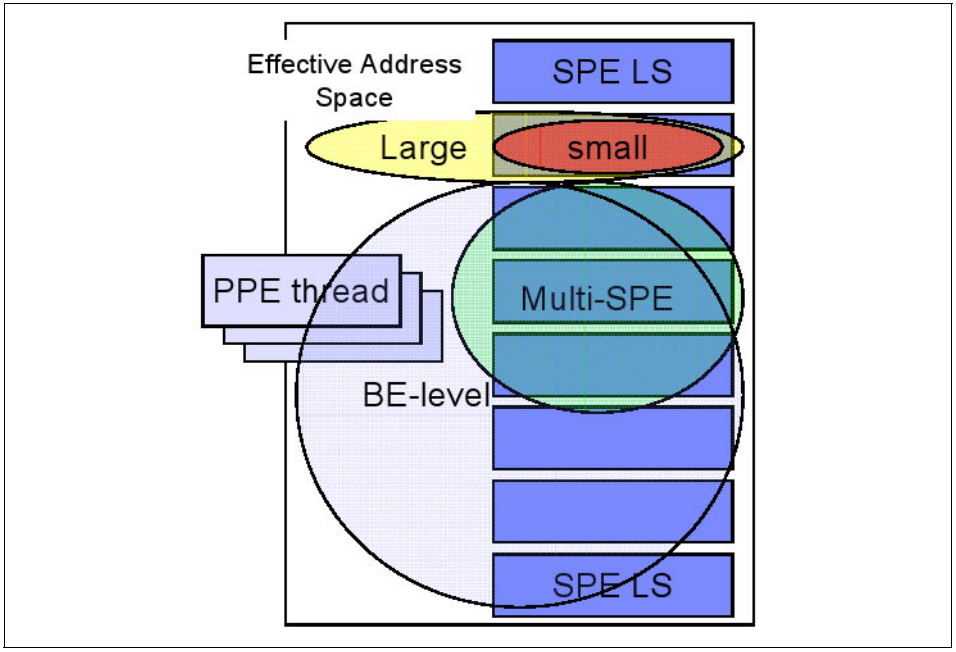


Figure 3-5 Various models on the Cell BE

We see here four different models :

- small single SPE program, where the whole code holds in the Local Store of a single SPE,
- large single SPE program, one SPE program accessing system memory,
- small multi-SPE program,
- general Cell BE program with multiple SPE and PPE threads.

When multiple SPE are used, they can be arranged in a data parallel, streaming mode, as depicted in Figure 3-6 on page 56.

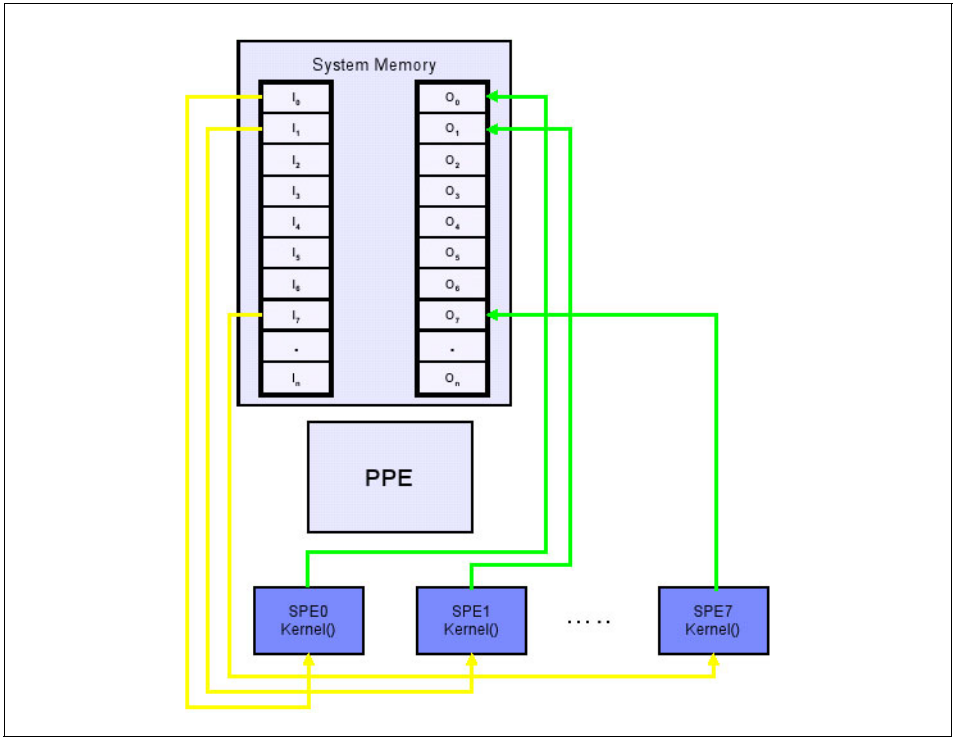


Figure 3-6 The streaming model

Each piece of input data (I0, I1, ...) is streamed through one SPE to produce a piece of output data (O0, O1, etc). The exact same code runs on all SPE. Sophisticated load balancing mechanisms can be applied here to account for differeing compute time per data chunk.

The SPE can also be arranged in a pipeline fashion, where the same piece of data undergoes various transformations as it moves from one SPE to the other. A general pipeline is shown on Figure 3-7 on page 57.

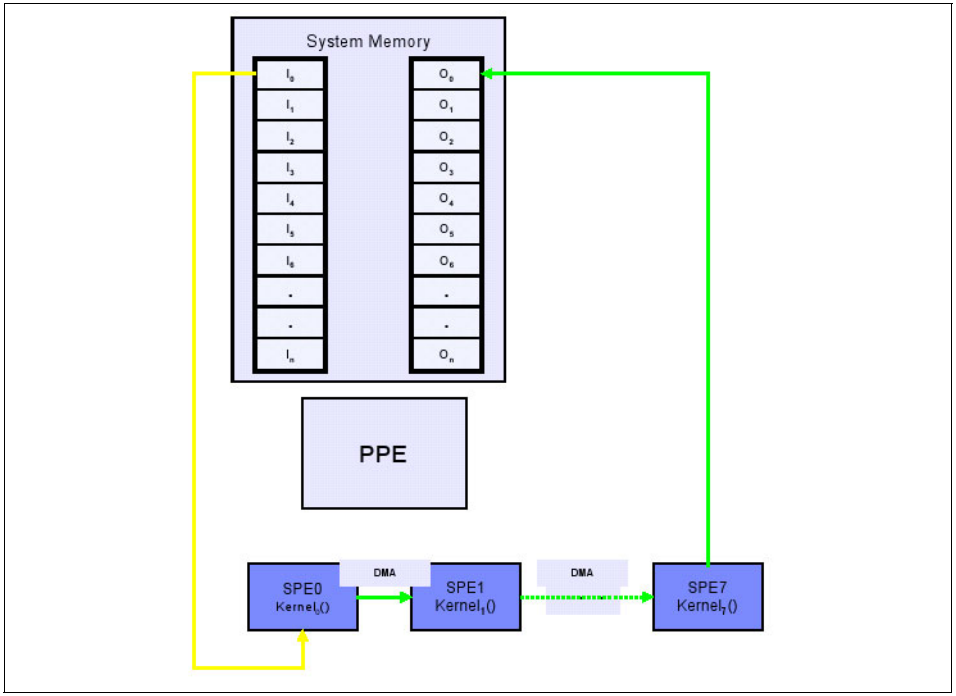


Figure 3-7 The pipeline model

The main benefit is that the we aggregate the code size of all the SPE participating in the pipeline We also benefit from the huge EIB bandwidth to transfer the data. One possible variation is to move the code instead of the data, whichever is the easiest or smallest to move around. A good load balancing is much more challenging as it relies on a constant per stage computational time.

3.3.3 More on the host-accelerator model

A common approach with Cell BE parallel programming is to use a function offload mechanism akin to the RPC model. The application flows on the PPE and only for selected, highly computational kernels do we call upon SPE acceleration. This is the easiest from a program development perspective as it limits the scope of source code changes and does not require much re-engineering at the application logic level. This is very much a fork/join model and care must be taken that we give enough work to the accelerator threads to compensate for the startup time. This is typically implemented with specialized workload libraries like BLAS, FFT or RNG for which there exists a Cell BE tuned version. BLAS is the only library that could be considered a “drop in replacement” at this time.

A variation of this model is to have general purpose accelerator progams running on the SPE, sitting in a loop, awaiting for being asked to provide services for the PPE threads. Having persistent threads on each SPE eliminates the startup time of SPE threads but requires that the accelerator programs be able to service various requests for various functions, possibly incurring the use of dynamic code uploading techniques. The ALF framework described in “ALF - Accelerated Library Framework” on page 291 and “Hybrid ALF” on page 456 is one implementation.

A general accelerator is shown in Figure 3-8.

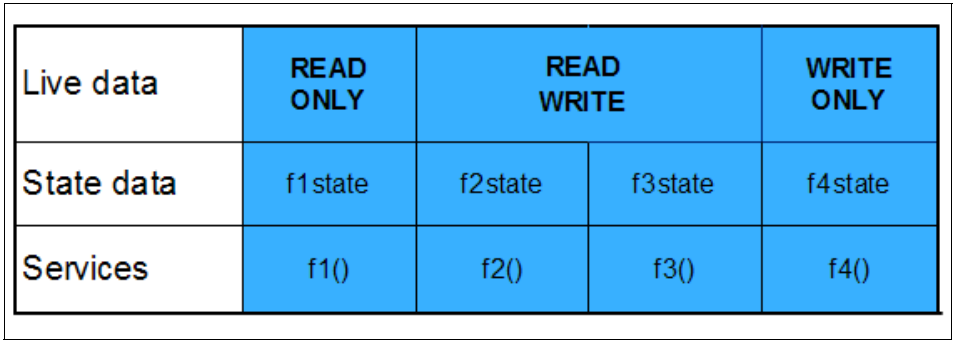


Figure 3-8 Memory structure of an accelerator

An accelerator can implement multiple services (functions f1 to f4). Each function may requires some “state” data, persistent across multiple invocations. The “live data” is the data in and out of the accelerator for each invocation. It is important to understand which is read, written or both to optimize the data transfers.

3.3.4 Summary

Among the various programming models and structures listed in tables 3-9 on 52 and 3-10 on 53, some will be easier to implement on Cell BE than others. The tables 3-12, 3-13 and 3-14 summarize the Cell BE specific issues.

Table 3-12 Cell BE suitability of algorithm structures and specific issues

Algorithm structure	Cell BE suitability (1 : poor, 5 : excellent)	Things to look at
Task parallelism	5	Load balancing Synchronization required for accessing the queue of work items

Algorithm structure	Cell BE suitability (1 : poor, 5 : excellent)	Things to look at
Divide and conquer	?	?
Geometric decomposition	5	DMA and double buffering required Code size
Tree	3	Random memory accesses
Pipeline	5	Load balancing EIB exploitation Move code or data ?
Event-based coordination	3	Code size, resulting code may be inefficent because operation required is not know until we get the even and data to process.

Table 3-13 Cell BE Suitability of code structures and specific issues

Code structure	Cell BE suitability (1 : poor, 5 : excellent)	Things to look at
SPMD	3	Code size The whole application control may not fit in Local Store and more PPE intervention may be required.
Master/Worker	5	Load balancing Synchronization required for accessing the queue of work items
Loop parallelism	3	PPE centric Task granularity Shared memory synchronization
Fork/Join	5	Fits the accelerator model Weigh the thread startup time with the amount of work and data transfer needed

Table 3-14 Data structures and Cell BE specific issues

Data structure	Things to look at
Shared data	Synchronization for accessing shared data, memory ordering and locks

Data structure	Things to look at
Shared queue	From PPE managed to SPE self-managed work queues
Distributed array	Data partitioning and DMA

3.4 Which Cell BE programming framework to use ?

We have now found which parallel programming model (work distribution, data distribution and synchronization) we wish to apply to the application. We may draw on more than one for a given application. We need to implement these using the available Cell BE frameworks. Some frameworks are very general; libspe2 with MFC services accessed through direct problem state (PPE) and channels (SPE) being the most versatile. And some frameworks are very specialized (ex. workload libraries). Choosing one is a matter of weighing the features, the ease of implementation and the performance. There are also application area specifics. For example, it seems that for radar applications, Gedae is almost mandatory.

We list in table 3-15 the various parallel programming constructs and give for each the most appropriate frameworks as a primary and secondary choices.

Table 3-15 Parallel programming constructs and frameworks

Programming construct	Primary	Secondary, comments
MPI	OpenMPI, nothing specific to the Cell BE. This is a cluster/PPE level construct.	MVAPICH, MPICH
pthreads	pthreads supported on the PPE. No direct support for a mapping between PPE pthreads and SPE pthreads. This would have to be implemented using libspe.	
OpenMP	XLC single source compiler	
UPC, CAF	Not supported	
X10	Not supported	
Task parallelism	libspe	This is function offload, see Fork/Join too

Programming construct	Primary	Secondary, comments
Divide and conquer		
Geometric decomposition	ALF if data blocks can be processed independantly	DaCS for more general data decomposition
Tree	software cache ?	
Pipeline	libspe	DaCS, Streamit
Event-based coordination	libspe	
SPMD	This is a PPE level consruct.	
Master/Worker	ALF	libspe
Loop parallelism	XLC single source compiler, with OpenMP support	
Fork/Join	Workload specific libraries if they exist and ALF otherwise. ALF can be used to create new workload specific libraries.	This is the accelerator model. Use DAV if we need to accelerate a Windows application.
Shared data	DaCS	MFC intrinsics, libsync
Shared queue	DaCS	MFC intrinsics, libsync
Distributed array	ALF	DaCS

3.5 The application enablement process

The process of enabling an application on Cell BE can be incremental and iterative. It is incremental in the sense that the hotspots of the application should be moved progressively off the PPE to the SPE. It is iterative as for each hotspot, the optimization can be refined at the SIMD, synchronization and data movement levels until satisfactory levels of performance are obtained.

As for the starting point, a thorough profiling of the application on a general purpose system (PPE is just fine for this) will give all the hotspots that need to be looked at. Then, for each hotspot, we can write a multi-SPE implementation with all the data transfer and synchronization between the PPE and the SPE. Once this first implementation is working, we then turn to the SIMDization and tuning of the SPE code. The last two steps can be repeated in a tight loop until we get a good performance. We can repeat the same process for all the major hotspots. This is shown in Figure 3-9.

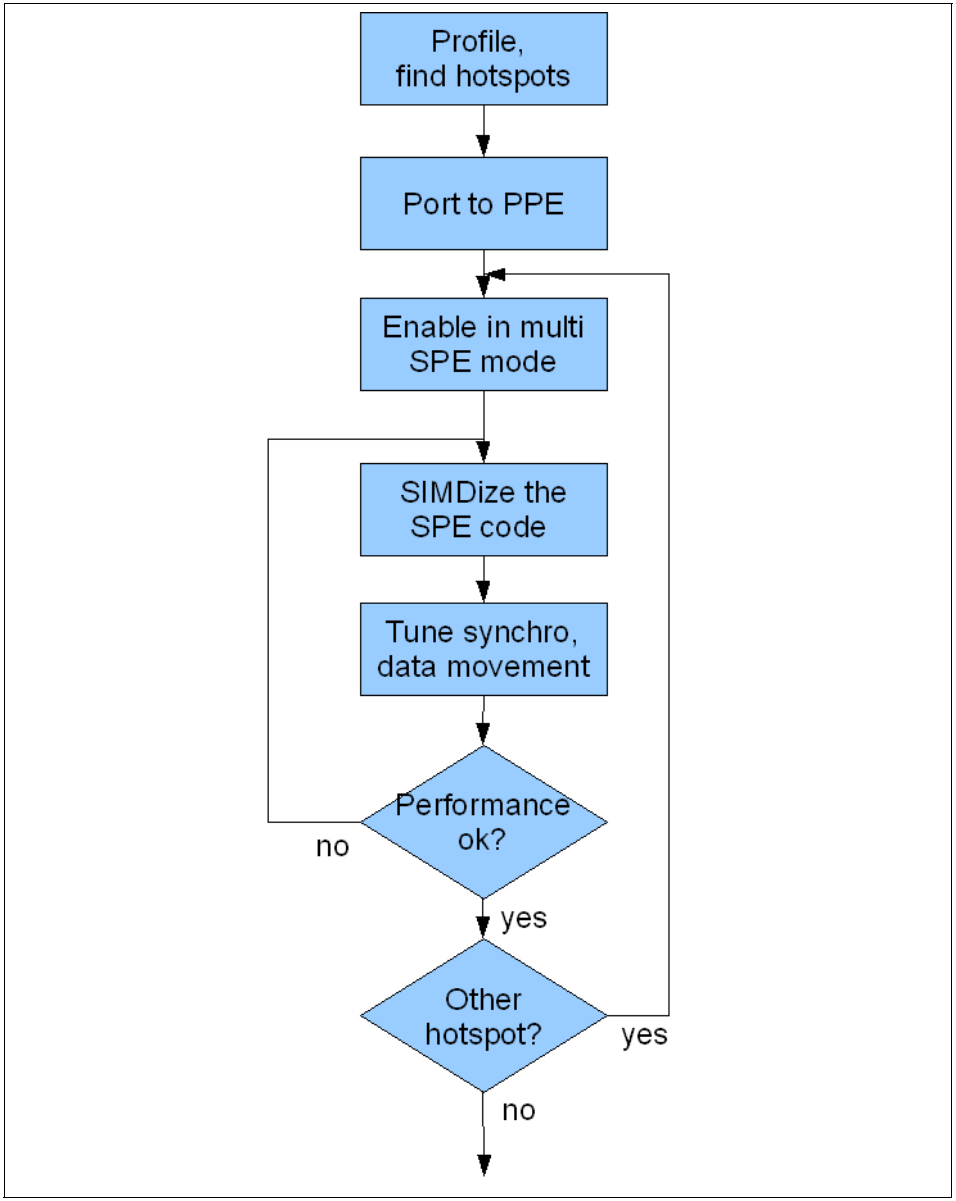


Figure 3-9 General flow for enabling an application on Cell BE

The figure above will change a bit depending on the framework that was chosen. If we are fortunate enough to have an application whose execution time is dominated by a function that happens to be part of a workload specific library

that has been ported to Cell BE, then the process to follow is shown in Figure 3-10 on page 63.

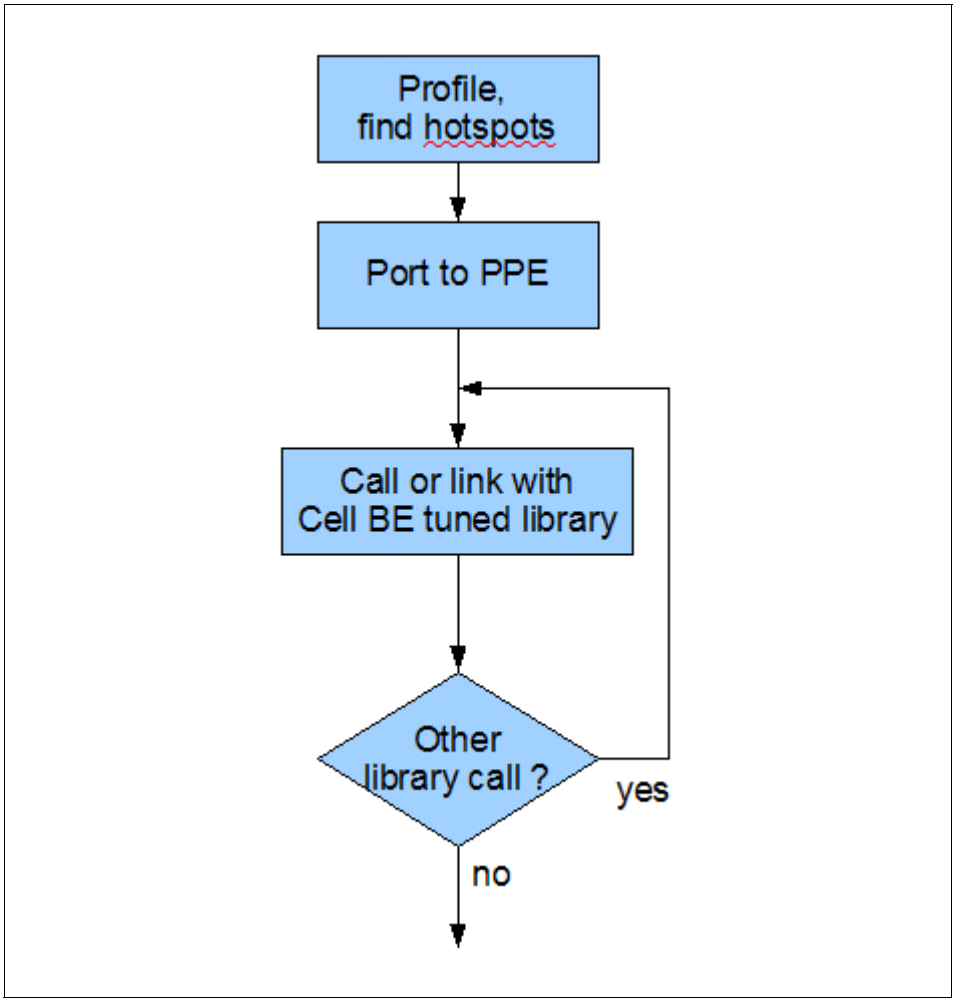


Figure 3-10 Implementing Cell BE tuned workload specific libraries

As for ALF, the process is described in Figure 3-11 on page 64.

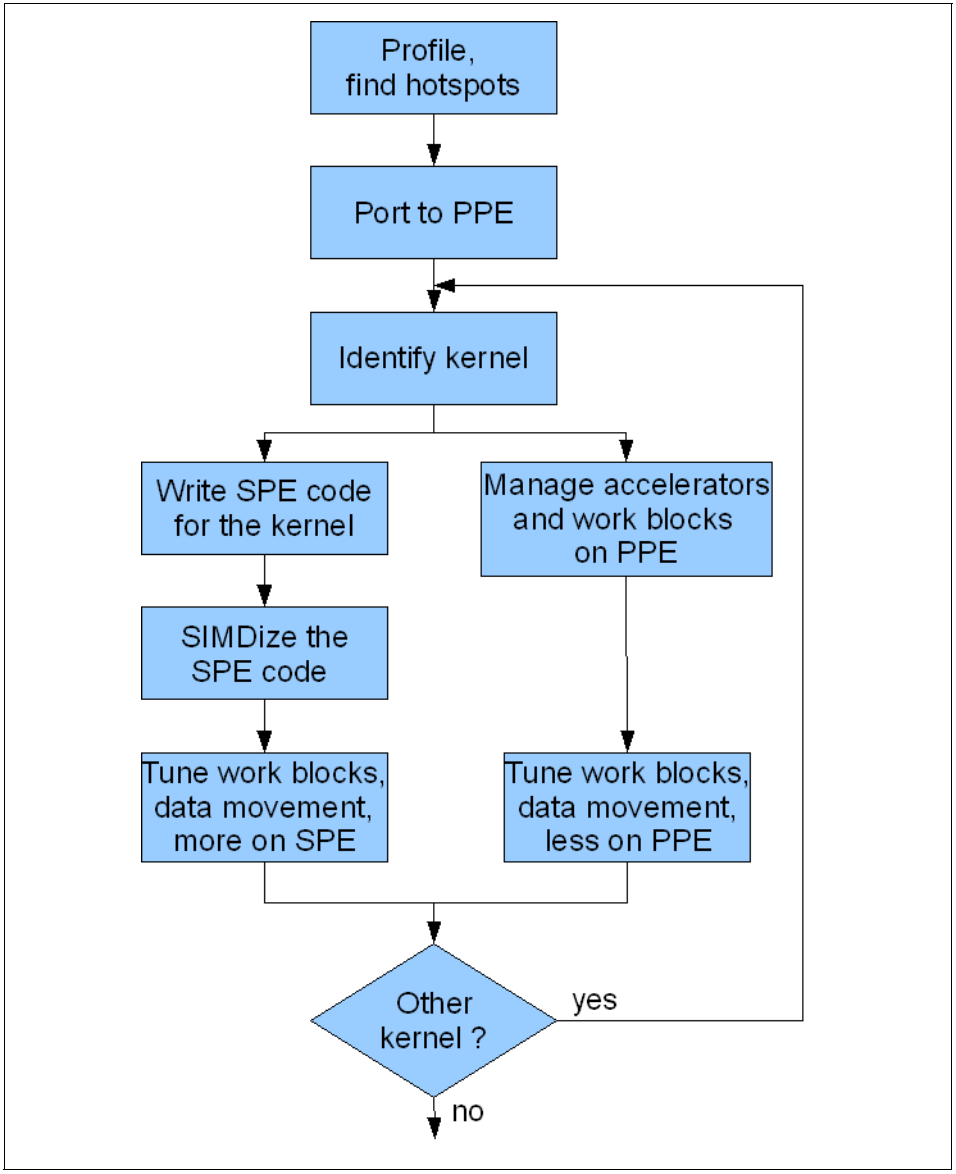


Figure 3-11 Enabling a Cell BE application with ALF

3.5.1 Performance tuning for Cell BE programs

Enabling applications on Cell BE is all about getting the best performance. This does not come for free and performance tuning is an integral part of the application enablement. Chapters “Cell BE programming” on page 75 and

Chapter 7, “Programming in distributed environments” on page 439 give many detailed performance tips for writing good SPE and PPE code. Mike Acton[10] in [25] gives very valuable advice using the experience he and his team at Insomniac Games gathered in the process of developing video games for the Sony Playstation 3 . His recommandations are reproduced below :

- let’s not hide the Cell BE architecture but exploit it instead
- for a succesful port to Cell BE :
  - understand the architecture
  - understand the data : movement, dependencies, generation, usage (read, write, or read-write)
  - do the hard work
- put more work on the SPE, less on the PPE
- do not to view the SPE as co-processors but rather view the PPE as a service provider for the SPE
- ban scalar code on the SPE
- less PPE/SPE synchronization, use deferred updates, lock-free synchonization (see “Shared storage synchronizing and data ordering” on page 213) and perform dataflow management as much as possible from the SPE.

3.6 A few scenarios

Here we review a few examples of Cell BE programs. In Figure 3-12 on page 66, we show the program flow for a typical function offload to a workload library. We picture the PPE thread, running useful work until it calls a function that is part of the Cell BE tuned library. What happens then is that the library will start SPE context and start execute the library code. The library on the SPE could be doing any kind of inter-SPE communication, DMA accesses, etc as figured by the cloud. Once the function has finished executing, the SPE contexts are terminated and the PPE thread resumes execution on the PPE.

[10] http://well.cellperformance.com

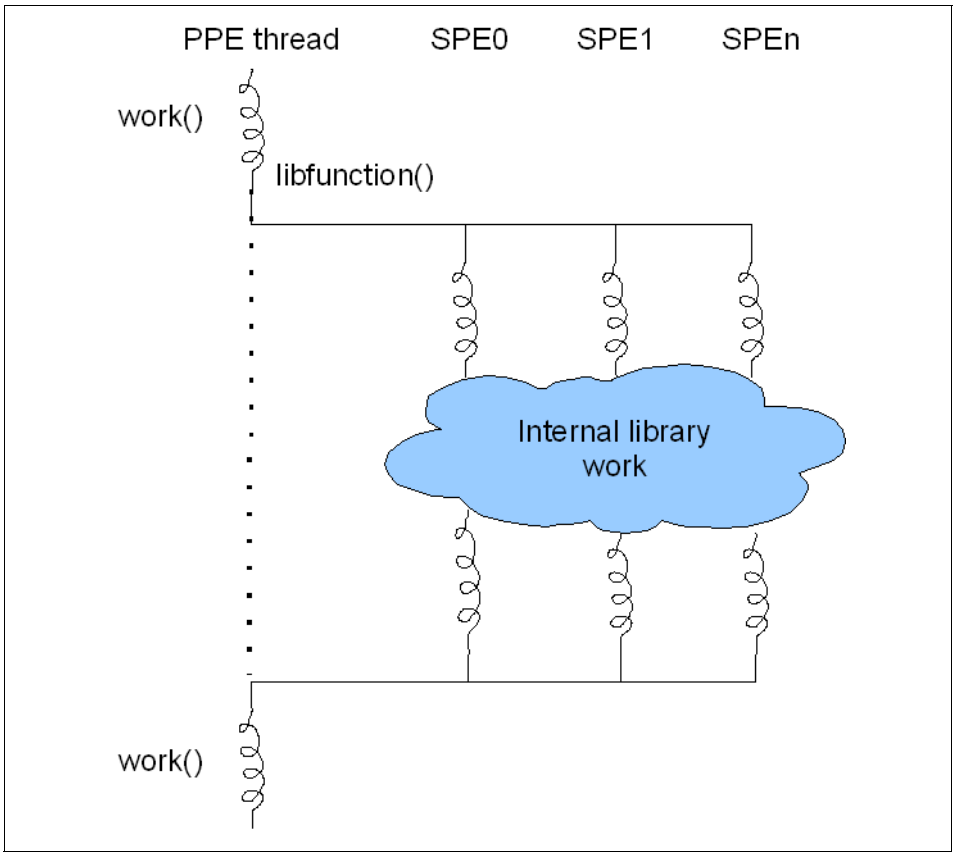


Figure 3-12 A typical work flow for Cell BE tuned workload libraries

Starting and terminating SPE contexts takes some time and we must ensure that the time spent in the library far exceeds the SPE context startup time.

A variation of this scheme is when the application calls the library repeatedly. In this case, it might be interesting to keep the library contexts running on the SPE and just set them to work with a lightweight mailbox operation for example. This is shown in Figure 3-13 on page 67.

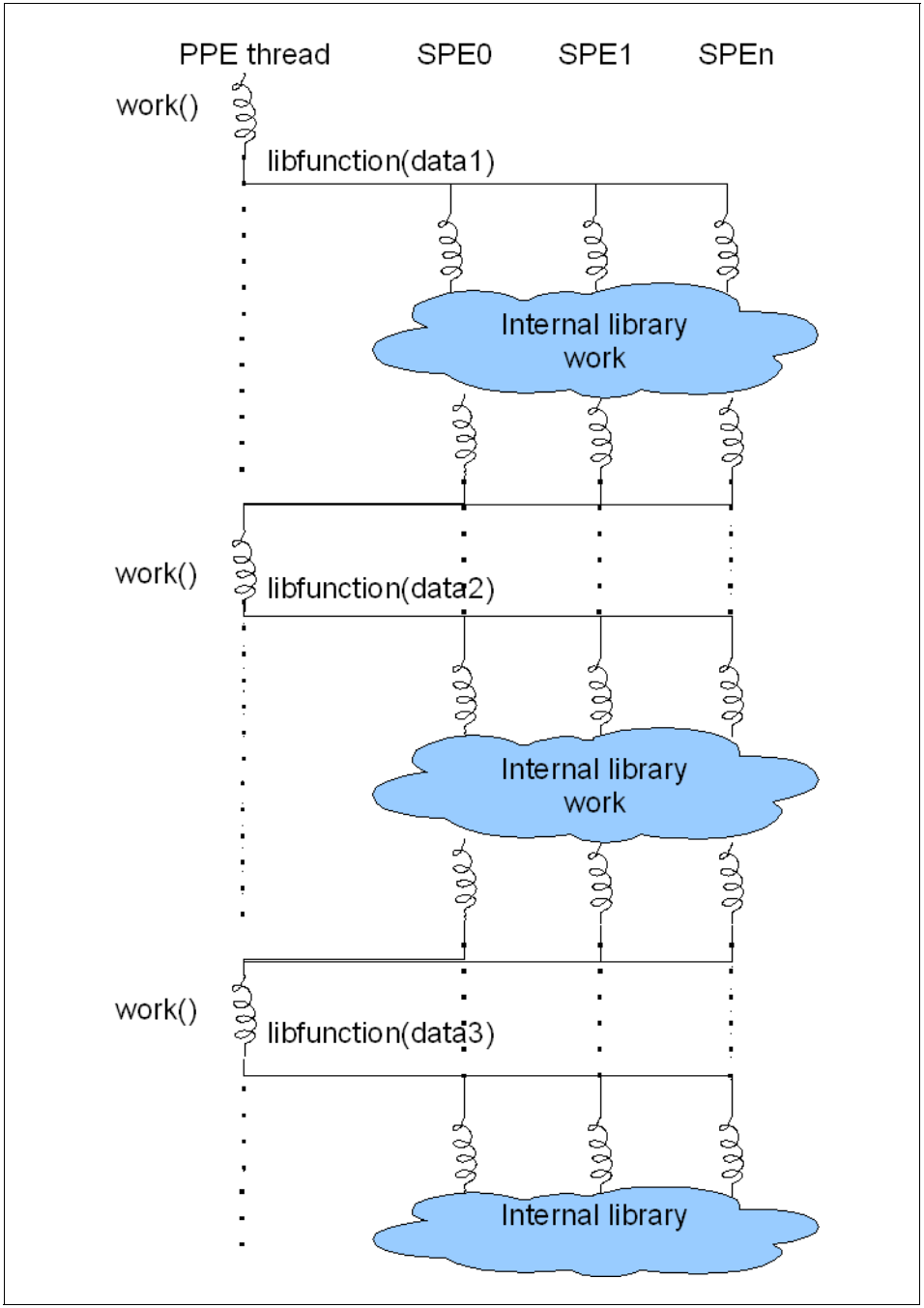


Figure 3-13 Successive invocations of the same library

Here we have three successive invocations of the library with data1, data2 and data3. The dotted lines indicate a SPE context that is active but waiting. This arrangement minimizes the impact of the SPE contexts creation but it can only work if the application has a single computational kernel that is called over and over.

In Figure 3-14, we show the typical workflow of an ALF application. The PPE thread will prepare the work blocks (numbered wb0 to wb12 here), and these will execute on the SPE accelerators.

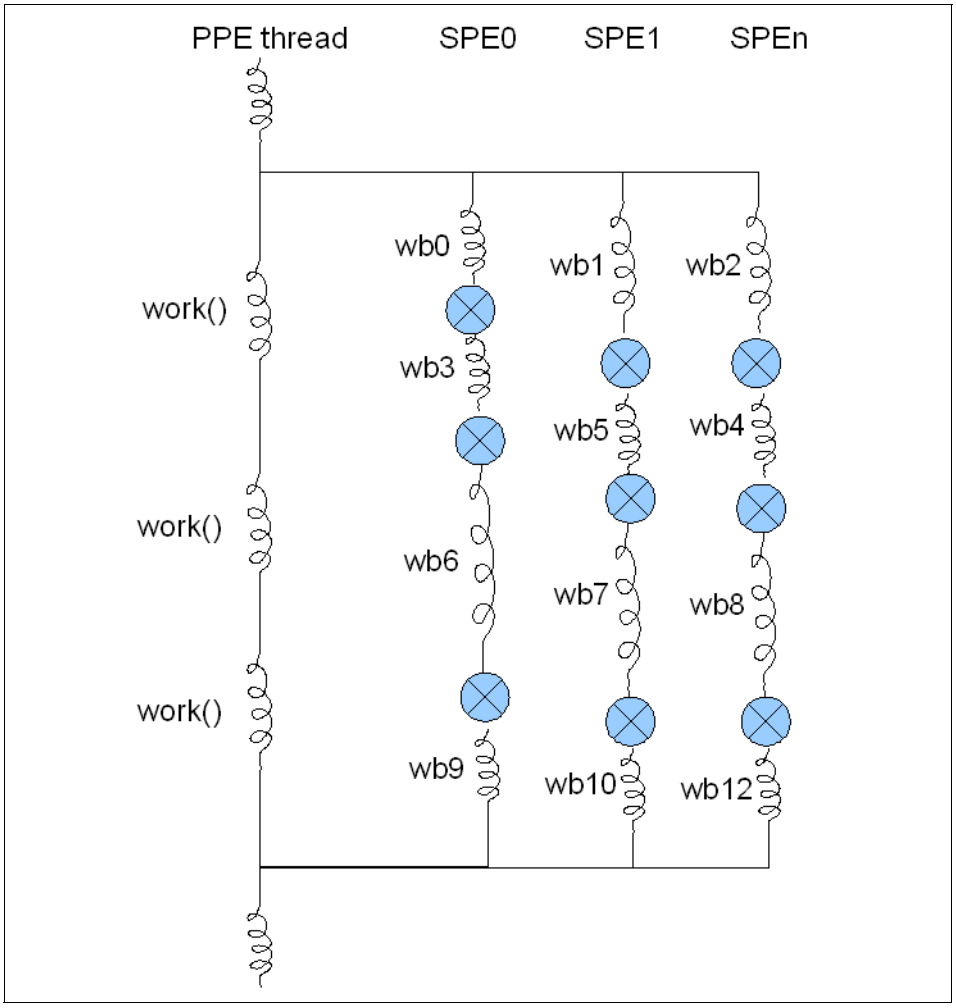


Figure 3-14 The ALF workflow

There is typically no communication between accelerators when they are processing a work block and the ALF runtime takes care of balancing the work among the accelerators. Also, the PPE thread may do useful work while the accelerators are crunching though the work blocks.

3.7 Design patterns for Cell BE programming

Design patterns were first introduced by Christoper Alexander in the field of town and building architecture. Gamma et al. in [3] applied the same principles to the computer programming and this has proved a very useful tool since then. Multiple definitions can be found for a pattern. In [4], Mattson et al. define a pattern as a “good solution to a recurring problem in a particular context”. Marc Snir, in [6], describes them as a “way to encode expertise”. Patterns are usually characterized by :

- a name,
- a problem,
- the forces shaping the solution,
- a solution to the problem

Using the same formalism, we have started to build a list of design patterns applied to Cell BE programming. This is clearly only a start and it is hoped that new patterns will emerge as we gain more and more expertise in porting code to the Cell BE environment.

We will look at five design patterns :

- a shared queue
- indirect addressing
- a pipeline
- multi-SPE software cache
- plugin

3.7.1 Shared queue

We wish to distribute work elements between multiple SPE. They are arranged in a FIFO queue in PPE memory.

Forces

The two main forces are the need for a good load balance betwen the SPE and minimal contention.

Solution

We can envision three solutions for dividing work between SPE. They are described as follows.

Fixed work assignment

Each SPE is statically assigned the same amount of work elements. This incurs no contention but may be a weak scheme if the time taken to process a work element is not constant.

Master/Worker

The PPE assigns the work elements to the SPE. The PPE could give the pieces of work one at a time to the SPE. When a SPE is finished with its piece of work, it signals the PPEwhich then feeds the calling SPE with a new item, automatically balancing the work. This scheme will be good for the load balance but may lead to some contention if many SPE are being used as the PPE may be overwhelmed by the task of assigning the work blocks.

Self managed by the SPE

The SPE will synchronize between themselves without PPE intervention. Once a SPE is finished with its work item, it will grab the next piece of work from the queue and process it. This is the best scheme as it ensures good load balance and does not put any load on the PPE. The critical part of the scheme is to make sure that the SPE remove work items off the queue atomically, possibly using the MFC atomic operations or using features from the sync library provided with the IBM SDK for Multicore Acceleration.

3.7.2 Indirect addressing

We wish to load in SPE memory a vector that is addressed through an index array. This is common in sparse matrix-vector product that arise for example when solving linear systems with conjugate gradients methods. The typical construct is shown in Example 3-1 where the matrix is stored in CSR (Compressed Sparce Row) format. This storage is described in [26].

Example 3-1 Matrix-vector product with a sparse matrix

```
float A[],x[],y[];
int ja[], idx[];
for(i=0;i<N;i++) {
   for(j=ja[i];j<ja[i+1];j++) {
```

```
        y[i]+=A[j]+x[idx[j]];
    }
}
```

Forces

The index array by with the x vector is accessed leads to random memory accesses.

Solution

However, the index array is known in advance and we can exploit this by using software pipelining with a multi buffering scheme and DMA lists. This is described in Figure 3-15.

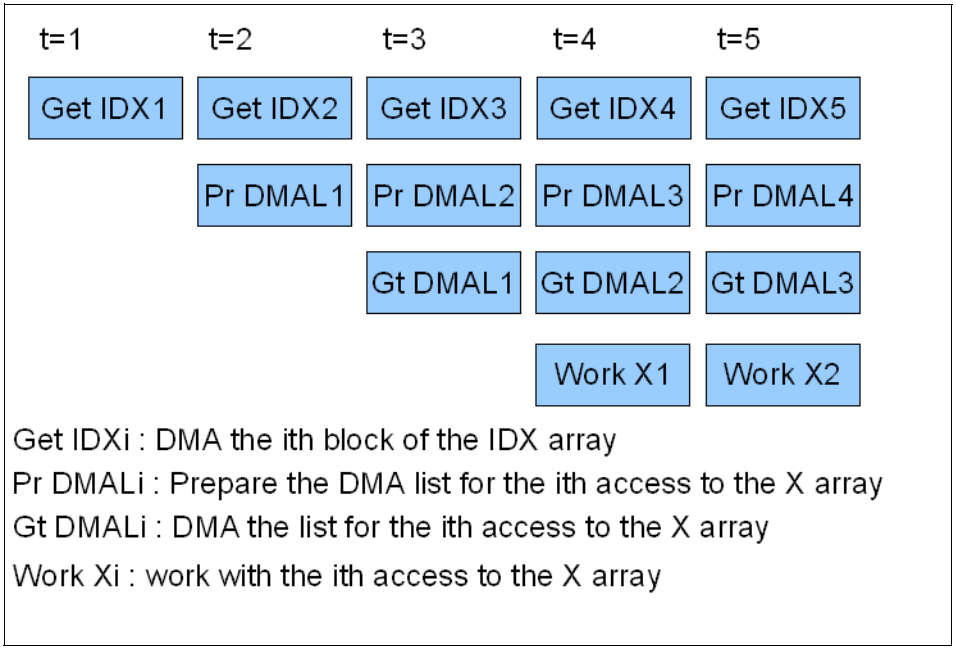


Figure 3-15 Software pipeling and multi-buffering

We do not to show the accesses to the matrix A and the array y. They are accessed sequentially and a simple multi-buffering scheme can also be applied.

3.7.3 Pipeline

We wish to arrange SPE contexts in a multi-stage pipeline manner.

Forces

We want to minimize the time it takes for the data to move from one pipeline stage to the other.

Solution

Using affinity functions as described in “Direct problem state access and LS to LS transfer” on page 143, we can make sure that successive SPE contexts are ideally placed on the EIB to maximize the LS to LS transfer speed. An alternative arrangement can be to move the code instead of the data, whichever is the fastest. Very often, when the programming model is a pipeline, some state data must reside on each stage and moving the function would also require moving the state data.

3.7.4 Multi-SPE software cache

We wish to define a large software cache that gathers Local Storage space from multiple participating SPEs.

Forces

We wish to push the software cache a bit further by allowing data to be cached not necessarily in the SPE that encounters a “miss” but also in other SPE’s Local Store. The idea here is to exploit the very high EIB bandwidth.

Solution

We do not have a solution for this yet. The first direction would be to look at cache coherency protocols (MESI, MOESI, MESIF)[11] in use today on multi processor systems and try to adapt them to the Cell BE.

3.7.5 Plugin

We wish to process data whose contents, and therefore its associated treatment, is discovered on the go.

Forces

This is similar to what happens with a Web browser when the flow of data coming from the Web server contains data that requires the loading of external plugins to be displayed. The challenge here is to be able to load on the SPE both the data and the code to process it as the data is being discovered.

[11] M=Modified, O=Owner, E=Exclusive, S=Shared, I=Invalid, F=Forwarding

Solution

The Cell BE has overlay support already so this could be one solution. However, there might be a better solution to this particular problem using dynamically loaded code. We can imagine loading code together with data using exactly the same DMA functions. Nothing in the SPE memory differentiates code from data. This has been implemented successfully by Eric Christensen et al. in [27]. The process is as follows :

1. compile the code,
2. dump the object code as binary,
3. load the binary code as data,
4. DMA the data (containing the code) just like regular data to the SPE, actual data could also be loaded during the same DMA operation,
5. on the SPE jump to the address location where the code has been loaded to pass the control to the plugin which has just been loaded.

4

Cell BE programming

The goal of this chapter is to provide an introductory guide for how to program an application on Cell BE. The chapter covers many aspects of Cell BE programing, from low level programing using intrinsics to higher level programming using frameworks that hide the processor unique architecture.

The chapter covers issues related to programming a single Cell BE processor or a Cell BE base blade system (e.g. QS21) that contains two Cell BE processors but shares the same operating system and memory map.

When describing the programming techniques we tried to keep a good balance between two opposite and complementary approaches:

- Keep the programing as high level a possible in order to reduce the development time and to produce a code which is as readable and simple as possible. This can be done for example using the SDK's C functions for accessing the different Cell BE hardware mechanisms (DMA, mailboxes, signals, etc.) and abstract high level libraries to manage the work with Cell BE (e.g. DaCS, ALF, software managed cache).
- Use low level programing in sections in the code where performance is critical. This can be done for example using the low level intrinsics which are mapped to a single or small number of assembly instructions.

When describing those techniques we usually emphasize the cases in which each of the approaches is suitable.

In addition, we tried to include a wide range of important issues related to Cell BE programming, that till this point were described in several different documents.

The programming techniques and libraries that are covered in this section are divided into sections according to the functionality within the program that they perform. We hope this approach is useful for the program developer as it enables to find the corresponding subject according to the current stage in development or according to the specific part of the program that is currently implemented.

This chapter is divided into the following sections:

- "Task parallelism and PPE programming" on page 78 - describes how to program the PPE and how to exploit task parallelism by distributing the work between the SPEs.
- "Storage domains, channels and MMIO interfaces" on page 95 discusses the different storage domains on the Cell BE and how either a PPE or SPE program can access them. The section also discusses how to use the MFC which is the main component for communicating between the processors and transferring data using DMA. It is useful to be familiar with this subject when deciding on the program's data partioning or when there is a need to use the MFC (as any Cell BE program does).
- "Data transfer" on page 109 discusses the various methods for performing data transfers in Cell BE between the different available memories. Obviously this is a key issue in any program development.
- "Inter-processor communication" on page 174 describes how to implement communication mechanisms between the different processors that run the Cell BE in parallel (e.g. mailbox, signals, events).
- "Shared storage synchronizing and data ordering" on page 213 discusses how that data transfer of the different processors can be ordered and synchronized. The Cell BE unique memory architecture requires the programer to be aware of this issue which in many cases need to be handled explicitly by the program using dedicated instructions.
- "SPU programming" on page 240 shows how to write an optimized SPU program. The intention here for programing issues related only to programing the SPU itself an without interacting with external components (e.g. PPE, other SPEs, main storage).
- "Frameworks and domain-specific libraries" on page 283 discusses some high level programing frameworks and libraries that aim to reduce the development efforts and hide the Cell BE specific architecture (e.g. DaCS, ALF and domain specific libraries). In some case using those frameworks provide similar performance as programing using the low level libraries.
- "Programming guidelines" on page 313 provides a collection of programming guidelines and tips.

- The section contains information gathered from various resources and also new items that we added.
- It discuss issues that are described in details in other chapters. A reference to the corresponding chapters is also mentioned.
- It may be a good idea for a programmer to read this chapter before starting developing a new application in order to understand the different consideration that need to be taken.

4.1 Task parallelism and PPE programming

The Cell BE has a single PowerPC Processor Element (PPE) which is intended primarily for running operating system, control the application process, managing system resources, and managing SPE threads. Execution of any user program also starts on this processor, and the PPE program may later off-load some of its functionality to run on one or more of the SPEs.

From the programming point of view, managing the work with the SPEs is similar to working with Linux threads, and the SDK contains libraries that assist in managing the code running on the SPE and communicate with this code during execution.

The PPE itself is conforms to the PowerPC Architecture so programs written for the PowerPC 970 processor, for example, should run on the Cell BE processor without modification. In addition, most program that run on a Linux based power system and uses the OS facilities should work properly on a Cell BE based system. Such facilities include accessing the file system, using sockets and MPI for communicate with remote nodes, and managing memory allocation.

It is important for the programmer to know that using the operating system facilities in any Cell BE application always take place on the PPE. While an SPE code may use those facilities, doing so will cause blocking the SPU code and let the PPE handle the system request. Only when the PPE complete handling the request, the SPE execution will continue.

In this section we cover the following topics:

- Chapter 4.1.1, “PPE architecture and PPU programming” on page 79 describes the PPE architecture and instruction set and general issues regarding programming code that runs on the PPU.
- Chapter 4.1.2, “Task parallelism and managing SPE threads” on page 83 discuss how PPU code may implement task parallelism using SPE threads. The section discuss how to create and execute those threads and how to create affinity between groups of threads.
- Chapter 4.1.3, “Creating SPEs affinity using gang” on page 93 discuss how to create affinity between SPE threads the meant to run together.

We include in this book the issues related to PPE programming that we found the most important when running most Cell BE applications. However, in case he reader is interested in learning more about this subject or need to know some specific detail that is not covered in this section, a good starting point to do so may be *PowerPC Processor Element* chapter in Cell Broadband Engine Programming Handbook.

4.1.1 PPE architecture and PPU programming

Programming the PPU is similar to programming any Linux based program that runs on a PowerPC processor system. Some of the key features of the PPE and its PPU instructions set are:

- A general-purpose, dual-threaded, 64-bit RISC processor.
- Conforms to PowerPC Architecture with Vector/SIMD multimedia extensions.
- Uses 32 bits instructions that are word-aligned.
- Dual-threaded.
- Support Vector/SIMD Multimedia Extension 32 bits and word-aligned instructions that works on 128 bits wide operands.
- 32 KB L1 instruction and data caches
- 512 KB L2 unified (instruction and data) cache.
- Cache line is 128 bytes.
- Instructions are executed in order.

The PPU supports two instruction sets: the PowerPC instruction set and the *Vector/SIMD Multimedia Extension instruction set*. In most cases it is preferred to use the eight SPEs to perform the massive SIMD operations and let the PPU program managing the application flow. However, it may be useful in some cases to add some SIMD computation on the PPU.

Although most of the coding for the Cell Broadband Engine will be in a high-level language like C or C++, an understanding of the PPE architecture and PPU instruction sets adds considerably to a developer's ability to produce efficient, optimized code. This is particularly true because C-language internals are provided for some of the PPU's instruction set. The following section discusses the PPU intrinsics (C/C++ language extensions) and how to use them. This section discuss both intrinsics that operate on scalars and also those that operate on vector data type.

C/C++ language extensions (intrinsics)

The intrinsics are essentially inline assembly-language instructions, in the form of function calls, that have syntax familiar to high-level programmers using the C language. The intrinsics provide explicit control of the PPU instructions without directly managing registers and scheduling instructions, as assembly-language programming requires. The compilers that come with the SDK package supports these C-language extensions.

Two main types of PPU intrinsics discussed in the following sections.

Scalar intrinsics

A minimal set of specific intrinsics to make the PPU instruction set accessible from the C programming language. Except for `__setflm`, each of these intrinsics has a one-to-one assembly language mapping, unless compiled for a 32-bit ABI in which the high and low halves of a 64-bit doubleword are maintained in separate registers.

The most useful intrinsics under this category are those related to shared memory access and synchronization and those related to cache management. Efficient use of those intrinsic may assist in improving the overall performance of the application.

The section “PPE ordering instructions” on page 217 discusses some of the more important the intrinsics that are related to the shared memory access and synchronization, such as ‘sync’, ‘lwsync’, ‘eieio’, and ‘isync’.

In addition, some of those scalar instruction provide access to the PPE registers and internal data structures which enables the programmer to use some of the PPE facilities.

All those intrinsics are declared in the `ppu_intrinsics.h` header file that need to be included in order to use those intrinsics. They may be either defined within this header as macros or implemented internally within the compiler.

By default, a call to an intrinsic with an out-of-range literal is reported by the compiler as an error. Compilers may provide an option to issue a warning for out-of-range literal values and use only the specified number of least significant bits for the out-of-range argument.

The intrinsics do not have a specific ordering unless otherwise noted. The intrinsics can be optimized by the compiler and be scheduled like any other instruction.

Additional information about PPU scalar intrinsics can be found in the following resources:

- *PPU Specific Intrinsics* chapter of C/C++ Language Extensions for Cell BE Architecture document - a list of the available intrinsics and their meaning.
- *PPE instruction sets* chapter of the Cell Broadband Engine Programming Tutorial document - a useful table that summarize those intrinsics.

Vector data types intrinsics

A set of intrinsics is provided in order to supports the *Vector/SIMD multimedia extension* (*VMX*) instructions. Those instructions follow the AltiVec™ standard.

The VMX model adds a set of fundamental data types, called vector types. The vector registers are 128 bits and can contain either sixteen 8-bit values (signed or unsigned), eight 16-bit values (signed or unsigned). four 32-bit values (signed or unsigned) or four single-precision IEEE-754 floating-point values.

The vector instructions include a reach set of operations that may be performed on those vectors, including arithmetic operations, rounding and conversion, floating-point estimate intrinsics, compare intrinsics, logical intrinsics, rotate and shift Intrinsics, load and store intrinsics, pack and unpack intrinsics and more.

Vector/SIMD Multimedia Extension data types and Vector/SIMD Multimedia Extension intrinsics can be used in a seamless way throughout a C-language program. The programmer do not need to setup, to enter a special mode. The intrinsics may be either defined as macros within the system header file or implemented internally within the compiler.

In order to use PPU's Vector/SIMD intrinsics the programmer should:

- Include system header file `altivec.h` which define the those intrinsics.
- Set `-qaltivec` and `-qenablevmx` flags in case XLC compilation is used.
- Set `-mabi=altivec` and `-maltivec` flags in case GCC compilation is used.

Example 4-1 demonstrates a simple PPU code that initiates two unsigned integer vectors and add them while putting the results into third similar vector.

Source code: The code of Example 4-1 is included in the additional material that is provided with this book. See "Simple PPU vector/SIMD code" on page 612 for more information.

Example 4-1 Simple PPU Vector/SIMD code

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

typedef union {
   int i[4];
   vector unsigned int v;
} vec_u;

int main()
{
   vec_u a, b, d;

   a.v = (vector unsigned int){1,2,3,4};
   b.v = (vector unsigned int){5,6,7,8};
```

```
   d.v = vec_add(a.v,b.v);

   return 0;
}
```

Additional information about PPU vector data type intrinsics can be found in the following resources:

- *AltiVec Technology Programming Interface Manual* - a detailed description of VMX intrinsics.
- *Vector Multimedia Extension Intrinsics* chapter of C/C++ Language Extensions for Cell BE Architecture document - a list of the available intrinsics and their meaning.
- *PPE instruction sets* chapter of the Cell Broadband Engine Programming Tutorial document - a useful table that summarize those intrinsics.

In most cases it is preferred to use the eight SPEs to perform the massive SIMD operations and let the PPU program managing the application flow. For that practical reason, we didn't discuss the issue of PPU Vector/SIMD operations in detail as we discuss the SPU SIMD instructions (see Chapter 4.6.4, "SIMD programming" on page 253).

However, it may be useful in some application to add some SIMD computation on the PPU. Another case when SIMD operation may take place on the PPU side is when a programmer start the application development on the PPU and optimize it to use SIMD instructions, and only later port the application to the SPU. We don't recommend to use this approach in most cases as it seems to consume more development time.

One of the reason for the additional time is that despite the strong similarity between the PPU's Vector/SIMD instructions set and SPU instruction, those instructions set are different. Most of the PPU Vector/SIMD instructions have equivalent SPU SIMD instructions and vice versus but not all.

SDK also provides as set of header files that aim to minimize the effort when porting PPU program to the SPU and vice versus.

- `vmx2spu.h` - macros and inline functions to map PPU Vector/SIMD intrinsics to generic SPU intrinsics.
- `spu2vmx.h` - macros and inline functions to map generic SPU intrinsics to PPU Vector/SIMD intrinsics.

- `vec_types.h` - a SPU header file which defines a set of single token vector data types that are available on both the PPU and SPU. The SDK3.0 provides both GCC and XLC versions of this header file.

In case the programmer would like to read more about this issue we recommend to read *SPU and PPU Vector Multimedia Extension Intrinsics* chapter and *Header Files* chapter in C/C++ Language Extensions for Cell BE Architecture document.

While the Vector/SIMD intrinsics contains various basic mathematical functions that are implemented by corresponding SIMD assembly instructions, more complex mathematical functions are not supported by those intrinsics. The SIMDmath library is provided in the SDK and address this issue by providing a set of functions that extend the SIMD intrinsics and support additional common mathematical functions. Similar to SIMD intrinsics, the library operates on short 128 bits vectors from different types.

The SIMDmath library is supported both by SPU and the PPU. The SPU version of this library is discussed in Chapter , “SIMDmath library” on page 257. The PPU version is similar, but the location of the library files are different:

- `simdmath.h` file is located in the `/usr/spu/include` directory
- inline headers are located in the `/usr/spu/include/simdmath` directory
- the library `libsimdmath.a` is located in the `/usr/spu/lib` directory.

4.1.2 Task parallelism and managing SPE threads

Programs running on the Cell BE typically partition the work among the eight available SPE as each SPE is assigned with a different task and data to work on. We suggest several programming models how to partition the work between the SPEs in Chapter 3.3, “Which parallel programming model ?” on page 51.

However, regardless the programming model, the main thread of the program is executed on the PPE which create sub-threads that run on the SPEs and off-load some function of the main program (to be run on the SPEs). It depends on the programming model how later the threads and tasks are managed, how the data is transferred and how the different processors communicate.

Managing the code running on the SPEs on a Cell BE based system can be done using the *libspe library* (*SPE runtime management library*) that is part of the SDK package. This library provides standardized low-level application programming interface (API) that enables application access the SPEs and run some of the program threads on those SPEs.

In general, applications running on the Cell BE do not have control over the physical SPE system resources as the operating system manages these resources. Instead, applications manage and use software constructs called SPE contexts. These SPE contexts are a logical representation of an SPE and holds all persistent information about a logical SPE. The libspe library operates on those contexts to manage the SPEs but the programmer should not access those objects directly.

The operating system schedules SPE contexts from all running applications onto the physical SPE resources in the system for execution according to the scheduling priorities and policies associated with the runable SPE contexts.

Note: Re-scheduling SPE and performing the context switching usually requires a fair amount of time as it required to store most of the 256 KB of the local store in memory and reload it with the code and data of the new thread. It is therefore recommended that application will not allow to do so by not allocating more SPE threads then the number of physical SPEs that are currently available (8 for a single Cell BE, 16 for QS20 or QS21 blade).

The programer is advised to run the SPE contexts on separate Linux thread which enables the operating system to actually run them parallel compare to the PPE threads and parallel compare to other SPEs.

SPE Runtime Management library document contains a detailed description of the API for managing the SPE threads. The library also implements API which provides the means for communication and data transfer between PPE threads and SPEs. For more information see 4.3, “Data transfer” on page 109 and 4.4, “Inter-processor communication” on page 174.

When creating SPE thread, similar to Linux’s threads, the PPE program may pass up to three parameters to this function. The parameters may be either 64 bits parameters or 128 bits vectors. Those parameter may be later used by the code running on the SPE. One common use is to place in those parameters an effective address of a control block that may be larger and contains additional information. The SPE can use this address to fetch this control block into its local store memory.

There are two main methods to load SPE programs:

1. *Static loading of SPE object*: statically compile the SPE object within the PPE program. At run time, the object is can be accessed as an external pointer that can be used by the programer to load the program into the local store. The loading itself is implemented internally by the library API using DMA.
2. *Dynamic loading of SPE executable*: compile the SPE as stand alone application. At run time open the executable file, map it into the main memory

and then load it into the SPE's local store. This method is more flexible as it allow to decide on run time which program to load (e.g. depends on run time parameters). Using this method saves linking the SPE program with the PPE program at the cost of lost encapsulation such that the program is now a set of files, not just a single executable

The following sections provides more information about the following subjects:

- "Running a single SPE program" on page 85 describe how to run code on a single SPE using static loading of SPE object.
- "Producing a multi-threaded program using the SPEs" on page 89 describes how to run code on multiple SPEs concurrently using dynamic loading of SPE executable.

Running a single SPE program

This chapter describes how the user may run code on a single SPE. In this example no Linux's threads are used, so the PPE program blocks until the SPE stops executing and the operating system returns from the system call that invoked the SPE execution.

Example 4-2 covers the following topics for the PPU code, ordered according to the same steps as executed in the code:

1. Initiate a control structure to point to input and output data buffers and initiate SPU executable's parameter to point to this structure (step 1in the code).
2. Create the SPE context using `spe_context_create` function.
3. Statically load the SPE object into the SPE context local store using `spe_program_load` function.
4. Run the SPE context using `spe_context_run` function.
5. Optionally print the reason why the SPE stopped (obviously end of its main function with return code 0 is the preferred one).
6. Destroy the SPE context using `spe_context_destroy` function.

The example covers the following topics for the SPU code:

- Use the parameters that the PPU code initiate in order to get the address a control block, and get the control block from main storage to local store.

Example 4-3 on page 86 show the PPU code, Example 4-4 on page 88 shows the SPU code while Example 4-2 on page 86 shows the common header file.

Please note that the `libspe2.h` header file should be included in order to run the SPE program.

Source code: The code in Example 4-2, Example 4-3, and Example 4-4 is included in the additional material that is provided with this book. See “Running a single SPE” on page 612 for more information.

Example 4-2 Running a single SPE - shared header file

```
// ====================================================================
// common.h file
// ====================================================================
#ifndef _COMMON_H_
#define _COMMON_H_

#define BUFF_SIZE 256

// the context that PPE forward to SPE
typedef struct{
   uint64_t ea_in; // effective address of input buffer
   uint64_t ea_out; // effective address of output buffer

} parm_context; // aligned to 16B

#endif // _COMMON_H_
```

Example 4-3 Running a single SPE - PPU code

```
#include <libspe2.h>

#include "common.h"

spe_program_handle_t spu_main; // a pointer to SPE object
spe_context_ptr_t spe_ctx;     // SPE context

// data structures to work with the SPE
//====================================================================
volatile parm_context ctx __attribute__ ((aligned(16)));
volatile char in_data[BUFF_SIZE]  __attribute__ ((aligned(128)));
volatile char out_data[BUFF_SIZE] __attribute__ ((aligned(128)));

// function for printing the reason for SPE thread to stop
// ===================================================================
void print_stop_reason( spe_stop_info_t *stop_info ){
```

```
   // result is a union that holds the SPE output result
   int result=stop_info->result.spe_exit_code;

    switch (stop_info->stop_reason) {
    case SPE_EXIT:
        printf(")PPE: SPE stop_reason=SPE_EXIT, exit_code=");
        break;
    case SPE_STOP_AND_SIGNAL:
        printf(")PPE: SPE stop_reason=SPE_STOP_AND_SIGNAL,
signal_code=");
        break;
    case SPE_RUNTIME_ERROR:
        printf(")PPE: SPE stop_reason=SPE_RUNTIME_ERROR,
runtime_error=");
        break;
    case SPE_RUNTIME_EXCEPTION:
        printf(")PPE: SPE stop_reason=SPE_RUNTIME_EXCEPTION,
runtime_exception=");
        break;
    case SPE_RUNTIME_FATAL:
        printf(")PPE: SPE stop_reason=SPE_RUNTIME_FATAL,
runtime_fatal=");
        break;
    case SPE_CALLBACK_ERROR:
        printf(")PPE: SPE stop_reason=SPE_CALLBACK_ERROR
callback_error=");
        break;
    default:
        printf(")PPE: SPE stop_reason=UNKNOWN, result=\n");
        break;
    }
    printf("%d, status=%d\n",result,stop_info->spu_status);
}

// main
//=============================================================================
int main( )
{
    spe_stop_info_t stop_info;
   uint32_t entry = SPE_DEFAULT_ENTRY;

   // STEP 1: initiate SPE control structure
   ctx.ea_in    = (uint64_t)in_data;
   ctx.ea_out   = (uint64_t)out_data;
```

```
   // STEP 2: create SPE context
   if ((spe_ctx = spe_context_create (0, NULL)) == NULL){
      perror("Failed creating context"); exit(1);
   }

   // STEP 3: Load SPE object into SPE context local store
   //         (SPU's executable file name is 'spu_main'.
   if (spe_program_load(spe_ctx, &spu_main)) {
      perror("Failed loading program"); exit(1);
   }

  // STEP 4: Run the SPE context (see 'spu_pthread' function above
  //         Note: this a synchronous call to the operating system
  //               which blocks until the SPE stops executing and the
  //               operating system returns from the system call that
  //               invoked the SPE execution.
  if(spe_context_run(spe_ctx,&entry,0,(void*)&ctx,NULL,&stop_info)<0){
      perror ("Failed running context"); exit (1);
   }

   // STEP 5: Optionally print the SPE thread stop reason
   print_stop_reason( &stop_info );

   // STEP 6: destroy the SPE context
   if (spe_context_destroy( spe_ctx   )) {
      perror("Failed spe_context_destroy"); exit(1);
   }
   return (0);
}
```

Example 4-4 Running a single SPE - SPU code

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

#include "common.h"

static parm_context ctx __attribute__ ((aligned (128)));

volatile char in_data[BUFF_SIZE]  __attribute__ ((aligned(128)));
volatile char out_data[BUFF_SIZE] __attribute__ ((aligned(128)));

int main(int speid , uint64_t argp)
```

```
{

   uint32_t tag_id;

   //STEP 1: reserve tag IDs
   if((tag_id=mfc_tag_reserve())==MFC_TAG_INVALID){ // allocate tag
      printf("SPE: ERROR - can't reserve a tag ID\n"); return 1;
   }

   //STEP 2: get context information from system memory.
   mfc_get((void*) &ctx, argp, sizeof(ctx), tag_id, 0, 0);
   mfc_write_tag_mask(1<<tag_id);
   mfc_read_tag_status_all();

   //STEP 3: get input buffer, process it, and put results in output
   //         buffer

   //STEP 4: release tag IDs
   mfc_tag_release(tag_id); // release tag ID before exiting
   return 0;
}
```

Producing a multi-threaded program using the SPEs

In order to get the best performance out of an application running on Cell BE, it is usually recommended to use multiple SPEs concurrently. In this case, the application must create at least as many threads as concurrent SPE contexts are required. Each of these threads may run a single SPE context at a time. If N concurrent SPE contexts are needed, it is common to have a main application thread plus N threads dedicated to SPE context execution

This chapter describes how the user may run code on a multiple SPEs concurrent using Linux's threads. We use a specific scheme which is the most common one for Cell BE programming, but depending on the specific application the programmer may use any other scheme.

The code example in this chapter execute two SPE threads and covers the following topics:

- Initiate SPEs control structures.
- Dynamically loading of SPE executable into several SPEs:
  - Create SPE contexts.
  - Open images of SPE programs and map them into main storage.
  - Load SPEs objects into SPE context local store

- Initiate Linux's thread and run the SPE executable concurrently on those threads. The PPU forward parameters the SPU programs.

Example 4-5 on page 90 show the PPU code, Example 4-6 on page 92 shows the SPU code. The common header file is the same as in Chapter , "Running a single SPE program" on page 85 and shown in Example 4-2 on page 86.

Please note that the `libspe2.h` header file should be included in order to run the SPE programs, and also `pthread.h` should be included to use Linux's threads.

Source code: The code of Example 4-5 and Example 4-6 is included in the additional material that is provided with this book. See "Running multiple SPEs concurrently" on page 613 for more information.

Example 4-5 Running multiple SPEs concurrently - PPU code

```
// ppu_main.c file ================================================
#include <libspe2.h>
#include <cbe_mfc.h>
#include <pthread.h>

#include "common.h"

#define NUM_SPES 2

// input and output data buffers
volatile char in_data[BUFF_SIZE]  __attribute__ ((aligned(128)));
volatile char out_data[BUFF_SIZE] __attribute__ ((aligned(128)));

// Data structures to work with the SPE
volatile parm_context ctx[NUM_SPES] __attribute__ ((aligned(16)));
spe_program_handle_t *program[BUFF_SIZE];

// data structure for running SPE thread ==============================
typedef struct spu_data {
  spe_context_ptr_t spe_ctx;
  pthread_t pthread;
  void *argp;
} spu_data_t;

spu_data_t data[NUM_SPES];

// create and run one SPE thread ======================================
void *spu_pthread(void *arg) {
```

```
   spu_data_t *datp = (spu_data_t *)arg;
   uint32_t entry = SPE_DEFAULT_ENTRY;

   if(spe_context_run(datp->spe_ctx,&entry,0,datp->argp,NULL,NULL)<0){
      perror ("Failed running context"); exit (1);
   }

   pthread_exit(NULL);
}

// main ================================================================
int main( )
{
   int num;

   // names of the two SPU executable file names
   char spe_names[2][20] = {"spu1/spu_main1","spu2/spu_main2"};

   // STEP 1: initiate SPEs control structures
   for( num=0; num<NUM_SPES; num++){
      ctx[num].ea_in = (uint64_t)in_data  + num*(BUFF_SIZE/NUM_SPES);
      ctx[num].ea_out= (uint64_t)out_data + num*(BUFF_SIZE/NUM_SPES);
      data[num].argp = &ctx;
   }

   // Loop on all SPEs and for each perform two steps:
   // STEP 2: create SPE context
   // STEP 3: open images of SPE programs into main storage
   //         'spe_names' variable store the executable name
   // STEP 4: Load SPEs objects into SPE context local store
   for( num=0; num<NUM_SPES; num++){
      if ((data[num].spe_ctx = spe_context_create (0, NULL)) == NULL) {
         perror("Failed creating context"); exit(1);
      }
      if (!(program[num] = spe_image_open(&spe_names[num][0]))) {
         perror("Fail opening image"); exit(1);
      }
      if (spe_program_load ( data[num].spe_ctx, program[num])) {
         perror("Failed loading program"); exit(1);
      }
   }

   // STEP 5: create SPE pthreads
   for( num=0; num<NUM_SPES; num++){
```

```
      if(pthread_create(&data[num].pthread,NULL,&spu_pthread,
         &data[num ])){
         perror("Failed creating thread");  exit(1);
      }
   }

   // Loop on all SPEs and for each perform two steps:
   //   STEP 6: wait for all the SPE pthread to complete
   //   STEP 7: destroy the SPE contexts
   for( num=0; num<NUM_SPES; num++){
      if (pthread_join (data[num].pthread, NULL)) {
         perror("Failed joining thread"); exit (1);
      }

      if (spe_context_destroy( data[num].spe_ctx   )) {
         perror("Failed spe_context_destroy"); exit(1);
      }
   }
   printf(")PPE:) Complete running all super-fast SPEs\n");
   return (0);
}
```

Example 4-6 Running multiple SPEs concurrently - SPU code version 1

```
// spu_main1.c file ==================================================
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include "common.h"

static parm_context ctx __attribute__ ((aligned (128)));

volatile char in_data[BUFF_SIZE]  __attribute__ ((aligned(128)));
volatile char out_data[BUFF_SIZE] __attribute__ ((aligned(128)));


int main(int speid , uint64_t argp)
{
   uint32_t tag_id;

   if((tag_id=mfc_tag_reserve())==MFC_TAG_INVALID){ // allocate tag
      printf("SPE: ERROR - can't reserve a tag ID\n"); return 1;
   }
```

```
   // get context information from system memory.
   mfc_get((void*) &ctx, argp, sizeof(ctx), tag_id, 0, 0);
   mfc_write_tag_mask(1<<tag_id);
   mfc_read_tag_status_all();

   printf("<SPE: Harel Rauch joyfully sleeps on the coach\n" );

   // get input, process it using method A, and put results in output

   mfc_tag_release(tag_id); // release tag ID before exiting
   return 0;
}
```

Example 4-7 Running multiple SPEs concurrently - SPU code version 2

```
// spu_main2.c file ================================================

// same variables and include as Example 4-6 on page 92

int main(int speid , uint64_t argp)
{
   // same prefix as Example 4-4 on page 88

   printf("<SPE: Addie Dvir would like to fly here.\n" );
   // get input, process it using method A, and put results in output

   mfc_tag_release(tag_id); // release tag ID before exiting
   return 0;
}
```

4.1.3 Creating SPEs affinity using gang

The libspe library enables the programmer to create gang, which is group of SPE contexts which should be executed together with certain properties. The mechanism enables to create SPE to SPE affinity, which means allowing a certain SPE context to be created and placed next to another previously created SPE context (affinity is always is always specified for pairs).

The SPEs scheduler, which is responsible to map the SPE logical context to physical SPE, honors this relationship by trying schedule the SPE contexts on physically adjacent SPUs. It depends on the current status of the system if he will be able to do so. If the PPE program tries to create such affinity when there are

no other code running on the SPEs (in this program or other program) there is no reason the schedule will not succeed doing so.

Using the SPE to SPE affinity can create performance advantages in some cases. The performance gain is based mainly on the following characteristics of the Cell BE architecture and systems:

1. On a Cell BE based SMP system, such as a Bladecenter QS21, communication between SPEs which are located on the same Cell BE are more efficient than data transfer between SPEs that are located on different Cell BE chips. This includes both data transfer (e.g. LS to LS) and other types of communication (e.g. mailbox and signals).
2. Similarly to #1 above, but on the same chip, communication between SPEs which are adjacent on the local EIB bus is more efficient then between SPEs which are not adjacent.

Given those characteristics, in case massive SPE to SPE communication is use it is recommended to physically locate specific SPEs next to each other.

Example 4-8 show a PPU code that creates such chain of SPEs. This example is inspired by the SDK code example named `dmabench` that is located in `/opt/cell/sdk/src/benchmarks/dma` directory.

Note: This example aims only to demonstrate how to create a chain of SPEs which are physically located one next to the other. SPEs pipeline which is based on this structure (e.g. each SPE execute DMA trasnfers from the local store of the previous SPE on the chain) will NOT provide the optimal results since only half of the EIB rings will be used (so half of the bandwidth is lost). On the other hand, once the physical location of the SPEs is known (using the affinity methods) the programer may use this information to locate the SPEs elsewhere on the SPE pipeline.
The article “Cell Broadband Engine Architecture and its first implementation - A performance view” provides information on the bandwidth that was measured for some SPE-to-SPE DMA transfers, which may be useful when deciding how to locate the SPEs related to each other on a given algorithm.

Example 4-8 PPU code for creating SPE physical chain using affinity

```
// take include files, ‘spu_data_t’ structure and the ‘spu_pthread’
// function from Example 4-5 on page 90

spe_gang_context_ptr_t gang;
spe_context_ptr_t ctx[NUM_SPES];

int main( )
```

```
{
   int i;

   gang = NULL;

   // create a gang
   if ((gang = spe_gang_context_create(0))==NULL) {
      perror("Failed spe_gang_context_create"); exit(1);
   }

   // create SPE contexts as part of the gang which preserve affinity
   // between each SPE pair.
   // SPEs' affinity is based on a chain architecture such as SPE[i]
   // and SPE[i+1] are physically adjacent.
   for (i=0; i<NUM_SPES; i++) {
      ctx[i]=spe_context_create_affinity(0,(i==0)?NULL:ctx[i-1],gang));

      if(ctx[i]==NULL){
         perror("Failed spe_context_create_affinity"); exit(1);
      }

   // ... Omitted section:
   // creates SPE contexts, load the program to the local stores,
   // run the SPE threads, and waits for SPE threads to complete.

   // (the entire source code for this example is comes with the book's
   // additional material).

   // See also section 4.1.2, "Task parallelism and managing SPE
threads"
}
```

4.2 Storage domains, channels and MMIO interfaces

This chapter describe the main storage domains of the Cell BE architecture. Cell BE has a unique memory architecture and understanding the those domains is a key issue in order to know how to program Cell BE application and how the data may be partitioned and transferred in such application. The storage domain is discussed in the first chapter - "Storage domains".

MFC is a hardware component that implements most of the Cell BE's inter-processor communication mechanism including the most significant means

to initiate data transfer - DMA data transfers. While located in each of the SPEs, the MFCs interfaces may be accessed by both program running on a SPU or a program running on the PPU. The MFC is discussed in the next three chapters:

- "MFC channels and MMIO interfaces and queues" on page 98 discuss the main features of the MFC and the two main interfaces is has with the programs (channels interface and MMIO interface).
- "SPU programming methods to access MFC's channel interface" discuss the programming methods for accessing the MFC interfaces and initiate its mechanisms from a SPU program.
- "PPU programming methods to access MFC's MMIO interface" discuss the programming methods for accessing the MFC interfaces and initiate its mechanisms from a PPU program.

While this chapter discussed the MFC interfaces and programming methods to program it, using the MFC mechanisms is described in other chapters:

- DMA data transfers and synchronization of data transfers is discussed in Chapter 4.3, "Data transfer" on page 109
- Communication mechanism between the different processors (PPE, SPEs) such as mailbox, signals and events, are discussed in Chapter 4.4, "Inter-processor communication" on page 174

4.2.1 Storage domains

Cell BE architecture defines three types of storage domains are defined in the Cell BE chip: one main-storage domain, eight SPE LS domains, and eight SPE channel domains. Figure 4-1 illustrates the storage domains and interfaces in Cell BE.

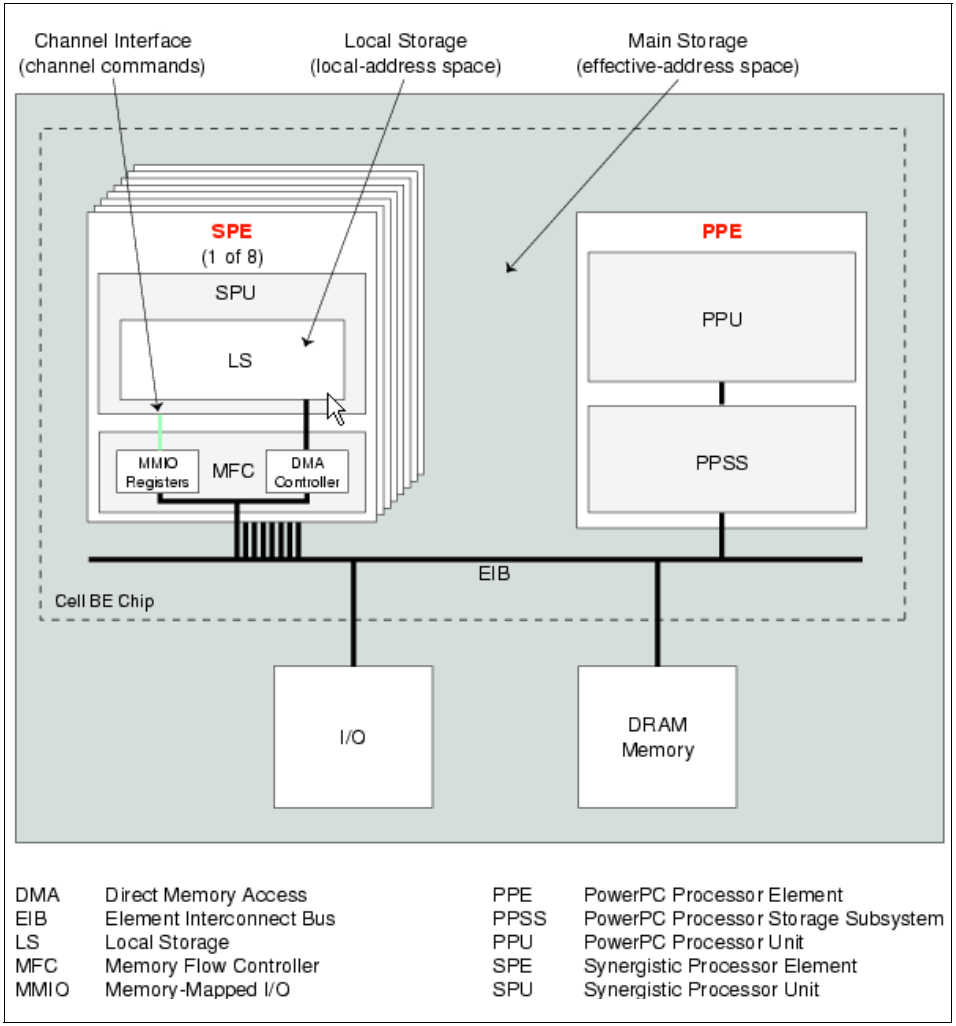


Figure 4-1 Cell BE storage domains and interfaces

The main-storage domain, which is the entire effective-address space, can be configured by the PPE operating system to be shared by all processors in the system. On the other hand, the local-storage and channel problem-state (user-state) domains are private to the SPE components. The main components in each SPE are the SPU, the LS and the *Memory Flow controller* (*MFC*) which handles the DMA data transfer.

Note: In this document we use the term *main storage* to describe any component that have an effective address mapping on the main storage domain.

An SPE program references its own LS using a *Local Store Address* (*LSA*). The LS of each SPE is also assigned a *Real Address* (*RA*) range within the system's memory map. This allows privileged software on the PPE to map LS areas into the *Effective Address* (*EA*) space, where the PPE, other SPEs, and other devices that generate EAs can access the LS like any regular component on the main storage.

A code that runs on an SPU can only fetch instructions from its own LS, and loads and stores can only access that LS.

Data transfers between the SPE's LS and main storage are primarily executed using DMA transfers controlled by the MFC DMA controller for that SPE. Each SPE's MFC serves as a data-transfer engine. DMA transfer requests contain both an LSA and an EA. Thus, they can address both an SPE's LS and main storage and thereby initiate DMA transfers between the domains. The MFC accomplishes this by maintaining and processing an MFC command queue.

The fact that the local stores may be mapped to the main storage, allows SPEs to use DMA operations to directly transfer data between their LS to another SPE’s LS. This mode of data transfer is very efficient, because the DMA transfers go directly from SPE to SPE on the high performance local bus without involving the system memory.

4.2.2 MFC channels and MMIO interfaces and queues

Each MFC have two main interfaces though which MFC commands may be initiated:

1. *Channels interface* - SPU can use this interface to interact with the associated MFC by executing a series of writes or reads to the various channels which in response enqueue MFC commands.
   Since accessing the channel remains local within a certain SPE it have low latency (for non blocking commands about 6 cycles if channel is not full) and also doesn’t have any negative influence EIB bandwidth.
2. *MMIO interface* - PPE or other SPUs can use this interface to interact with any MFC by accessing the MFC’s Command-Parameter Registers. Those registers can be mapped to the system's real-address space so the PPE or SPUs may access them by executing MMIO reads and writes to the corresponding effective address.

For detailed description of the Channels and MMIO interfaces see *SPE Channel and Related MMIO Interface* chapter in Cell Broadband Engine Programming Handbook.

Accessing those two interfaces insert commands into one of the two MFC independent command queues:

- Channels interface is associated with MFC SPU command queue
- MMIO interface is associated with MFC Proxy command queue.

Regarding the channels interface, each channel may be defined as either blocking or non-blocking. When SPE reads or writes a non-blocking channel, the operation executes without delay. However, when SPE software reads or writes a blocking channel, the SPE might stall for an arbitrary length if the associated channel count (which is its remaining capacity) is ‘0’. In this case, the SPE will remain stalled until the channel count becomes ‘1’ or more.

The stalling mechanism reduces SPE software complexity and also allows an SPE to minimize the power consumed by message based synchronization. To avoid stalling on access to a blocking channel, SPE software can read the channel count to determine the available channel capacity. In addition, many of the channels have a corresponding and independent event that can be enabled to cause an asynchronous interrupt.

Accessing the MMIO interface on the other hand is always non-blocking. If a PPE (or other SPE) write a command while the queue is full then the last entry in the queue is override with no indication to the software. Therefore, the PPE (or other SPE) should first verify if there is available space in the queue by reading the queue status register and only if it is not full - write a command to it. The programer should be aware that waiting for available space by continuously reading this register in a loop have negative affect on the performance of the entire chip as it involve transactions on the local EIB bus.
Similarly, reading from a MMIO register when a queue is empty returns an invalid data. Therefore, the PPE (or other SPE) should first read the corresponding status register and only if there is a valid entry (queue is not empty) the MMIO register itself should be read.

Table 4-1 summarizes the main attributes of MFC’s two main interfaces.

Table 4-1 MFC interfaces

Interface	Queue	Initiator	Blocking	Full	Description
Channels	MFC SPU Command Queue	Local SPU	blocking or non blocking	wait till queue has available entry	For MFC commands sent from the SPU through the channel interface.
MMIO	MFC Proxy Command Queue	PPE or other SPEs	always non blocking	overwrite last entry	For MFC commands sent from the PPE, other SPUs, or other devices through the MMIO registers.

4.2.3 SPU programming methods to access MFC's channel interface

Software running on a SPU may access the MFC facilities through the channel interface. This chapter discuss the four main programing methods to access this interface, as listed from the most abstract to the most low level one:

1. MFC functions
2. Composite intrinsics
3. Low level intrinsics
4. Assembly-language instructions

Note: The simplest way programming point of view to access the DMA mechanism is through the first option - the MFC functions. Therefore. most of the examples in this document, besides the examples in this chapter, are written using MFC functions. However, from performance point of view, using the MFC functions will not always provide the best results, especially when invoked from a PPE program.

Many code examples in the SDK package also uses this method. However, the examples in the SDK documentation rely mostly on the next two methods - composite intrinsics and low level intrinsics. Many such examples are available in Cell Broadband Engine Programming Tutorial document.

In this chapter we illustrate the differences between the four methods using issuing of a DMA 'get' command which moves data from some component on main storage to local storage. This is done only for demonstration and similar implementation may be performed for using each of the other MFC facilities

(mailboxes, signals, events, etc.). In case the reader is not familiar with the MFC DMA commands, it may first go over the chapter the discuss this issue, “Data transfer” on page 109, before continuing with current chapter.

There are some parameters that are common to all DMA transfer commands. Those parameters are described in Table 4-2:

Table 4-2 DMA transfer parameters

Name	Type	Description
lsa	void*	local-storage address
► ea or ► eah ► eal	► uint64_t or ► uint32_t ► uint32_t	► effective address in main storage [a]. or ► effective addr. higher bits in main storage[b]. ► effective addr. lower bits in main storagemult_.
size	uint32_t	DMA transfer size in bytes
tag	uint32_t	DMA group tag ID.
tid	uint32_t	Transfer class identifier[a].
rid	uint32_t	Replacement[a].

a. Used for MFC functions only.
b. Used for methods other than MFC functions.

For all alternatives, we assume that the DMA transfer parameters, described in Table 4-2, are defined previously to executing the DMA command.

The following sections describe the four major methods to access MFC facilities.

MFC functions

MFC functions are a set of convenience functions, each perform a single DMA command (e.g. get, put, barrier). The functions are implemented either as macros or as built-in functions within the compiler, causing the compiler to map each of those functions to a certain composite intrinsic (similar to those discussed in chapter “Composite intrinsics”) with the corresponding operands.

A list and a brief description of all the available MFC functions is in Table 4-3 on page 112. For a more detailed description see *Programming Support for MFC Input and Output* chapter in C/C++ Language Extensions for Cell BE Architecture document.

To use those intrinsics the programmer must include `spu_mfcio.h` header file.

Example 4-9 illustrates initiating of a single ‘get’ command using MFC functions.

Example 4-9 SPU MFC function ‘get’ example

```
#include “spu_mfcio.h“

mfc_get(lsa, ea, size, tag, tid, rid);

// Implemented as the following composite intrinsic:
//       spu_mfcdma64(lsa, mfc_ea2h(ea), mfc_ea2l(ea), size, tag,
//                    ((tid<<24)|(rid<<16)|MFC_GET_CMD));

// wait until DMA transfer is complete (or do other things before that)
```

Composite intrinsics

The SDK3.0 defines a small number of composite intrinsics to handle DMA commands. Each composite intrinsics handles one DMA commands and is actually constructed from a series of low-level intrinsics (similar to those discussed in chapter “Low level intrinsics”). These intrinsics are further described in *Composite Intrinsics* chapter in C/C++ Language Extensions for Cell BE Architecture document, and also in Cell Broadband Engine Architecture document.

To use those intrinsics the programmer must include `spu_intrinsics.h` header file.

In addition, the header file `spu_mfcio.h` includes some useful predefined values of the DMA commands (e.g. `MFC_GET_CMD` in the example below). The programmer may include this file and use those predefined values instead of explicitly writing the corresponding value.

Example 4-10 illustrates the initiating of a single ‘get’ command using composite intrinsics.

Example 4-10 SPU composite intrinsics ‘get’ example

```
#include <spu_intrinsics.h>
#include “spu_mfcio.h“

spu_mfcdma64(lsa, eah, eal, size, tag, MFC_GET_CMD);

// Implemented using the six low level intrinstics in Example 4-11

// MFC_GET_CMD is defined as 0x0040 in spu_mfcio.h
```

Low level intrinsics

A series of few low level intrinsics (means generic or specific intrinsics) should be executed in order to executed a single DMA transfer. Each intrinsics is mapped to a single assembly instruction.
The relevant low level intrinsic are described in *Channel Control Intrinsics* chapter in C/C++ Language Extensions for Cell BE Architecture document.

To use those intrinsics the programmer must include `spu_intrinsics.h` header file.

Example 4-11 illustrates the initiating of a single 'get' command using low level intrinsics.

Example 4-11 SPU low level intrinsics 'get' example

```
spu_writech(MFC_LSA, lsa);
spu_writech(MFC_EAH, eah);
spu_writech(MFC_EAL, eal);
spu_writech(MFC_Size, size);
spu_writech(MFC_TagID, tag);
spu_writech(MFC_CMD, 0x0040);
```

Assembly-language instructions

Assembly-language instructions are similar to low level intrinsics (intrinsics are a series of ABI-compliant assembly language instructions executed for a single DMA transfer). Each of the low level intrinsics represents one assembly instruction). From practical point of view, the only case where we can recommend using this method instead the low level intrinsics is when the program is written in assembly.

Example 4-12 illustrates the initiating of a single 'get' command using assembly-language instructions.

Example 4-12 SPU assembly-language instructions 'get' example

```
   .text
   .global  dma_transfer
dma_transfer:

wrch$MFC_LSA, $3
wrch$MFC_EAH, $4
wrch $MFC_EAL, $5
wrch $MFC_Size, $6
wrch $MFC_TagID, $7
```

```
wrch $MFC_Cmd, $8
bi $0
```

4.2.4 PPU programming methods to access MFC's MMIO interface

Software running on a PPU may access the MFC facilities through the MMIO interface. There are two main methods to access this interface and this chapter discuss those methods. The two main methods, listed from the most abstract to the most low level one, are:

1. MFC functions
2. Direct problem state access (or Direct SPE access)

Unlike the SPU case when using the channel interface, in the PPU case it is not always recommended to use the MFC functions. The list below summarizes the main differences between the two methods and when is recommended to use either of them:

1. MFC functions are simpler from programming point of view and therefore using this method may reduce development time and make the code more readable.
2. Direct problem state access enables the programmer more flexibility and therefore when non standard mechanism should be implemented.
3. Direct problem state access have significant better performance in many cases (e.g. writing the inbound mailbox). Two of the reasons for the reduce performance for the MFC functions is the call overhead and also the mutex locking associated with the library functions being thread safe.
   It is therefore recommended in cases where the performance (e.g. latency) of the PPE access to the MFC is important to use the direct SPE access.

Note: If the performance (e.g. latency) of the PPE access to the MFC is important it is recommended to use the direct SPE access which may have significant better performance over the MFC functions. For more consideration on deciding the preferred method - see the three items above.

Most of the examples in this document as well as many code examples in the SDK package use the MFC functions method. However, the examples in the SDK documentation relay mostly on the second method - direct SPE access. Many such examples are available in Cell Broadband Engine Programming Tutorial.

In this chapter we are illustrating the differences between the two methods using the DMA 'get' command to move data from some component on main storage to local storage. This is done only for demonstration and similar implementation

may be performed for using each of the other MFC facilities (mailboxes, signals, events, etc.). We used the same parameters that are defined in Table 4-2 on page 101, but additional parameter is added in the PPU case:

- `spe_context_ptr_t spe_ctx` : a pointer to the context of the relevant SPE. This context is created when the SPE thread is created.

The following sections describe the two main methods for a PPE to access MFC facilities.

MFC functions

MFC functions are a set of convenience functions. Each implements a single DMA command (e.g. get, put, barrier). A list and brief description of all the available functions is in Table 4-3 on page 112. For a more detailed description see *SPE MFC problem state facilities* chapter in SPE Runtime Management library document.

The implementation of the MFC functions for the PPE, unlike the SPE implementation, usually involves accessing the operating system kernel which add a non negligible number of cycles and increase the latency of hose functions.

To use those intrinsics the programmer must include `libspe2.h` header file.

Example 4-13 illustrates the initiating of a single ‘get’ command using MFC functions.

Example 4-13 PPU MFC functions ‘get’ example

```
#include “libspe2.h“

spe_mfcio_get ( spe_ctx, lsa, ea, size, tag, tid, rid);

// wait till data was transfered to LS, or do other things...
```

Direct problem state access

The second option for PPE software to access the MFC facilities, is explicitly interact with the relevant MMIO interface of the relevant SPE. In order to do so, the software should perform the following steps:

1. Map corresponding the problem state area of the relevant SPE to the PPE thread address space. The programmer can do so using `spe_ps_area_get` function in the libspe library (include `libspe2.h` file to use this function).
2. Once the corresponding problem state area is mapped, the programer can access it using one of the following methods:

- Use one of the inline functions for direct problem state access that are defined in the `cbe_mfc.h` header file. This header file makes using direct problem state as easy as using the libspe functions. For example, the function `_spe_sig_notify_1_read` reads the SPU_Sig_Notify_1 register, function `_spe_out_mbox_read` reads a value from the SPU_Out_Mbox mailbox register, and `_spe_mfc_dma` function enqueues a DMA request.
- Use direct memory load or store instruction to access the relevant MMIO registers. The easiest way to do so is using enum and structs that describe the problem state areas and the offset of the MMIO registers. Thosestructs and enum are defined in `libspe2_types.h` and `cbea_map.h` header files (see Example 4-15 on page 107) but in order to use them the programer should simply include only `libspe2.h` file.

Please notice that once the problem state area is mapped, directly accessing this area by the application doesn't involve the kernel and therefore has a smaller latency then the corresponding MFC function.

Note: PPE programer must set the `SPE_MAP_PS` flag when creating the SPE context (in `spe_context_create` function) of the SPE whose problem state area the programmer later try to map (using `spe_ps_area_get` function). See Example 4-14.

Example 4-14 shows the PPU code for mapping SPE problem state to the thread address space and initiating a single 'get' command using direct SPE access.

Source code: The code of Example 4-14 is included in the additional material that is provided with this book. See "Simple PPU vector/SIMD code" on page 612 for more information.

Example 4-14 PPU dIrect SPE access 'get' example

```
#include <libspe2.h>
#include <cbe_mfc.h>
#include <pthread.h>

spe_context_ptr_t spe_ctx;
uint32_t lsa, eah, eal, tag, size, ret, status;
volatile spe_mfc_command_area_t* mfc_cmd;
volatile char data[BUFF_SIZE] __attribute__ ((aligned (128)));

// create SPE context: must set SPE_MAP_PS flag to access problem state
spe_ctx = spe_context_create (SPE_MAP_PS, NULL);
```

```
// - open an SPE executable and map using 'spe_image_open' function
// - load SPU program into LS using 'spe_program_load' function
// - create SPE pthread using 'pthread_create' function

// map SPE problem state using spe_ps_area_get
if ((mfc_cmd = spe_ps_area_get( data.spe_ctx, SPE_MFC_COMMAND_AREA)) ==
      NULL) {
   perror ("Failed mapping MFC command area"); exit (1);
}


// lsa = LS space address that SPU code provide
// eal = ((uintptr_t)&data) & 0xffffffff;
// eah = ((uint64_t)(uintptr_t)&data)>>32;
// tag = number from 0 to 15 (as 16-31 are used by the kernel)
// size= .....

while( (mfc_cmd->MFC_QStatus & 0x0000FFFF) == 0);

do{
   mfc_cmd->MFC_LSA = lsa;
   mfc_cmd->MFC_EAH = eah;
   mfc_cmd->MFC_EAL = eal;
   mfc_cmd->MFC_Size_Tag = (size<<16) | tag;
   mfc_cmd->MFC_ClassID_CMD = MFC_PUT_CMD;

   ret = mfc_cmd->MFC_CMDStatus;

} while(ret&0x3); //enqueuing until success

//following 2 lines are commented in order to be similar to
Example 4-13
//ret=spe_mfcio_tag_status_read(spe_ctx, 1<<tag, SPE_TAG_ALL, &status);
//if( ret !=0) printf("error in GET command");
```

The SDK3.0 header files `libspe2_types.h` and `cbea_map.h` contain several enums and structs that define the problem state areas and registers which makes the programming more convenient when accessing the MMIO interface from the PPE. Example 4-15 shows those enum and structs

Example 4-15 Structs and Enums for defining problem state areas and registers

```
// From libspe2_types.h header file
// ================================================================
```

```
enum ps_area { SPE_MSSYNC_AREA, SPE_MFC_COMMAND_AREA, SPE_CONTROL_AREA,
SPE_SIG_NOTIFY_1_AREA, SPE_SIG_NOTIFY_2_AREA };

// From cbea_map.h header file
// ================================================================
SPE_MSSYNC_AREA: MFC multisource synchronization register area
typedef struct spe_mssync_area {
   unsigned int MFC_MSSync;
} spe_mssync_area_t;

// SPE_MFC_COMMAND_AREA: MFC command parameter queue control area
typedef struct spe_mfc_command_area {
   unsigned char reserved_0_3[4];
   unsigned int MFC_LSA;
   unsigned int MFC_EAH;
   unsigned int MFC_EAL;
   unsigned int MFC_Size_Tag;
   union {
      unsigned int MFC_ClassID_CMD;
      unsigned int MFC_CMDStatus;
   };
   unsigned char reserved_18_103[236];
   unsigned int MFC_QStatus;
   unsigned char reserved_108_203[252];
   unsigned int Prxy_QueryType;
   unsigned char reserved_208_21B[20];
   unsigned int Prxy_QueryMask;
   unsigned char reserved_220_22B[12];
   unsigned int Prxy_TagStatus;
} spe_mfc_command_area_t;

// SPE_CONTROL_AREA: SPU control area
typedef struct spe_spu_control_area {
   unsigned char reserved_0_3[4];
   unsigned int SPU_Out_Mbox;
   unsigned char reserved_8_B[4];
   unsigned int SPU_In_Mbox;
   unsigned char reserved_10_13[4];
   unsigned int SPU_Mbox_Stat;
   unsigned char reserved_18_1B[4];
   unsigned int SPU_RunCntl;
   unsigned char reserved_20_23[4];
   unsigned int SPU_Status;
   unsigned char reserved_28_33[12];
   unsigned int SPU_NPC;
```

```
} spe_spu_control_area_t;

// SPE_SIG_NOTIFY_1_AREA: signal notification area 1
typedef struct spe_sig_notify_1_area {
   unsigned char reserved_0_B[12];
   unsigned int SPU_Sig_Notify_1;
} spe_sig_notify_1_area_t;

// SPE_SIG_NOTIFY_2_AREA: signal notification area 2
typedef struct spe_sig_notify_2_area {
   unsigned char reserved_0_B[12];
   unsigned int SPU_Sig_Notify_2;
} spe_sig_notify_2_area_t;
```

4.3 Data transfer

The Cell BE has a radical organization of storage and asynchronous DMA transfers between *local store* (*LS*) and main storage. While this architecture enables high performance it requires the application programer to explicitly handle the data trasnfers between LS and main memory or other local stores. Programing efficient data transfers is a key issue not only for preventing errors (e.g. synchronization errors which are hard to debug) but also for having the optimized out of a program running on a Cell BE based system.

Programming the DMA data transfer can be done by either an SPU program using the channel interface, or by the a PPU program using the MMIO interface. Using those interfaces is discussed in Chapter 4.2, “Storage domains, channels and MMIO interfaces” on page 95.

Regarding issue of DMA commands to the MFC command, the channel interface has 16 entries in its corresponding MFC SPU command queue, which stands for up to 16 DMA commands that may be handle simultaneously by the MFC. The corresponding MMIO interface on the other hand has only 8 entries in its corresponding MFC proxy command queue. For this reason as well as for other reasons (smaller latency in issuing the DMA commands, less overhead on the internal EIB bus, etc.) the programer should prefer issuing DMA commands from the SPU program rather then from the PPU.

This section explains about DMA data transfer methods as well as other data transfer methods (e.g. direct load and store) that may be used in order to transfer data between a LS and main memory or between one LS to another LS.

The part contains the following sections:

- 4.3.1, “DMA commands” on page 111 - the first chapter provides an overview over the DMA commands that are supported by the MFC, whether the commands are initiated by the SPE of the PPE.

The next three sections discuss how to initiate various data transfer using the SDK3.0 core libraries:

- 4.3.2, “SPE initiated DMA transfer between LS and main storage” on page 119 - discuss how a program running on a SPU may initiate DMA commands between its LS and the main memory using the associated MFC.
- 4.3.3, “PPU initiated DMA transfer between LS and main storage” on page 137 discuss how a program running on a PPU may initiate DMA commands between the LS of some SPE and the main memory using the MFC which is associated with this SPE.
- 4.3.4, “Direct problem state access and LS to LS transfer” on page 143 discuss two different issues. The first is how a LS of some SPE can be accessed directly by the PPU or by an SPU program running on other SPE.

The next two sections discuss two alternatives (other the core libraries) that comes with SDK3.0 and can be used for simpler intuiting of data transfer between the LS and main storage:

- 4.3.5, “Facilitate random data access using SPU software cache” on page 146 discuss how to use the SPU software managed cache and in which cases it is recommended to use it.
- 4.3.6, “Automatic software caching on SPE” on page 155 discuss an automated version of the SPU software cache which provides even simpler programing method but with possibly reduced performance.

The next three sections describes several fundamental techniques for programming performance efficient data transfers:

- 4.3.7, “Efficient data transfers by overlapping DMA and computation” on page 157 discuss the double buffering and multibuffering techniques that enable to overlap between DMA transfers and computation. Doing so very often provides a significant performance improvement.
- 4.3.8, “Improving page hit ratio using huge pages” on page 163 discuss how to configure huge pages in the system and when it may be useful.
- 4.3.9, “Improving memory access using NUMA” on page 168 discuss how to use the NUMA features on a Cell BE bases system.

Another topic which is very relevant to the data transfer and in not covered in those chapters is the ordering between different data turnovers and

synchronization techniques. This topic is discussed in Chapter 4.5, “Shared storage synchronizing and data ordering” on page 213.

4.3.1 DMA commands

MFC supports a set of DMA commands which provide the main mechanism that enables data transfer between the LS and main storage. It also supports a set of synchronization commands which used to control the order in which storage accesses are performed and maintaining synchronization with other processors and devices in the system.

Each MFC has an associated *Memory Management Unit* (*MMU*) that holds and processes address-translation and access-permission information supplied by the PPE operating system. While this MMU is distinct from the one used by the PPE, to process an effective address provided by a DMA command, the MMU uses the same method as the PPE memory-management functions. Thus, DMA transfers are coherent with respect to system storage. Attributes of system storage are governed by the page and segment tables of the PowerPC Architecture.

The following sections discuss several issues related to the supported DMA commands.

DMA commands

MFC supports a set of DMA commands:

- DMA commands may initiate or monitor the status of data transfers.
- Each MFC can maintain and process up to 16 in-progress DMA command requests and DMA transfers which are executed asynchronous to the code execution.
- The MFC can also autonomously manage a sequence of DMA transfers in response to a DMA-list command from its associated SPU. DMA lists are a sequence of eight-byte list elements, stored in an SPE’s LS, each of which describes a single DMA transfer.
- Each DMA command is tagged with a 5-bit Tag ID (which defines up to 32 IDs) and the software can use this identifier to check or wait on the completion of all queued commands in one or more tag groups.

The supported and recommended values for the DMA parameters are describe in “Supported and recommended values for DMA parameters” on page 115.

The supported and recommended parameters of a DMA list are described on “Supported and recommended values for DMA-list parameters” on page 116

A summary of all the DMA commands which are supported by the MFC are described in Table 4-3. For each command, we also mention the SPU and the PPE MFC functions that implement it, it any (blank means that this command is not supported by either the SPE or PPE). For detailed information on the MFC commands, see *DMA Transfers and Inter-processor Communication* chapter on Cell Broadband Engine Programming Handbook.

The SPU functions are defined in `spu_mfcio.h` header file and described in C/C++ Language Extensions for Cell BE Architecture .

The PPE functions are defined in `libspe2.h` header file and described in SPE Runtime Management library document.

SDK3.0 defines another set of PPE inline functions for handling the DMA data transfer in `cbe_mfc.h` file which is preferred from performance point of view over the `libspe2.h` functions. While the `cbe_mfc.h` functions are not well described in the official SDK documents they are quite straight forward and easy to use. In order to enqueue a DMA command the programmer may issue `_spe_mfc_dma` function with 'cmd' parameter indicating the DMA command that should be enqueued (e.g. set 'cmd' parameter to MFC_PUT_CMD for 'put' command, set it to MFC_GETS_CMD for 'gets' command, etc.)

Table 4-3 DMA commands supported by the MFC

Command	Function		Description
	SPU	PPE	
Put commands			
put	mfc_put	spe_mfcio_put	Moves data from LS to the effective address.
puts	*unsupported*	None[a]	Moves data from LS to the effective address and starts the SPU after the DMA operation completes.
putf	mfc_putf	spe_mfcio_putf	Moves data from LS to the effective address with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
putb	mfc_putb	spe_mfcio_putb	Moves data from LS to the effective address with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).

Command	Function		Description
	SPU	PPE	
putfs	*unsupported*	None[a]	Moves data from LS to the effective address with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue) and starts the SPU after the DMA operation completes.
putbs	*unsupported*	None[a]	Moves data from LS to the effective address with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue) and starts the SPU after the DMA operation completes.
putl	mfc_putl	*unsupported*	Moves data from LS to the effective address using an MFC list.
putlf	mfc_putlf	*unsupported*	Moves data from LS to the effective address using an MFC list with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
putlb	mfc_putlb	*unsupported*	Moves data from LS to the effective address using an MFC list with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).
get commands			
get	mfc_get	spe_mfcio_get	Moves data from the effective address to LS.
gets	*unsupported*	None[a]	Moves data from the effective address to LS, and starts the SPU after the DMA operation completes.
getf	mfc_getf	spe_mfcio_getf	Moves data from the effective address to LS with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
getb	mfc_getb	spe_mfcio_getb	Moves data from the effective address to LS with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).

Command	Function		Description
	SPU	PPE	
getfs	*unsupported*	None[a]	Moves data from the effective address to LS with fence (this command is locally ordered with respect to all previously issued commands within the same tag group), and starts the SPU after the DMA operation completes.
getbs	*unsupported*	None[a]	Moves data from the effective address to LS with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue), and starts the SPU after the DMA operation completes.
getl	mfc_getl	*unsupported*	Moves data from the effective address to LS using an MFC list.
getlf	mfc_getlf	*unsupported*	Moves data from the effective address to LS using an MFC list with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
getlb	mfc_getb	*unsupported*	Moves data from the effective address to LS using an MFC list with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).

a. While this command may be issued by the PPE there is no MFC function that supports it

The suffixes in Table 4-4 are associated with the DMA commands, and extend or refine the function of a command. For example, a ‘putb’ command moves data from LS to the effective address similar to the ‘put’ command, but also adds a barrier.

Table 4-4 MFC commands suffixes

Mnemonic	Possible Initiator		Description
	SPU	PPE	
s		Yes	*Start SPU*. Starts the SPU running at the address in the SPU Next Program Counter Register (SPU_NPC) after the MFC command completes.

Mnemonic	Possible Initiator		Description
	SPU	PPE	
f	Yes	Yes	*Tag-specific fence*. Command is locally ordered with respect to all previously issued commands in the same tag group and command queue.
b	Yes	Yes	*Tag-specific barrier*. Command is locally ordered with respect to all previously issued and all subsequently issued commands in the same tag group and command queue.
l	Yes		*List command*. Command processes a list of DMA list elements located in LS. Up to 2048 elements in a list; each list element specifies a transfer of up to 16 KB.

Supported and recommended values for DMA parameters

The following list summarizes the MFC's supported or recommended values for the parameters of the DMA commands:

- *Direction:* Data transfer may be in any of the two directions as referenced from the perspective of an SPE:
  - get commands: transfer data to a LA from the main storage.
  - put commands: transfers data out of the LS to the main storage.
- *Size*: Transfer size should obey the following guidelines:
  - Supported transfer sizes are 1, 2, 4, 8, or 16 bytes, and multiples of 16-bytes
  - Maximum transfer size is 16 KB.
  - Peak performance is achieved when transfer size is a multiple of 128 bytes.
- *Alignment*: Alignment of the LSA and the EA should obey the following guidelines:
  - 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 (bits 28 through 31 must be ‘0’).
- Peak performance is achieved when both source and destination are aligned on a 128-byte boundary (bits 25 through 31 cleared to ‘0’).

Note: The header file `spu_mfcio.h` contains some useful definitions the supported parameter of DMA command (e.g. `MFC_MAX_DMA_SIZE`)

If a transaction have illegal size or the address is invalid (due to a segment fault, a mapping fault, or other address violation) there will be no error during compilation. Instead, during run time the corresponding DMA command queue processing is suspended and an interrupt is raised to the PPE. The application usually terminated in this case and a “Bus Error” message is printed.
The MFC checks the validity of the effective address during transfers. Partial transfers can be performed before the MFC encounters an invalid address and raises the interrupt to the PPE.

Supported and recommended values for DMA-list parameters

The following list summarizes the MFC’s supported or recommended values for the parameters of the DMA-list commands:

- The parameters of each transfer (e.g. size, alignment) should be according to the described in “Supported and recommended values for DMA parameters” on page 115.
- All the data transfers that are issued in a single DMA-list command have the same high 32 bits of a 64 bit effective address.
- All the data transfers that are issued in a single DMA-list command share the same tag ID.

In addition, the supported and recommended parameters of the DMA list itself are the following:

- *Length*: A DMA list command can specify up to 2048 DMA transfers, defining up to 16 KB of memory in the LS to maintain the list itself. Since each such transfer have up to 16 KB length, a DMA list command can transfer up to 32 MB, which is 128 times the size of the 256 KB LS.
- *Continuity*: DMA list can move data between a contiguous area in a LS and possibly non-contagious area in the effective address space.
- *Alignment*: The local store address of the DMA list itself must be aligned on an eight-byte boundary (bits 29 through 31 must be ‘0’).

Note: The header file `spu_mfcio.h` contains some useful definitions the supported parameter of DMA-list command (e.g. `MFC_MAX_DMA_LIST_SIZE`)

Synchronization and atomic commands

MFC also support a set of synchronization and atomic commands that can be used to control the order in which DMA storage accesses are performed. Those commands include four atomic commands, three send-signal commands and three barrier commands. Synchronization may be performed for all the transactions in a queue or only to a group of them as explained in Chapter , “DMA-command tag groups” on page 118

While this chapter provide a brief overview of those commands, a more detailed description is in Chapter 4.5, “Shared storage synchronizing and data ordering” on page 213.

The synchronization and atomic command supported by the MFC are described in Table . For each command, we also mention the SPU and the PPE MFC functions that implement it, it any (blank means that this command is not supported by either the SPE or PPE). For detailed information on the MFC commands, see *DMA Transfers and Inter-processor Communication* chapter on Cell Broadband Engine Programming Handbook.

The SPU MFC functions are defined in `spu_mfcio.h` header file and are described in C/C++ Language Extensions for Cell BE Architecture .

The PPE’s are defined in *libspe2.h* header file and are described in SPE Runtime Management library.

*Synchroniz*ation commands supported by the MFC

Command	Possible Initiator		Description
	SPU	PPE	
Synchronization commands			
barrier	mfc_barrier	*unsupported*	Barrier type ordering. Ensures ordering of all preceding DMA commands with respect to all commands following the barrier command in the same command queue. The *barrier* command has no effect on the immediate DMA commands: *getllar*, *putllc*, and *putlluc*.

Command	Possible Initiator		Description
	SPU	PPE	
mfceieio	mfc_eieio	_eieio[a]	Controls the ordering of *get* commands with respect to **put** commands, and of *get* commands with respect to *get* commands accessing storage that is caching inhibited and guarded. Also controls the ordering of *put* commands with respect to *put* commands accessing storage that is memory coherence required and not caching inhibited.
mfcsync	mfc_sync	__sync[a]	Controls the ordering of DMA *put* and *get* operations within the specified tag group with respect to other processing units and mechanisms in the system.
sndsig	mfc_sndsig	spe_signal_write	Write SPU Signal Notification Register in another device.
sndsigf	mfc_sndsigf	*unsupported*	Write SPU Signal Notification Register in another device, with fence.
sndsigb	mfc_sndsigb	*unsupported*	Write SPU Signal Notification Register in another device, with barrier.
Atomic commands			
getllar	mfc_getllar	lwarx/ldarx [a]	Get lock line and reserve.
putllc	mfc_putllc	stwcx/stdcx[a]	Put lock line conditional.
putlluc	mfc_putlluc	*unsupported*	Put lock line unconditional.
putqlluc	mfc_putqlluc	*unsupported*	Put queued lock line unconditional.

a. No function call for implementing this command but instead implemented as intrinsic that is defined in `ppu_intrinsics.h` file.

DMA-command tag groups

All DMA commands except the atomic ones can be tagged with a 5-bit Tag Group ID. By assigning a DMA command or group of commands to different tag groups, the status of the entire tag group can be determined within a single

command queue. Software can use this identifier to check or wait on the completion of all queued commands in one or more tag groups.

Notice that tag groups can be formed separately within any of the two MFC command queues. Thus, tags assigned to commands in the SPU command queue are independent of the tags assigned to commands in the MFC's proxy command queue.

Tagging is useful when using barriers to control the ordering of MFC commands within a single command queue. DMA commands within a tag group can be synchronized with a fence or barrier option by appending an 'f' or 'b', respectively, to the command mnemonic:

- Execution of a *fenced* command option is delayed until all previously issued commands within the same tag group have been performed.
- Execution of a *barrier* command option and all subsequent commands is delayed until all previously issued commands in the same tag group have been performed.

4.3.2 SPE initiated DMA transfer between LS and main storage

Software running on a SPU initiate DMA data transfer by accessing the local MFC facilities through the channel interface. In this chapter we describe how such SPU code can initiate basic data transfers between main storage and LS.

We illustrate it though *'get'* command which transfer data from main storage to the LS, and *'put'* command that transfer data in the opposite direction. We also describe the *'getl'* and *'putl'* commands which transfer data using DMA list.

The MFC support additional data transfer commands which guarantees ordering between data transfer (e.g. 'putf', 'putlb', 'getlf', 'getb'). Those commands are initiated in similar way to the basic 'get' and 'put' commands, but their behavior is different.

For detailed information on the channel interface and information on the MFC commands, see *SPE Channel and Related MMIO Interface* chapter and *DMA Transfers and Interprocessor Communication* chapter respectively on Cell Broadband Engine Programming Handbook document.

Tag manager

The *tag manager* facilitates the management of tag identifiers used for DMA operations in an SPU application. It is implemented through a set of functions that the programmer should use in order to reserve tag IDs before initializing DMA transactions and release them when he/she is done.

The functions are defined in `spu_mfcio.h` header file and are described in C/C++ Language Extensions for Cell BE Architecture document. The main functions are:

- `mfc_tag_reserve`: reserve a single tag ID.
- `mfc_tag_release`: release a single tag ID.

Some tags may be pre-allocated and being used by the operating environment (e.g software managed cache, PDT: Performance Analysis Tool). The implementation of the tag manager therefore does not guarantee to make all 32 architected tag IDs available for user allocation. If the programmer uses some fixed value of tag IDs instead of using the tag manager to do so, it can lead to possible inefficiencies caused by waiting for DMA completions on tag groups containing DMAs issued by other software components.

Note: When programming a SPU application that initiate DMAs, it is necessary to use the tag manager's functions in order to reserve a tag ID or a set of IDs and not use some random or fixed values. It is recommended that tag allocation services be used to ensure that the other SW component's use of tag ID's does not overlap with the application's use of tags. However, it is not required.

The usage of the tag manager is illustrated through Example 4-16 on page 122.

Basic DMA transfer between LS and main storage

This chapter describes how SPU software can transfer data between the LS and main storage using basics DMA commands. That term 'basic' commands implies to commands that should be explicitly issued for each DMA transaction separately. Another alternative is using the DMA list commands which may initialize a sequence of DMA transfers as explained in Chapter , "DMA list data transfer" on page 124.

The next sections describe how to initialize basic 'get' and 'put' DMA commands. We illustrate it through a code example which also includes the use of the tag manager.

Initiate a DMA transfer

To initialize a DMA transfer the SPE programmer can call one of the corresponding functions of `spu_mfcio.h` header file. Each of those functions implements a single command, such as:

- `mfc_get`: implements 'get' command.
- `mfc_put`: implements 'put' command.

These functions are non-blocking in terms of issuing the DMA command - the software will continue its execution after enqueueing the commands into the MFC SPU command queue but will not block till the DMA commands are actually issued on the EIB bus. However, these functions will block if the command queue is full and will wait till there is available space in that queue. The full list of the supported commands are shown in Table 4-3 on page 112.

The programmer should be aware of the fact that the implementation of those functions actually involve a sequence of the following six channel writes:

1. Write LSA (local store address) parameter to MFC_LSA channel.
2. Write EAH (effective address higher bits) parameter to MFC_EAH channel.
3. Write EAL (effective address lower bits) parameter to MFC_EAL channel.
4. Write transfer size parameter to MFC_Size channel.
5. Write tag ID parameter to MFC_TagID channel.
6. Write class ID and command opcode to MFC_Cmd channel. The opcode defines the transfer type (e.g. ‘get’, ‘put’).

Note: The supported and recommnded value of the different DMA parameters are described in “Supported and recommended values for DMA parameters” on page 115.

Waiting for completion of a DMA transfer

After DMA command was initiated, the software may wait for a completion of the DMA transaction. Programmer may do so by calling to one of the functions that are implemented in `spu_mfcio.h` header file. The two main functions to do so are:

1. `mfc_write_tag_mask:` write the tag mask which determines to which tag IDs a completion notification is needed (done using the two functions below).
2. `mfc_read_tag_status_any`: wait until *any* of the specified tagged DMA commands is completed
3. `mfc_read_tag_status_all`: wait until *all* of the specified tagged DMA commands are completed

The last two functions are blocking so it will cause the software to halt till all DMA transfer related to the tag ID are complete. Full list of the supported commands are in Table 4-3 on page 112.

The implementation of the first function generates the following channel operations:

1. Set the bit that represents the tag ID by writing the corresponding value (all bits are ‘0’ beside bit number tag ID) to the MFC_WrTagMask channel.

The implementation of the next two functions involve a sequence of the following two channel operations:

1. Write MFC_TAG_UPDATE_ALL or MFC_TAG_UPDATE_ANY mask to MFC_WrTagUpdate channel.
2. Read MFC_RdTagStat channel.

Basic DMA 'get' and 'put' transfers - code example

This chapter illustrate how SPU code can perform basic 'get' and 'put' commands and also illustrate some other relevant issues. The examples includes in this chapter demonstrate the following techniques:

- SPU code uses the tag manager to reserve and release tag ID
- SPU code uses 'get' command to transfer data from main storage to LS.
- SPU code uses 'put' command to transfer data from LS to main storage.
- SPU code waiting for completion of the 'get' and 'put' commands.
- SPU macro for waiting to completion of DMA group related to input tag.
- PPU macro for rounding input value to the next higher multiple of either 16 or 128 (to fulfill MFC's DMA requirements).

As mentioned in Chapter 4.2.3, "SPU programming methods to access MFC's channel interface" on page 100, we use the MFC functions method to access the DMA mechanism. Each of such functions actually implements few of the steps that were mentioned above causing the code to be simpler. From programmer point of view it is important to be familiar with the number of commands that are involve in order to understand the impact on its application execution.

Example 4-16 and Example 4-17 contains the corresponding SPU and PPU code respectively.

Source code: The code of Example 4-16 and Example 4-17 is included in the additional material that is provided with this book. See "SPU initiated basic DMA between LS and main storage" on page 613 for more information.

Example 4-16 SPU initiated basic DMA between LS and main storage - SPU code

```
#include <spu_mfcio.h>

// Macro for waiting to completion of DMA group related to input tag:
// 1. Write tag mask
// 2. Read status which is blocked untill all tag's DMA are completed
#define waitag(t) mfc_write_tag_mask(1<<t); mfc_read_tag_status_all();
```

```
// Local store buffer: DMA address and size alignment:
//    - MUST be 16B aligned otherwise a bus error is generated
//    - may be 128B aligned to get better performance
// In this case we use 16B becuase we don't care about performance
volatile char str[256]  __attribute__ ((aligned(16)));

// argp - effective address pointer to the string in main storage
// envp - size of string in main memory in bytes
int main( uint64_t spuid , uint64_t argp, uint64_t envp ){
   uint32_t tag_id = mfc_tag_reserve();

   // reserve a tag from the tag manager
   if (tag_id==MFC_TAG_INVALID){
      printf("SPE: ERROR can't allocate tag ID\n"); return -1;
   }

   // get data from main storage to local store
   mfc_get((void *)(str), argp, (uint32_t)envp, tag_id, 0, 0);

   // wait for 'get' command to complete. wait only on this tag_id.
   waitag(tag_id);

   printf("SPE: %s\n", str);
   strcpy(str, "Am I there? No! I'm still here! I will go there
again....");

   // put data to main storage from local store
   mfc_put((void *)(str), argp, (uint32_t)envp, tag_id, 0, 0);

   // wait for 'get' command to complete. wait only on this tag_id.
   waitag(tag_id);

   // release the tag from the tag manager
   mfc_tag_release(tag_id);

   return (0);
}
```

Example 4-17 SPU initiated basic DMA between LS and main storage - PPU code

```
#include <libspe2.h>

// macro for rounding input value to the next higher multiple of either
// 16 or 128 (to fulfill MFC's DMA requirements)
#define spu_mfc_ceil128(value)  ((value + 127) & ~127)
```

```
#define spu_mfc_ceil16(value)   ((value +  15) &  ~15)

volatile char str[256]  __attribute__ ((aligned(16)));

int main(int argc, char *argv[])
{
   void *spe_argp, *spe_envp;
   spe_context_ptr_t spe_ctx;
   spe_program_handle_t *program;
   uint32_t entry = SPE_DEFAULT_ENTRY;

   // Prepare SPE parameters
   strcpy( str, "I am here but I want to go there!");
   printf("PPE: %s\n", str);

   spe_argp=(void*)str;
   spe_envp=(void*)strlen(str);
   spe_envp=(void*)spu_mfc_ceil16((uint32_t)spe_envp);//round up to 16B

   // Initialize and run the SPE thread using the four functions:
   // 1) spe_context_create 2) spe_image_open
   // 3) spe_program_load 4) spe_context_run

   // Wait for SPE thread to complete using spe_context_destroy
   // function (blocked untill SPE thread was complete).

   printf("PPE: %s\n", str); // is he already there?
   return (0);
}
```

DMA list data transfer

A DMA list is a sequence of transfer elements (or list elements) that, together with an initiating DMA-list command, specifies a sequence of DMA transfers between a single *continuous* area of LS and *possibly discontinuous areas* in main storage. DMA lists can therefore be used to implement scatter-gather functions between main storage and the LS. All the data transfers that are issued in a single DMA-list command share the same tag ID and are the same type of commands (‘getl’, or ‘putl’, or other command). The DMA list is self is stored in the LS of the same SPE.

The next three chapter describes the three steps that a programmer who wish to initiate sequence of transfers using a DMA-list should typically performs:

1. “Creating a DMA list” on page 125 - create and initialize the DMA list in an SPE’s LS. This step can be done by either the local SPE, the PPE or other SPE.
2. “Initiating DMA list command” on page 126 - issue a DMA-list command such as ‘getl’ or ‘putl’. Such DMA-list commands can only be issued by programs running on the local SPE.
3. “Waiting for completion of data transfer” on page 126 - wait for completion of the data transfers.

The last chapter, “DMA list transfer - code example” on page 127, provide a code example which illustrate this sequence of steps.

Creating a DMA list

Each transfer element in the DMA list contains three parameters:

- *notify*: stall-and-notify flag that can be used to suspend list execution after transferring a list element whose stall-and-notify bit is set.
- *size*: transfer size in bytes.
- *eal*: lower 32-bits of an effective address in main storage.

Note: The supported and recommended values of the DMA-list parameters are described in “Supported and recommended values for DMA-list parameters” on page 116.

SPU software creates the list and stores it in the LS. The list basic element is a `mfc_list_element` structure that describes a single data transfer. This structure, that is defined in `spu_mfcio.h` header file as shown in Example 4-18:

Example 4-18 DMA list basic element - mfc_list_element struct

```
typedef struct mfc_list_element {
   uint64_t notify   : 1;  // optional stall-and-notify flag
   uint64_t reserved : 16; // the name speaks for itself
   uint64_t size     : 15; // transfer size in bytes
   uint64_t eal      : 32; // lower 32-bits of an EA in main storage
} mfc_list_element_t;
```

Transfer elements are processed sequentially in the order they are stored. If the `notify` flag is set for a transfer element, the MFC will stop processing the DMA list after performing the transfer for that element until the SPE program send

acknowledge. This procedure is described in “Waiting for completion of data transfer” on page 126.

Initiating DMA list command

After the list is stored in the LS, the execution of the list is initiated by a DMA-list command, such as ‘getl’ or ‘putl’, from the SPE whose LS contains the list. To initialize a DMA list transfer the SPE programmer can call one of the corresponding functions of `spu_mfcio.h` header file. Each of those functions implements a single DMA list command, such as:

- `mfc_getl`: implements ‘getl’ command.
- `mfc_putl`: implements ‘putl’ command.

These functions are non-blocking in terms of issuing the DMA command - the software will continue its execution after enqueueing the commands into the MFC SPU command queue but will not block till the DMA commands are actually issued on the EIB bus. However, these functions will block if the command queue is full and will wait untill there is available space in that queue. The full list of supported commands are in Table 4-3 on page 112.

Initializing a DMA-list commands requires similar steps and parameters as when initializing basic DMA command. Those steps are described in “Initiate a DMA transfer” on page 120. However, a DMA-list command requires two different types of parameters than those required by a single-transfer DMA command:

- EAL which is written to the MFC_EAL channel should be the starting local store address (LSA) of the DMA list (rather then with the EAL which is specified in each transfer element separately).
- Transfer size which is written to MFC_Size channel should be the size in bytes of the DMA list itself (rather then the transfer size which is specified in each transfer element separately). The list size is equal to the number of transfer elements, multiplied by the size of `mfc_list_element` structure (8 bytes).

The starting LSA and the EA-high (EAH) are specified only once, in the DMA-list command that initiates the transfers. The LSA is internally increment based on the amount of data transferred by each transfer element. However, if the starting LSA for each transfer element in a list does not begin on a 16-byte boundary, then hardware automatically increments the LSA to the next 16-byte boundary. The EAL for each transfer element is in the 4-GB area defined by EAH.

Waiting for completion of data transfer

There are two main mechanism that enables the software to verify the completion of the DMA transfers.

The first mechanism is the same as basic (non-list) DMA commands using MFC_WrTagMask and MFC_RdTagStat channels and can be used to notify the software on the completion of the entire transfer in the DMA list. This procedure is explained in “Waiting for completion of a DMA transfer” on page 121.

The second mechanism is using the *stall-and-notify* flag that enables the software to be notified on the completion of subset of the transfers in the list by the MFC. The MFC halt it transfer on this list (but not only the operations) till it is acknowledged by the software. This mechanism may be useful if the software needs to update the characteristics of a stalled subsequent trasnfers depends on the data that was just transferred to the LS on the previous transfers. In any case the number of elements in the queued DMA list cannot be changed.

To use this mechanism, the following steps are performed by the SPE software and the local MFC:

1. Software enables *DMA List Command Stall-And-Notify* event.
   This step is illustrated in `notify_event_enable` function of Example 4-20.
2. Software sets the `notify` bit in a certain element in the DMA list
   (SW says: “*let me know when you’re done...*”)
3. Software issues a DMA-list command on this list
   (SW says: “*do it...*”)
4. MFC stop processing the DMA list after performing the transfer for that specific element which activates *DMA List Command Stall-And-Notify* event.
   (MFC says: “*I’ve completed working on this - its yours now...*”)
5. Software handles the event, optionally modify subsequent transfer elements before they are processed by the MFC and then acknowledge the MFC.
   This step is illustrated in `notify_event_handler` function of Example 4-20.
   (SW says: “*Got it - I’m checking the incoming data. Go back to your next task...*”)
6. MFC continue processing the subsequent transfer elements in the list (until maybe another element sets the `notify` bit).

DMA list transfer - code example

This section contains a code example on how SPU program may initiate DMA list transfer. The example demonstrate the following techniques:

- SPU code creates a DMA list on the LS.
- SPU code activates stall-and-notify bit in some of the elements in the list.
- SPU code `spe_mfcio.h` definitions to check if DMA transfer attributes are legal.
- SPU code issue ‘getl’ command to transfer data from main-storage to LS.

- SPU code issue 'putl' command to transfer data from LS to main-storage.
- SPU code implements *event handler* to the stall-and-notify events.
- SPU code for dynamically updating DMA list according to the data that was just transferred into the LS.
- PPU code mark the SPU code to stop transferring data after some data elements using the stall-and-notify mechanism.
- PPU and SPU code for synchronizing the completion of SPE's writing the output data. Implemented using a notification flag in main-storage and barrier between writing the data to memory and updating this notification flag.

Example 4-19 shows the shared header file, Example 4-20 shows the SPU code, while Example 4-20 shows the corresponding PPU code.

Source code: The code of Example 4-19 and Example 4-20 is included in the additional material that is provided with this book. See "SPU initiated DMA list transfers between LS and main storage" on page 613 for more information.

Example 4-19 SPU initiated DMA list transfers between LS and main storage - shared header file

```
// common.h file ======================================================

// DMA list parameters
#define DMA_LIST_LEN  512
#define ELEM_PER_DMA   16 // Guarantee alignment to 128 B
#define NOTIFY_INCR    16

#define TOTA_NUM_ELEM  ELEM_PER_DMA*DMA_LIST_LEN
#define BUFF_SIZE      TOTA_NUM_ELEM+128

#define MAX_LIST_SIZE 2048  // 2K

// commands and status definitions
#define CMD_EMPTY 0
#define CMD_GO    1
#define CMD_STOP  2
#define CMD_DONE  3

#define STATUS_DONE      1234567
#define STATUS_NO_DONE ~(STATUS_DONE)

// data elements that SPE should work on
#define DATA_LEN 15
```

```
typedef struct {
   char cmd;
   char data[DATA_LEN];
} data_elem; // aligned to 16B

// the context that PPE forward to SPE
typedef struct{
   uint64_t ea_in;
   uint64_t ea_out;
   uint32_t elem_per_dma;
   uint32_t tot_num_elem;
   uint64_t status;
} parm_context; // aligned to 16B

#define MIN(a,b) (((a)>(b)) ? (b) : (a))
#define MAX(a,b) (((a)>(b)) ? (a) : (b))

// dummy function for calculating the output from the input
inline char calc_out_d( char in_d ){
   return in_d-1;
}
```

Example 4-20 SPU initiated DMA list transfers between LS and main storage - SPU code

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

#include "common.h"

// Macro for waiting to completion of DMA group related to input tag
#define waitag(t) mfc_write_tag_mask(1<<t); mfc_read_tag_status_all();

static parm_context ctx __attribute__ ((aligned (128)));

// DMA data structures and and data buffer
volatile data_elem lsa_data[BUFF_SIZE] __attribute__ ((aligned (128)));
volatile mfc_list_element_t dma_list[MAX_LIST_SIZE] __attribute__
((aligned (128)));
volatile uint32_t status __attribute__ ((aligned(128)));

// global variables
```

```
int elem_per_dma, tot_num_elem, byte_per_dma, byte_tota, dma_list_len;
int event_num=1, continue_dma=1;
int notify_incr=NOTIFY_INCR;

// enables stall-and-notify event
//=====================================================================
static inline void notify_event_enable(  )
{
   uint32_t eve_mask;

   eve_mask = spu_read_event_mask();
   spu_write_event_mask(eve_mask | MFC_LIST_STALL_NOTIFY_EVENT);
}

// updates the remaining DMA list according to data that was already
// transferred to LS
//=====================================================================
static inline void notify_event_update_list( )
{
   int i, j, start, end;

   start = (event_num-1)*notify_incr*elem_per_dma;
   end   = event_num*notify_incr*elem_per_dma-1;

   // loop on only data elements that were transffered since last event
   for (i=start; i<=end; i++){
      if ( lsa_data[i].cmd == CMD_STOP){

         // PPE wants us to stop DMAs - zero remaing DMAs
         dma_list[event_num*notify_incr+1].size=0;
         dma_list[dma_list_len-1].size=0;
         for (j=event_num*notify_incr; j<dma_list_len; j++){
            dma_list[j].size = 0;
            dma_list[j].notify = 0;
         }
         continue_dma = 0;
         break;
      }
   }
}

// handle stall-and-notify event include acknowledging the MFC
//=====================================================================
static inline void notify_event_handler( uint32_t tag_id )
{
```

```
   uint32_t eve_mask, tag_mask;

   // blocking function to wait for even
   eve_mask = spu_read_event_mask();

   spu_write_event_mask(eve_mask | MFC_LIST_STALL_NOTIFY_EVENT);

   // loop for checking that event is on the correct tag_id
   do{
      // loop for checking that stall-and-notify event occured
      do{
         eve_mask = spu_read_event_status();

      }while ( !(eve_mask&(uint32_t)MFC_LIST_STALL_NOTIFY_EVENT)  );

      // disable event stall-and-notify event
      eve_mask = spu_read_event_mask();
      spu_write_event_mask(eve_mask & (~MFC_LIST_STALL_NOTIFY_EVENT));

      // acknowledge stall-and-notify event
      spu_write_event_ack(MFC_LIST_STALL_NOTIFY_EVENT);

      // read the tag_id that caused the event. no infomation is
      // provided on which DMA list command in the tag group has
      // stalled or which element in the DMA list command has stalled
      tag_mask = mfc_read_list_stall_status();

   }while ( !(tag_mask & (uint32_t)(1<<tag_id))  );

   // update DMA list according to data that was just transferred to LS
   notify_event_update_list( );

   // acknowlege the MFC to continue
   mfc_write_list_stall_ack(tag_id);

   // re-enable the event
   eve_mask = spu_read_event_mask();
   spu_write_event_mask(eve_mask  | MFC_LIST_STALL_NOTIFY_EVENT);
}

void exit_handler( uint32_t tag_id ){

   // update the status so PPE knows that all data is in place
   status = STATUS_DONE;
```

```
   //barrier to ensure data is written to memory before writing status
   mfc_putb((void*)&status, ctx.status, sizeof(uint32_t), tag_id,0,0);
   waitag(tag_id);

   mfc_tag_release(tag_id); // release tag ID before exiting

   printf("<SPE: done\n");
}

int main(int speid , uint64_t argp){
   int i, j, num_notify_events;
   uint32_t addr, tag_id;

   // enable the stall-and-notify
   //=================================================================
   notify_event_enable( );

   // reserve DMA tag ID
   //=================================================================
   tag_id = mfc_tag_reserve();

   if(tag_id==MFC_TAG_INVALID){
      printf("SPE: ERROR - can't reserve a tag ID\n");
      return 1;
   }

   // get context information from system memory.
   //=================================================================
   mfc_get((void*) &ctx, argp, sizeof(ctx), tag_id, 0, 0);
   waitag(tag_id);

   // initalize DMA tranfer attributes
   //=================================================================
   tot_num_elem = ctx.tot_num_elem;
   elem_per_dma = ctx.elem_per_dma;
   dma_list_len = MAX( 1, tot_num_elem/elem_per_dma );
   byte_tota = tot_num_elem*sizeof(data_elem);
   byte_per_dma = elem_per_dma*sizeof(data_elem);

   // initalize data buffer
   //=================================================================
   for (i=0; i<tot_num_elem; ++i){
      lsa_data[i].cmd = CMD_EMPTY;
   }
```

```
// use spe_mfcio.h definitions to check DMA attributes' legitimate
//=================================================================
if (byte_per_dma<MFC_MIN_DMA_SIZE || byte_per_dma>MFC_MAX_DMA_SIZE){
   printf("SPE: ERROR - illegal DMA transfer's size\n");
   exit_handler( tag_id ); return 1;
}
if (dma_list_len<MFC_MIN_DMA_LIST_SIZE||
    dma_list_len>MFC_MAX_DMA_LIST_SIZE){
   printf("SPE: ERROR - illegal DMA list size.\n");
   exit_handler( tag_id ); return 1;
}
if (dma_list_len>=MAX_LIST_SIZE){
   printf("SPE: ERROR - DMA list size bigger then local list \n");
   exit_handler( tag_id ); return 1;
}

if(tot_num_elem>BUFF_SIZE){
   printf("SPE: ERROR - dma length bigger then local buffer\n");
   exit_handler( tag_id ); return 1;
}

// create the DMA lists for the 'getl' comand
//=================================================================
addr = mfc_ea2l(ctx.ea_in);

for (i=0; i<dma_list_len; i++) {
   dma_list[i].size = byte_per_dma;
   dma_list[i].eal = addr;
   dma_list[i].notify = 0;
   addr += byte_per_dma;
}

// update stall-and-notify bit EVERY 'notify_incr' DMA elements
//=================================================================
num_notify_events=0;
for (i=notify_incr-1; i<(dma_list_len-1); i+=notify_incr) {
   num_notify_events++;
   dma_list[i].notify = 1;
}

// issue the DMA list 'getl' command
//=================================================================
mfc_getl((void*)lsa_data, ctx.ea_in, (void*)dma_list,
          sizeof(mfc_list_element_t)*dma_list_len,tag_id,0,0);
```

```
// handle stall-and-notify events
//=====================================================================
for (event_num=1; event_num<=num_notify_events; event_num++) {
   notify_event_handler( tag_id );

   if( !continue_dma ){ // stop dma since PPE mark us to do so
      break;
   }
}

// wait for completion of the 'getl' command
//=====================================================================
waitag(tag_id);


// calculate the output data
//=====================================================================
for (i=0; i<tot_num_elem; ++i){
   lsa_data[i].cmd = CMD_DONE;
   for (j=0; j<DATA_LEN; j++){
      lsa_data[i].data[j] = calc_out_d( lsa_data[i].data[j] );
   }
}

// + update the existing DMA lists for the 'putl' comand
// + update only the address since the length is the same
//=====================================================================
addr = mfc_ea2l(ctx.ea_out);

for (i=0; i<dma_list_len; i++) {
   dma_list[i].eal = addr;
   dma_list[i].notify = 0;
   addr += byte_per_dma;
}

// + no notification is needed for the 'putl' command

// issue the DMA list 'getl' command
//=====================================================================
mfc_putl((void*)lsa_data,ctx.ea_out,(void*)dma_list,
         sizeof(mfc_list_element_t)*dma_list_len,tag_id,0,0);

// wait for completion of the 'putl' command
//=====================================================================
waitag(tag_id);
```

```
   exit_handler(tag_id);
   return 0;
}
```

Example 4-21 SPU initiated DMA list transfers between LS and main storage - PPU code

```
#include <libspe2.h>
#include <cbe_mfc.h>

#include "common.h"

// data structures to work with the SPE
volatile parm_context ctx __attribute__ ((aligned(16)));
volatile data_elem in_data[TOTA_NUM_ELEM] __attribute__
((aligned(128)));
volatile data_elem out_data[TOTA_NUM_ELEM] __attribute__
((aligned(128)));
volatile uint32_t status __attribute__ ((aligned(128)));

// take ‘spu_data_t’ structure and ‘spu_pthread’ function from
// Example 4-5 on page 90

int main(int argc, char *argv[])
{
   spe_program_handle_t *program;
   int i, j, error=0;

   printf(")PPE: Start main \n");
   status = STATUS_NO_DONE;

   // initiate input and output data
   for (i=0; i<TOTA_NUM_ELEM; i++){
      in_data[i].cmd  = CMD_GO;
      out_data[i].cmd = CMD_EMPTY;

      for (j=0; j<DATA_LEN; j++){
         in_data[i].data[j]  = (char)j;
         out_data[i].data[j] = 0;
      }
   }

   // ================================================================
```

```
   // tell the SPE to stop in some random element (number 3) after 10
   // stall-and-notify events.
   // ================================================================
   in_data[3+10*ELEM_PER_DMA*NOTIFY_INCR].cmd  = CMD_STOP;

   // initiate SPE parameters
   ctx.ea_in    = (uint64_t)in_data;
   ctx.ea_out   = (uint64_t)out_data;
   ctx.elem_per_dma = ELEM_PER_DMA;
   ctx.tot_num_elem = TOTA_NUM_ELEM;
   ctx.status   = (uint64_t)&status;

   data.argp = (void*)&ctx;

   // ... Omitted section:
   // creates SPE contexts, load the program to the local stores,
   // run the SPE threads, and waits for SPE threads to complete.

   // (the entire source code for this example is part of the book's
   // additional material).

   // This subject of is also described in section 4.1.2, "Task
parallelism and managing SPE threads"

   // wait for SPE data to be written into memory
   while (status != STATUS_DONE);

   for (i=0, error=0; i<TOTA_NUM_ELEM; i++){
      if (out_data[i].cmd != CMD_DONE){
         printf("ERROR: command is not done at index %d\n",i);
         error=1;
      }
      for (j=0; j<DATA_LEN; j++){
         if ( calc_out_d(in_data[i].data[j]) != out_data[i].data[j]){
            printf("ERROR: wrong output : entry %d char %d\n",i,j);}
            error=1; break;
      }
      if (error) break;
   }
   if(error){ printf(")PPE: program was completed with error\n");
   }else{     printf(")PPE: program was completed successfully\n");}

   return (0);
}
```

4.3.3 PPU initiated DMA transfer between LS and main storage

Software running on a PPU initiate DMA data transfers between the main storage and LS of some SPE by accessing the MFC facilities of this SPE through the MMIO interface. In this chapter we describe how such PPU code can initiate basic data transfers between main storage and LS of some SPE.

For detailed information on the MMIO (or Direct Problem State) interface and information on the MFC commands, see *SPE Channel and Related MMIO Interface* chapter and *DMA Transfers and Interprocessor Communication* chapter respectively on Cell Broadband Engine Programming Handbook.

Note: The tag ID used for the PPE initiated DMA transfer is not related to the tag ID used by the software that runs on this SPE - each of them related to a different queue of the MFC. There is currently no mechanism for allocating tag IDs on the PPE side (like the SPE's tag manager) so the programmer should use some predefined tag ID. Since tag IDs 16 to 31 are reserved for the Linux kernel, the user must use only tag IDs 0 to 15.

Another alternative for a PPU software to access the LS of some SPE is mapping the LS to main storage and then use regular direct memory access. This issue is discussed in chapter "Direct PPE access to LS of some SPE" on page 143.

Basic DMA transfer between LS and main storage

This chapter describes how PPU software can transfer data between the LS of some SPE and main storage using basics DMA commands. That term 'basic' commands implies to commands that should be explicitly issued for each DMA transaction separately, unlike DMA list commands.

We describe how the PPU may initialize those command and illustrate it using a code example of 'get' and 'put' commands. The available DMA commands are described in Chapter , "DMA commands" on page 111.

Please note that the naming of the commands is based on a SPE centric view, for example, 'put' means a transfer from the SPE LS to an effective address.

Note: The programmer should try to avoid initiating DMA commands from the PPE and prefer initiating them by the local SPE. First reason is that accessing the MMIO by the PPE is executed on the interconnect bus which has larger latency then the SPU accessing the local channel interface. (The latency is high because the SPE problem state is mapped as guarded, cache inhibited memory.) Second, by adding this traffic it reduces the available bandwidth for other resources on the interconnect bus. Third, the PPE is expensive resource anyway so it is better to have the SPEs to do more work instead.

Initiate a DMA transfer

To initialize a DMA transfer the PPE programmer can call one of the corresponding functions of `libspe2.h` header file. Each of those functions implements a single command, such as:

- `spe_mfcio_get`: implements 'get' command.
- `spe_mfcio_put`: implements 'put' command.

Those functions are nonblocking so the software will continue its execution after issuing those commands. Full list of the supported commands are in Table 4-3 on page 112.

Another alternative is using the inline function that is defined in `cbe_mfc.h` file and support all the DMA commands:

- `_spe_mfc_dma` : Enqueues a DMA request using the values provided. The function supports all types of DMA commands according to the value of 'cmd' input parameter (e.g. 'cmd' parameter set to MFC_PUT_CMD for 'put' command, set it to MFC_GETS_CMD for 'gets' command, etc.). This function will block until the MFC queue has space available (in the `cbe_mfc.h` file) and is preferred from a performance point of view over the `libspe2.h` functions.

Note: When issuing DMA commands from the PPE, using the `cbe_mfc.h` functions are preferred from performance point of view over the `libspe2.h` functions. While the `cbe_mfc.h` functions are not well described in the SDK documentation they are quite straight forward and easy to use. In our examples we used the `libspe2.h` functions.

The programmer should be aware of the fact that the implementation of those functions actually involve a sequence of the following commands:

The programmer should be aware of the fact that the implementation of those functions actually involve a sequence of the following commands:

1. Write LSA (local store address) parameter to MFC_LSA register.

2. Write EAH (effective address higher bits) and EAL (effective address lower bits) parameters to MFC_EAH registers respectively.
   Software can implement is by two 32-bit stores or one 64-bit store.
3. Write transfer size and tag ID parameters to MFC_Size and MFC_TagID registers respectively.
   Software can implement it by one 32-bit store (MFC_Size in upper 16 bits, MFC_TagID in lower 16 bits) or along MFC_ClassID_CMD in one 64-bit store.
4. Write class ID and command opcode to MFC_ClassID_CMD register. The opcode defines the transfer type (e.g. ‘get’, ‘put’).
5. Read the MFC_CMDStatus register using a single 32 bits store to determine the success or failure of the attempt to enqueue a DMA command, as indicated by the 2 least-significant bits of returned value:
   - 0: The enqueue was successful.
   - 1 – Sequence error occurred while enqueuing the DMA (e.g. interrupt occurred, then another DMA was started within interrupt handler). Software should restarted the DMA sequence by going to step 1.
   - 2: The enqueue failed due to insufficient space in command queue. Software could either wait for space to become available before attempting the DMA transfer again, or can simply continue attempting to enqueue the DMA until successful (go to step 1).
   - 3: Indicates that both errors occurred.

Note: The supported and recommnded value of the different DMA parameters are described in “Supported and recommended values for DMA parameters” on page 115.

Waiting for completion of a DMA transfer

After DMA command is initiated, the software may wait for a completion of the DMA transaction. Programmer may do so by calling to one of the functions that are defined in `libspe2.h` header. For example:

- `spe_mfcio_tag_status_read`: The function input parameters include a mask which defines group ID (as explained below) and blocking behavior (continue waiting until completion or quit after one read).

The programmer should be aware of the fact that the implementation of this function include a sequence of the following commands:

1. Set the Prxy_QueryMask register to the groups of interest. Each tag ID is represented by one bit (tag 31 is assigned the most-significant bit and tag 0 is assigned the least-significant bit).

2. Issue an eieio instruction before reading the Prxy_TagStatus register to ensure the effects of all previous stores complete
3. Read the Prxy_TagStatus register.
4. If the value is nonzero, at least one of the tag groups of interest has completed. If waiting for all the tag groups of interest to complete, XOR the tag group status value with the tag group query mask. A result of '0' indicates that all groups of interest are complete.
5. Repeat steps 3 and 4 until the tag groups of interest are complete.

Another alternative is using the inline functions that is defined in `cbe_mfc.h` file:

- `_spe_mfc_write_tag_mask` : A nonblokcing function which writes the mask value to the Prxy_QueryMask register.
- `_spe_mfc_read_tag_status_immediate` : A nonblokcing function which reads the Prxy_TagStatus register and returns the value read. Beore calling this function, the `_spe_mfc_write_tag_mask` function should be called to set the tag mask.

There are various other methods to wait for the completion of the DMA transfer o as described in chapter *PPE-Initiated DMA Transfers* in Cell Broadband Engine Programming Handbook document. We chose to show the simplest one.

Basic DMA 'get' and 'put' transfers - code example

This section contains a code example on how PPU program may initiate basic DMA transfers between the LS and main storage. This example demonstrate the following techniques:

- PPU code maps the LS to share memory and retrieve pointer to its EA base.
- SPU code uses the mailbox to send PPU the offset to its data buffer in LS.
- PPU code initiates DMA 'put' command to transfer data from LS to main storage (please notice that the direction of this commands may be confusing).
- PPU code wait for the completion of the 'put' command before using the data.

Example 4-22 shows the PPU code while Example 4-23 shows the corresponding SPU code.

We use the MFC functions method to access the DMA mechanism from the PPU side. Each of such functions actually implements few of the steps that were mentioned above causing the code to be simpler. From programmer point of view it is important to be familiar with the number of commands that are involve in order to understand the impact on its application execution.

Source code: The code of Example 4-22 and Example 4-23 is included in the additional material that is provided with this book. See “PPU initiated DMA transfers between LS and main storage” on page 614 for more information.

Example 4-22 PPU initiated DMA transfers between LS and main storage - PPU code

```
#include <libspe2.h>
#include <cbe_mfc.h>

#define BUFF_SIZE 1024

spe_context_ptr_t spe_ctx;

uint32_t ls_offset;  // offset from LS base of the data buffer in LS

// PPU's data buffer
volatile char my_data[BUFF_SIZE] __attribute__ ((aligned(128)));

int main(int argc, char *argv[]){

   int ret;
   uint32_t tag, status;

   // MUST use only tag 0-15 since 16-31 are used by kernel
   tag = 7; // choose my lucky number

   spe_ctx = spe_context_create (....); // create SPE context
   spe_program_load (....);            // load SPE program to memory
   pthread_create (....);             // create SPE pthread

   // collect from the SPE the offset in LS of the data buffer. NOT the
   // most efficient using mailbox- but sufficient for initialization
   while(spe_out_mbox_read( data.spe_ctx, &ls_offset, 1)<=0);

   //intiate DMA 'put' command to transfer data from LS to main storage
   do{
      ret=spe_mfcio_put( spe_ctx, ls_offset, (void*)my_data, BUFF_SIZE,
                         tag, 0,0);
   }while( ret!=0);

   // wait for completion of the put command
   ret = spe_mfcio_tag_status_read(spe_ctx,0,SPE_TAG_ALL, &status);

   if(ret!=0){
```

```
      perror ("Error status was returned");
      // 'status' variable may provide more information
      exit (1);
   }

   // MUST issue synchronization command before reading the 'put' data
   __lwsync();

   printf("SPE says: %s\n", my_data);

   // continue saving the world or at least managing the 16 SPEs

   return (0);

}
```

Example 4-23 PPU initiated DMA transfers between LS and main storage - SPU code

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

#define BUFF_SIZE 1024

// SPU's data buffer
volatile char my_data[BUFF_SIZE] __attribute__ ((aligned(128)));

int main(int speid , uint64_t argp)
{
   strcpy((char*)my_data, "Racheli Paz lives in La-Paz.\n" );

   // send to PPE the offest the data buffer- stalls if mailbox is full
   spu_write_out_mbox((uint32_t)my_data);

   // continue helping PPU saving the world or at least do what he says

   return 0;
}
```

DMA list data transfer

A DMA list is a sequence of transfer elements (or list elements) that, together with an initiating DMA-list command, specifies a sequence of DMA transfers

between a single continuous area of LS and possibly discontinuous areas in main storage.

The PPE software may participate in the initiating of such DMA-list command by create and initialize the DMA list in an SPE's LS. The way on which a PPU software can access the LS is described in Chapter , "Direct PPE access to LS of some SPE" on page 143. Once such list was created only a code running on this SPU may proceed with the execution of the command itself. This entire process is described in Chapter , "DMA list data transfer" on page 124.

4.3.4 Direct problem state access and LS to LS transfer

This chapter describes how applications can access directly an SPE's LS. The intention is for applications that do not run on this SPE but runs on either the PPE or other SPEs.

PPE access to the LS is described in the first chapter "Direct PPE access to LS of some SPE". Programmer should try to avoid massive use of this technique because of performance considerations.

Other SPEs accessing the LS is described on the next chapter - "SPU initiated LS to LS DMA data transfer". For memory bandwidth reasons it is highly recommended to prefer this technique whenever it fit the application structure.

Direct PPE access to LS of some SPE

In this chapter we describe how the PPE can directly access the LS of some SPE. The programmer should try to avoid frequent PPE direct access to the LS and should try to use DMA transfer instead. However, it may be useful to use direct PPE to LS access of occasionally with small amount of data in order to control the program flow, for example to write a notification.

A code running on the PPU can access the LS by performing the following steps:

1. Map the LS to the main storage and provides an effective address pointer to the LS base address. Function `spe_ls_area_get` of the `libspe2.h` header file implements this step as described in SPE Runtime Management library document. Note that this type of memory access is not cache coherent.
2. Optionally, get from the SPE the offset compare to the LS base of the data to be read or written. May be implemented using the mailbox mechanism.
3. Access this region like any regular data access to main storage using direct load and store instructions.

Note: The LS stores the SPU program's instructions, program stack as well as global data structure. The PPU code should therefore be cautious in accessing the LS in order to prevent override those SPU program's components. The recommended way to do so is letting the SPU code manage its LS space. Using any other communication technique, the SPU code can send to the PPE the offset of the region in LS that the PPE may access.

Example 4-24 shows code that illustrates how a PPE may access the LS of an SPE:

- The SPU program forwards the offset of the corresponding buffer in the LS to the PPE using the mailbox mechanism.
- The PPE program maps the base address of the LS to the main storage using libspe function
- The PPE program adds the buffer offset to the LS base address to retrieve the buffer effective address.
- The PPE program uses the calculated buffer address to access the LS (and copy some data to it).

Source code: The code of Example 4-24 is included in the additional material that is provided with this book. See "Direct PPE access to LS of some SPE" on page 614 for more information.

Example 4-24 Direct PPE access to LS of some SPE

```
// Take 'spu_data_t' stucture and 'spu_pthread' function from
// Example 4-5 on page 90

#include <ppu_intrinsics.h>

uint64_t ea_ls_base; // effective address of LS base
uint32_t ls_offset;  // offset (LS address) of SPE's data buffer
uint64_t ea_ls_str;  // effective address of SPE's data buffer

#define BUFF_SIZE 256

int main(int argc, char *argv[])
{
   uint32_t mbx;

   // create SPE thread as shown in Example 4-3 on page 86
```

```
   // map SPE's LS to main storage and retrieve its effective address
   if( (ea_ls_base = (uint64_t)(spe_ls_area_get(data.spe_ctx)))==NULL){
      perror("Failed map LS to main storage"); exit(1);
   }

   // read from SPE the offset to the data buffer on the LS
   while(spe_out_mbox_read( data.spe_ctx, &ls_offset, 1)<=0);

   // calculate the effective address of the LS data buffer
   ea_ls_str = ea_ls_base + ls_offset;

   printf("ea_ls_base=0x%llx, ls_offset=0x%x\n",ea_ls_base, ls_offset);

   // copy a data string to the LS
   strcpy( (char*)ea_ls_str, "Ofer Thaler is lemon's lemons");

   // make sure that writing the string to LS is complete before
   // writing the mailbox notification
   __lwsync();

   // use mailbox to notify SPE that the data is ready
   mbx = 1;
   spe_in_mbox_write(data.spe_ctx, &mbx,1,1);

   // wait SPE thread completion as shown in Example 4-3 on page 86

   printf("PPE: Complete this educating (but useless) example\n" );

   return (0);
}
```

SPU initiated LS to LS DMA data transfer

This section contains a code example on how SPU program may access LS of another SPE in the chip. The LS is mapped to an effective address in the main storage which allows SPEs to use ordinary DMA operations to transfer data to and from this LS.

It is highly recommended to prefer LS to LS data transfer whenever it fits the application structure. This type of data transfer is very efficient because it goes directly from SPE to SPE on the internal EIB bus without involving the main memory interface. The internal bus have much higher bandwidth then the memory interface (up to 10 times faster) and lower latency.

The following steps may be taken to enable a group of SPEs to initiated DMA transfer between each other local stores:

1. PPE maps the local stores to the main storage provides an effective address pointer to the local stores base addresses.
   Function `spe_ls_area_get` of `libspe2.h` header file implement this step as described in SPE Runtime Management library document.
2. SPEs send to the PPE the offset compare to the LS base of the data to be read or written.
3. PPE provides the SPEs the LS base addresses and the data offsets of other SPEs. May be implemented using mailbox also.
4. SPEs access this region like regular DMA transfer between LS and effective address on the main storage.

Source code: An code example that uses LS to LS data transfer to implement a multistage pipeline programming mode is available as part the additional material that is provided with this book. See “Multistage pipeline using LS to LS DMA transfer” on page 614 for more information.

4.3.5 Facilitate random data access using SPU software cache

This chapter discuss the software cache[1] library which is a part of the SDK package and is based on the following principles:

- Provides a set of SPU functions calls to manage the data on the LS and to transfer data between the LS and main storage.
- The library maintain a cache memory that is statically allocated on the LS.
- From the programer point of view accessing the data using the software cache is similar to using ordinary load and store instructions, unlike the SPU’s typical DMA interface for transferring data.
- For each data on the main storage that the program try to access, the cache mechanism first check if it is already located in the cache (i.e. in LS). If it does, the data is simply taken from there and by that saves the latency of bringing the data from the main storage. Otherwise - the cache automatically and transparent to the programer perform DMA transfer.
- the cache also provides asynchronous interface which, like double buffering, enables the programmer to hide the memory access latency by overlapping between data transfer and computation.

[1] While in this chapter we call this library ‘software cache’, its full name is actually ‘SPU software managed cache’. We use the shorted name for simplicity. The library specification is in Example Library API Reference document.

The library has the following advantages over using standard SDK functions call to activate the DMA transfer:

- Better performance in some applications[2] can be achieved by taking advantage of locality of reference and save redundant data transfers if the corresponding data is already in LS.
- Use familiar load/store instructions with effective address which are easier to program in most cases.
- Since the topology and behavior of the cache is configurable, can be easily optimized to match data access patterns (unlike most hardware cache).
- Decreases the development time that is needed to port some application to SPE.

However, since the software cache functions add some computation overhead compare to ordinary DMA data transfers, in case the data access pattern is sequential it is therefore preferred from performance point of view to use ordinary DMA data transfer instead.

Note: The software cache activity is local to a single SPU, managing the data access of such SPU program to the main storage and LS.
The software cache does not coordinate between data accesses of different SPUs to main storage neither take care of coherency between them.

The chapter discuss the following issues:

- “Main features of the software cache” - summarizes the main features of the software cache and how the programer may configure them.
- “Software cache operation modes” - discuss the two different modes that are supported by the cache to perform either synchronous or asynchronous data transfer.
- “Programing using software cache” - shows how to practically program using the software cache include some code examples.
- “When and how to use the software cache” - provides some examples of application where using the software cache is beneficial.

Main features of the software cache

Many features related to the software cache topology and behavior can be configured by the programmer which creates an advantage to the software cache over hardware cache in some cases. Configuring those features allows the

[2] Chapter , “When and how to use the software cache” on page 153 provides some examples for applications for which the software cache provides a good solution.

programer to iteratively adjust those cache attributes to what best suites the specific application that currently runs in order to achieve optimal performance.

The main feature of the software cache are:

- Associativity: direct mapped, 2-way, or 4-way set associative.
- Access mode: read-only or read-write (the former has better performance).
- Cache line size[3]: 16 B to 4 KB (a power of two values)
- Number of lines: 1 to 4K lines (a power of two values)
- Data type: any valid data type (but all the cache has the same type).

In order to set those software cache attributes the programer should statically add the corresponding definition to the program code. It means that the cache attributes are taking into account during compilation of the SPE program (i.e. and not on run time) when many of the cache structures and functions are constructed.

In addition, the programer should assign a specific name to the software cache which allows to define several separate caches in the same program. This may be useful in case several different data structures are accessed by the program and each structure has different attributes (e.g. some structures are read-only and some are read-write, some are integers and some single precision)

By default, the software managed cache may use the entire range of the 32 tag ID which are available for DMA transfers and doesn't not take into account other application uses of tag IDs. If a program also initiate DMA transfers (which require separate tag IDs) the programmer should limit the number of tag ID used by the software cache by explicitly configure range of tag IDs that the software cache may use.

Software cache operation modes

The software cache support two different modes in which the programer may use the cache once it was created. The two supported modes are 'safe mode' and 'unsafe mode' which are discussed in next two chapters.

Safe mode and synchronous interface

The safe interfaces provide the programmer with a set of functions to access data simply by using the data's effective address. The software cache library performs the data transfer between LS and the main memory transparently to the programer and manages the data that is already in LS.

[3] Unless explicitly mentioned otherwise, we use the term 'cache line' for the entire chapter to define the software cache line. While hardware cache line is fixed to 128 B, the software line can be configured to any power of 2 value between 16B to 4 KB.

One of the advantage using this method is that the programmer doesn't need to worry the LS addresses and can simply use effective addressees like any other PPE program. From programing point of view it is therefore very simple.

Data access function call using this mode are done synchronously and are performed according to the following guidelines:

- If data is already in LS (in the cache) - a simple load from LS is performed.
- if data is not currently in LS - software cache function perform DMA between LS and main storage and the program is blocked until the DMA is completed.

Such synchronous interface has the disadvantage of having a long latency when the software cache needs to transfer data between LS and main storage.

Unsafe mode and asynchronous interface

The unsafe provides a more efficient means of accessing the LS compared to the safe services. Software cache provides functions to map effective addresses to LS addresses. The programer should later use those LS address to access the data (unlike in safe mode where the effective addresses are used).

In this mode, like in safe mode, the software cache keeps tracking which data is already in the LS and perform data transfer between LS and main storage only if the data is not currently in LS.

One of the advantages when using this method, is that the programer may ask the software cache to asynchronously prefetch the data by 'touching' it. The programer can implement double buffering like data transfer, letting the software cache start transferring the next data to be processed while the program can continue performing computation on the current data.

The disadvantage of using this mode is that programming is slightly more complex. The programmer should access the data using the LS address and use software cache functions to lock the data in case the data is updated. For optimal performance software cache functions for prefetching the data may be called.

Programing using software cache

This chapter demonstrate how the programmer can use the software cache in an SPU program. The following programming techniques are shown in the next sections:

- "Constructing a software cache" shows how to construct a software cache and define its topology and behavior attributes.
- "Synchronous data access using safe mode" shows how to perform synchronous data access using the library's safe mode.

- “Asynchronous data access using unsafe mode” shows how to perform asynchronous data access using the library’s unsafe mode.

Source code: The code of examples that are presented in this section - Example 4-25 through Example 4-27, is included in the additional material that is provided with this book. See “SPU software managed cache” on page 614 for more information.

Constructing a software cache

In order to construct a software cache on a SPU program the programer should define a couple of required attributes may define other optional attributes which defines the cache topology and behavior. If the programer choose not to define those attributes the library sets default values to those attributes. In either case it is recommended to be familiar with all the attributes since their values may effect the performance.

The next thing in the code following those definition must be the including of the cache header file that is located at:

```
/opt/cell/sdk/usr/spu/include/cache-api.h
```

Multiple caches may be defined in the same program by re-defining these attributes and re-including the `cache-api.h` header file. The only restriction is that the CACHE_NAME must be different for each cache.

Example 4-25 shows a code example of using the software cache. The example shows how to:

- Construct a software cache named `MY_CACHE` and define both its mandatory and optional attributes.
- Reserve tag ID to be used by the software cache.

Example 4-25 Constructing software cache

```
unsigned int tag_base; // should be defined before the cache

// Manadatory attributes
#define CACHE_NAME         MY_CACHE // name of the cache
#define CACHED_TYPE        int      // type of basic element in cache

// Optional attributes
#define CACHE_TYPE    CACHE_TYPE_RW // rw type of cache
#define CACHELINE_LOG2SIZE 7        // 2^7 = 128 bytes cache line
#define CACHE_LOG2NWAY     2        // 2^2 = 4-way cache
#define CACHE_LOG2NSETS    4        // 2^4 = 16 sets
#define CACHE_SET_TAGID(set) (tag_base + (set & 7)) // use 8 tag IDs
```

```
#define CACHE_STATS//collect statistics
#include <cache-api.h>

int main(unsigned long long spu_id, unsigned long long parm)
{

   // reserve 8 tags for the software cache
   if((tag_base=mfc_multi_tag_reserve(8))==MFC_TAG_INVALID){
      printf( "ERROR: can't reserve a tags\n"); return 1;
   }

   // can use the cache here
   ...
```

Synchronous data access using safe mode

Once the caches was define, the programer can use its function calls to access data. This chapter shows how to perform synchronous data access using the safe mode.

Please notice that using this mode only the effective addresses are used to access the main memory data and there is no need for the programmer to be aware of the LS address to which the data was transferred (i.e. by the software cache).

The code in Example 4-26 shows how to do the following:

- Use safe mode to perform synchronous data access.
- Flush the cache so the modified data will be written into main memory.
- Reads variables from main memory using their effective address, modify them and write them back to memory using their effective address.

Example 4-26 Synchronous data access using safe mode

```
Take Example 4-25 code to construct the cache and initialize tag IDs

   int a, b;
   unsigned eaddr_a, eaddr_b;

   // initialize effective addresses from PPU parameter
   eaddr_a = parm;
   eaddr_b = parm + sizeof(int);

   // read a and b from effective address
   a = cache_rd(MY_CACHE, eaddr_a);
```

```
b = cache_rd(MY_CACHE, eaddr_b);

// write values into cache (no write-through to main memory)
cache_wr(MY_CACHE, eaddr_b, a);
cache_wr(MY_CACHE, eaddr_a, b);

// at this point only the variables in LS are modified

// writes all modified (dirty) cache lines back to main memory
cache_flush(MY_CACHE);

...
```

Asynchronous data access using unsafe mode

This chapter shows how to perform asynchronous data access using the unsafe mode.
In addition the chapter show how the programer can print cache statistics that provide information about the cache activity. Those statistics may later be used to tune the cache topology and behavior.

Please notice that using this mode the software cache maps the effective address of the data in the main memory into local store. The programer should later use the mapped local addresses to use the data.

The code in Example 4-27 shows how to do the following:

- Use unsafe mode to perform synchronous data access.
- Touch a variable so the cache will start asynchronous prefetching of this variable from main memory to local store.
- Wait till the prefetched data is present in LS before modifying it.
- Flush the cache so the modified data will be written into main memory.
- Print software cache statistics.

Example 4-27 Asynchronous data access using unsafe mode

```
// Take the begining of the program from Example 4-26
   ...

   int *a_ptr, *b_ptr;

   // asynchronously touch data 'b' so cache will start to prefetch it
   b_ptr = cache_touch(MY_CACHE, eaddr_b);
```

```
    // synchronously read data 'a' - blocked till data is present in LS
    a_ptr = cache_rw(MY_CACHE, eaddr_a);

    // MUST lock variables in LS since it will be modified.
    // ensures that it will not cast out while the reference is held.
    cache_lock(MY_CACHE,a_ptr);

    // 'a' is locked in cache - can now safely be modified through ptr
    *a_ptr = *a_ptr+10;

    // blocking function that waits till 'b' is present in LS
    cache_wait(MY_CACHE, b_ptr);

    // need to lock 'b' since it will be updated
    cache_lock(MY_CACHE,b_ptr);

    // now 'b' is in cache - can now safely be modified through ptr
    *b_ptr = *b_ptr+20;

    // at this point only the variables in LS are modified

    // writes all modified (dirty) cache lines back to main memory
    cache_flush(MY_CACHE);

    //print software cache statistics
    cache_pr_stats(MY_CACHE);

    mfc_multi_tag_release(tag_base, 8);
    return (0);
}
```

When and how to use the software cache

In this chapter we discuss two main cases in which we recommend to use the software cache. Each of the next four sections define one such case and also discuss how it is recommended to use the software cache in this case (mainly regarding which safe/unsafe mode should be selected).

Note: Using unsafe mode and performing asynchronous data access provide better performance then using safe mode's synchronous access. It depends on the specific program if the performance improvement is indeed significant. However, programing is slightly more complex using the unsafe mode.

Case 1: First pass SPU implementation

This section refer to cases in which there is a need to develop a first pass implementation of an application on the SPU in relatively short time. If the programer use the software cache in safe mode the program is not significantly different neither requires more programming compare to other single processor program (e.g. program that runs on the PPE).

Case 2: Random data access with high cache hit rate

Some application has random or not predicted data access pattern which make it hard to implement a simple and efficient double or multi buffering mechanism. In this section we mainly refer to streaming applications which contain many iterations, and in each iteration few blocks of data are read and are used as an input for computing some output blocks. Examples for such applications include:

- The data blocks that are accessed by the program are scattered in memory[4] and are relatively small.
- Indirect mechanism, in which the program should first read index vectors from main storage and those vectors contains the location of the next blocks of data that need t be processed.
- Only after computing the results of current iteration the program know which blocks should be read next.

If those application also have high cache hit rate then the software cache can provided better performance compare to other techniques. Such high rate may be occur if blocks that are read in one iteration are likely to be used in the sequential iterations. Or another similarly is if blocks that are read in some iteration are close enough to the blocks of previous iteration (i.e. in the same software cache line).

A high cache hit rate ensures that in most cases when a structure is accessed by the program, the corresponding data is already in the LS so the software cache will be smart enough to take the data form the LS instead of transferring it again from main memory.

If the hit rate is significantly high, performing synchronous data access using safe mode will provide good performance since waiting for data transfer completion will not occur too often. However, in most case the programer may try to use asynchronous data access of unsafe mode and measure whether performance improvement is indeed achieved.

[4] The intention is not that the data itself is necessarily scattered in memory, but each iteration of the algorithm uses several different blocks which aren't continuos in memory (scattered).

4.3.6 Automatic software caching on SPE

A simple way for an SPE to reference data on the main storage can be achieved through an extension to the C language syntax which enables to share data in this way between an SPE and the PPE or between SPEs. This extension makes it easier to pass pointers so that the programer can use the PPE to perform certain functions on behalf of the SPE. Similarly this mechanism can be used to share data between all SPEs through variables in PPE address space.

This mechanism is based on the software cache safe mode mechanism that was discussed in Chapter 4.3.5, “Facilitate random data access using SPU software cache” on page 146 but provides a more user friendly interface to activate it.

In order to use this mechanism the programer should use the `__ea` address space identifier when declaring a variable to indicate to the SPU compiler that a memory reference is in the remote (or effective) address space, rather than in local store. The compiler automatically generates code to DMA these data objects into local store and caches references to these data objects.

This identifier can be used as an extra type qualifier like `const` or `volatile` in type and variable declarations. The programer can qualify variable declarations in this way, but not variable definitions.

Accessing an `__ea` variable from an SPU program creates a copy of this value in the local storage of the SPU. Subsequent modifications to the value in main storage are not automatically reflected in the copy of the value in local store. It is the programer’s responsibility to ensure data coherence for `__ea` variables that are accessed by both SPE and PPE programs.

The following are examples on how to use this variable:

```
// Variable declared on the PPU side.
extern __ea int ppe_variable;

// Can also be used in typedefs.
typedef __ea int ppe_int;

// SPU pointer variable point to memory in main storage address space
__ea int *ppe_pointer;
```

The SPU program should initiate this pointer to a valid effective address:

```
// Init the SPU pointer according to ‘ea_val’ which is a valid effective
// address that PPU forward to the SPU (e.g. using mailbox)
ppe_pointer = ea_val;
```

After the pointer was initiated, it can be used as an any SPU pointer while the software cache will map it into DMA memory access, for example:

```
for (i = 0; i < size; i++) {
     *ppe_pointer++ = i;  // memory accesses use software cache
}
```

Another case if pointers in the SPE's LS space that can be cast to pointers in the main storage address space. Doing this transforms an LS address into an equivalent address in the main storage (as the LS is mapped also to the main storage domain). The following is an example.

```
int x;
__ea int *ppe_pointer_to_x = &x;
```

The pointer variable `ppe_pointer_to_x` can be passed to the PPE process by of a mailbox and used by PPE code to access the variable `x` in the LS. The programer should be aware or the ordering issues in case both the PPE access this variable (from the main storage) and SPE access it (from the LS domain). Similarly this pointer can be used to transfer data between on LS to another by the SPEs.

GCC for the SPU provides the following command line options to control the runtime behavior of programs that use the `__ea` extension. Many of these options specify parameters for the software-managed cache. In combination, these options cause GCC to link the program to a single software-managed cache library that satisfies those options. Table 4-5 describes these options:

Table 4-5 GCC options for supporting main storage access from the SPE

Option	Description
-mea32	Generate code to access variables in 32-bit PPU objects. The compiler defines a preprocessor macro `__EA32__` to allow applications to detect the use of this option. This is the default.
-mea64	Generate code to access variables in 64-bit PPU objects. The compiler defines a preprocessor macro `__EA64__` to allow applications to detect the use of this option.
-mcache-size=X	Specify an X KB cache size (X=8, 16, 32, 64 or 128)
-matomic-updates	Use DMA atomic updates when flushing a cache line back to PPU memory. This is the default.
-mno-atomic-updates	This negates the -matomic-updates option.

A complete example using __ea qualifiers to implement a quick sort algorithm on the SPU accessing PPE memory can be found in the SDK's `/opt/cell/sdk/src/examples/ppe_address_space` directory.

4.3.7 Efficient data transfers by overlapping DMA and computation

One of the unique features of the Cell BE architecture is the DMA engines in each of the SPEs which enables asynchronous data transfer. In this chapter we discuss fundamental techniques to achieve overlapping between data transfers and computation using the DMA engines. This is an important topic as it enables to dramatically increase the performance of many applications.

Motivation

Consider a simple SPU program that repeating the following steps:

1. DMA incoming data from main storage to LS buffer B.
2. Wait for the transfer to complete.
3. Compute on data in buffer B.

This sequence is not efficient because it waste a lot of time waiting for the completion of the DMA transfer and has no overlap between data transfer and computation. The time graph for such scenario is illustrate in Figure 4-2:

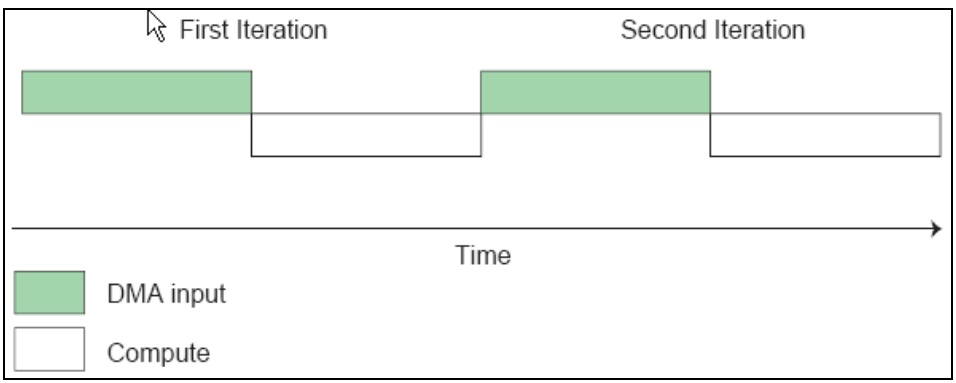


Figure 4-2 Serial computation and data transfer

Double buffering

We can significantly speed up the process described above by allocating two buffers, B_0 and B_1, and overlapping computation on one buffer with data transfer in the other. This technique is called *double buffering* whose flow diagram scheme is shown in Figure 4-3:

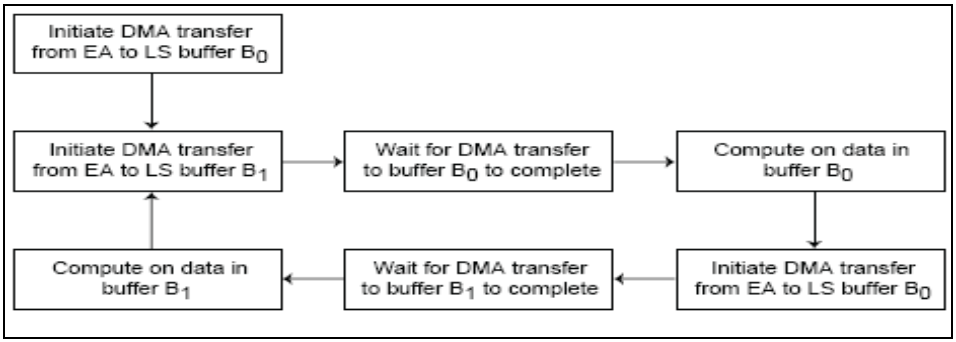


Figure 4-3 Double buffering scheme

Double buffering is a private class of *multibuffering*, which extends this idea using multiple buffers in a circular queue instead of only the two buffers of double buffering. While In most cases using the two buffers in the double buffering case are enough to guarantee overlapping between computation and data transfer.

However, in case the software still need to wait for completion of the data transfer, the programmer may consider extending the number of buffers and move to multibuffering scheme. Obviously this requires more memory on the LS which may be a problem in some cases. The multibuffering technique is described in “Multibuffering” on page 163.

Below is an example code for double buffering mechanism. Example 4-28 is the header file which is common to the SPE and PPE side, Example 4-29 is the SPU code that contains the double buffering mechanism, and Example 4-30 is the corresponding PPU code.

Source code: The code of Example 4-28 through Example 4-30, is included in the additional material that is provided with this book. See “Double buffering” on page 615 for more information.

The code also demonstrate the use of barrier on the SPE side to ensure that all the output data that SPE updates in memory is written into memory before PPE tries to read it.

Example 4-28 Double buffering code - common header file

```
// common.h file -------------------------------------------------

#define ELEM_PER_BLOCK 1024 // # of elements to process by the SPE
#define NUM_OF_ELEM 2048*ELEM_PER_BLOCK // total # of elements
```

```
#define STATUS_DONE      1234567
#define STATUS_NO_DONE ~(STATUS_DONE)

typedef struct {
   uint32_t *in_data;
   uint32_t *out_data;
   uint32_t *status;
   int size;
} parm_context;
```

Example 4-29 Double buffering mechanism - SPU code

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include "common.h"

// Macro for waiting to completion of DMA group related to input tag
#define waitag(t) mfc_write_tag_mask(1<<t); mfc_read_tag_status_all();

// Local store structures and buffers.
volatile parm_context ctx __attribute__ ((aligned(16)));;
volatile uint32_t ls_in_data[2][ELEM_PER_BLOCK] __attribute__
((aligned(128)));
volatile uint32_t ls_out_data[2][ELEM_PER_BLOCK] __attribute__
((aligned(128)));
volatile uint32_t status __attribute__ ((aligned(128)));

uint32_t tag_id[2];

int main(unsigned long long spu_id, unsigned long long argv)
{
   int buf, nxt_buf, cnt, nxt_cnt, left, i;
   volatile uint32_t *in_data, *nxt_in_data, *out_data, *nxt_out_data;

   tag_id[0] = mfc_tag_reserve();
   tag_id[1] = mfc_tag_reserve();

   // Fetch the  parameter context, waiting for it to complete.
   mfc_get((void*)(&ctx), (uint32_t)argv, sizeof(parm_context),
            tag_id[0], 0, 0);
   waitag(tag_id[0]);

   // Init parameters
   in_data  = ctx.in_data;
```

```
out_data = ctx.out_data;
left     = ctx.size;
cnt = (left<ELEM_PER_BLOCK) ? left : ELEM_PER_BLOCK;

// Prefetch first buffer of input data.
buf = 0;
mfc_getb((void *)(ls_in_data), (uint32_t)(in_data),
         cnt*sizeof(uint32_t), tag_id[0], 0, 0);

while (cnt < left) {
   left -= SPU_Mbox_Statnt;

   nxt_in_data  = in_data + cnt;
   nxt_out_data = out_data + cnt;
   nxt_cnt = (left<ELEM_PER_BLOCK) ? left : ELEM_PER_BLOCK;

   // Prefetch next buffer so it is available for next iteration.
   // IMPORTANT: Put barrier so that we don't GET data before
   //            the previous iteration's data is PUT.
   nxt_buf = buf^1;

   mfc_getb((void*)(&ls_in_data[nxt_buf][0]),
            (uint32_t)(nxt_in_data) , nxt_cnt*sizeof(uint32_t),
            tag_id[nxt_buf], 0, 0);

   // Wait for previously prefetched buffer
   waitag(tag_id[buf]);

   for (i=0; i<ELEM_PER_BLOCK; i++){
      ls_out_data[buf][i] = ~(ls_in_data[buf][i]);
   }

   // Put the output buffer back into main storage
   mfc_put((void*)(&ls_out_data[buf][0]), (uint32_t)(out_data),
            cnt*sizeof(uint32_t),tag_id[buf],0,0);

   // Advance parameters for next iteration
   in_data  = nxt_in_data;
   out_data = nxt_out_data;

   buf = nxt_buf;
   cnt = nxt_cnt;
 }

 // Wait for previously prefetched buffer
```

```
    waitag(tag_id[buf]);

    // process_buffer
    for (i=0; i<ELEM_PER_BLOCK; i++){
      ls_out_data[buf][i] = ~(ls_in_data[buf][i]);
    }
    // Put the output buffer back into main storage
    // Barrier to ensure all data is written to memory before status
    mfc_putb((void*)(&ls_out_data[buf][0]), (uint32_t)(out_data),
            cnt*sizeof(uint32_t), tag_id[buf],0,0);

    // Wait for DMAs to complete
    waitag(tag_id[buf]);

    // Update status in memory so PPE knows that all data is in place
    status = STATUS_DONE;

    mfc_put((void*)&status, (uint32_t)(ctx.status), sizeof(uint32_t),
             tag_id[buf],0,0);
    waitag(tag_id[buf]);

    mfc_tag_release(tag_id[0]);
    mfc_tag_release(tag_id[1]);

   return (0);
}
```

Example 4-30 Double buffering mechanism - PPU code

```
#include <libspe2.h>
#include <cbe_mfc.h>
#include <pthread.h>

#include "common.h"

volatile parm_context ctx __attribute__ ((aligned(16)));
volatile uint32_t in_data[NUM_OF_ELEM] __attribute__ ((aligned(128)));
volatile uint32_t out_data[NUM_OF_ELEM] __attribute__ ((aligned(128)));

volatile uint32_t status __attribute__ ((aligned(128)));

// Take ‘spu_data_t’ stucture and ‘spu_pthread’ function from
// Example 4-5 on page 90
```

```
int main(int argc, char *argv[])
{
   spe_program_handle_t *program;
   int i, error;

   status = STATUS_NO_DONE;

   // Init input buffer and zero output buffer
   for (i=0; i<NUM_OF_ELEM; i++){
      in_data[i] = i;
      out_data[i] = 0;
   }

   ctx.in_data  = in_data;
   ctx.out_data = out_data;
   ctx.size     = NUM_OF_ELEM;
   ctx.status   = &status;

   data.argp = &ctx;

   // ... Omitted section:
   // creates SPE contexts, load the program to the local stores,
   // run the SPE threads, and waits for SPE threads to complete.

   // (the entire source code for this example is part of the book's
   // additional material).

   // This subject of is also described in 4.1.2, "Task parallelism and
managing SPE threads"


   // Wait for SPE data to be written into memory
   while (status != STATUS_DONE);

   for (i=0, error=0; i<NUM_OF_ELEM; i++){
      if (in_data[i] != (~out_data[i])){
         printf("ERROR: wrong output at index %d\n", i);
         error=1; break;
      }
   }
   if(error){ printf(")PPE: program was completed with error\n");
   }else{     printf(")PPE: program was completed successfully\n"); }

   return 0;
}
```

Multibuffering

Data buffering can be extended to use more than two buffers if there is no complete overlapping between computation and data transfer, causing the software to significantly wait to the completion of the DMA transfers. Extending the number of buffer will obviously extend the amount of memory needed to store those buffer, so the programmer should guarantees that there is enough space is available in the LS for doing so.

Building on similar concepts to double buffering, the multibuffering uses multiple buffers in a circular queue. Example 4-31 show a pseudo code of a multibuffering scheme:

Example 4-31 Multibuffering buffering scheme

```
1. Allocate multiple LS buffers, B0..Bn.
2. Initiate transfers for buffers B0..Bn. For each buffer Bi, apply tag
   group identifier i to transfers involving that buffer.
3. Beginning with B0 and moving through each of the buffers in round
   robin fashion:
   - Set tag group mask to include only tag i, and request conditional
     tag status update.
   - Compute on Bi.
   - Initiate the next transfer on Bi.
```

This algorithm waits for and processes each B_i in round-robin order, regardless of when the transfers complete with respect to one another. In this regard, the algorithm uses a strongly ordered transfer model. Strongly ordered transfers are useful when the data must be processed in a known order as happens in many streaming model applications.

4.3.8 Improving page hit ratio using huge pages

In this chapter we discuss how the programer may use huge pages in order to enhance the data access and performance of a given application. The chapter contain the commands required for configuring huge pages on a system and also short code example on how using the huge pages within a program.

Another code example which also shows how to use huge pages with NUMA API is presented in Example 4-34 on page 170.

The huge page support on the SDK aim to address the issue of reducing the latency of address translation mechanism on the SPEs. This mechanism is implemented using 256 entry translation lookaside buffers (TLBs) which reside on the on the SPEs and store the information regarding address translation. The operating system on the PPE is responsible to manage those buffers.

The following process runs whenever the SPE try to access some data on the main storage:

1. SPU code initiate MFC DMA command for accessing data on main storage and provide the effective address of the data in main storage.
2. SPE SMM[5] checks if the effective address falls within one of the TLB entries:
   - If exists (page hit): use this entry to translate to real address and exit the translation process.
   - If not (page miss): continue to step 3.
3. SPU halts program execution and generates an external interrupt to the PPE.
4. Operating systems on the PPE allocate the page and writes the require information into the TLB of this particular SPE using memory access to the problem state of this SPE.
5. PPE signal the SPE that translation is complete.
6. MFC starts transferring the data and SPU code continue running.

This mechanism causes the SPU program to halt until the translation process is complete which may take significant amount of time. This may be not efficient in case the process repeats itself many times during the program execution.

However, the process is taken place only for the first time a page is accessed, unless and the translation information in the TLB is replaced by information of other pages which are later accessed.

Hence, using very large pages may significantly improve the performance in cases where the application operates on large data sets. In those cases, using very large pages can significantly reduce the number of time this process occurs (only once for each page).

The SDK supports the huge TLB file system, which allows the programmer to reserve 16 MB huge pages of pinned, contiguous memory. For example, if 50 pages are configured, it provides 600 MB of pinned contiguous memory. In the worst case where each SPE accesses the entire memory range, a TLB miss will occur only once for each of the 50 pages since the TLB will have enough room to store all those pages. For comparison, the size of ordinary pages on the operating system that runs on Cell BE is either 4 KB or 64 KB.

[5] SMM (synergistic memory management) unit is responsible for address translation in the SPE.

Note: It is recommended to use huge pages in cases where the application uses large data sets. This can significantly improve the performance in many cases and usually it requires only minor changes in the software.
The number of pages that the programmer should allocated depends on the specific application and that way data is partitioned on this application.

Few issues related to the huge pages mechanism:

- Configuring the huge pages is not related to a specific program but to the operating system.
- Any program that runs on the system may use the huge pages by explicitly mapping data buffers on the corresponding huge files.
- The area that is used by the huge pages is pinned in the system memory, so it equivalently reduces the amount of system memory bytes available for other purposes (i.e. any memory allocation that doesn't explicitly use huge pages).

In order to configure huge pages, a 'root' user needs to execute a set of commands. Those commands may be executed at any time and create memory mapped files at /huge/ path that will store the huge pages content.

The first part of Example 4-32 shows the commands required to set 20 huge pages which provided 320 MB of memory. The last four commands in this part (`groupadd`, `usermod`, `chgrp`, `chmod` commands) provide permission to the user the huge pages files. Without those executing those commands, only the root user will later be able to access those files and use the huge pages.
The second part of this example demonstrates how to verify if the huge pages were successfully allocated.

However, in many cases the programmer may have difficulties configuring adequate huge pages usually because the memory is fragmented. Rebooting the system is required in those cases.

The alternative and recommended way is to add the first part of the command sequence shown Example 4-32 to the startup initialization script, such as `/etc/rc.d/rc.sysinit`, so that the huge TLB file system is configured during the system boot.

Some programmer may use huge pages while also using *NUMA* (Non-Uniform Memory Architecture) to restrict memory allocation to a specific node (as described in 4.3.9, "Improving memory access using NUMA" on page 168). The number of available huge pages for the specific node in this case is half of what is reported in `/proc/meminfo`. This is because on Cell based blade systems the huge pages are equally distributed across both memory nodes.

Example 4-32 Configuring huge pages

```
> Part 1: Configuring huge pages:

mkdir -p /huge
echo 20 > /proc/sys/vm/nr_hugepages
mount -t hugetlbfs nodev /huge
groupadd hugetlb
usermod -a -G hugetlb <user>
chgrp -R hugetlb /huge
chmod -R g+w /huge

> Part 2: Verify that huge pages are successfully configured:

cat /proc/meminfo

The following output should be printed:
MemTotal:   1010168 kB
MemFree:     155276 kB
. . .
HugePages_Total: 20
HugePages_Free:  20
Hugepagesize: 16384 kB
```

Once the huge pages are configured, any application may allocate data on the corresponding memory mapped file. This can be done by explicitly invoking mmap of a /huge file of the specified size.

Example 4-33 shows a code example which opens a huge page file using the `open` function and allocates 32 MB of private huge paged memory using `mmap` function (32 MB indicated by the 0x2000000 parameter of `mmap` function).

Source code: The code of Example 4-33 is included in the additional material that is provided with this book. See “Huge pages” on page 615 for more information.

Note: The `mmap` function 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 process is terminated and the message “killed” is emitted. The programmer must therefore ensure that the number of huge pages requested does not exceed the number available.

Another useful standard Linux library that handles the access to huge pages from the program code is *libhugetlbfs*[6]. This library provides an API for dynamically managing the huge pages in a way which is very similar to working with ordinary pages.

Example 4-33 PPU code for using huge pages

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

int main(int argc, char *argv[])
{
   void *ptr;
   int fmem;
   char *mem_file = "/huge/myfile.bin";

   // open a huge pages file
   if ( (fmem = open(mem_file, O_CREAT|O_RDWR, 0755))==-1){
      perror("ERROR: Can't open huge pages file"); exit(1);
   }
   remove(mem_file);

   // map 32MB (0x2000000) huge pages file to main sotrage
   // get pointer to effective address
   ptr = mmap(0, 0x2000000, PROT_READ|PROT_WRITE, MAP_PRIVATE,fmem,0);
   if(ptr==NULL){
      perror("ERROR: Can't map huge pages"); exit(1);
   }

   printf("Map huge pages to 0x%llx\n",(unsigned long long int)ptr);

   // now we can use ‘ptr’ effective addr. pointer to store our data
   // for example forward to the SPEs to use it

   return (0);
}
```

[6] See http://sourceforge.net/projects/libhugetlbfs

4.3.9 Improving memory access using NUMA

The first two cell based blade system, the BladeCenter QS20 and BladeCenter QS21 are both Non-Uniform Memory Architecture (NUMA) systems, which consist of two Cell BE processors, each with its own system memory. The two processors are interconnected through a FlexIO interface using the fully coherent BIF protocol.

Since coherent access is guaranteed, from software point of view a program that runs on either of the two processors can coherently access either of the two attached memories. The programer may therefore can choose to ignore the NUMA architecture having the data stored in two different memories and program as if the program runs on an SMP system. However, in many cases doing so will results in performance which are far from optimal.

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, the programmer can specify CPU and memory binding to either reduce FlexIO traffic or exploit the aggregated bandwidth of the memory available on both nodes.

Note: Applications that are memory bandwidth-limited should consider allocating memory on both nodes and exploit the aggregated memory bandwidth. The optimal case is in which the data and tasks execution can be perfectly divided between nodes (processor on node 0 primarily access memory on this node, and the same for node 1)

Linux provide NUMA API[7] to address this issue and to enable allocating memory on specific node. For doing so, the programmer may use the NUMA API in the following way:

- Use NUMA API to allocate memory on the same processor on the current thread runs.
- Use NUMA API to guarantee that this thread keep running on a specific processor (node affinity).

The following chapters discuss the two separate interfaces that Linux provides to control and monitor the NUMA policy and also some program consideration regarding NUMA:

[7] A NUMA API for LINUX, Technical Linux Whitepaper

- Chapter , “NUMA program level API (libnuma library)” on page 169 discuss the ‘libnuma’ shared library which provides an application level API.
- Chapter , “NUMA command utility (numactl)” on page 173 discuss the ‘numactl’ command utility.
- Chapter , “NUMA policy considerations” on page 173 present the main consideration the programer should take when deciding if and how to use NUMA.

Once NUMA was configured and the application completed its execution, the programer can use NUMA’s `numastat` command to retrieve some statistics regarding the status of NUMA allocation and data access on each of the nodes. This information can be used to estimate the effectiveness of the current NUMA configuration.

NUMA program level API (libnuma library)

Linux provides s shared library name ‘*libnuma*’ that implements set of API for controlling and tracing the NUMA policy. The library functional calls can be called from any application level program which allow programing flexibility and also have the advantage of creating self contained program that manage the NUMA policy unique to them.

In order to use the NUMA API the programer should do the following:

- Include the `numa.h` header file in the source code.
- Add the `-lnuma` flag to the compilation command in order to link the library to the application.

Additional information is available in the man pages of this library that can be retrieved using the `man numa` command.

A suggested method for using NUMA is described through Example 4-34 which shows a corresponding PPU code. The example is inspired by the SDK’s matrix multiply demo which is in `/opt/cell/sdk/src//demos/matrix_mul` directory.

Please note that NUMA terminology uses the term ‘node’ that in the example below refer to as one Cell BE processor (having two of those on a Cell BE blade).

The main principles behind the NUMA example that we present are:

1. Use NUMA API to allocate two memory continuos memory regions - one on each of the nodes’ memories.
2. The allocation is done using huge pages to minimize SPE’s page miss. Notice that the huge pages are equally distributed across both memory nodes on a Cell BE based blade systems. Huge pages are further discussed in Chapter 4.3.8, “Improving page hit ratio using huge pages” on page 163

3. Duplicate the input data structures (matrix and vector in this case) by initiating two different copies - one on each of the regions that were allocated in step 1.
4. Use NUMA to split the SPE threads so each half of the threads is initiate and runs on a separate node.
5. The threads that runs on node number 0 are assigned to work on the memory region that was allocated on this node, and node 1's threads are assigned to work on node 1's memory region.

In this example there was a need to duplicate the input data since the entire input matrix is needed for any of the threads. While this is not the optimal solution. in many other applications there is no need to do so and the input data can simply be divided between the two nodes (e.g. when adding two matrixes one half of those matrixes can be located on one node's memory and second half on the other node's memory).

Two more comments regarding combining NUMA API with other SDK's functions:

- SDK's `spe_cpu_info_get` function can be use to retrieve the number of physical Cell BE processors and in specific number of physical SPEs that are currently available. Using this function is demonstrated in Example 4-34.
- SDK's SPEs affinity mechanism may be used in conjunction with NUMA API in order to add affinity between SPEs to the SPEs to near memory binding that is provided by NUMA. The SPE affinity relevant mainly when there is significant SPE to SPE communication and is discussed in Chapter 4.1.3, "Creating SPEs affinity using gang" on page 93.

Example 4-34 Code example for using NUMA

```
#include <numa.h>

char *mem_file = "/huge/matrix_mul.bin";
char *mem_addr0=NULL, *mem_addr1=NULL;

#define MAX_SPUS16
#define HUGE_PAGE_SIZE(size_t)(16*1024*1024)


// main=================================================================
int main(int argc, char *argv[])
{
   int i, nodes, phys_spus, spus;
   unsigned int offset0, offset1;
   nodemask_t mask0, mask1;

   // calculate the number of SPU for the program
```

```
spus = <number of required SPUs>;

phys_spus = spe_cpu_info_get(SPE_COUNT_PHYSICAL_SPES, -1);

if (spus > phys_spus) spus = phys_spus;

// check NUMA availability and intiate NUMA data structures
if (numa_available() >= 0) {
   nodes = numa_max_node() + 1;

   if (nodes > 1){
      // Set NUMA masks; mask0 for node # 0, mask1 for node # 1
      nodemask_zero(&mask0);
      nodemask_set(&mask0, 0);
      nodemask_zero(&mask1);
      nodemask_set(&mask1, 1);
   }else{
      printf("WARNING: Can't use NUMA - insufficient # of nodes\n");
   }
}else{
   printf("WARNING: Can't use NUMA - numa is not available.\n");
}

// calculate offset on the huge pages for input buffers
offset0 = <offset for node 0's buffer>
offset1 = <offset for node 1's buffer>

// allocate inout buffers - mem_addr0 on node 0, mem_addr1 on node 1
mem_addr0 = allocate_buffer(offset0, &mask0);
mem_addr1 = allocate_buffer(offset1, &mask1);

//  Initialize the data in mem_addr0 and mem_addr1

// Create each of the SPU threads
for (i=0; i<spus; i++){

   if (i < spus/2) {
      // lower half of the SPE threads uses input buffer of ndoe 0
      threads[i].input_buffer = mem_addr0;

      // binds the current thread and its children to node 0
      // they will only run on the CPUs of node 0 and only be able
      // to allocate memory from this node
      numa_bind(&mask0);
```

```
      }else{
         // similarly - second half use buffer of node 1
         threads[i].input_buffer = mem_addr1;
         numa_bind(&mask1);
      }

      // create SPE thread - will be binded to run only on
      // NUMA's specified node
      spe_context_create(...);
      spe_program_load(...);
      pthread_create(...);
   }

   for (i=0; i<spus; i++) {
      pthread_join(...); spe_context_destroy(...);
   }
}

// alocate_buffer=======================================================
// allocate a cacheline aligned memory buffer from huge pages or the
char * allocate_buffer(size_t size, nodemask_t *mask)
{
   char *addr;
   int fmem = -1;
   size_t huge_size;

   // sets memory allocation mask. The thread will only allocate memory
   // from the nodes set in 'mask'.
   if (mask) {
      numa_set_membind(mask);
   }

   // map huge pages to memory
   if ((fmem=open (mem_file, O_CREAT|O_RDWR, 0755))==-1) {
      printf("WARNING: unable to open file (errno=%d).\n", errno);
      exit(1);
   }
   remove(mem_file);
   huge_size = (size + HUGE_PAGE_SIZE-1) & ~(HUGE_PAGE_SIZE-1);

   addr=(char*)mmap(0, huge_size, PROT_READ|PROT_WRITE,
                    MAP_PRIVATE,fmem,0);

   if (addr==MAP_FAILED) {
      printf("ERROR: unable to mmap file (errno=%d).\n", errno);
```

```
      close (fmem); exit(1);
   }

   // Perform a memset to ensure the memory binding.
   if (mask) {
      (void*)memset(addr, 0, size);
   }
   return addr;
}
```

NUMA command utility (numactl)

Linux provide a command utility names '*numactl*' that enable control and trace on the NUMA policy. The programer may combine the 'numctl' commands in a script that execute the appropriate those commands and later runs the application.

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 ./matrix_mul
```

A shorter version command that perform the same action is:

```
numactl -c 0 -m 0 ./matrix_mul
```

To read the man pages of this command run the `man numactl` command.

One of the advantages of using this method is that there is no need to recompile the program to run with different setting of NUMA configuration. On the other hand, using the command utility enables less flexibility to the programer compare to calling the API of 'libnuma' library from the program itself.

Controlling NUMA policy using the command utility is usually sufficient in cases where all SPU threads can run on a single Cell BE processor. If more then one processor is needed (usually because more then 8 threads are needed) and the application required dynamic allocation of data, it is usually hard to use only the command utility. Using 'libnuma' library API from the program itself is more appropriate and allow greater flexibility in this case.

NUMA policy considerations

Choosing an optimal NUMA policy depends upon the application's data access patterns and communication methods. We suggest the following guidelines when the programmer need to decide if using NUMA commands or API is needed and which NUMA policy should be implemented:

- 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.
- The programmer should choose a NUMA policy compatible with typical system usage patterns. For example, if the multiple applications are expected to run simultaneously, the programmer should 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 not to constrain the Linux scheduler with any specific bindings.
- If the access patterns are not predictable and SPE are allocated on both nodes, then using a interleaved memory policy will improve overall memory throughput.
- In Cell BE system node 0 usually have better memory access performance so it should be preferred over node 1 if possible.
- The programmer should consider the operating system services when choosing 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. In those specific cases, it may be advisable to bind the application to node 1.
- The programmer should 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 when avoiding over-committing resources.

4.4 Inter-processor communication

The Cell BE contains several mechanisms that enable communication between the PPE and SPEs and between the SPEs to themselves. These mechanisms are mainly implemented by the MFCs (one instance of MFC exists in every of the eight SPEs). The code that runs on an SPU may interact with the MFC of the associated SPE using the channels interface while PPU code or code that runs on the other SPUs may interact with this MFC using the MMIO interface.

The following chapters discuss four of the primary communication mechanisms between the PPE and SPEs are:

- 4.4.1, "Mailboxes" on page 176 discuss the mailbox mechanism which allows to send 32-bits messages to and from the SPE. Should be used mainly to control communication between an SPE and the PPE or between SPEs to themselves. other devices. Mailboxes hold

- 4.4.2, “Signal notification” on page 187 discuss the signal notifications (signaling) mechanism which allows to send 32-bits messages to an SPE. Used for control communication from the PPE or other SPEs to another SPE. Can be configured for one-sender-to-one-receiver signalling or many-senders-to-one-receiver signalling.
- 4.4.3, “SPE events” on page 199 discuss how event may be used to create asynchronous communication between the processors.
- 4.4.4, “Using atomic unit and the atomic cache” on page 206 discuss how to implement a fast shared data structure for inter-processor communication using the Atomic Unit and the Atomic Cache hardware mechanism.

The MFC interfaces and the different programing methods in which a programs may interact with the MFC are described in Chapter 4.2, “Storage domains, channels and MMIO interfaces” on page 95. In our chapter we use only the MFC functions method in order interact with the MFC.

Another mechanism that can be used to apply inter-processor communication is DMA data transfers. For example, and SPE may compute an output data and use DMA to transfer this data to the main memory. Later the SPE can notify the PPE that the data is ready using additional DMA to a notification variable in the memory which the PPE polls. The available data transfer mechanisms and how the programmer may initiate them are described in Chapter 4.3, “Data transfer” on page 109.

Both mailboxes and signals are mechanism that may be use for program control and sending short messages between the different processors. While those mechanisms have a lot in common, there are some differences between the two mechanisms. As general, mailbox implements a queue for sending separate 32-bits messages, while signaling is more similar to interrupts which are may be accumulated when being written and are reset when being read. Table 4-6 compares between the two mechanisms.

Table 4-6 Comparison between mailboxes and signals

Attribute	Mailboxes	Signals
Direction	One inbound, two outbound	Two inbound (toward the SPE).
Interrupts	One mailbox can interrupt PPE. Two mailbox-available event interrupts.	Two signal-notification event interrupts.
Message accumulation	No	Yes, using logical OR mode (many-to-one). Other alternative is overwrite mode (one-to-one),

Attribute	Mailboxes	Signals
Unique SPU commands	No; programs use channel reads and writes.	Yes, ‘sndsig’, ‘sndsigf’, and ‘sndsigb’ enables and SPU to send signals to another SPE.
Destructive read	Reading a mailbox consumes an entry.	Reading a channel resets all 32 bits to ‘0’.
Channel count	Indicates number of available entries.	Indicates waiting signal.
Number	Three mailboxes: 4-deep incoming, 1-deep outgoing, 1-deep outgoing with interrupt.	Two signal registers.

4.4.1 Mailboxes

This chapter discuss the mailbox mechanism which is an easy to use mechanism that enables to send 32-bits messages between the different processors on the chip (PPE and SPEs).

The following chapters discuss the following topics:

- “Mailbox overview” on page 176 provides an overview on this mechanism and its hardware implementation.
- “Programing interface for accessing mailboxes” on page 179 describes the main software interfaces for a SPU or PPU program to use the mailbox mechanism.
- “Blocking versus non-blocking access to the mailboxes” on page 180 discusses how the programer may implement either blocking or nonblocking access to the mailbox on either a SPU or PPU program.
- “Practical scenarios and code examples for using mailboxes” on page 181 provides some practical scenarios and techniques for using the mailboxes and also emphasize some code examples.

Please notice that monitoring the mailbox status may be done asynchronously using events that are generated whenever a new mailbox was written or read by external source (e.g. PPE or other SPE). While this chapter do not discuss the mailbox events, Chapter 4.4.3, “SPE events” on page 199 discuss the events mechanism in general.

Mailbox overview

Mailboxes is an easy to use mechanism that enables the software to exchange 32-bit messages between the local SPU and the PPE or local SPU and other

SPEs[8]. The term *local SPU* stands for the SPU of the same SPE where the mailbox is located. The mailboxes are access from the local SPU using the channel interface and from the PPE or other SPEs using the MMIO interface.

Mailbox mechanism is similar in some sense to the signaling mechanism. Table 4-6 on page 175 displays a comparison between the two mechanism.

Note: Local SPU access to the mailbox are internal to that SPE and have very small latency (<=6 cycles for non blocking access). On the other hand, PPE or other SPEs access to the mailbox are done through the local memory EIB bus. The result is larger latency and also overloading the bus bandwidth (especially when polling to wait for mailbox to become available).

The MFC of each SPE contains three mailboxes divided into two categories:

1. *Outbound mailboxes*: Two mailboxes that are used to send messages from the local SPE to the PPE or other SPEs:
   a. SPU Write Outbound mailbox (`SPU_WrOutMbox`)
   b. SPU Write Outbound Interrupt mailbox (`SPU_WrOutIntrMbox`)
2. *Inbound mailbox*: One mailbox that is used to send messages to the local SPE from the PPE or other SPEs:
   c. SPU Read Inbound mailbox (`SPU_RdInMbox`)

The main attributes of those mailboxes and the differences between outbound mailboxes and inbound mailbox are summarized in Table 4-7. This table also describes the differences between accessing the mailboxes from the SPU programs and accessing them from the PPU other SPEs programs.

[8] Mailboxes can also be used as a communications mechanism between SPEs. This is accomplished by an SPE DMAing data into the other SPE's mailbox using the effective addressed problem state mapping.

Table 4-7 Attributes of inbound and outbound mailboxes

Attribute	Inbound mailboxes	Outbound mailboxes
Direction	Messages from the PPE or another SPEs to the local SPE.	Messages from the local SPE to the PPE or another SPEs.
Read/Write	► Local SPE reads. ► PPE[b] writes.	► Local SPE write. ► PPE[b] reads.
# mailboxes	1	2
# entries	4	1
Counter[a]	Counts number of valid entries: ► Decremented when SPU program reads from mailbox. ► Incremented when PPU program[b] writes to mailbox.	Counts number of empty entries: ► Decremented when SPU program writes to mailbox . ► Incremented when PPU program[b] reads from mailbox.
Buffer	A first-in-first-out (FIFO) queue - SPU program reads the oldest data first.	A first-in-first-out (FIFO) queue - SPU program reads the oldest data first.
Overrun	PPU program[b] writing new data when buffer is full overrun the last entry in this fifo.	SPU program writing new data when buffer is full blocks till there is available space in the buffer (e.g PPE[b] reads from the mailbox).
Blocking	► SPU program blocks when trying to read an empty buffer and will continues only when there is a valid entry (e.g PPE[b] write to the mailbox). ► PPU program[b] never block. Writing to mailbox when full overide the last entry and the PPU immediately continues.	► SPU program blocks when trying to write to the buffer when it is full and will continues only when there is an empty entry (e.g PPE[b] reads from the mailbox). ► PPU program[b] never block. Reading from mailbox when it is empty returns a in-valid data and the PPU program immediately continues.

a. This per-mailbox counter may be read by local SPU program using a separate channel or by the PPU or other SPUs program using separate MMIO register.
b. Or other SPE that access the mailbox of the local SPE.

Programing interface for accessing mailboxes

The simplest way to access the mailboxes is through the MFC functions that are part of SDK package library.

- Local SPU program can access the mailboxes using `spu_*_mbox` functions in `spu_mfcio.h` header file.
- PPU program can access the mailboxes using `spe_*_mbox*` functions in `libspe2.h` header file.
- Other SPUs program may access the mailboxes using DMA functions of `spu_mfcio.h` header file which enables to read or write the mailboxes that are may be mapped to main storage as part of the problem state of local SPU.

The `spu_mfcio.h` functions are described in *SPU Mailboxes* chapter in C/C++ Language Extensions for Cell BE Architecture document. The `libspe2.h` functions are described in *SPE mailbox functions* chapter in SPE Runtime Management library document.

Table 4-7 summarizes the simple functions in those files for accessing the mailboxes from a local SPU program or from a PPU program.

In addition to the value of the mailboxes messages, the counter that is mentioned in Table 4-7 can also be read by software using the SPU's `*_stat_*` functions of PPU's `*_status` functions.

Table 4-8 MFC functions for accessing the mailboxes

Name	SPU code functions (channel interface)	Blocking	PPU code functions (MMIO interface)	Blocking
SPU write outbound mailbox	spu_write_out_mbox	Yes	spe_out_mbox_read	No
	spu_stat_out_mbox	No	spe_out_mbox_status	No
SPU write outbound int. mailbox	spu_write_out_intr_mbox	Yes	spe_out_intr_mbox_read	User [a]
	spu_stat_out_intr_mbox	No	spe_out_intr_mbox_status	No
SPU read inbound mailbox	spu_read_in_mbox	Yes	spe_in_mbox_write	User
	spu_stat_in_mbox	No	spe_in_mbox_status	No

a. A user parameter to this function chooses whether the function is blocking or not blocking.

In order to access a mailbox of the local SPU from other SPU several steps should be taken:

1. PPU code map the SPU's controls area to main storage using `libspe2.h` file's `spe_ps_area_get` function with `SPE_CONTROL_AREA` flag set.
2. PPE forward the SPU's control area base address to another SPU.
3. The other SPU uses ordinary DMA transfers to access the mailbox. Effective address should be control area base plus offset to specific mailbox register.

Blocking versus non-blocking access to the mailboxes

Using the SDK library functions for accessing the mailboxes (which are described in chapter "Programing interface for accessing mailboxes") enables the programmer to implement either blocking or non blocking mechanisms.

As for the SPU, the instructions to access the mailbox are blocking by nature and are stalled when the mailbox is non available (empty for read or full for write). The SDK simply implement those instructions.

For the PPU, the instructions to access the mailbox are nonblocking by nature. SDK functions provides software abstraction of blocking behavior functions for some of the mailboxes (which is implemented by polling the mailbox counter till there is available entries).

In case the programer wants to explicitly read the mailbox status (the counter that is mentioned in Table 4-7) is can be done by calling `*_stat_*` functions for SPU program and `*_status` functions for PPU program.

Note: Nonblocking approach are slightly more complicated to program but enables the program to perform other tasks in case the fifo is empty instead being stalled waiting for a valid entry.

Different programming approaches for performing either blocking or nonblocking access to the mailbox on a PPU or SPU program are summarized on Table 4-9:

Table 4-9 Blocking versus nonblocking access to mailboxes - programming approaches

Proc.	Mailbox	Blocking	Nonblocking
SPU	In	Simply read the mailbox using `spu_read_in_mbox` function.	Before reading the mailbox poll the counter using `spu_stat_in_mbox` function till fifo is not empty.
	Out	Simply write to mailbox using `spu_write_out_mbox` function.	Before writing to mailbox poll the counter using `spu_stat_out_mbox` function function till fifo is not full.
	OutIntr	Write to mailbox using `spu_write_out_intr_mbox` function.	Before writing to mailbox poll the counter using `spu_stat_out_intr_mbox` function function till fifo is not full.
PPU	In	Call `spe_in_mbox_write` and set 'behavior' parameter to blocking.	Call `spe_in_mbox_write` and set 'behavior' parameter to nonblocking.
	Out	Not implemented.[a]	Call `spe_out_mbox_read` function.
	OutIntr	Call `spe_out_intr_mbox_read` and set 'behavior' parameter to blocking.	Call `spe_out_intr_mbox_read` and set 'behavior' parameter to nonblocking.

a. Programmer should check the function return value to see that the data that was read it valid.

Practical scenarios and code examples for using mailboxes

When using the mailbox it is important to be aware of the following attributes of the mailboxe's access:

- Local SPU access is internal to that SPE and has very small latency (<=6 cycles for non blocking access).
- Local SPU access to not available mailbox (empty for read or full for write) is blocking. To avoid blocking, the program may first read the counter as explained below.
- PPE or other SPEs access is done through the local memory EIB bus, so they have larger latency and also overload the bus bandwidth (especially when polling the mailbox counter waiting for mailbox to become available).

- PPU or local SPU access to the mailboxes may be either blocking or nonblocking using SDK library functions, as discussed in Chapter , “Blocking versus non-blocking access to the mailboxes” on page 180.

The following sections describe different scenarios for using the mailbox mechanism and provide a mailbox code example.

Using mailbox to notify PPE on data transfer completion

Mailbox may be useful when a SPE need to notify the PPE about completion of transferring data that was previously computed by the SPE to the main memory.

Such mechanism may be implemented using the following steps:

1. SPU code places computational results in main storage via DMA
2. SPU code waits for the DMA transfer to complete.
3. SPU code writes to an outbound mailbox to notify the PPE that its computation is complete. This ensures only that the SPE’s LS buffers are available for reuse but does not guarantee that data has been coherently written to main storage.
4. PPU code reads the SPE’s outbound mailbox and is notified that computation is complete.
5. PPU code issue an ‘lwsync’ instruction to be sure that results are coherently written to memory.
6. PPU code reads the results from memory.

Please notice that in order to implement step 4 the PPU may need to poll the mailbox status to see if there is a valid data in this mailbox. Doing so is not very efficient since it cause overhead on the bus bandwidth which may effect other data transfer on this bus such as SPEs reading from main memory.

Comment: Alternatively, an SPU can notify the PPU that it has completed computation by using a fenced DMA to write notification to some address in the main storage. The PPU may poll this area on the memory which may be local to the PPE in case the data is in the L2 cache so it minimizes the overhead on the EIB bus and memory subsystem. Example 4-19 on page 128 and the following Example 4-20 and Example 4-21 provide code for such mechanism.

Using mailbox to exchange parameters between PPE and SPE

Mailbox may be used for any short-data transfer purpose, such as sending of storage effective addresses from PPE to SPE.

Because the operating system runs on the PPE, only the PPE is originally aware of the effective addresses of different variables in the program. For example when the PPE dynamically allocate data buffers or when it maps the SPE's local stores or problem state to an effective address on the main storage. The inbound mailboxes may be use to transfer those addresses to the SPE. Example 4-35 on page 183 and the following Example 4-36 provides code for such mechanism.

Similarly, any type of function or command parameters may be forwarded from the PPE to the SPE using this mechanism.

On the other direction, an SPE may use the outbound mailbox to notify the PPE about a local store offset of some buffer which is located on the local store and may be later accessed by either the PPE or another SPE. Chapter "Code example for using mailboxes" provides code example for such mechanism.

Code example for using mailboxes

The code example below covers the following techniques:

- Example 4-35 show the PPU code which access SPEs' mailboxes using either non blocking methods for (most of the methods described in the list above are illustrated) and blocking methods.
  This example also shows how to map the control area of the SPEs to the main storage to enable SPEs to access each other's mailbox.
- Example 4-36 show the SPU code which access the local mailboxes using either non blocking methods for (most of the methods described in the list above are illustrated) and blocking methods. The code also send mailbox to another SPE's mailbox.
- The functions who implement the writing to a mailbox of another SPE using DMA transactions is in Example 4-39 on page 195. The code also contains functions for reading the status of other SPE's mailbox

Source code: The code of Example 4-35, Example 4-36 and Example 4-39 is included in the additional material that is provided with this book. See "Simple mailbox" on page 615 for more information.

Example 4-35 PPU code for accessing SPEs' mailboxes

```
// add the ordinary SDK and C libraries header files...
// take 'spu_data_t' structure and 'spu_pthread' function from
// Example 4-5 on page 90

extern spe_program_handle_t spu;
volatile parm_context ctx[2] __attribute__ ((aligned(16)));
volatile spe_spu_control_area_t* mfc_ctl[2];
```

```
// main===================================================================
int main()
{
   int num, ack;
   uint64_t ea;
   char str[2][8] = {"0 is up","1 is down"};

   for( num=0; num<2; num++){
      // SPE_MAP_PS flag should be set when creating SPE context
      data[num].spe_ctx = spe_context_create(SPE_MAP_PS,NULL);
   }

   // ... Omitted section:
   // load the program to the local stores, and run the SPE threads.

   // (the entire source code for this example is part of the book's
   // additional material)

   // This is also described in 4.1.2, "Task parallelism and managing
SPE threads"

   // STEP 0: map SPEs' MFC problem state to main storage (get EA)
   for( num=0; num<2; num++){
      if ((mfc_ctl[num] = (spe_spu_control_area_t*)spe_ps_area_get(
                          data[num].spe_ctx, SPE_CONTROL_AREA))==NULL){
         perror ("Failed mapping MFC control area");exit (1);
      }
   }
   // STEP 1: send each SPE its number using BLOCKING mailbox write
   for( num=0; num<2; num++){

      // write 1 entry to in_mailbox
      // we don't know if we have availalbe space so use blocking
      spe_in_mbox_write(data[num].spe_ctx,(uint32_t*)&num,1,
                        SPE_MBOX_ALL_BLOCKING);
   }

   // STEP 2: send each SPE the EA of other SPE's MFC area and a string
   //         Use NON-BLOCKING mailbox write after first verifying
   //         availability of space.
   for( num=0; num<2; num++){

      ea = (uint64_t)mfc_ctl[(num==0)?1:0];
```

```
      // loop till we have 4 entries available
      while(spe_in_mbox_status(data[num].spe_ctx)<4){
         // PPE can do other things meanwhile before check status again
      }

      //write 4 entries to in_mbx- we just checked having 4 entries
      spe_in_mbox_write(data[num].spe_ctx, (uint32_t*)&ea,2,
                        SPE_MBOX_ANY_NONBLOCKING);
      spe_in_mbox_write(data[num].spe_ctx, (uint32_t*)&str[num],2,
                        SPE_MBOX_ANY_NONBLOCKING);
   }

   // STEP 3: read acknowledge from SPEs using NON-BLOCKING maibox read
   for( num=0; num<2; num++){
      while(!spe_out_mbox_status(data[num].spe_ctx)){
         // simulate the first second after the universe was created or
         // do other computations before check status again
      };
      spe_out_mbox_read(data[num].spe_ctx, (uint32_t*)&ack, 1);
   }

   // ... Omitted section:
   // waits for SPE threads to complete.

   // (the entire source code for this example is part of the book's
   // additional material)

   return (0);
}
```

Example 4-36 SPU code for accessing local mailboxes and other SPE's mailbox

```
// add the ordinary SDK and C libraries header files...
#include "spu_mfcio_ext.h" // the file described in Example 4-39

uint32_t my_num;

// Macro for waiting to completion of DMA group related to input tag:
#define waitag(t) mfc_write_tag_mask(1<<t); mfc_read_tag_status_all();

int main( )
{
   uint32_t data[2],ret, mbx, ea_mfc_h, ea_mfc_l, tag_id;
   uint64_t ea_mfc;
```

```
   if ((tag_id= mfc_tag_reserve())==MFC_TAG_INVALID){
      printf("SPE: ERROR can't allocate tag ID\n"); return -1;
   }

   // STEP 1: read from PPE my number using BLOCKING mailbox read
   while( spu_stat_in_mbox()<=0 );
   my_num   = spu_read_in_mbox();

   // STEP 2: receive from PPE the EA of other SPE's MFC and string
   //         use BLOCKING mailbox, but to avoid bloking we first read
   //         status to check that we have 4 valid entries
   while( spu_stat_in_mbox()<4 ){
      // SPE can do other things meanwhile before check status again
   }

   ea_mfc_h = spu_read_in_mbox(); // read EA lower bits
   ea_mfc_l = spu_read_in_mbox(); // read EA higher bits

   data[0] = spu_read_in_mbox(); // read 4 bytes of string
   data[1] = spu_read_in_mbox(); // read 4 more bytes of string

   ea_mfc = mfc_hl2ea( ea_mfc_h, ea_mfc_l);

   // STEP 3: send my ID as acknowledge to PPE using BLOCKING mbx write
   spu_write_out_mbox(my_num+1313000); //add dummy constant to pad MSb

   // STEP 4: write message to other SPE's mailbox using BLOCKING write
   mbx = my_num + 1212000; //add dummy constant to pad MSb

   ret =  write_in_mbox( mbx, ea_mfc, tag_id);
   if (ret!=1){ printf("SPE: fail sending to other SPE\n");return -1;}

   // STEP 5: read mailbox written by other SPE
   data[0] = spu_read_in_mbox();

   return 0;
}
```

4.4.2 Signal notification

This chapter discuss the signal notification mechanism which is an easy to use mechanism that enables a PPU program or SPU program to signal a program running on another SPU.

The following chapters discuss the following topics:

- "Signal notification overview" on page 187 provides an overview on this mechanism and its hardware implementation.
- "Programing interface for accessing signaling" on page 189 describes the main software interfaces for a SPU or PPU program to use the signal notification mechanism.
- "Practical scenarios and code examples for using signaling" on page 189 provide some practical scenarios and techniques for using the signal notification and also emphasize some code examples.

  This chapter also contains printing macros for tracing inter-processors communication, such as sending mailboxes and signaling between PPE and SPE and SPE and SPE. Those macro can be useful when tracing a flow of a given parallel program.

Please notice that monitoring the signals status may be done asynchronously using events that are generated whenever a new signal is set by external source (e.g. PPE or other SPE). While this chapter do not discuss the signal events, Chapter 4.4.3, "SPE events" on page 199 discuss the events mechanism in general.

Signal notification overview

Signal notification is an easy to use mechanism that enables a PPU program to signal an SPE using 32-bit registers. It also enables a SPU program so signal a program running on another SPU using the other SPU's signal mechanism. The term *local SPU* is used in this chapter to define the SPU of the same SPE where the signal register is located.

Each SPE contain two identical signal notification registers named Signal Notification 1 (`SPU_RdSigNotify1`) and Signal Notification 2 (`SPU_RdSigNotify2`).

Unlike the mailboxes. the signal notification has only one direction and enables to send information toward the SPU that resides in the same SPE as the signal registers (and not vice versus). Programs may access the signals using the following interfaces:

- Local SPU program reads the signal notification using the channel interface.
- PPU program signals a SPE by writing to it the MMIO interface.

- SPU program signals another SPE using special signaling commands ('sndsig', 'sndsigf', and 'sndsigb'). Those commands are actually implemented using DMA 'put' commands and optionally contain ordering information ('f' and 'b' suffix in the commands above indicate 'fence' and 'barrier' respectively).

When the local SPU program reads a signal notification, the value of the signals register is reset to '0'. Reading the signal's MMIO (or problem state) register by the PPU or other SPUs does not reset their value.

Regarding writing of PPU or other SPUs to the signals registers, there are two different modes that can be configured:

- *OR mode* (*many-to-one*): MFC accumulates several write to the signal-notification register by combining all the values written to this register using a logical OR operation. The register is reset when the SPU reads it.
- *Overwrite mode* (*one-to-one*): writing a value to a signal-notification register overwrites the value in this register. This mode is actually very similar to using inbound mailbox and have similar performance.

Configuring signaling mode can be done by the PPU when it creates the coresponding SPE context.

Note: OR mode allows the signal producers to send their signals at any time and independently of other signal producers (no synchronization is needed). When SPU program reads the signal notification register, it becomes aware of all the signals that have been sent since the most recent read of the register.

Similar to the mailboxes, the signal notification register maintain a counter, which had different behavior in the signaling case:

- The counter indicates only if there are pending signals (at least one bit set) and not how many writes to the this register have taken place.
- Reading a value of '1' indicates that there is at least one event pending and value of '0' indicates that no signals are pending.
- May be read by program running on either the local SPU, PPU or other SPUs.

Regarding the blocking behavior, the accessing the signal notification has the following characters:

- PPU code writing to the signal register is nonblocking. It may override its previous value or not depends on the configured mode (OR or overwrite mode as explained above).
- SPU code writing to signal register of another SPU behaves similar to DMA 'put' command and blocks only if the MFC fifo is full.

- Local SPU reading the signal register is blocking when no events are pending. Reading is completed immediately in case there is at least one pending event.

The similarities and differences between the signal notification and mailbox mechanism are summarized in Table 4-6 on page 175.

Programing interface for accessing signaling

The simplest way to access the signal notification mechanism is through the MFC functions that are part of SDK package library.

- Local SPU program can read the local SPE's signals using `spu_read_signal*` and `spu_stat_signal*` functions in `spu_mfcio.h` header file for reading the signals register and the status (counter) respectively.
- Other SPUs' program can signal other SPU using the functions `mfc_sndsig*` (* is 'b', 'f' or blank) in `spu_mfcio.h` header file, which enables to signal the other SPU by doing write operation on its memory mapped problem state.
- PPU program can access the signals using two main functions in `libspe2.h` header file. The function `spe_signal_write` to send a signal to an SPU and optionally setting SPE_CFG_SIGNOTIFY1_OR flag when creating the SPE context (`spe_context_create` function) to enable OR mode.

The `spu_mfcio.h` functions are described in *SPU Signal Notification* chapter in C/C++ Language Extensions for Cell BE Architecture document. The `libspe2.h` functions are described in *SPE SPU signal notification functions* chapter in SPE Runtime Management library document.

In order to signal local SPU from other SPU several steps should be taken:

1. PPU code map the SPU's signaling area to main storage using `libspe2.h` file's `spe_ps_area_get` function with `SPE_SIG_NOTIFY_x_AREA` flag set.
2. PPE forward the SPU's signaling area base address to another SPU.
3. Other SPU uses `spu_mfcio.h` file's `mfc_sndsig` function to access the signals. Effective address should be signaling area base plus offset to specific signal register.

The programer may take either blocking or nonblocking approach when reading the signals from the local SPU. The programming methods to do so are similar to those discuss for the mailboxes in chapter Chapter , "Blocking versus non-blocking access to the mailboxes" on page 180. However, setting signals from the PPU program or other SPUs is always nonblocking.

Practical scenarios and code examples for using signaling

Similar to the mailboxes mechanism, local SPU access to the signals notification is internal to that SPE and have very small latency (<=6 cycles for non blocking

access), and PPE or other SPEs MMIO access to the mailbox are done through the local memory EIB bus which has larger latency. However, since it is not common (and usually not useful) to poll the signal notification from the MMIO side, overloading the bus bandwidth is usually not a significant issue.

Regarding blocking behavior, local SPU reading the signals register when no bits are set is blocking. To avoid blocking, the program may first read the counter as explained below. PPE or other SPEs signaling some SPU is always non-blocking.

When using the OR mode the PPE or other SPEs usually don't need to poll the signals counter since events are accumulated. Otherwise (overwrite mode) the signals have similar behavior to inbound mailboxes.

The following two chapters describes two different scenarios for using the signals notification mechanism. Next, the third chapter provide a signals code example.

Since in overwrite mode the signals behave similar to mailboxes, the scenarios for using this mode are similar to the described in Chapter , "Practical scenarios and code examples for using mailboxes" on page 181.

Using signal value as processor ID

This chapter describe one useful scenario for using OR mode. This mode can be useful when one processor needs to asynchronously send some notification (i.e. about reaching a certain step in the program) to a SPE and uses the signal value to identify which processor has sent the signal. In this scenario it is assumed that a SPE may receive notification from different sources.

Following are suggested steps to implement such mechanism:

- Each processor (PPE, SPE) is assigned with one bit in the signaling register.
- A processor that wants to signal some SPE, sends write to the SPE's signal register with the processor's corresponding is set to 1 and other bits are 0.
- A SPE that reads its signal register check which bits are set. For each bit that is set, the SPE knows that the corresponding processor has send a signal to this SPE.
- The SPE that received the signal may then get more information from the sending processor, for example by reading its mailbox or memory.

Using signal value as event ID

This chapter describe one useful scenario for using OR mode. This mode can be useful when a single source processor, usually the PPE, needs to asynchronously send notification about some event (i.e. about the need to execute some command) to a SPE (or few SPEs). In this scenario there are several different events in the program and the signal value is used to identify which event has occurred at this time.

Following are suggested steps to implement such mechanism:

- Each event in the program is assigned with one bit in the signaling register.
- A PPE that wants to signal some SPE about some event write to the SPE's signal register with the event's corresponding bit set to 1 and other bits are 0.
- A SPE that reads its signal register check which bits are set. For each bit that is set, the SPE knows that the corresponding event occurred and handles it.

Code example for using signals notification

The code example below shows covers the following techniques:

- Example 4-37 show the PPU code which signals a SPE. Since the SPE is configured to OR mode we use non blocking access.
  This example also shows how to map the signaling area of the SPEs to the main storage to enable SPEs to signal each other.
- Example 4-38 show the SPU code which reads the local signals using either non blocking methods and blocking methods. The SPUs signals each other in a loop till they receive asynchronous signal from the PPU to stop.
- Example 4-39 show a SPU code that contains functions who implement both signaling another SPE. The code also contains functions for writing the other SPE's mailbox and reading the mailbox status using DMA transactions.
- Example 4-40 show PPU and SPU printing macros for tracing inter-processors communication, such as sending mailboxes and signaling between PPE and SPE and SPE and SPE.

Source code: The code of Example 4-37 through Example 4-40 is included in the additional material that is provided with this book. See "Simple signals" on page 616 for more information.

Example 4-37 PPU code for signaling the SPEs

```
// add the ordinary SDK and C libraries header files...
#include <cbea_map.h>
#include <com_print.h> // the code from Example 4-40

extern spe_program_handle_t spu;

// EA pointer to SPE's singnal1 and singnal2 MMIO registers
volatile spe_sig_notify_1_area_t *ea_sig1[2];
volatile spe_sig_notify_2_area_t *ea_sig2[2];

// main=============================================================
int main()
```

```
{
   int num, ret[2],mbx[2];
   uint32_t sig=0x80000000; // bit 31 indicates signal from PPE
   uint64_t ea;

   for( num=0; num<2; num++){
      // SPE_MAP_PS flag should be set when creating SPE context
      data[num].spe_ctx = spe_context_create(SPE_MAP_PS,NULL);
   }

   // ... Omitted section:
   // load the program to the local stores. and run the SPE threads

   // (the entire source code for this example is part of the book's
   // additional material).

   // This subject of is also described in TBD_REF: Chapter 4.1.2 Task
parallelism and managing SPE threads

   // STEP 0: map SPE's signals area to main storage (get EA)
   for( num=0; num<2; num++){
      if ((ea_sig1[num] = (spe_sig_notify_1_area_t*)spe_ps_area_get(
                  data[num].spe_ctx, SPE_SIG_NOTIFY_1_AREA))==NULL){
         perror("Failed mapping Signal1 area");exit (1);
      }
      if ((ea_sig2[num] = (spe_sig_notify_2_area_t*)spe_ps_area_get(
                  data[num].spe_ctx, SPE_SIG_NOTIFY_2_AREA))==NULL){
         perror("Failed mapping Signal2 area");exit (1);
      }
   }

   // STEP 1: send each SPE the EA of the other SPE's signals area
   // first time writing to SPE so we know mailbox has 4 entries empty
   for( num=0; num<2; num++){
      spe_in_mbox_write(data[num].spe_ctx, (uint32_t*)&num,1,
                        SPE_MBOX_ANY_NONBLOCKING);
      ea = (uint64_t)ea_sig1[(num==0)?1:0];
      spe_in_mbox_write(data[num].spe_ctx, (uint32_t*)&ea,2,
                        SPE_MBOX_ANY_NONBLOCKING);

      // wait we have 2 entries free and then send the last 2 entries
      while(spe_in_mbox_status(data[num].spe_ctx)<2);

      ea = (uint64_t)ea_sig2[(num==0)?1:0];
      spe_in_mbox_write(data[num].spe_ctx, (uint32_t*)&ea,2,
```

```
                        SPE_MBOX_ANY_NONBLOCKING);
}

// STEP 2: wait for SPEs to start signaling loop
for( num=0; num<2; num++){
   while(!spe_out_mbox_status(data[num].spe_ctx));
   spe_out_mbox_read(data[num].spe_ctx, (uint32_t*)&mbx[num], 1);
   prn_p_mbx_s2m(4,1,sig);
};

// STEP 3: wait for  while - let SPEs signal one to another
for( num=0; num<20000000; num++){
   mbx[0] = mbx[0] *2;
}

// STEP 4: send the SPEs a signal to stop
prn_p_sig_m2s(4,0,sig);
prn_p_sig_m2s(4,1,sig);
ret[0]= spe_signal_write(data[0].spe_ctx, SPE_SIG_NOTIFY_REG_1,sig);
ret[1]= spe_signal_write(data[1].spe_ctx, SPE_SIG_NOTIFY_REG_2,sig);

if (ret[0]==-1 || ret[1]==-1){
   perror ("Failed writing signal to SPEs"); exit (1);
}

// STEP 5: wait till SPEs tell me that they're done
for( num=0; num<2; num++){
   while(!spe_out_mbox_status(data[num].spe_ctx));
   spe_out_mbox_read(data[num].spe_ctx, (uint32_t*)&mbx[num], 1);
   prn_p_mbx_s2m(5,num,mbx[num]);
};

// STEP 6: tell SPEs that they can omplete execution
for( num=0; num<2; num++){
   mbx[num] = ~mbx[num];
   spe_in_mbox_write(data[num].spe_ctx, (uint32_t*)&mbx[num],2,
                        SPE_MBOX_ANY_NONBLOCKING);
   prn_p_mbx_m2s(6,num,mbx[num]);
}

// ... Omitted section:
// waits for SPE threads to complete.

// (the entire source code for this example is part of the book's
// additional material).
```

```
   return (0);
}
```

Example 4-38 SPU code for reading local signals and signaling other SPE

```
// add the ordinary SDK and C libraries header files...
#include "spu_mfcio_ext.h" // the code from Example 4-39
#include <com_print.h> // the code from Example 4-40

#define NUM_ITER 1000

uint32_t num;

// Macro for waiting to completion of DMA group related to input tag:
// 1. Write tag mask. 2. Read status untill all tag's DMA are completed
#define waitag(t) mfc_write_tag_mask(1<<t); mfc_read_tag_status_all();

int main( )
{
   uint32_t in_sig,out_sig,mbx,idx,i,ea_h,ea_l,tag_id;
   uint64_t ea_sig[2];

   if ((tag_id= mfc_tag_reserve())==MFC_TAG_INVALID){
      printf("SPE: ERROR can't allocate tag ID\n"); return -1;
   }

   // STEP 1: read from PPE my number using BLOCKING mailbox read
   num     = spu_read_in_mbox();
   idx     = (num==0)?1:0;
   out_sig = (1<<num);

   // STEP 2: receive from PPE EA of other SPE's signal area and string
   while( spu_stat_in_mbox()<4 ); //wait till we have 4 entries
   for (i=0;i<2;i++){
      ea_h = spu_read_in_mbox(); // read EA lower bits
      ea_l = spu_read_in_mbox(); // read EA higher bits
      ea_sig[i] = mfc_hl2ea( ea_h, ea_l);
   }

   // STEP 3: Tell the PPE that we are going to start loopoing
   mbx = 0x44332211; prn_s_mbx_m2p(3,num,mbx);
   spu_write_out_mbox( mbx );
```

```
   // STEP 4: Start looping-  signal other SPE and read my signal
   if (num==0){
      write_signal2( out_sig, ea_sig[idx], tag_id);
      while(1){
         in_sig = spu_read_signal1();

         if (in_sig&0x80000000){ break; } // PPE signals us to stop

         if (in_sig&0x00000002){ // receive signal from other SPE
            prn_s_sig_m2s(4,num,out_sig);
            write_signal2( out_sig, ea_sig[idx], tag_id);
         }else{
            printf("}}SPE%d<<NA:   <%08x>\n",num,in_sig); return -1;
         }
      }
   }else{ //num==1
      while(1){
         in_sig = spu_read_signal2();
         if (in_sig&0x80000000){ break; } // PPE signals us to stop

         if (in_sig&0x00000001){ // receive signal from other SPE
            prn_s_sig_m2s(4,num,out_sig);
            write_signal1( out_sig, ea_sig[idx], tag_id);
         }else{
            printf("}}SPE%d<<NA:   <%08x>\n",num,in_sig); return -1;
         }
      }
   }
   prn_s_sig_p2m(4,num,in_sig);

   // STEP 5: tell tell the PPE that we're done
   mbx = 0x11223344*(num+1); prn_s_mbx_m2p(5,num,mbx);
   spu_write_out_mbox( mbx );

   // STEP 6: block mailbox from PPE- to not finish before other SPE
   mbx = spu_read_in_mbox(); prn_s_mbx_p2m(5,num,mbx);

   mfc_tag_release(tag_id);
   return 0;
}
```

Example 4-39 SPU code for accessing other SPE's mailbox and signals

```
spu_mfcio_ext.h ====================================================
```

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

static uint32_t msg[4]__attribute__ ((aligned (16)));

// mailbox status register definitions
#define SPU_IN_MBOX_OFFSET      0x0C // offset from control area base
#define SPU_IN_MBOX_OFFSET_SLOT 0x3  // 16B alignment= (OFFSET&0xF)>>2

// mailbox status register definitions
#define SPU_MBOX_STAT_OFFSET     0x14 // offset from control area base
#define SPU_MBOX_STAT_OFFSET_SLOT 0x1 // 16B alignment= (OFFSET&0xF)>>2

// signal notify 1 and 2 registers definitions
#define SPU_SIG_NOTIFY_OFFSET    0x0C  // offset from signal areas base
#define SPU_SIG_NOTIFY_OFFSET_SLOT 0x3 // 16B alignment (OFFSET&0xF)>>2


// returns the value of mailbox status register of remote SPE
inline int status_mbox(uint64_t ea_mfc, uint32_t tag_id)
{
   uint32_t status[4], idx;
   uint64_t ea_stat_mbox = ea_mfc + SPU_MBOX_STAT_OFFSET;

   idx = SPU_MBOX_STAT_OFFSET_SLOT;

   mfc_get((void *)&status[idx], ea_stat_mbox, sizeof(uint32_t),
           tag_id, 0, 0);
   mfc_write_tag_mask(1<<tag_id);
   mfc_read_tag_status_any();

   return status[idx];
}

// returns the status (counter) of inbound_mailbox of remote SPE
inline int status_in_mbox(uint64_t ea_mfc, uint32_t tag_id)
{
   int status = status_mbox( ea_mfc, tag_id);
   status = (status&0x0000ff00)>>8;
   return status;
}

// returns the status (counter) of outbound_mailbox of remote SPE
inline int status_out_mbox(uint64_t ea_mfc, uint32_t tag_id)
```

```
{
   int status = status_mbox( ea_mfc, tag_id);
   status = (status&0x000000ff);
   return status;
}

//returns status (counter) of inbound_interrupt_mailbox of remote SPE
inline int status_outintr_mbox(uint64_t ea_mfc, uint32_t tag_id)
{
   int status = status_mbox( ea_mfc, tag_id);
   status = (status&0xffff0000)>>16;
   return status;
}

// writing to a remote SPE's inbound mailbox
inline int write_in_mbox(uint32_t data, uint64_t ea_mfc,
                         uint32_t tag_id)
{
   int status;
   uint64_t ea_in_mbox = ea_mfc + SPU_IN_MBOX_OFFSET;
   uint32_t mbx[4], idx;

   while( (status= status_in_mbox(ea_mfc, tag_id))<1);

   idx = SPU_IN_MBOX_OFFSET_SLOT;
   mbx[idx] = data;

   mfc_put((void *)&mbx[idx], ea_in_mbox,sizeof(uint32_t),tag_id, 0,0);
   mfc_write_tag_mask(1<<tag_id);
   mfc_read_tag_status_any();

   return 1; // number of mailbox being written
}

// signal a remote SPE's signal1 register
inline int write_signal1(uint32_t data, uint64_t ea_sig1,
                         uint32_t tag_id)
{
   uint64_t ea_sig1_notify = ea_sig1 + SPU_SIG_NOTIFY_OFFSET;
   uint32_t idx;

   idx = SPU_SIG_NOTIFY_OFFSET_SLOT;
   msg[idx] = data;

   mfc_sndsig( &msg[idx], ea_sig1_notify, tag_id, 0,0);
```

```
   mfc_write_tag_mask(1<<tag_id);
   mfc_read_tag_status_any();

   return 1; // number of mailbox being written
}

// signal a remote SPE's signal1 register
inline int write_signal2(uint32_t data, uint64_t ea_sig2, uint32_t
tag_id)
{
   uint64_t ea_sig2_notify = ea_sig2 + SPU_SIG_NOTIFY_OFFSET;
   uint32_t idx;

   idx = SPU_SIG_NOTIFY_OFFSET_SLOT;
   msg[idx] = data;

   mfc_sndsig( &msg[idx], ea_sig2_notify, tag_id, 0,0);
   mfc_write_tag_mask(1<<tag_id);
   mfc_read_tag_status_any();

   return 1; // number of mailbox being written
}
```

Example 4-40 PPU and SPU macros for tracing inter-processor communication

```
com_print.h ========================================================
// add the ordinary SDK and C libraries header files...

// Printing macros for tracing PPE-SPE and SPE-SPE communication
// Syntax: prn_X_Y_Z2W:
//   X: 'p' when printing from the PPE, 's' printing from SPE
//   Y: 'mbx' for mailbox, 'sig' for signaling
//   Z: 'm' source is me, 's' source SPE, 'p' source PPE
//   W: 'm' destination is me, 's' destination SPE, 'p' destination PPE
// Paremeters (i,s,m) stands for:
//   i: some user-defined index for example step # in program execution
//   s: For PPE - # of SPE with-which we communicate,
//      For SPE - # of local SPE
//   m: message value (mailbox 32b value, signal value)

#define prn_p_mbx_m2s(i,s,m) printf("%d)PPE>>SPE%02u: <%08x>\n",i,s,m);
#define prn_p_mbx_s2m(i,s,m) printf("%d)PPE<<SPE%02u: <%08x>\n",i,s,m);
#define prn_p_sig_m2s(i,s,m) printf("%d)PPE->SPE%02u: <%08x>\n",i,s,m);
#define prn_p_sig_s2m(i,s,m) printf("%d)PPE<-SPE%02u: <%08x>\n",i,s,m);
```

```
#define prn_s_mbx_m2p(i,s,m) printf("%d{SPE%02u>>PPE: <%08x>\n",i,s,m);
#define prn_s_mbx_p2m(i,s,m) printf("%d{SPE%02u<<PPE: <%08x>\n",i,s,m);
#define prn_s_mbx_m2s(i,s,m) printf("%d{SPE%02u<-SPE: <%08x>\n",i,s,m);
#define prn_s_mbx_s2m(i,s,m) printf("%d{SPE%02u->SPE: <%08x>\n",i,s,m);
#define prn_s_sig_m2p(i,s,m) printf("%d{SPE%02u->PPE: <%08x>\n",i,s,m);
#define prn_s_sig_p2m(i,s,m) printf("%d{SPE%02u<-PPE: <%08x>\n",i,s,m);
#define prn_s_sig_m2s(i,s,m) printf("%d{SPE%02u->SPE: <%08x>\n",i,s,m);
#define prn_s_sig_s2m(i,s,m) printf("%d{SPE%02u<-SPE: <%08x>\n",i,s,m);
```

4.4.3 SPE events

This chapter discuss the SPE events mechanism that enables a code that runs on the SPU to trace events which are external to the program execution. SDK package provide software interface that also enables a PPE program to trace events that occurred on the SPEs.

The following chapters discuss the following topics:

- “SPE events overview” on page 199 provides an overview on this mechanism and its hardware implementation.
- “Programing interface for accessing events” on page 201 describes the main software interfaces for a SPU or PPU program to use the SPE events mechanism.
- “Practical scenarios and code example for using events” on page 202 provides some practical scenarios and techniques for using the mailboxes and also emphasize some code examples.

SPE events overview

Events is an SPE mechanism that enables a code that runs on the SPU to trace events which are external to the program execution. Those event can be set either internally by the hardware of this specific SPE or due to external events such as sending mailbox messages of signal notification by the PPE or the SPEs.

In addition, the SDK package provides software interface that enables a PPE program to trace events that occurred on the SPEs, and create event handler to service those events. Please notice that only a subset of four events are supported by this mechanism. This mechanism is discussed in Chapter , “Programing interface for accessing events” on page 201.

The main events that may be monitored falls into the following categories:

- *MFC DMA*: Events related to MFC's DMA commands. In specific, a code example for handling 'MFC direct memory access (DMA) list command stall-and-notify' is shown in Example 4-20 on page 129.
- *Mailbox or signals*: External write or read to mailbox or signal notification registers.
- *Synchronization events*: Events related to multi source synchronization or lock line reservation (atomic) operation.
- *Decrementer*: events that are set whenever the decrementer's elapsed time has expired.

The events are generated asynchronous to the program execution but software may choose to monitor and correspond to those events either synchronous or asynchronous:

- *synchronous monitoring*: program explicitly check the events status in one of the following ways:
  - nonblocking: poll for pending events by testing the events counts in a loop.
  - blocking: read the event status which stalls when no events are pending.
- *asynchronously monitoring*: implement an event interrupt handler.
- *intermediate approach*: sprinkle 'bisled' instructions, either manually or automatically using code-generation tools, throughout application code so that they are executed frequently enough to approximate asynchronous event detection.

There are four different 32 bits channels that enables an SPU software to manage the events mechanism. The channels have identical bit definition while each event is represented by a single bit. The typical steps that a SPE software should take in order to deals with SPE events are:

1. Initialize event handling by write to 'SPU Write Event Mask' channel and set the bits that correspond to the events that the program wish to monitor.
2. Monitor that some events are pending using either synchronous, asynchronous or intermediate approach as described above.
3. Recognize which events are pending by reading from the 'SPU Read Event Status' channel and see which bits were set.
4. Clear events by writing a value to 'SPU Write Event Acknowledge' and set the bit correspond to the pending events in the written value.
5. Service the events by executing application-specific code for handle the specific events that are pending.

Similarly to the mailbox or signal notification mechanism, each of those registers maintains a counter that may be read by the SPU software. The only counter that

is usually relevant to the software is the one related to ‘SPU Read Status’ channel which may be read by the software to know how many events are pending. Reading the counter returns ‘0’ if no enabled events are pending, and it returns ‘1’ if *enabled* events have been raised since the last read of the status.

A summery of the four available channels is in Table 4-10:

Table 4-10 SPE event channels

Name	RW	Description
SPU Write Event Mask (`SPU_WrEventMask`)	W	To enable only the events that are relevant to its operation, SPU program can initializes a mask value with event bits set to ‘1’ only for the relevant events
SPU Read Event Status (`SPU_RdEventStat`)	R	Reading this channel reports events that are both pending at the time of the channel read and are enabled (the corresponding bit is set in ‘SPU Write Event Mask’).
SPU Write Event Acknowledgment (`SPU_WrEventAck`)	W	Before SPE program services the events reported in ‘SPU Read Event Status’, it should write a value to the ‘SPU Write Event Acknowledge’ to acknowledge (clear) the events that will be processed. Each bit in the written value acknowledge the corresponding event.
SPU Read Event Mask (`SPU_RdEventMask`)	R	Enables the software to read the value that was recently written to ‘SPU Write Event Mask’.

Programing interface for accessing events

The simplest way to access the events is through the MFC functions that are part of SDK package library:

- The SPU programer can manages events with the following functions in `spu_mfcio.h` header file:
  - Enable events using `spu_read_event_mask` and `spu_write_event_mask` functions which access ‘Event Mask’ channel
  - Monitor events using `spu_read_event_status` and `spu_stat_event_status` functions which reads the value and counter of ‘Event Status’ channel.
  - Acknowledge events using `spu_write_event_ack` function which write into ‘Event Acknowledgment’ channel.
  - Retrieve which event are pending using MFC_*_EVENT defines (e.g. `MFC_SIGNAL_NOTIFY_1_EVENT` and `MFC_OUT_MBOX_AVAILABLE_EVENT`)
- PPU program can trace the events that are set on the SPE and implemented an event handler using several functions in libspe library (SPE Runtime Management, defined in `libspe2.h` header file):

- `spe_event_handler_*` functions to create, register, unregister and destroy the event handler.
- `spe_event_wait` function to synchronously wait for events. This is a semi-blocking function - it is stalled till the input timeout parameter expired.
- Use other functions to service the detected event. Depends on the events the appropriate function should be used (e.g. functions for reading the mailbox when mailbox related event occurred).

The `spu_mfcio.h` functions are described in *SPU Event* chapter in C/C++ Language Extensions for Cell BE Architecture document. The `libspe2.h` functions are described in *SPE event handling* chapter in SPE Runtime Management library document.

The programer may take either blocking or nonblocking approach when reading events from the local SPU. The programming methods to do so are similar to those discussed for the mailboxes in chapter Chapter , “Blocking versus non-blocking access to the mailboxes” on page 180. However, reading events form the PPE side is a semi-blocking function which is stalled till the input timeout parameter expired.

There is not specific mechanism to allow on SPE to trace the events of another, but it may be possible to implement such mechanism in software. However, we don’t see such mechanism as practical in most cases.

Practical scenarios and code example for using events

Similar to the mailboxes mechanism, local SPU access to the event channels is internal to the SPE and has very small latency (<=6 cycles for non blocking access). PPE or other SPE access to the event registers has higher latency.

Regarding blocking behavior, local SPU reading the events register when no bits are set is blocking. To avoid blocking, the program may first read the counter as explained below.

Based on the event that is monitored, the events may be used for the following scenarios:

- *DMA list dynamic updates*: Monitor stall-notify-event to update the DMA list according to the data that was transferred to local store from the main storage. A code example for such scenario is in Example 4-20 on page 129.
- *Profiling or watchdog of SPU program:* Use the decrementer to periodically profile the program or implement a watchdog about the program execution.

Another example scenario for using the SPE events is in Example 4-41 on page 204, which provide a code example for implementing an event handler on the PPU.

SPU as computation server

SPE event may be used to implements mechanism in which the SPU act as a computation server who execute commands that are generated and forward to it by the PPU code.

One option to implement it as *asynchronous* computation server. SPU program implements asynchronous events handler mechanism for handling incoming mailboxes from the PPE:

1. SPU code asynchronously wait for inbound mailbox event.
2. PPU code forward to the SPU which commands should be executed (and maybe some other information) by writing commands to the inbound mailbox.
3. SPU code monitor the pending mailbox event and understand which command should be executed.
4. Additional information may be forward from the PPU to the SPU using more mailboxes messages or DMA transfer.
5. SPU process the command.

The SPU side for such mechanism can be implemented as an interrupt (events) handler as described in *Developing a Basic Interrupt Handler* chapter in Cell Broadband Engine Programming Handbook document.

Another option is to implement *synchronous* computation server on the SPU side and implement the event handler on the PPU side:

- SPU code synchronously poll and execute the command that are defined in its inbound mailbox.
- PPU code implement event handler for the SPU events. Whenever PPU monitors that SPU has read the mailbox it write the next command to the SPU mailbox.

“PPU code example for implementing SPE events handler” suggest how to implement such event handler on the PPU.

The second synchronous computation server may have advantages when compared to the asynchronous version since it allows overlapping between different commands as PPU can write to SPU the next command in the same time SPU is working on the current command.

Please notice that there is a large latency between he generation of the SPE event till the execution of corresponding PPU event handler (which involves running some kernel functions). For the reason, only if the delay between one command to another is large, then using the second synchronous computation server make since and provides good performance results.

PPU code example for implementing SPE events handler

This chapter demonstrates how a PPU code may implement handler for SPE events. The code contains a simplified version of the PPU program for implementing the synchronous computation server that is described in Chapter , “SPU as computation server” on page 203.

Please notice that there is a large latency between he generation of the SPE event till the execution of corresponding PPU event handler (roughly 100K cycles) since it involves running some kernel functions.

Example 4-41 contains the corresponding PPU code that creates and registers an event handler for monitoring whenever the inbound mailbox is not full anymore. Any time the mailbox is not full, which indicates that the SPU has read a command from it, the PPU puts new commands in this mailbox.

Please notice that the example aims only to demonstrate how to implement PPE handler for SPE events and uses the event of SPE read from inbound mailbox only as an example. While supporting only this type of event may not always be practical, it can be easily extended to support few different types of other events. For example, it can support also event indicating that an SPE has stopped execution, PPE-initiated DMA operations have completed, or SPE has written to the outbound mailbox. a callback to the PPE-side of the SPE thread (stop and signal mechanism) as described in *PPE-assisted library facilities* chapter in SPE Runtime Management library document.

The SPU code is not shown, but at generally it should include a simple loop that reads coming message from the mailbox and process them.

Source code: The code of Example 4-41 is included in the additional material that is provided with this book. See “PPE event handler” on page 616 for more information.

Example 4-41 Event handler on the PPU

```
// include files....
#include <com_print.h> // the code from Example 4-40

#define NUM_EVENTS 1
#define NUM_MBX 30

// take ‘spu_data_t’ structure and ‘spu_pthread’ function from
// Example 4-5 on page 90

int main()
{
```

```
int i, ret, num_events, cnt;
spe_event_handler_ptr_t event_hand;
spe_event_unit_t event_uni, pend_events[NUM_EVENTS];
uint32_t mbx=1;

data.argp = NULL;

// SPE_EVENTS_ENABLE flag should be set when creating SPE thread
// to enable events tracing
if ((data.spe_ctx = spe_context_create(SPE_EVENTS_ENABLE,NULL))
     ==NULL){
   perror("Failed creating context"); exit(1);
}

// create and register handle event handler
event_hand = spe_event_handler_create();
event_uni.events = SPE_EVENT_IN_MBOX;
event_uni.spe = data.spe_ctx;
ret = spe_event_handler_register(event_hand, &event_uni);

// more types of events may be registered here

// load the program to the local stores, and run the SPE threads.
if (!(program = spe_image_open("spu/spu"))) {
   perror("Fail opening image"); exit(1);
}

if (spe_program_load (data.spe_ctx, program)) {
   perror("Failed loading program"); exit(1);
}

if (pthread_create (&data.pthread, NULL, &spu_pthread, &data)) {
   perror("Failed creating thread");  exit(1);
}

// write 4 first messages to make the mailbox queue full
for (mbx=1; mbx<5; mbx++){
   prn_p_mbx_m2s(mbx,0,mbx);
   spe_in_mbox_write(data.spe_ctx, &mbx,1,SPE_MBOX_ANY_BLOCKING);
}

// loop on all pending events
for ( ; mbx<NUM_MBX; ) {
   // wait for events to be set
   num_events =spe_event_wait(event_hand,pend_events,NUM_EVENTS,-1);
```

```
      // few events were set - handle them
      for (i = 0; i < num_events; i++) {
         if (pend_events[i].events & SPE_EVENT_IN_MBOX){

            // SPE read from mailbox- write to mailbox till it is full
            for (cnt=spe_in_mbox_status(pend_events[i].spe);cnt>0;
                                                          cnt--){
               mbx++;
               prn_p_mbx_m2s(mbx,0,mbx);
               ret = spe_in_mbox_write(pend_events[i].spe, &mbx,1,
                                       SPE_MBOX_ANY_BLOCKING);
            }
         }

         //if we register more types of events- we can handle them here
      }
   }

   // wait for all the SPE pthread to complete
   if (pthread_join (data.pthread, NULL)) {
      perror("Failed joining thread"); exit (1);
   }

   spe_event_handler_destroy(event_hand); //destroy event handle

   // destroy the SPE contexts
   if (spe_context_destroy( data.spe_ctx   )) {
      perror("Failed spe_context_destroy"); exit(1);
   }

   return (0);
}
```

4.4.4 Using atomic unit and the atomic cache

This chapter discuss how to implement a fast shared data structure for inter-processor communication using the Atomic Unit and the Atomic Cache hardware mechanism.

The following chapters discuss the following topics:

- “Atomic unit and the atomic cache overview” on page 207 provides an overview on this mechanism.
- “Programing interface for accessing atomic unit and cache” on page 207 describes the main software interfaces for a SPU or PPU program to use the atomic unit and cache mechanism.
- “Code example for using atomic unit and cache” on page 209 provides a code example for using the atomic unit and cache.

Atomic unit and the atomic cache overview

All the atomic operations supported by the SPE are implemented by a specific Atomic Unit inside each MFC, which contains a dedicated local cache for cache line reservations. This cache is called the Atomic Cache.

The Atomic Cache has a total capacity of six 128-byte cache lines, of which four are dedicated to atomic operations.

When all the SPEs and the PPE perform atomic operations on a cache line with identical Effective Address, and therefore a reservation for that cache line is present in at least one of the MFC units, the cache snooping and update processes are performed by transferring that cache line contents to the requesting SPE or PPE over the Element Interconnect Bus, without requiring a read/write to main system memory.

This constitutes effectively a hardware support for very efficient atomic operations on shared data structures consisting of up to 512 bytes divided in four 128-bytes blocks mapped on a 128-bytes aligned data structure in the SPEs' Local Store, which can be effectively used as a fast broadcast inter-processor communication system.

The approach to exploiting this facility is to extend the principles behind the handling of a mutex lock or an atomic addition, ensuring that the operations involved affect always the same four cache lines.

Programing interface for accessing atomic unit and cache

Two programing methods are available to exploit this functionality:

1. The simplest method involves using two procedures on both SPU and PPU: `atomic_read` and `atomic_set`. These procedures provide access to individual shared 32 bits variables, which can be atomically set to specific values, or atomically modified by simple arithmetic operations using `atomic_add`, `atomic_inc`, and `atomic_dec`.
   Those atomic procedures are part of ‘sync’ library that is delivered with SDK3.0 and is implemented using more basic reservation related

instructions. This library is further dicussed in Chapter 4.5.3, “Using sync library facilities” on page 234.

2. The more powerful method is to allow multiple simultaneous atomic updates to a shared structure, also using more complex update logic. The size of the shared variable can be up to the four lines of 128 bytes atomic cache (aasuming no other mechanism uses this cache).
   In order to use this facility the same sequence of operations required to handle a shared lock is performed using atomic instructions as described below.

The sequence of operations to be performed in the SPU program in order to set a lock on a shared variable is:

1. Perform the reservation for the cache line designated to contain the part of the shared data structure to be updated using `mfc_getllar`. This operation triggers the data transfer from the Atomic Unit containing the most recent reservation or from the PPU cache to the requesting SPE's Atomic Unit over the Element Interconnect Bus.
2. The data structure mapped in the SPU Local Store now contains the most up-to-date values, thus the code can copy the values to a temporary buffer and update the structure with modified values according with the program logic.
3. Attempt the conditional update for the updated cache line using `mfc_putllc`, and if unsuccessful repeat the process from step 1.
4. Upon successful update of the cache line the program can continue having both the previous structure values contained in the temporary buffer, and the modified values in the Local Store mapped structure.

The sequence of operations to be performed in the PPU program in order to set a lock on a shared variable is:

1. Perform the reservation for the cache line designated to contain the part of the shared data structure to be updated using `__lwarx` or `__ldarx`. This operation triggers the data transfer from the Atomic Unit containing the most recent reservation to the PPU cache over the Element Interconnect Bus.
2. The data structure contained at the specified Effective Address, which resides in the PPU cache, now contains the most up-to-date values, thus the code can copy the values to a temporary buffer and update the structure with modified values according with the program logic.
3. Attempt the conditional update for the updated cache line using `__stwcx` or `__stdcx`, and if unsuccessful repeat the process from step 1.

4. Upon successful update of the cache line the program can continue having both the previous structure values contained in the temporary buffer, and the modified values in the structure at the specified Effective Address.

A fundamental difference between the PPE and SPE behavior in managing atomic operations is worth noting: while both use the cache line size (128 bytes) as the reservation granularity, the PPU instructions operate on a maximum of 4 bytes (`__lwarx` and `__stwcx`) or 8 bytes (`__ldarx` and `__stdcx`) at once, whereas the SPE atomic functions update the entire cache line contents.

More details on how to use the atomic instructions on the SPE (`mfc_getllar` and `mfc_putllc`) and on the PPE (`__lwarx`, `__ldarx`, `__stwcx`, and `__stdcx`) is in Chapter 4.5.2, “Atomic synchronization” on page 229.

Provided the Atomic Cache in one of the MFC units or the PPE cache always holds the desired cache lines before another SPE or the PPE requests a reservation on those lines, the data refresh relies entirely on the internal data bus, which offers a very high performance.

Because the libsync synchronization primitives also use the cache line reservation facility in the SPE's MFC, special care must be used to avoid conflicts that may occur when simultaneously exploiting manual usage of the Atomic Unit and other atomic operations provided by libsync.

Code example for using atomic unit and cache

This chapter provides a code example which demonstrates how to use the atomic unit and atomic cache to communicate between the SPEs. The code example shows how to use the atomic instructions on the SPEs (`mfc_getllar` and `mfc_putllc`) to synchronize the access some shred structure.

Example 4-42 shows a PPU code which intiates the shared structure, runs the SPE threads and when the threads complete it reads the shared variable. No atomic access ot this structure is done by the PPE.

Example 4-43 show the SPU code which make use of the atomic instructions to synchronize the access to the shared variables between the SPEs.

Example 4-42 PPU code for using atomic unit and cache

```
// add the ordinary SDK and C libraries header files...
// take ‘spu_data_t’ structure and ‘spu_pthread’ function from
// Example 4-5 on page 90

#define SPU_NUM 8
```

```
spu_data_t data[SPU_NUM];

typedef struct {
   int processingStep; // contains the overall workload processing step
   int exitSignal;     // variable to signal end of processing step
   uint64_t accumulatedTime[8]; // contains workload dynamic execution
                                // statistics (max. 8 SPE)
   int accumulatedSteps[8];
   char  _dummyAlignment[24]; // dummy variables to set the structure
                              // size equal to cache line (128 bytes)
} SharedData_s;


// Main memory version of the shared structure
// size of this structure is a single cache line
static volatile SharedData_s SharedData __attribute__ ((aligned(128)));

int main(int argc, char *argv[])
{
   int i;
   spe_program_handle_t *program;

   // Initialize the shared data structure
   SharedData.exitSignal = 0;
   SharedData.processingStep = 0;

   for( i = 0 ; i < SPU_NUM ; ++i ) {
      SharedData.accumulatedTime[i] = 0;
      SharedData.accumulatedSteps[i] = 0;
      data[i].argp = (void*)&SharedData;
      data[i].spu_id = (void*)i;
   }

   // ... Omitted section:
   // creates SPE contexts, load the program to the local stores,
   // run the SPE threads, and waits for SPE threads to complete.

   // (the entire source code for this example is part of the book's
   // additional material).

   // This subject of is also described in TBD_REF: Chapter 4.1.2 Task
parallelism and managing SPE threads

   // Output the statistics
   for( i = 0; i < SPU_NUM ; ++i) {
```

```
      printf("SPE %d - Avg. processing time (decrementer steps):
              %lld\n", i, SharedData.accumulatedTime[i] /
              SharedData.accumulatedSteps[i]);
   }

   return (0);
}
```

Example 4-43 SPU code for using atomic unit and cache

```
// add the ordinary SDK and C libraries header files...

Same ‘SharedData_s’ structure definition as in Example 4-42

// local version of the shared structure
// size of this structure is a single cache line
static volatile SharedData_s SharedData __attribute__ ((aligned(128)));

// effective address of the shared sturture
uint64_t SharedData_ea;

// argp - effective address pointer to shared structure in main memory
// envp - spu id of the spu
int main( uint64_t spuid , uint64_t argp, uint64_t envp )
{
   unsigned int status, t_start, t_spu;
   int exitFlag = 0, spuNum = envp, i;
   SharedData_ea = argp;

   // Initialize random number generator for fake workload example
   srand( spu_read_decrementer() );

   do{
      exitFlag = 0;

      // Start performace profile information collection
      spu_write_decrementer(0x7fffffff);
      t_start = spu_read_decrementer();

      // Data processing here
      // ...
      // Fake example workload:
```

```
// 1) The first random number < 100 ends first step of the
//    process
// 2) The first number < 10 ends the second step of the process
// Different SPEs process a different amount of data to generate
// different execution time statistics.
// The processingStep variable is shared, so all the SPEs will
// process the same step until one encounters the desired result
// Multiple SPEs can reach the desired result, but the first one
// to reach it will trigger the advancement of processing step

switch( SharedData.processingStep ){
   case 0:
      for( i = 0 ; i < (spuNum * 10) + 10 ; ++i ){
         if( rand() <= 100 ){ //found the first result
            exitFlag = 1;
            break;
         }
      }
      break;

   case 1:
      for( i = 0 ; i < (spuNum * 10) + 10 ; ++i ){
         if( rand() <= 10 ){ // found the second result
            exitFlag = 1;
            break;
         }
      }
   break;
}
// End performance profile information collection
t_spu = t_start - spu_read_decrementer();

// ...
// Because we have statistics on all the SPEs average workload
// time we can have some inter-SPE dynamic load balancing,
// especially for workloads that operate in pipelined fashion
// using multiple SPEs

do{
   // get and lock the cache line of the shared structure
   mfc_getllar((void*)&SharedData, SharedData_ea, 0, 0);
   (void)mfc_read_atomic_status();

   // Update shared structure
   SharedData.accumulatedTime[spuNum] += (uint64_t) t_spu;
```

```
            SharedData.accumulatedSteps[spuNum]++;

            if( exitFlag ){
               SharedData.processingStep++;
               if(SharedData.processingStep > 1)
                  SharedData.exitSignal = 1;
            }

            mfc_putllc((void*)&SharedData, SharedData_ea, 0, 0);
            status = mfc_read_atomic_status() & MFC_PUTLLC_STATUS;

         }while (status);

      }while (SharedData.exitSignal == 0);

      return 0;
}
```

4.5 Shared storage synchronizing and data ordering

While the Cell BE processor executes instructions in program order, it loads and stores data using a “weakly” consistent storage model. This storage model allows storage accesses to be reordered dynamically, which provides an opportunity for improved overall performance and reduced effect of memory latency on instruction throughput.

This model puts a lot of responsibility on the programmer which needs to explicitly order accesses to storage using special synchronization instruction, whenever it is needed that stores occur in the program order. Lack of doing so correctly may result in difficult to debug real time bugs. Program may run correctly on one system and fail on another, or run correctly on one execution and fail on another on the same system.

On the other hand, over usage of those synchronization instruction may significantly reduce the performance as they mostly take a lot of time to complete.

In this chapter we discuss the Cell BE storage model as well as software utilities to control the data transfer ordering. From the reasons mentioned above it is important to understand this topic in order to get efficient and correct results. Further reading on this topic is on *Shared-Storage Synchronization* chapter in Cell Broadband Engine Programming Handbook document.

The following chapters discuss the following issues:

- "Shared Storage model" on page 216 discuss the Cell BE shared storage model and how the different components on the system may force ordering between their data transfers using special ordering instructions. The chapter contains three different sections:
  - "PPE ordering instructions" discuss how a code that runs the PPU may order the PPE data transfers on the main storage with respects to all other elements in the system (e.g. other SPEs).
  - "SPU ordering instructions" discuss how the code that runs the SPU to order SPU data access to the LS with respects to all other elements that may access it, such as the MFC and other elements in the system that access the LS through the MFC (e.g. PPE, other SPEs). Also synchronize the access to the MFC channels.
  - "MFC ordering mechanisms" discuss the MFC ordering mechanism Those instructions are similar to PPU ordering instructions but from the SPU side as they enables the SPU code to order SPE data transfers on the main storage (done by the MFC) with respects to all other elements in the system (e.g. PPE and other SPEs).
- "Atomic synchronization" on page 229 discuss instructions that enables the different components on the Cell BE chip to synchronization atomic access to some shared data structures.
- "Using sync library facilities" on page 234 describe the sync library which provide more high level synchronization functions (based on the instructions mentioned above). The supported C functions closely match those found in current traditional operating systems such as mutex, atomic increment and decrements of variables and conditional variables.
- "Practical examples using ordering and synchronization mechanisms" on page 235 describe some specific useful real-life scenarios for using the ordering and synchronization instructions that are discussed in previous chapters.

Table 4-11 on page 215 summarizes the effects of the different ordering and synchronization instructions, that are discussed on all other chapters, on three storage domains - main storage, local store and channels interface.
It shows effects of instructions issued by different components - the PPU code, the SPU code, and the MFC. Regarding the MFC, the intention here is for data transfers are executed by the MFC following commands that were issued toward the MFC by either the SPU code (using the channel interface) or PPU code (using MMIO interface).

Table 4-11 Effects of Synchronization on Address and Communication Domains[1]

Issuer	Instruction, Command, or Facility	Main-Storage Domain			LS Domain[2]			Channel Domain[3]
		Accesses by PPE		Accesses by All Other Processor Elements and Devices	Accesses by Issuing SPU	Accesses by Issuing SPU's MFC	Accesses by All Other Processor Elements and Devices	Accesses by Issuing SPU
		Issuing Thread	Both Threads					
PPU	sync[4]	all accesses				Unreliable. Use MFC Multisource Synchronization Facility[5]		
	lwsync[6]	accesses to memory-coherence-required locations						
	eieio	accesses to caching-inhibited and guarded locations		accesses to caching-inhibited and guarded locations		Unreliable. Use MFC Multisource Synchronization Facility[5]		
	isync	instruction fetches						
SPU	sync				all accesses			all accesses
	dsync				load and store accesses	all accesses		
	syncc				all accesses			

1. Gray shading in a table cell means that the instruction, command, or facility has no effect on the referenced domain.
2. The LS of the issuing SPE.
3. The channels of the issuing SPE.
4. This is the PowerPC **sync** instruction with L = '0'.
5. These accesses can exist only if the LS is mapped by the PPE operating system to the main-storage space. This can only be done if the LS is assigned caching-inhibited and guarded attributes.
6. This is the PowerPC **sync** instruction with L = '1'.

Table 4-11 Effects of Synchronization on Address and Communication Domains[1]

Issuer	Instruction, Command, or Facility	Main-Storage Domain			LS Domain[2]			Channel Domain[3]
		Accesses by PPE		Accesses by All Other Processor Elements and Devices	Accesses by Issuing SPU	Accesses by Issuing SPU's MFC	Accesses by All Other Processor Elements and Devices	Accesses by Issuing SPU
		Issuing Thread	Both Threads					
MFC	mfcsync			all accesses		Unreliable. Use MFC Multisource Synchronization Facility[5]		
	mfceieio			accesses to caching-inhibited and guarded locations				
	barrier			all accesses				
	<f>, <b>			all accesses for the tag group				
	MFC Multisource Synchronization Facility			all accesses	all accesses			

1. Gray shading in a table cell means that the instruction, command, or facility has no effect on the referenced domain.
2. The LS of the issuing SPE.
3. The channels of the issuing SPE.
4. This is the PowerPC **sync** instruction with L = '0'.
5. These accesses can exist only if the LS is mapped by the PPE operating system to the main-storage space. This can only be done if the LS is assigned caching-inhibited and guarded attributes.
6. This is the PowerPC **sync** instruction with L = '1'.

4.5.1 Shared Storage model

Unlike the in-order execution of instructions in Cell BE, the processor loads and stores data using a weakly consistent storage model. This means that the order in which any following three are executed might be different from each other:

- Order of any processor element (PPE or SPE) perform storage accesses.
- Order in which those accesses are performed with respect to another processor element.
- Order in which those accesses are performed in main storage.

In order to ensure that accesses to shared storage are performed in program order, software must place memory-barrier instructions between storage accesses.

The term storage access means an access to main storage caused by a load, a store, a direct memory access (DMA) read, or a DMA write. There are two orders to consider:

- Order of instructions execution. Cell BE is in-order machine, which means that from the programmer viewpoint it appears that the instructions are executed in the order specified by the program.
- Order shared-storage accesses. The order in which shared-storage accesses are performed might be different from both program order and the order in which the instructions that caused the accesses are executed.

PPE ordering instructions

PPU ordering instructions enable the code that runs the PPU to order the PPE data transfers on the main storage with respects to all other elements in the system (e.g. other SPEs). Ordering of storage accesses and instruction execution may be explicitly controlled by the PPE program using barrier instructions. These instructions can be used between storage-access instructions to define a memory barrier that divides the instructions into those that precede the barrier instruction and those that follow it.

PPE supported barrier instructions are defined as intrinsics in `ppu_intrinsics.h` header file so the programmer can easily use them in any C code application. There are two types of such instruction - storage barriers and instruction barriers as described in Table 4-12.

Table 4-12 PPE barrier intrinsics

Intrinsic	Description	Usage
Storage Barriers		

Intrinsic	Description	Usage
__sync()	Known as the heavyweight sync, ensures that all instructions preceding the sync appear to have completed before the sync instruction completes, and that no subsequent instructions are initiated until after the sync instruction completes. This does not mean that the previous storage accesses have completed before the 'sync' instruction completes.	To ensure that the results of all stores into a data structure, caused by store instructions executed in a critical section of a program, are seen by other processor elements before the data structure is seen as unlocked.
__lwsync()	Also known as light weight sync, creates the same barrier as the sync instruction for storage accesses that is memory coherence. Therefore, unlike 'sync' instruction, it orders only PPE's main-storage accesses and has no effect on the main-storage accesses of other processor elements.	When ordering is required only for coherent memory, because it executes faster than 'sync'.
__eieio()	Enforce in-order execution of I/O means that all main-storage accesses caused by instructions proceeding the 'eieio' have completed, with respect to main storage, before any main-storage accesses caused by instructions following the 'eieio'. The eieio instruction does not order accesses with differing storage attributes. For example, if an eieio is placed between a caching-enabled store and a caching-inhibited	Managing shared data structures, accessing memory-mapped I/O (such as SPEs MMIO interface), and preventing load or store combining.
Instruction Barrier		
__isync()	ensures that all PPE instructions proceeding the isync are completed before isync is completed. causes issue stall and blocks all other instructions from both PPE threads until the isync instruction completes.	In conjunction with self-modifying PPU code, executed after an instruction is modified and before it is executed. Also may be used during context switching when the MMU translation rules are being changed.

Table 4-13 summarizes the use of the storage barrier instructions for two common types of main-storage memory:

- *System memory*: The coherence main memory of the system. The XDR main memory falls into this category and so are the local stores when they are accessed from the EIB bus (by the PPE or SPEs other then theirs).
- *Device memory*: Memory that is caching-inhibited and guarded. In Cell a BE system it is typical of memory-mapped I/O devices such as the DDR that is attached to the south bridge. Mapping of SPEs' LS to main storage is caching-inhibited but not guarded.

In these tables, "yes" (and "no") mean that the instruction performs (or does not perform) a barrier function on the related storage sequence, "rec" (stands for "recommended") means that the instruction is the preferred one, "not rec" means that the instruction will work but is not the preferred one, and "not req" (stands for "not required") and "no effect" mean the instruction has no effect.

Table 4-13 Storage-barrier ordering of accesses to system memory and device memory

Storage-Access Instruction Sequence	System memory			Device memory		
	sync	lwsync	eieio	sync	lwsync	eieio
load-*barrier*-load	yes	rec	no affect	yes	no affect	yes
load-*barrier*-store	yes	rec	no affect	yes	no affect	yes
store-*barrier*-load	yes	no	no affect	yes	no affect	yes
store-*barrier*-store	yes	rec	not rec	not req [a]	no affect	not req[a]

a. Two stores to caching-inhibited storage are performed in the order specified by the program, regardless if they are separated by a barrier instruction or not

SPU ordering instructions

SPU ordering instructions enable the code that runs the SPU to order SPU data access to the LS with respects to all other elements that may access it, such as the MFC and other elements in the system that access the LS through the MFC (e.g. PPE, other SPEs). They also synchronize the access to the MFC channels. An LS can experience asynchronous interaction from the following streams that access it:

- Instruction fetches by the local SPU
- Data loads and stores by the local SPU
- DMA transfers by the local MFC or another SPE's MFC
- Loads and stores in the main-storage space by other processor elements.

With regard to an SPU, the Cell BE's in-order execution model guarantees only that SPU instructions that access that SPU's LS appear to be performed in

program order with respect to that SPU but not necessarily with respect to external accesses to that LS or with respect to the SPU's instruction fetch.

Hence, from architecture point of view, in case an SPE write some data to the LS and immediately later generate MFC 'put' command that read this data (and transfer it to the main storage), then without synchronization instructions it is not guaranteed that the MFC will read the latest data (since it is not guaranteed that MFC reading the data is perform after the SPU writing the data). However, from practical point of view there is no need to add the synchronization command to guarantee this ordering since executing the six commands for issuing the DMA always takes more then executing the former write to the LS.

From the programmer point of view, it means that in the absence of external LS writes, an SPU load from an address in its LS returns the data written by that SPU's most-recent store to that address. However, this statement is not necessarily true for an instruction fetch from that address which may not guaranteed to return that recent data. The following statement regarding instruction fetch effect only cases in of self-modifying code.

Please notice that in case the LS and MFC resources of some SPE (which are mapped to the system-storage address space) are accessed by software running on the PPE or other SPEs, it is not guaranteed that two accesses that are made to two different are ordered, unless some synchronization command (e.g. 'eieio' or 'sync' are explicitly executed by the PPE or other SPEs, as explained in "PPE ordering instructions" on page 217.

In the descriptions bellow, we use the terms "SPU load" and "SPU store" to described accesses by the same SPU that executes the synchronization instruction.

Several practical example for using the SPU ordering instructions are discussed in *Synchronization and Ordering* chapter of Synergistic Processor Unit Instruction Set Architecture Version 1.2 document. In specific sub-chapter *External Local Storage Access* which demonstrates how those instructions may be used when processor which is external to the SPE (e.g. PPE) access the LS for example in order to write some data to this LS and later notify the code that runs on the associated SPU that the writing the data is completed by writing to another address in this same LS.

The SPU instruction set provides three synchronization instructions. The easiest way to use those instructions is through intrinsics and in order to do so the programmer should include the `spu_internals.h` header file. A brief description of these intrinsics and their main usage is summarized in Table 4-14.

Table 4-14 SPU ordering instructions

Intrinsic	Description	Usage
spu_sync	'sync' (synchronize) instruction causes the SPU to waits until all pending instructions of loads and stores to LS and channel accesses have been completed before fetching the next instruction.	This instruction is most often used in conjunction with self-modifying SPU code. It must be used before attempting to execute new code that either arrives through DMA transfers or is written with store instructions.
spu_dsync	'dsync' (synchronize data) instruction ensures that data has been stored in the LS before the data becomes visible to the local MFC or other external devices.	Architecturally, Cell BE DMA transfers may interfere with store instructions and the store buffer, so 'dsync' meant to ensure that all DMA store buffers are flushed to the LS (i.e., all previous stores to LS will be seen by subsequent LS accesses. However, current Cell implementation does not require 'dsync' instruction for doing so as it handle it by HW.
spu_sync_c	'syncc' (synchronize channel) ensures channel synchronization followed by the same synchronization provided by the 'sync' instruction.	To ensure that the effects on SPU state caused by prior write to some nonblocking channel are propagated and influence the execution of the following instructions.

The instructions have both coherency and instruction-serializing effects which are summarized Table 4-15.

Table 4-15 Effects of the SPU ordering instructions

Intrinsic	Ensures these coherency effects	Forces this instruction serialization effects
spu_dsync	1. Subsequent external read access data written by prior SPU stores. 2. Subsequent SPU loads access data written by external writes.	► Forces SPU load and SPU store access of LS due to instructions before the 'dsync' to be completed before completion of 'dsync'. ► Forces read channel operations due to instructions before the 'dsync' to be completed before completion of the 'dsync'. ► Forces SPU load and SPU store access of LS due to instructions after the 'dsync' to occur after completion of the 'dsync'. ► Forces read-channel and write-channel operations due to instructions after the 'dsync' to occur after completion of the 'dsync'.

Intrinsic	Ensures these coherency effects	Forces this instruction serialization effects
spu_sync	► Effects 1 and 2 of spu_dsync 3. Subsequent instruction fetches access data written by prior SPU stores and external writes.	► All access of LS and channels due to instructions before the 'sync' to be completed before completion of 'sync'. ► All access of LS and channels due to instructions after the 'sync' to occur after completion of the 'sync'.
spu_sync_c	► Effects 1 and 2 of spu_dsync ► Effects 3 of spu_sync 4. Subsequent instruction processing is influenced by all internal execution states modified by previous write instructions to some channel.	► All access of LS and channels due to instructions before the 'syncc' to be completed before completion of 'syncc'. ► All access of LS and channels due to instructions after the 'syncc' to occur after completion of the 'syncc'.

Table 4-16 shows which SPU synchronization instructions are required between LS writes and LS reads to ensure that reads access data written by prior writes as it highlights the differences between different initiators:

Table 4-16 Synchronization instructions for accesses to an LS

Writer	Reader		
	SPU instruction fetch	SPU load	External read[a]
SPU Store	'sync'	nothing required	'dsync'
External Write[a]	'sync'	'dsync'	NA

a. By any DMA transfer (from the local MFC or a non-local MFC), the PPE, or other device—other than the SPU that executes the synchronization instruction

MFC ordering mechanisms

SPU may use the MFC channel interface to issue commands to the associated MFC. The PPU or other SPUs outside this SPE may similarly use the MFC's MMIO interface in order to send commands to a particular MFC. For each of those interfaces independently, the MFC accepts only queueable commands which are entered into one of the MFC SPU command queue (one queue for the channels interfaces and another for the MMIO). The MFC then processes these commands, possibly out of order to improve efficiency.

However, MFC supports ordering mechanism that may be activated through each of those two main interfaces:

- *Channel interface*: allows an SPU code to control the order in which the MFC execute the commands that have been previously issued by this SPU using the channel interface.
- *MMIO interface*: similarly but independently, PPU or other SPUs may use the MMIO interface to control the order in which MFC issue command that have been previously queued on its MMIO interface.

It is important to mention that the effect of those commands, regardless if they are issued through either of the two interfaces, actually controls the order of the MFC data transfers on the main storage with respects to all other elements in the system (e.g. PPE and other SPEs).

There are two types of ordering commands:

- *Fence or barrier command options*: tag specific mechanism that is activated by appending a 'fence', or 'barrier' options to either data transfer or signaling commands.
- *Barrier commands*: a separate barrier command can be issued to order the command against all preceding and all succeeding commands in the queue, regardless of tag group.

The following section describes more about those two types of commands.

Fence or barrier command options

The *fence* or *barrier command options* ensure local ordering of storage accesses made through the MFC with respect to other devices in the system. The 'local' ordering ensures ordering of the MFC commands with respect to that particular MFC tag group (commands that have similar tag ID) and command queue (i.e. MFC proxy command queue and MFC SPU command queue). Both ordinary DMA and list DMA commands are supported and well as signalling commands.

Programmers can enforce ordering among DMA commands in a tag group with a fence or barrier option by appending an 'f' for 'fence', or a 'b', for 'barrier', to the signaling commands (e.g. 'sndsig') or data transfer commands (e.g. 'getb' and 'putf'). The simplest way to do so is using the supported MFC functions:

- SPU code may use the functions call defined by `spu_mfcio.h` header file. For example use `mfc_getf` and `mfc_putb` functions to issue 'fenced get' command and 'barrier put' command respectively.
- PPU code may use the functions call defined by `libspe2.h` header file. For example use `spe_mfcio_getf` and `spe_mfcio_putb` functions to issue 'fenced get' command and 'barrier put' command respectively.

Table 4-17 lists the supported tag-specific ordering commands:

Table 4-17 MFC tag-specific ordering commands

Option	Commands
barrier	getb, getbs, getlb, putb, putbs, putrb, putlb, putrlb, sndsigb
fence	getf, getfs, getlf, putf, putfs, putrf, putlf, putrlf, sndsigf

The ‘fence’ and ‘barrier’ option has different effects:

- *Fenced command* is not executed until all previously issued commands within the same tag group have been performed; commands issued after the fenced command might be executed before the fenced command.
- *Barrier command* and all the commands issued after the barrier command are not executed until all previously issued commands in the same tag group have been performed.

Once those data transfers were issues, the storage system may complete requests in an order different then the order in which they are issued, depending on the storage attributes. However, in specific it is guaranteed that accesses to the main memory (which has caching-inhibited storage) and other SPE’s LS and problem state are completed in the same order in which they are issued.

The different effects of the ‘fenced’ and ‘barrier’ command are illustrated in Figure 4-4. The row of white boxes represents command-execution slots, in real-time, in which the DMA commands (red and green boxes) might execute. Each DMA command is assumed to transfer the same amount of data, thus, all boxes are the same size. The arrows show how the DMA hardware, using out-of-order execution, might execute the DMA commands over time.

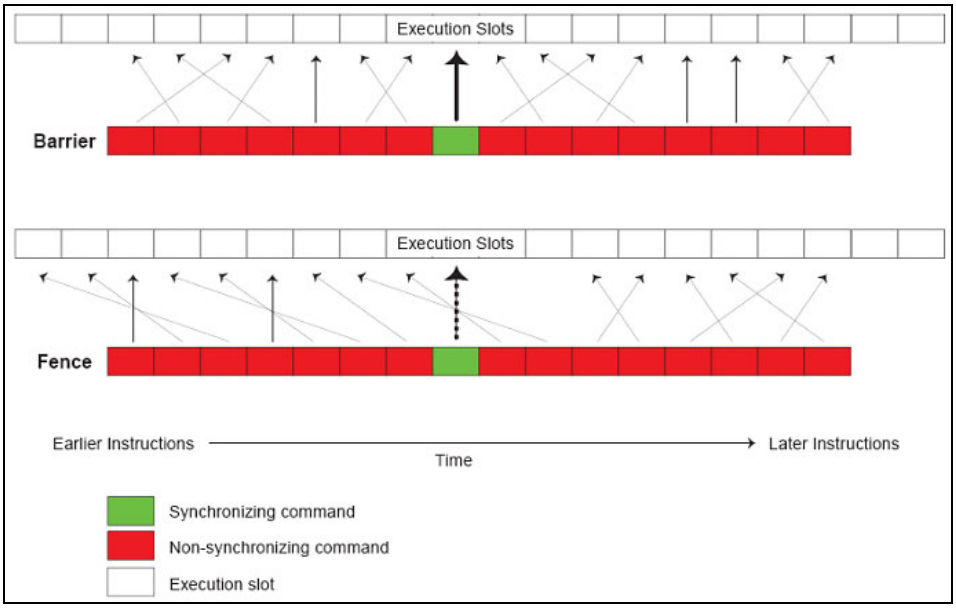


Figure 4-4 Barrier and fence effect

Those commands are very useful and efficient in synchronizing SPU code data access to the shared storage with access of other elements in the system. One of the common use of those command is in double buffering mechanism, as explained in Chapter , “Double buffering” on page 157 and illustrated through the code of Example 4-29 on page 159. For more scenarios examples see Chapter 4.5.4, “Practical examples using ordering and synchronization mechanisms” on page 235.

Barrier commands

*T*he *barrier commands* order storage accesses made through the MFC with respect to all other MFCs, processor elements, and other devices in the system. While the CBEA specifies those commands as having tag-specific effects (controls only the order in which transfers related to one tag-ID group are executed compare to each other), the current Cell BE implementation have no tag-specific effects.

Those commands may activated only by the SPU that is associated with the MFC using the channel interface. There is no support from the MMIO interface. However, the PPU may achieve similar effects by using the not-MFC-specific ordering instructions that described in Chapter , “PPE ordering instructions” on page 217.

The easiest way to use those instructions from the SPU side is through intrinsics. In order to do so the programmer should include the `spu_mfcio.h` header file. A brief description of these intrinsics and their main usage is summarized in Table 4-18.

Table 4-18 MFC ordering commands

Intrinsic	Description	Usage
mfc_sync	'mfcsync' command is similar to PPE 'sync' instruction and controls the order in which MFC commands are executed with respect to storage accesses by all other elements and in the system.	Designed to be used inter-processor/device synchronization. Since it creates a large load on the memory system, should be used only between commands involving storage with different storage attributes - otherwise other synchronization commands should be preferred.
mfc_eieio	'mfceieio' command controls the order in which DMA commands are executed with respect to storage accesses by all other system elements, only when the storage being accessed has the attributes of caching-inhibited and guarded (typical for I/O devices). The command is similar to PPE 'eieio' instruction - for more details regarding the effects on accessing different types of memories - seeTable 4-13 on page 219.	Managing shared data structures, performing memory-mapped I/O, and preventing load and store combining in main storage. The 'fence' and 'barrier' options of other commands are preferred from performance point of view so should be use in case they are sufficient.
mfc_barrier	'barrier' command orders all subsequent MFC commands with respect to all MFC commands preceding the barrier command in the DMA command queue, independent of tag groups. The barrier command will not complete until all preceding commands in the queue have completed. After the command completes, subsequent commands in the queue may be started.	Managing data structures which are located in main storage and are shared by other elements in the system.

MFC multisource synchronization facility

The Cell BE processor contains multiple address and communication domains - the main-storage domain, eight local LS-address domains, and eight local channel domains. MFC multisource synchronization facility ensure cumulative ordering of storage accesses performed by multiple sources (e.g PPE and SPEs) across all those address domains, unlike the PPE 'sync' instruction and other

similar instructions which provides such cumulative ordering only with respect to the main-storage domain.

The MFC multisource synchronization facility addresses this cumulative-ordering need by providing two independent multisource synchronization-request methods:

- *MMIO interface* - allows the PPE or other processor elements or devices to control synchronization from the main-storage domain.
- *Channel interface* - allows an SPE to control synchronization from its LS-address and channel domain.

Each of these two synchronization-request methods ensures that all write transfers to the associated MFC are sent and received by the MFC before the MFC synchronization-request is completed. This facility does not ensure that read data is visible at the destination when the associated MFC is the source.

The two methods operate independently so synchronization request through the MMIO register has no effect on synchronization requests through the channel, and vice versa.

MMIO interface of MFC multisource synchronization facility

MFC multisource synchronization facility may be accessed from the main storage domain by the PPE or other processor elements or devices using the MMIO MFC_MSSync (MFC multisource synchronization) register. A programmer can access this facility through two functions that are defined in `libspe2.h` header file and are further described in SPE Runtime Management library document.

Example 4-44 illustrates how the PPU programmer may achieve cumulative ordering using the two corresponding `libspe2.h` functions:

Example 4-44 MMIO interface of MFC multisource synchronization facility

```
#include “libspe2.h”

// Do dome MFC DMA operation between memory and LS of some SPE
// PPE/other-SPEs use our MFC to transfer data between memory and LS

int status;

spe_context_ptr_t spe_ctx;

// init one or more SPE threads (also init ‘spe_ctx’ variable)

// Send a request to the MFC of some SPE to start tracking outstanding
// transfers which are sent to this MFC by either te associated SPU or
```

```
// PPE or other-SPEs.

status = spe_mssync_start();

if (status==-1){
   // do whatever need to do on ERROR but do not continue to next step
}

// Check if all the transfers that are being tracked are completed.
// Repeat this step till the function returns 0 indicating the
// completions of those transfers

while(1){
   status = spe_mssync_status(spe_ctx); // nonblocking function

   if (status==0){
      break; // synchronization was completed
   }else{
      if (status==-1){
         // do whatever need to do on ERROR
         break; //unless we already exit program because of the error
      }
   }
};
```

Channel interface of MFC multisource synchronization facility

MFC multisource synchronization facility may be accessed by the local SPU code form the LS domain using the MFC_WrMSSyncReq (MFC write multisource synchronization request) channel. A programmer can access this facility through two functions that are defined in `spu_mfcio.h` header file and are further described in C/C++ Language Extensions for Cell BE Architecture document. Example 4-45 illustrates how the SPU programmer may achieve cumulative ordering using the two corresponding `spu_mfcio.h` functions:

Example 4-45 Channel interface of MFC multisource synchronization facility

```
#include “spu_mfcio.h”


uint32_t status;

// Do dome MFC DMA operation between memory and LS
// PPE/other-SPEs use our MFC to transfer data between memory and LS
```

```
// Send a request to the associated MFC to start tracking outstanding
// transfers which are sent to this MFC by either this SPU or PPE or
// other-SPEs

mfc_write_multi_src_sync_request();


// Check if all the transfers that are being tracked are completed.
// Repeat this step till the function returns 1 indicating the
// completions of those transfers

do{
   status = mfc_stat_multi_src_sync_request(); // nonblocking function
} while(!staus);
```

Other alternative is using asynchronous event that may be generated by the MFC to indicate the completion of the requested data transfer. Chapter *MFC Multisource Synchronization Facility* in the Cell Broadband Engine Programming Handbook describes more about this alternative and other issues related to MFC multisource synchronization facility.

4.5.2 Atomic synchronization

This section the atomic operations that are supported by Cell BE. Those operation are implemented on both the SPE and the PPE and enables the programmer to create synchronization primitives such as semaphores and mutex locks in order to synchronize storage access or other functions. Those feature should be use with special care in order to avoid livelocks and deadlocks.

The atomic operation that are implemented in Cell BE are not blocking so this enables the programmer to implement algorithms that are lock-free and wait-free.

Atomic operations are described in details in *PPE Atomic Synchronization* chapter in Cell Broadband Engine Programming Handbook document. the chapter also contains some usage examples and how those atomic operations may be used to implement synchronization primitives such as mutex, atomic addition (part of semaphore implementation), and condition variables. We recommend to the advanced programmer to read this chapter in order to further understand this mechanism.

Atomic synchronization instructions

The atomic mechanism enables to set a lock (called 'reservation') on some aligned unit of real storage (called a 'reservation granule') containing the address that we wish to lock. The Cell BE reservation granule is 128 bytes, corresponding to the size of a PPE cache line, means the programmer may set a lock on block of 128 bytes which is also aligned to 128 bytes address.

The atomic synchronization mechanism include the following instructions:

- *Load-and-reserve instructions*: load the addressed value from memory and then set a reservation on the reservation granule that containing the address.
- *Store-conditional instructions*: verify that the reservation is still set on the granule and only if it is set the store operation is carried out. Other wise (reservation does not exist) the instruction completes without altering storage. The hardware set indication bit in CRT register to enables the programmer to determine if the store was successful.

The reservation is cleared by setting another reservation or by executing a conditional store to any address. Another processor element may also clear the reservation by accessing the same reservation granule.

A pair of load-and-reserve and store-conditional instructions permits atomic update of variable in main storage which enables the programmer to implement various synchronization primitives such as semaphore, mutex lock, test-and-set, fetch-and-increment, and any atomic update of a single aligned word or doubleword in memory. Example 4-46 Illustrate how this mechanism may be used to implement a semaphore:

Example 4-46 implementing semaphore using load-and-reserve and store-conditional

```
1. read a semaphore using load-and-reserve.
2. compute a result based on the value of the semaphore.
3. use store-conditional to write the new value back into the semaphore
   location only if that location has not been modified (i.e. by other
   processor) since it was read in step 1.
4. determine if the store was successful;
   – if successful: the sequence of instructions from steps 1 to step 3
     appears to have been executed atomically.
   – otherwise: other processor accessed the semaphore so the software
     may repeat this process (back to step 1).
```

These instructions also control the order in which memory operations are completed with respect to asynchronous events, and the order in which memory operations are seen by other processor elements or memory access mechanisms.

PPE and SPE atomic implementation

The atomic synchronization is implemented on both the PPE and the SPE:

- *PPE side*: atomic synchronization is implemented through a set of assembly instructions. A set of specific intrinsics to make those PPU instructions easily accessible from the C programming language as each of these intrinsics has a one-to-one assembly language mapping. Programer should include `ppu_intrinsics.h` header file to use them.
- *SPE side*: atomic synchronization is implemented on the SPE with a set of MFC synchronization commands which are accessible through a set of functions provided by `spu_mfcio.h` file.

Table 4-19 summarized both the PPE and SPE atomic instructions. For each PPE instruction attached an SPE's MFC commands that implement similar mechanism. Please notice that for all the PPE instructions, reservation (lock) is set for the entire cache line in which this word resides.

Table 4-19 Atomic primitives of PPE and SPE

PPE	Description	SPE (MFC)	Description
Load and reserve instructions			
__ldarx	Load a doubleword (cache line) and set reservation.	mfc_getllar	Transfer cache line from LS to main storage and created a reservation (lock). Not tagged and is executed immediately (not queued behind other DMA commands.
__lwarx	Load a word and set reservation[a].	-	-
Store conditional instructions			
__stdcx	Store a doubleword (cache line) only if reservation (lock) exists.	mfc_putllc	Transfer cache line from LS to main storage only if reservation (lock) exists.
__stwcx	Store a word only if reservation exists[a].	-	-

PPE	Description	SPE (MFC)	Description
-	-	mfc_putlluc	Put lock-line unconditional (regardless if reservation exists. Executed immediately.
-	-	mfc_putqlluc	Put lock-line unconditional (regardless if reservation exixts.is placed into the MFC command queue, along with other MFC commands.

a. Reservation (lock) is set for the entire cache line in which this word resides.

There are two pairs of atomic instructions that are implemented on both the PPE and the SPE. The first pair is `__ldarx/_lwarx` and `mfc_getllar` and the second pair is `__stdcx/_stwcx` and `mfc_putllc` for PPE and SPE respectively. This functions provide atomic read-modify-write operations that may be used to derive other synchronization primitives between a program that runs on the PPU and a program code that runs on the SPU (or SPUs).

PPU code and SPU code examples of implementing a mutex-lock mechanism using those two pairs of atomic instructions is shown Example 4-47 and Example 4-48 for PPE and SPE respectively.

Chapter 4.5.3, “Using sync library facilities” on page 234 illustrates how *sync library* implement many of the standard synchronization mechanisms, such as mutex and semaphore, using the atomic instructions. Example 4-47 is actually based on a ‘sync’ library code from `mutex_lock.h` header file.

Note: Programmer should consider using the various synchronization mechanisms that are implemented in sync library instead of explicitly using the atomic instructions that are described in this chapter. For more information see Chapter 4.5.3, “Using sync library facilities” on page 234.

Chapter *PPE Atomic Synchronization* in Cell Broadband Engine Programming Handbook document provides more code examples on how synchronization mechanisms may be implemented on both a PPE program and SPE program. using those instructions in order to achieve synchronization between the two programs. Example 4-48 is based on one of those examples.

Example 4-47 PPE implementation of mutex_lock function in sync library

```
#include “ppu_intrinsics.h”

// assumes 64 bit compelation of the code
```

```
void mutex_lock(uint64_t mutex_ptr) {

   uint32_t done = 0;

   do{
      if (__lwarx((void*)mutex_ptr) == 0)
         done = __stwcx((void*)mutex_ptr, (uint32_t)1);

   }while (done == 0); // retry if the reservation was lost

   __isync(); // synchronize with other data transfers
}
```

Example 4-48 SPE implementation of mutex lock

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

void mutex_lock(uint64_t mutex_ptr) {
   uint32_t offset, status, mask;

   volatile char buf[256], *buf_ptr;
   volatile int *lock_ptr;

   // determine the offset to the mutex word within its cache line.
   // align the effective address to a cache line boundary.
   uint32_t offset   = mfc_ea2h(mutex_ptr) & 0x7F;
   uint32_t mutex_lo = mfc_ea2h(mutex_ptr) & ~0x7F;
   mutex_ptr = mfc_hl2ea(mfc_ea2h(mutex_ptr), mutex_lo);

   // cache line align the local stack buffer.
   buf_ptr = (char*)(((uint32_t)(buf) + 127) & ~127);
   lock_ptr = (volatile int*)(buf_ptr + offset);

   // setup for use possible use of lock line reservation lost events.
   // detect and discard phantom events.
   mask = spu_read_event_mask();
   spu_write_event_mask(0);

   if (spu_stat_event_status()) {
      spu_write_event_ack( spu_read_event_status());
   }
   spu_write_event_mask( MFC_LLR_LOST_EVENT );
```

```
   do{ //get-and-reservation the cache line containing mutex lock word.

      mfc_getllar( buf_ptr, mutex_ptr, 128, tag_id,0,0);
      mfc_read_atomic_status();

      if (*lock_ptr) {
         // The mutex is currently locked. Wait for the lock line
         // reservation lost event before checking again.
         spu_write_event_ack( spu_read_event_status());

         status = MFC_PUTLLC_STATUS;
      } else {
         // The mutex is not currently locked. Attempt to lock.
         *lock_ptr = 1;

         // put-conditionally, the cache line containing the lock word.
         mfc_putllc( buf_ptr, mutex_ptr, 128, tag_id,0,0);
         status = mfc_read_atomic_status() & MFC_PUTLLC_STATUS;
      }
   } while (status); // retry if the reservation was lost.

   spu_write_event_mask(mask); // restore the event mask
}
```

4.5.3 Using sync library facilities

The ‘sync’ library provides several simple general purpose synchronization constructs. The supported C functions closely match those found in current traditional operating systems.

Most of the functions of this library are supported by both the PPE and the SPE, but small portion of them are supported only by the SPE. The functions are all based upon the Cell BE load-and-reserve and store-conditional functionality that is described in Chapter 4.5.2, “Atomic synchronization” on page 229.

In order to use the facilities of the ‘sync’ library, the programmer should refer to the following files:

- `libsync.h`: the header file that should be included as it contains most of the definitions.
- `libsync.a`: the library that contains the implementation and should be linked to the program.

- *function specific header files*: each function is defined by a separate header file so the programmer may include this header file instead of the `libsync.h` file. For example the programmer may include `mutex_init.h` header file when 'mutex_lock' operation is needed. In this case an underline should be added when calling the function (e.g. `_mutex_lock` function when including `mutex_init.h` file instead of `mutex_lock` when including `libsync.h`.

Note: Using the function specific header files is preferred from performance point of view since those function are defined as *inline*, unlike the definition of the corresponding function in *libsync.h* file. However, similar effect may be achieved by setting the appropriate compilation flags.

The 'sync' library provides five sub-classes of synchronization primitives:

1. *atomic operations*: atomically adds or subtract a value from some 32 bits integer variable.
2. *mutexes*: routines that operate on mutex (mutual exclusion) objects and are used to ensure exclusivity. Enables the programer to atomically 'lock' the mutex before accessing some shared structure and 'unlock' it when done.
3. *condition variables*: routines that operate on condition variables and have two main operations. When a thread calls the 'wait' operation on some condition it is suspended and waits on that condition variable signal until another thread signals (or broadcasts) the condition variable using the 'signal' operation.
4. *completion variables*: enables one thread to notify other threads that are waiting on the completion variable that the completion is true.
5. *reader/writer locks*: routines that enables a thread to lock some 32 bits word variable in memory using two types of locks. A 'read lock' is a non-exclusive mutex which allow multiply simultaneous readers. A 'writer lock' is a exclusive mutex which allows a single writer.

4.5.4 Practical examples using ordering and synchronization mechanisms

This section includes some practical examples showing how the storage-ordering and synchronization facilities of the Cell BE processor can be used. Most of those exampled are mainly based on chapter *Shared-Storage Ordering* in Cell Broadband Engine Programming Handbook document.

SPE writing notifications to PPE using fenced-option

The most common use of the fenced DMA is when writing back notifications. Example 4-49 illustrated such scenario:

Example 4-49 Writing notification using fenced-option

```
SPU program:
1. Compute some data.
2. Issue several DMA ‘put’ commands to writes the computed data back to
   main storage.
3. Use fenced ‘putf’ command to write a notification that the data is
   available. This notification might be to any type memory (main
   memory, I/O memory, a signal-notification or MMIO register, or
   another SPE’s mailbox).
4. If the program to reuse the LS buffers it waits for the completion
   of all the the commands issued in previous steps.
PPU program:
1. Wait for notification (poll the notification flag).
2. Operate on the computed SPE data (it is guaranteed that that the
   updated data is available in memory since SPE used fence between
   writing the computed data and the notification).
```

To ensure ordering of the DMA writing of the data (step 2) and of the notification (step 3) the notification may be sent using a fenced DMA command. This guarantee that the notification is not sent until all previous DMA commands of the group are issued.

In this example, the writing of both the data and that notification should have the same tag ID in order to guarantee that the fence will work.

Ordering reads followed by writes using barrier-option

A barrier option might be useful when a buffer read takes multiple commands and must be performed before writing the buffer, which also takes multiple commands. Example 4-50 illustrated such scenario:

Example 4-50 Ordering reads follows by writes using barrier-option

```
1. Issue several ‘get’ commands to read data into the LS.
2. Issue a single barrier ‘putb’ command to write data to main
   storage from LS. The barrier guarantee that the ‘putb’ command and
   the subsequent ‘put’ commands issued in step 3 will occur only after
   the `get` commands of step 1 are complete.
3. Issue a more ordinary (without barrier) ‘put’ command to    write
   data to main storage.
4. Waits for the completion of all the the command issued in
   previous steps.
```

Using the barrier-form for the first command to write the buffer (step 2) allows the commands used to put the buffer (step 2 to 3) to be queued without waiting for the completion of the ‘get’ commands (step 1). The programmer may take advantage of this mechanism to overlap those data transfers (read and write) with computation allowing the hardware to manage the ordering.

This scenario may occur on either the SPU or PPU who uses the MFC to initiate data transfers.

The ‘get’ and ‘put’ commands should have the same tag ID in order to guarantee that the barrier option (i.e. that comes with the ‘get’ and ‘put’ commands) will ensure writing the buffer just after data is read. If the ‘get’ and ‘put’ commands are issued using multiple tag IDs, then a MFC `barrier` command can be inserted between the ‘get’ and ‘put’ command instead of using a ‘put’ with barrier option for the first ‘put’ command.

If multiple commands are used to read and write the buffer, using the barrier option allows the read commands to be performed in any order and the write commands to be performed in any order, which provides better performance but forces all reads to finish before the writes start.

Double buffering using barrier-option

Barrier commands are also useful when performing double-buffered DMA transfers in which the data buffers used for the input data are the same as the output data buffers. Example 4-51 illustrated such scenario:

Example 4-51 Ordering SPU reads follows by writes using barrier-option

```
int i;
i = 0;
‘get’ buffer 0
while (more buffers) {
   ‘getb’ buffer i^1 //‘mfc_getb’ function (with barrier)
   wait for buffer i //‘mfc_write_tag_mask’ & ‘mfc_read_tag_status_all’
   compute buffer i
   ‘put’ buffer i    //‘mfc_put’ function
   i = i^1;
}
wait buffer i       //‘mfc_write_tag_mask’ & ‘mfc_read_tag_status_all’
compute buffer i
‘put’ buffer i      //‘mfc_put’ function
```

In the ‘put’ command at the end of each loop iteration data is written from the same local buffer to which data is later read in the beginning of next iteration’s ‘get’ command. It is critical there for to barrier the ‘get’ command to ensure that the writes complete before the reads are started preventing the wrong data to be written.

The code of SPU program which implements such double buffering mechanism is shown in Example 4-29 on page 159.

PPE-to-SPE communications using storage barrier instruction

For some applications the PPE is used as an application controller which manages and distributes work to the SPEs. Example 4-51 show a typical scenario for such applications and how a ‘sync’ storage barrier instruction may be used in this case to guarantee the correct ordering:

Example 4-52 Ordering SPU reads follows by writes using barrier-option

```
1. PPE write main storage with the data to be processed
2. PPE issue ‘sync’ storage barrier instruction.
3. PPE notifies the SPE by writing to either the inbound mailbox or one
   of the SPE’s signal-notification registers.
4. SPE read the notification and understands that data is ready.
5. SPE read the data and process it.
```

To make this feasible, it is important that the data storage performed in step 1 be visible to the SPE before receiving the work-request notification (steps 3 and 4). To ensure guaranteed ordering, a ‘sync’ storage barrier instruction must be issued by the PPE between the final data store in memory and the PPE write to the SPE mailbox or signal-notification register. This barrier instruction appears as step 2 in the example.

SPEs updating shared structures using atomic operation

In some case several SPEs may maintain a shared structure, for example when using the following programing model:

- A list of work elements in the memory defines the work that needs to be done. Each of the element defines one task out of the overall work that may be executed in parallel to the others.
- A shared structure contains the pointer to the next work element and potentially other shared information.
- An SPE that available to execute the next work element atomically reads the shared structure to evaluate the pointer to the next work element and update

it to point to the next element. Then he can get the relevant information and process it.

Atomic operations are useful in such cases when several SPE need to atomically read and update the value of the shared structure. Potentially, the PPE may also update this shared structure using atomic instructions on the PPU program.

Example 4-53 illustrated such scenario and how the SPEs can manage the work and access to the shared structure using the atomic operations:

Example 4-53 SPEs updating shared using atomic operation

```
// local version of the shared structure
// size of this structure is a single cache line
static volatile vector shared_var ls_var __attribute__ ((aligned
(128)));


// effective address of the shared sturture
uint64_t ea_var;

int main(unsigned long long spu_id, unsigned long long argv){

   unsigned int status;

   ea_var = get from PPE pointer to shared structure's effective addr.

   while (1){

      do {
         // get and lock the cache line of the shared sahred structure
         mfc_getllar((void*)&ls_var, ea_var, 0, 0);
         (void)mfc_read_atomic_status();

         if (valus in 'ls_var' indicate that the work was complete){

            (comment: 'ls_var' may contain total # of work tasks and #
                      of complete task - SPE can compare those values)
            break;
         }

         // else - we have a new work to do

         ls_var = progress the var to point to the next work to be done
```

```
            mfc_putllc((void*)&ls_var, ea_var, 0, 0);

            status = mfc_read_atomic_status() & MFC_PUTLLC_STATUS;

         } while (status); // loop till the atomic operation succeeds

         //get data of current work, process it, and put results in memory
      }

}
```

4.6 SPU programming

The eight SPEs are optimized for compute-intensive applications in which a program's data and instruction needs can be anticipated and transferred into the local store (LS) by DMA while the SPE computes using previously transferred data and instructions. However, the SPEs are not optimized for running programs that have significant branching, such as an operating system.

The following chapters are included in this section:

1. "Architecture overview and its impact on programming" on page 241 provide an overview on the main SPE architecture features and how they affect the SPU programing.
2. "SPU instruction set and C/C++ language extensions (intrinsics)" on page 244 provides an overview of the SPU instruction set and the SPU intrinsics which are simpler high level programing interface to access the SPE hardware mechanisms and assembly instructions.
3. "Compiler directives" on page 251 describes the compiler directive that are most likely to be use when writing an SPU program.
4. "SIMD programming" on page 253 discuss how the programer can explicitly exploit the SIMD instructions to the SPU.
5. "Auto-SIMDizing by compiler" on page 264 describe how the programer can use compilers to automatically convert a scalar code into a SIMD code.
6. "Using scalars and converting between different vector types" on page 271 described how to work with different vector data types and how to convert between vectors and scalars and vice versus.

7. “Code transfer using SPU code overlay” on page 276 describe how the programer can use the SDK3.0 SPU code overlay to face situations in which the code is too big to fit into the local store.
8. “Eliminating and predicting branches” on page 277 describes how write an efficient code when branches are required.

This chapter cover mainly issues related to writing a program that runs efficiently on the SPU while fetching instructions form the attached LS. However, in most cases an SPU program should also interact with the associated MFC to transfer data between the main storage and communicate with other processors on the Cell BE chip. It is therefore very important to understand those issues, which are covered in other chapters of the book:

- Chapter 4.3, “Data transfer” on page 109 discuss how SPU can transfer data between the LS and main storage.
- Chapter 4.4, “Inter-processor communication” on page 174 discuss how SPU communicate with other processors on the chip (PPE and SPEs).
- Chapter 4.5, “Shared storage synchronizing and data ordering” on page 213 discuss how the data transfer of the SPU and other processors are ordered and who the SPU can synchronize with other processors.

4.6.1 Architecture overview and its impact on programming

This chapter describes the main features of SPE architectures with emphasis on the impact that those features have on the programming of SPU applications.

The chapter is divided into sections such as that each section discuss specific component of the SPU architecture. Inside each section, we provide a list of the

Memory and data access

This chapter summarizes the main features related to memory and data access.

Local store (LS)

- From programing point of view this is the storage domain that the program directly refer to when doing load and store instruction or use pointers.
- Its size is 256KB. This size is relatively compact so usually the programer should explicitly transfer data between main memory and the LS.
- Holds both the instructions, stack, and data (global and local variables).
- Accessed directly by load and store instructions which are deterministic, have no address translation and have low latency.

- 16 byte per cycle load and store bandwidth and quadword aligned only. When the programer store data that is smaller then that (e.g. scalar) the program will actually perform 16 byte read, shuffle to set alignment, modify the data and 16 byte store. Obviously this is not very efficient.

Main storage and DMA

- Accessing the main storage is done by the programmer explicitly issuing DMA transfer between the LS and main storage.
- The effective address of the main storage data should be supplied by the program runs on the PPE.
- DMA transfer are done asynchronously with program execution allowing the programer to overlap between data transfer and computation.
- 16 bytes of data are transfered per cycle.

Register file

- Large register file of 128 entries of 128-bits each.
- Unified register file such as all types (floating point, integers, vectors, pointers, etc.) are stored in the same registers.
- The large and unified file allows for instruction-latency hiding using deep pipeline without speculation.
- Big-endian data ordering (lowest-address byte and lowest-numbered bit are the most-significant byte and bit, respectively).

LS arbitration

Arbitration to the LS is done according the following priorities (high first):

1. MMIO, DMA, and DMA list transfer element fetch.
2. ECC scrub.
3. SPU load/store; hint instruction prefetch.
4. Inline instruction prefetch.

Instruction set and instruction execution

This chapter summarizes the SPE instruction set, the way in which instructions are executed (pipeline and dual issue).

Instructions set:

- Supports the single-instruction, multiple-data (SIMD) instruction architecture that works on 128b vectors.
- Scalar instructions are also supported.

- The programer should try to use SIMD instruction as much as possible as the have significantly preferred performance. Can be done by using functions that are defined by SDK's language extensions for C/C++, or using auto-vectorization feature of the compiler.

Floating-point operations:

- Single-precision instructions are performed in 4-way SIMD fashion and are fully pipelined. Since those instructions have good performance it is recommended for the programer to try to use them if the application allows to.
- Double-precision instructions are performed in 4-way SIMD fashion, are only partially pipelined, and will stall dual issue of other instructions. The performance of these instructions makes Cell BE less attractive for applications that have massive use of double-precision instructions.
- Data format follows the IEEE Standard 754 definition, but the single precision results are not fully compliant with this standard (different overflow and underflow behavior, support only for truncation rounding mode, different denormal results).
  The programer should be aware that in come cases the computation results will not be identical to IEEE Standard 754

Branches:

- No branch prediction cache, branches assume to be not taken so in case a branch s taken there a stall that have negative effect on the performance.
- Special branch hint commands can be used in the code to direct the hardware that a coming branch will be taken and by that avoid the stall.
- There are no hint intrinsics. Instead programmers can improve branch prediction by either utilizing the __builtin_expect compiler directive or utilize feedback directed optimization supported by the IBM xl compilers or FDPRPro.

Pipeline and dual issue:

- Has two pipelines, named even (pipeline 0) and odd (pipeline 1). Whether an instruction goes to the even or odd pipeline depends on its instruction type.
- Issue and complete up to two instructions per cycle, one in each pipeline.
- Dual-issue occurs when a fetch group has two issueable instructions with no dependencies in which the first instruction can be executed on the even pipeline and the second instruction can be executed on the odd pipeline.
- Advances programers can write low level code that fully utilize the two pipelines by separating between instructions that goes to the same pipeline (i.e. put instruction that goes to the other pipeline between them) or separating between instructions that have data dependencies.

- However, in many cases the programer may relies on the compiler or other performance tools (e.g. FDPRPro) to utilize the two pipelines. However, it is recommended to analyze the results either statically (e.g. using SPU static timing tool or Code Analyzer tool) or using profiling (e.g. load FDPRPro profiming data into Code Analyzer tool).

Features which are not supported by the SPU

The SPU doesn't support many of the features provided in most general purpose processors:

- No direct (SPU-program addressable) access to main storage. The SPU accesses main storage only by using the MFC's DMA transfers.
- No direct access to system control, such as page-table entries. PPE privileged software provides the SPU with the address-translation information that its MFC needs.
- With respect to accesses by its SPU, the LS is unprotected and un-translated storage.

4.6.2 SPU instruction set and C/C++ language extensions (intrinsics)

The SPU Instruction Set Architecture (*ISA*) is fully documented in Synergistic Processor Unit Instruction Set Architecture document. SPU ISA operates primarily on SIMD vector operands, both fixed-point and floating-point, with support for some scalar operands.

Another recommended source of information is *SPU Instruction Set and Intrinsics* chapter in Cell Broadband Engine Programming Handbook document which its appendix provides a table of all the supported instructions as well as their latency.

SDK provided rich set of language extensions for C/C++ which define SIMD data types and intrinsics that map to one or more assembly-language instructions into C language functions. This gives the programmer very convenient and productive control over code performance without the need for assembly-language programming.

From the programer point of view it is generally highly recommended:

- Use SIMD operations where ever possible as they provide the maximum performance which can be up to 4 times (for single precision float or 32b integers) or 16 times (for 8 bit chars) faster then scalar processor.
  This important topic is further discussed at Chapter 4.6.4, "SIMD programming" on page 253.

- Can use any scalar operations on the C/C++ code and the compiler will take care of mapping them to one or more SIMD operations (e.g. read-modify-write) if the appropriate scalar assembly instruction does not exist. The programer should try to minimize those operations (e.g. to use them only for control) since their performance is not as good as the SIMD ones or in some cases not as good as executing similar commands on ordinary scalar machine.

SPU Instruction Set Architecture (ISA)

The SPU Instruction Set Architecture (*ISA*) operates primarily on SIMD 128 bits vector operands, both fixed-point and floating-point. the architecture have support for some scalar operands.

There are 204 instructions in the ISA and they are grouped into several classes according to their functionality. Most of the instructions are mapped into either generic intrinsics or specific intrinsics that may be called as C functions from the program.Full description of the instructions set is in the Synergistic Processor Unit Instruction Set Architecture document.

ISA provides a reach set of SIMD operations that can be performed on 128 bits vectors of several fixed point or floating point elements. Instructions are also available to access any of the MFC channels in order to initiate DMA transfers or communicate with other processors.

The following chapters provide additional information on some of the main types of instructions.

Memory access SIMD operations

Load and store instructions are performed on the LS memory and uses 32 bits LS address. The instruction operates on 16 bytes elements which are quadword aligned. The SPU can perform a one such instruction in every cycle and their latency is about 6 cycles.

Channels access

A set of instructions are provided in order to access the MFC channels. Those instructions can be used to initiate DMA data transfer, communicate with other processors, access the SPE decrementer and more. The SPU interface with the MFC channel is further described in the prefix of Chapter 4.6, “SPU programming” on page 240

SIMD operations

ISA SIMD instructions provides a reach set of operations (logical, arithmetical, casting, load and store, etc.) that can be performed on 128 bits vectors of either fixed point or floating point values. The vectors can contain various sizes of

variables - 8, 16, 32 or 64 bits. The performance of the program can be significantly effected by the way the SIMD instructions are used. For example, using SIMD instructions on 32 bits variables (single precision floating point or 32 bits integer) can speed up the program by at least four times compare to equivalent scalar program since in every cycle the instruction works on four different elements in parallel (since there are four 32 bits variables for one 128 vector).

Figure 4-5 shows one example of such SPU add SIMD instruction of four 32 bits elements vector. This instruction simultaneously adds four pairs of floating-point vector elements, stored in registers VA and VB, and produces four floating-point results, written to register VC.

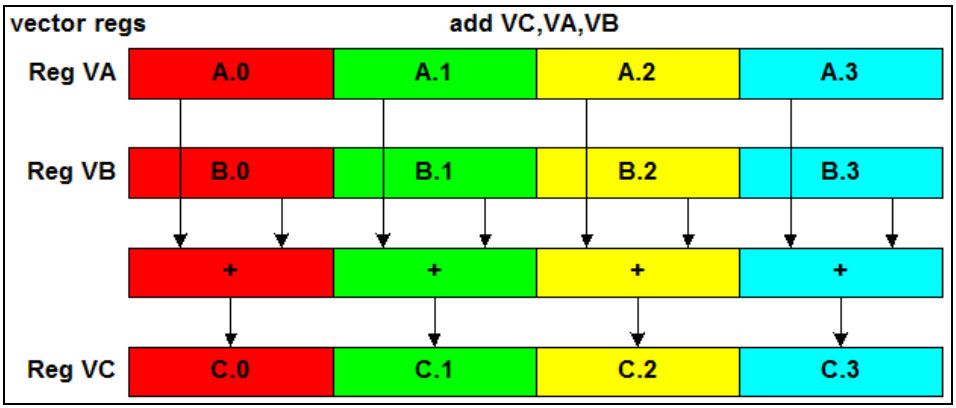


Figure 4-5 SIMD add instruction

Scalar related instructions

ISA also provides instructions to access scalars. A set of store assist instructions is available in order to help store bytes, halfwords, words, and doublewords in the 128-bit vector registers. Similarly, instructions are provided in order to extract such scalar from the vector registers. Rotate instructions are also available and can be used to move data into the appropriate locations in the vector.

Those instructions may be used by the programer whenever there is a need to operate on specific element from a given vector (e.g. summarize the elements of one vector).

In addition, those instructions are often used by the compiler. Whenever the high level C/C++ function operated on scalars, the compiler translate it into a set of 16 bytes read, modify, and 16 bytes write operations. In this process, the compiler use the store assist and extract instruction to access the appropriate scalar element.

ISA provides some instructions that use or produce scalar operands or addresses. In this case, the values are set into in the preferred slot in the 128-bit vector registers as illustrated in Figure 4-6. The compiler may use the scalar store assist and extract instructions when a non aligned scalar are used in order to shift it into the preferred slot.

In order to eliminate the need for such shift operations, the programer may explicitly define the alignment of frequently used scalar variables so they will be located in the preferred slot. The compiler optimization and after link optimization tools that comes with the SDK (e.g. FDPRPro) will also try to help in this process by statically align scalar variables into the preferred slot.

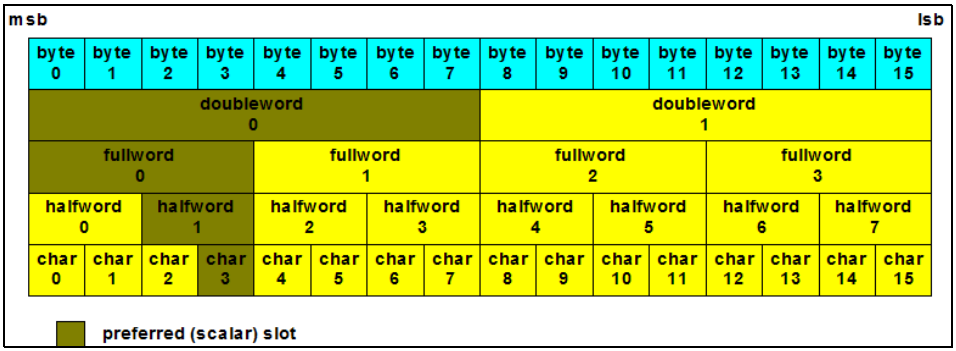


Figure 4-6 Scalar Overlay on SIMD in SPE

SIMD "cross-element" shuffle instructions.

ISA provides a set of shuffle instructions for reorganizing data in given vector which are very useful in SIMD programming. In one instruction the programmer can reorder all the vector elements into an output vector. Other less efficient alternative to do so to perform a series of several scalar based instructions for extracting the scalar from a vector, and store in the appropriate location in a vector.

Figure 4-7 shows an example instruction. Bytes are selected from vectors VA and VB based on byte entries in control vector VC. Control vector entries are indices of bytes in the 32-byte concatenation of VA and VB. While the shuffle operation is purely byte oriented it can also be applied to more than byte vectors (e.g. vectors of floating points or 32 bits integers).

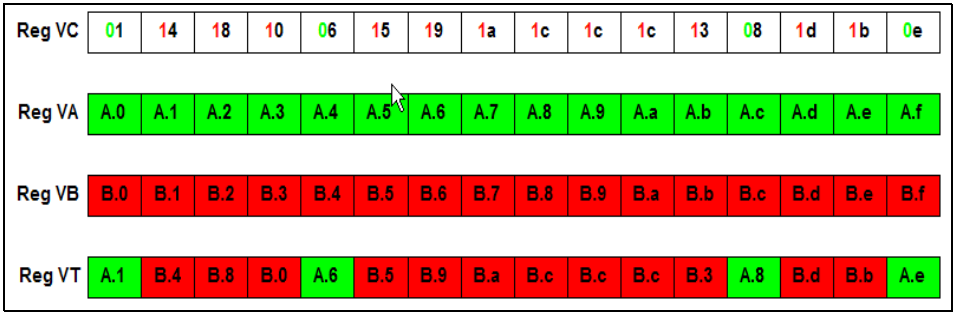


Figure 4-7 Shuffle/Permute example: shufb VT,VA,VB,VC instruction

SPU C/C++ language extensions (intrinsics)

A large set of SPU C/C++ language extensions (intrinsics) make the underlying SPU Instruction Set Architecture and hardware features conveniently available to C programmers

- Intrinsics are essentially in-line assembly-language instructions in the form of C-language function calls.
- Intrinsics can be used in place of assembly-language code when writing in the C or C++ languages.
- A single intrinsics map one or more assembly-language instructions.
- Intrinsics provide the programmer with explicit control of the SPE SIMD instructions without directly managing registers.
- Intrinsics provide the programmer access to all MFC channels as well as other system registers (e.g. decrementer, SPU state save/restore register).

Full description of those extensions is in C/C++ Language Extensions for Cell BE Architecture V2.4 document.

Note: The SPU intrinsics are defined in `spu_intrinsics.h` system header file which should be included in case the programer wish to use them.
The directory in which this file is located varies depends on which compiler is used: `/usr/lib/gcc/spu/4.1.1/include/` when using GCC
`/opt/ibmcmp/xlc/cbe/9.0/include/` when using XLC

The SDK compiler supports these intrinsics will emit efficient code for the SPE architecture, similar to using the original assembly instructions. The techniques used by compilers to generate efficient code include register coloring, instruction scheduling (dual-issue optimization), loop unrolling and auto vectorization, up-stream placement of branch hints and more.

For example, an SPU compiler provides the intrinsic `t=spu_add(a,b)` as a substitute for the assembly-language instruction `fa rt,ra,rb`. The compiler will generate a floating-point add instruction `fa rt, ra, rb` for the SPU intrinsic `t=spu_add(a,b)`, assuming t , a , and b are vector float variables.

The PPU and the SPU instruction sets have similar, but distinct, SIMD intrinsics. It is important to understand the mapping between the PPU and SPU SIMD intrinsics when developing applications on the PPE that will eventually be ported to the SPEs. Chapter 4.1.1, “PPE architecture and PPU programming” on page 79 discuss this issue.

Intrinsics data types

Many of the intrinsics can accept parameters from different types but the intrinsic name remain the same. For example, `spu_add` function can add two `signed int` vectors into one output `signed int` vector, or add two `float` vectors (single precision) into one output `float` vector, and few other types of vectors.

The translation from function to instruction dependent on datatype of arguments. For example, spu_add(a,b) can translate to a floating add or a signed int add depends on the input parameters.

Some operations cannot be performed on all data types, for example multiply using `spu_mul` can be performed only on floating point data types. A detailed information about all the intrinsics include the data type that is supported but each of them, is in C/C++ Language Extensions for Cell BE Architecture .

Note: It is recommended for the programer to be familiar with this issue early in the development stage while defining the program’s data types in order to prevent unpleasant surprises during the later development of the algorithm, in case some crucial operation is not supported on the chosen data types.

Intrinsics classes

SPU intrinsics are grouped into the three classes:

- *Specific Intrinsics*: intrinsics that have a one-to-one mapping with a single assembly-language instruction and are provided for all instructions except some branch and interrupt related ones. All specific intrinsics are named using the SPU assembly instruction prefixed by the string, `si_` (e.g. the specific intrinsic that implements the ‘stop’ assembly instruction is `si_stop`). Programmers rarely need these intrinsics since all of them are mapped into generic (see next bullet) which are more convenient.
- *Generic and Builtin intrinsics*: intrinsics that map to one or more assembly-language instructions as a function of the type of input parameters and are often implemented as compiler built-ins. Intrinsics of this group are

very useful and covers almost all the assembly-language instructions including the SIMD ones. Instructions who are not covered are naturally accessible through the C/C++ language semantics.
All of the generic intrinsics are prefixed by the string `spu_`. For example, the intrinsic that implements the 'stop' assembly instruction is named `spu_stop`.

- *Composite and MFC related intrinsics* — Convenience intrinsics constructed from a sequence of specific or generic intrinsics. Those intrinsics are further discussed in other chapters in the document that discuss DMA data transfer and inter-processor communication using the MFC.

Intrinsics: functional types

SPU generic intrinsics, which construct the main class of intrinsics, are grouped into the several types according to their functionality:

- *Constant formation* (example: `spu_splats`): replicate a single scalar value across all elements of a vector of the same type.
- *Conversion* (example: `spu_convtf, spu_convts`): convert from one type of vector to another. Using those intrinsics is the correct approach to do cast between two vectors of different types.
- *Scalar* (example: `spu_insert, spu_extract, spu_promote`): allow programmers to efficiently coerce scalars to vectors, or vectors to scalars which enables to easily perform operations between vectors and scalars.
- *Shift and rotate* (example: `spu_rlqwbyte, spu_rlqw`): shift and rotate the elements within a single vector.
- *Arithmetic* (example: `spu_add, spu_madd, spu_nmadd`) : perform arithmetic operation on all the elements of the given vectors.
- *Logical* (example: `spu_and, spu_or`) : logical operation on the entire vectors.
- *Byte operations* (example: `spu_absd, spu_avg`): operations between bytes of the same vector.
- *Compare, branch and halt* (example: `spu_cmpeq, spu_cmpgt`): different operations to control the flow of the program.
- *Bits and masks* (example: `spu_shuffle, spu_sel`): bitwise operation like counting the number of bits equal '1' or the number of leading zeros.
- *Control* (example: `spu_stop, spu_ienable, spu_idisable`) - several control operation such as stop and signal the PPE and controlling the interrupts.
- *Channel Control* (example: `spu_readch, spu_writech`) - read from and write to MFC's channels.
- *Synchronization and Ordering* (example: `spu_dsync`) - synchronize and order data transfer as related to external components.

In addition, the composite intrinsics contain intrinsics (`spu_mfcdma32`, `spu_mfcdma64`, `spu_mfcstat`) that enable to issue DMA commands to the MFC and check their status.

In the next chapter we discuss some of the more useful intrinsics.

A list that summarizes all the SPU intrinsics is presented in table 18 in the Cell Broadband Engine Programming Tutorial document.

4.6.3 Compiler directives

Like compiler intrinsics, *compiler directives* are crucial programming elements. In this chapter we summarize some of the more important ones for SPU programming.

aligned attribute

The `aligned` attribute is very important in Cell BE programming and is used to ensure proper alignment of variables in the program.

There are two main cases where this attributes may be used:

- To ensure proper alignment of the DMA source or destination buffer. A 16 bytes alignment is mandatory for data transfer of more then 16 bytes while 128 bytes alignment is optional but provides better performance.
- To ensure proper alignment of the scalar. Whenever a scalar is often used it is recommended to align it with the preferred slot in order to save shuffle operations while it is read or modified.

The syntax of this attribute for the SDK gcc and xlc implementations to align a variable into quadword (16 bytes) is:

```
float factor __attribute__((aligned (16)));
```

Please note that the compilers currently do not support alignment of automatic (stack) variables to an alignment that is stricter then the alignment of the stack itself (16 bytes).

volatile keyword

The `volatile` keyword can be set when some variable is defined. It instructs the compiler that this variable may be changed for some reason that is not related to the program execution itself (i.e. program instructions). This prevent the compiler from doing optimizations that assumes that the memory does not change unless a store instruction wrote new data to it.

Such scenario happens when some hardware component besides the processor itself may modify this variable.

For a Cell BE program (either SPU or PPE) it is recommended to define buffers that are written by the DMA as volatile, for example a buffer on the LS to which a ‘get’ command write data. Doing so ensures that buffers are not accessed by SPU load or store instructions until after DMA transfers have completed.

The syntax of this keyword for the SDK is shown as follows:

```
volatile float factor;
```

__builtin_expect directive

Since branch mispredicts are relatively expensive, `__builtin_expect` provides a way for the programmer to direct branch prediction. This example:

```
int __builtin_expect(int exp, int value)
```

returns the result of evaluating `exp` , and means that the programmer expects `exp` to equal value. The value can be a constant for compile-time prediction, or a variable used for run-time prediction.

Using this directive is further discussed, including some useful code examples in Chapter , “Branch hint” on page 281.

_align_hint directive

The `_align_hint` directive helps compilers “auto-vectorize”. Although it looks like an intrinsic, it is more properly described as a compiler directive, since no code is generated as a result of using the directive. The example:

```
_align_hint(ptr, base, offset)
```

informs the compiler that the pointer `ptr` points to data with a base alignment of `base`, with a byte offset from the base alignment of `offset`. The base alignment must be a power of two. Giving 0 as the base alignment implies that the pointer has no known alignment. The offset must be less than the base, or, zero. The `_align_hint` directive should not be used with pointers that are not naturally aligned.

restrict qualifier

The `restrict` qualifier is well-known in many C/C++ implementations, and it is part of the SPU language extension. When the `restrict` keyword is used to qualify a pointer, it specifies that all accesses to the object pointed to are done through the pointer. For example:

```
void *memcpy(void * restrict s1, void * restrict s2, size_t n);
```

By specifying s1 and s2 as pointers that are restricted, the programmer is specifying that the source and destination objects (for the memory copy) do not overlap.

4.6.4 SIMD programming

This chapter discuss how to write SIMD operation based programs to be run on the SPU and in specific how the programer can convert code that is based on scalar data types and operations to a code that is based on vector data types and SIMD operations.

algorithm vectorized by the programmer

- “Vector data types” discuss the vector data types that are supported for SIMD operation.
- “SIMD operations” discuss which SIMD operation are supported in the different libraries and how to use them
- “Loop unrolling for converting scalar data to SIMD data” discuss the main technique for converting a scalar code to a SIMD one by unrolling long loops.
- “Data organization - AOS versus SOA” discuss the two main data organization methods for SIMD programming and also how a scalar code may be converted to SIMD using the more common data organization method among the two (SOA).

Another alternative to covert scalar code into SIMD code is to let the compiler perform automatically conversion of the code. This approach is called auto-SIMDizing and is further discussed in Chapter 4.6.5, “Auto-SIMDizing by compiler” on page 264.

Vector data types

SPU SIMD programming operates on vectors data types. Following are few of the main attributes of those data types:

- 128 bits (16B) long.
- Aligned on quadword (16B) boundaries.
- Different data type are supported: fixed point (e.g. char, short, int, signed or unsigned) and floating point (e.g. float and double).
- Contain from 1 to 16 elements per vector depends on the corresponding type.
- Stored in memory similar to array of the corresponding data types (e.g. vector of integer is like array of four 32b integers).

In order to use the data types the programer should include `spu_intrinsics.h` header file.

As general, the vector data types shared a lot in common with ordinary C language scalar data types:

- Pointers to vector types can be defined and so are operations on those pointers. For example, in case the pointer `vector float *p` is defined then `p+1` points to the next vector (16B) after that pointed to by `p`.
- Arrays of vectors can be define and so as operations on those arrays. For example, in case the array `vector float p[10]` is defined then `p[3]` is the third variable in this array.

The vector data types can be used in two different formats:

- Full names which are combination of the data type of the elements that this vector consist of, together with vector prefix (e.g. `vector signed int`).
- Single token typedefs (e.g. `vec_int4`) which are more recommended since they are shorter and are also compatible with using the same code for PPE SIMD programming.

Table 4-20 summarizes the different data types that are supported by the SPU including both the full and the corresponding single token typedefs.

Table 4-20 Vector data types

Vector data type	Single-Token Typedef	Content
vector unsigned char	vec_uchar16	Sixteen 8-bit unsigned chars
vector signed char	vec_char16	Sixteen 8-bit signed chars
vector unsigned short	vec_ushort8	Eight 16-bit unsigned halfwords
vector signed short	vec_short8	Eight 16-bit signed halfwords
vector unsigned int	vec_uint4	Four 32-bit unsigned words
vector signed int	vec_int4	Four 32-bit signed words
vector unsigned long long	vec_ullong2	Two 64-bit unsigned doublewords
vector signed long long	vec_llong2	Two 64-bit signed doublewords
vector float	vec_float4	Four 32-bit single-precision floats
vector double	vec_double2	Two 64-bit double precision floats
qword	-	quadword (16-byte)

SIMD operations

This chapter discuss how the programer may perform SIMD operations on an SPU program vectors. There are four main options to perform SIMD operations as discuss in the following four chapters:

1. “SIMD arithmetic and logical operators” - SDK3.0 compilers support a vector version of some of the common arithmetic and logical operators. Those operator work on each element of the vector.
2. “SIMD low level intrinsics” - high level C functions which support almost all the SPU’s SIMD assembler instructions. Those intrinsics contains basic logical and arithmetic operations between 128 bits vectors from different types, and also some operations between elements of a single vector.
3. “SIMDmath library” - extend the low level intrinsic and provides functions that implement more complex mathematical operations (e.g. root square and trigonometric operations) on 128 bit vectors.
4. “MASS and MASSV libraries”- MASS library provided similar functions as SIMDmath library but optimized to have better performance in the price of having redundant accuracy. MASSV perform similar operations on longer vectors who has any multiple of 4 length.

SIMD arithmetic and logical operators

SDK compilers support a vector version of some of the common arithmetic and logical operators. This is the easiest way to program SIMD operations as the syntax is identical to programing with scalar variables. When those operators are applied on vector variables, the compiler translate it to operators that work separately on each element of the vectors.

While the compilers support some basic arithmetic, logical and rational operators, not all the existing operators are currently supported. In case the required operator is not supported, the programer should use the other alternatives that are described in the following chapters.

The operator that are supported are:

- Vector subscripting: [ ]
- Unary operators: ++, --, +, -, ~
- Binary operators: +, -, *, /, unary minus, %, &, |, ^, <<, >>
- Relational Operators: ==, !=, <, >, <=, >=

More details about this subject are in the *Operator Overloading for Vector Data Types* chapter in C/C++ Language Extensions for Cell BE Architecture document.

Example 4-54 shows a simple code that uses some SIMD operators.

Example 4-54 Simple SIMD operator code

```
#include <spu_intrinsics.h>

vector float vec1={8.0,8.0,8.0,8.0}, vec2={2.0,4.0,8.0,16.0};

vec1 = vec1 + vec2;
vec1 = -vec1;
```

SIMD low level intrinsics

SDK 3.0 provides a reach set of low-level specific and generic intrinsics which support the SIMD instructions that are supported by the SPU assembler instruction set (e.g. `c=spu_add(a,b)` intrinsic stands for `add vc,va,vb` instruction). Those are C level functions that are implemented either internally within the compiler or as macros.

The intrinsics are grouped into several types according to their functionality, as described in Chapter , “Intrinsics: functional types” on page 250. The three groups which contains the most significant SIMD operations are:

- Arithmetic intrinsics which perform arithmetic operation on all the elements of the given vectors (e.g. `spu_add, spu_madd, spu_nmadd`, ...)
- Logical intrinsics which perform logical operation on all the elements of the given vectors (`spu_and, spu_or`, ...).
- Byte operations which perform operations between bytes of the same vector (e.g. `spu_absd, spu_avg`,...).

The intrinsics support different data types and it is up to the compiler to translate the intrinsics to the correct assembly instruction depends on the type of the intrinsic operands.

In order to use those the SIMD intrinsics the programer should include the `spu_intrinsics.h` header file.

Example 4-55 shows a simple code that uses low level SIMD intrinsics.

Example 4-55 Simple SIMD intrinsics code

```
#include <spu_intrinsics.h>

vector float vec1={8.0,8.0,8.0,8.0}, vec2={2.0,4.0,8.0,16.0};
```

```
vec1 = spu_sub( (vector float)spu_splats((float)3.5), vec1);
vec1 = spu_mul( vec1, vec2);
```

SIMDmath library

While SIMD intrinsics contains various basic mathematical functions that are implemented by corresponding SIMD assembly instructions, more complex mathematical functions are not supported by those intrinsics. The SIMDmath library is provided with SDK3.0 and address this issue by providing a set of functions that extend the SIMD intrinsics and support additional common mathematical functions. The library, like the SIMD intrinsics, operates on short 128 bits vectors from different types (e.g. single precision float, 32 bit integer) are supported. It depends on the specific function which vector types are supported.

The SIMDmath library provide functions for the following categories:

1. Absolute value and sign functions: remove or extract the signs from values.
2. Classification and comparison functions: return boolean values from comparison or classification of elements.
3. Divide, multiply, modulus, remainder and reciprocal functions: standard arithmetic operations.
4. Exponentiation, root, and logarithmic functions: functions related to exponentiation or the inverse.
5. Gamma and error functions: probability functions.
6. Minimum and maximum functions: return the larger, smaller or absolute difference between elements.
7. Rounding and next functions: convert floating point values to integers.
8. Trigonometric functions: sin, cos, tan and their inverses.
9. Hyperbolic functions: sinh, cosh, tanh and their inverses.

The SIMDmath library is an implementation of most of the C99 math library (-lm) that operates on short SIMD vectors. The library functions conform as closely as possible to the specifications set out by the scalar standards. However, fundamental differences between scalar architectures and the Cell BE architecture require some deviations, including the handling of rounding, error conditions, floating-point exceptions and special operands such as NaN and infinities.

The SIMDmath library can be used in two different versions:

- *linkable library archive* - a static library that contains all the library functions. Using this version is more convenient to code since it only requires the

inclusion of a single header file, but it produces slower, and potentially larger binaries (depends on the frequency of invocation) due to the branching instructions necessary for function calls. The function calls also reduces the number of instructions available for scheduling and leveraging the large SPE register file.

- *set of inline function headers* - a set of standalone inline functions. This version require extra header files to be included for each math function used, but produce faster and smaller (unless inlined multiple times) binaries, because the compiler is able to reduce branching and often achieves better dual-issue rates and optimization.
  The functions names are prefixed with an underscore character '_' compare to the linkable library format (e.g. inline version of `fabsf4` is `_fabsf4`).

To use the SIMDmath library the programer should do the following:

- For the linkable library archive version, include the primary header file `/usr/spu/include/simdmath.h`
- For the linkable library archive version, link the SPU application with the /usr/spu/lib/`libsimdmath.a` library.
- For the inline functions version include a distinct header file for each function used. Those header files are in `/usr/spu/include/simdmath` directory. For example, add `#include <simdmath/fabsf4.h>` to use `_fabsf4` inline function.
- In addition, some classification functions require inclusion of `math.h` file.

Additional information about this library exist in the following:

- Code example and additional usage instruction is in Chapter 8, "Case study: Monte Carlo Simulation" on page 493.
- Function calls format is in SIMDmath Library API Reference document.
- Function specification is in SIMD Math Library Specification.

MASS and MASSV libraries

This chapter discuss two libraries that are part of SDK3.0 and implement various SIMD functions:

- MASS (mathematical acceleration subsystem) library: functions which operates on short 128 bits vectors. The interface of those functions is similar to SIMDmath library that is described in Chapter , "SIMDmath library" on page 257.
- MASSV (MASS vector) library: functions which can operate on longer vectors. Vector length can be any number which is multiple of 4.

Similar to the SIMDmath library, the MASS libraries can be used in two different versions - linkable library archive version, and inline functions version.

However, the implementation of the MASS and MASSV libraries are different from SIMDmath library on the following aspects:

- SIMDmath is focused on accuracy while MASS and MASSV are focused on having better performance. For a performance comparison between the two see “Performance Information for the MASS Libraries for CBE SPU” document.
- SIMDmath has support across the entire input domain while MASS and MASSV may restrict the input domain.
- MASS and MASSV library support a subset of the SIMDmath library functions
- MASSV library can work on long vectors whose length is any number which is multiple of 4.
- The functions of MASSV library have similar names as SIMDmath and MASS functions but with “vs” prefix.

In order to use those the MASS library the programer should do the following:

- For both versions above, include the `mass_simd.h` and `simdmath.h` header files in `/usr/spu/include/` directory in order to use the MASS functions, and include `massv.h` header files for MASSV functions.
- For both versions above, link the SPU application with the libmass_simd`.a` header file for MASS functions and with `libmassv.a` file for MASSV functions. Both files are in `/usr/spu/lib/` directory.
- In addition, for the inline functions version include a distinct header file for each function used. Those header files are in `/usr/spu/include/mass` directory. For example, include `acosf4.h` header file to use acosf4 inline function.

Additional information about this library exist in the following:

- Function call format and brief description is in “MASS C/C++ function prototypes for CBE SPU” document.
- Usage instructions is in “Using the MASS libraries on CBE SPU” document.

Loop unrolling for converting scalar data to SIMD data

This chapter discuss the loop unrolling programming technique which is one of the most common methods for practicing SIMD programming.

The process of loop unrolling related to SIMD programming involves:

- The programer expand a loop such as each new iteration contains several of what used to be an old iteration (before the unrolling).

- Few operations on scalar variables on the old iteration are joined to a single SIMD operation on a vector variable in the new iteration. The vector variable contains several of the original scalar variables.

The loop unrolling can provide significant performance improvement when applied on relatively long loops in an SPU program. The improvement is approximately in the factor which is equal to the number of elements in a unrolled vector (e.g. unrolling a loop that operated on single precision float provide four time speedup since four such 32b float exists in the 128b vector).

Example 4-56 shows a code that practice the loop unrolling technique. The code contains two version of multiply between two inout array of float. The first version is an ordinary scalar version (`mult_` function) and the second is loop-unrolled SIMD version (`vmult_` function).

Source code: The code of Example 4-56 is included in the additional material that is provided with this book. See “SPE loop unrolling” on page 616 for more information.

Please notice that in this example we requires that the arrays are quadword aligned and the array length is divisible by 4 (stands for 4 float elements in a vector of floats).

Two general comments regarding the *alignment and length of the vectors*:

- We insure that the quadword alignment using the `aligned` attribute which is recommended in most cases. If this is not the case a scalar prefix may be added to the unrolled loop to handle the first not aligned elements.
- It is recommended to try to work with arrays whose length are divisible by 4. If this is not the case, a suffix may be added to the unrolled loop to handle the last elements.

Example 4-56 SIMD loop unrolling

```
#include <spu_intrinsics.h>

#define NN 100

// multiply - scalar version
void mult_(float *in1, float *in2, float *out, int N){
   int i;
   for (i=0; i<N; i++){
      out[i] = in1[i] * in2[i];
   }
}
```

```
// multiply - SIMD loop-unrolled version
// assume the arrays are quadword aligned and N is divisible by 4
void vmult_(float *in1, float *in2, float *out, int N){
   int i, Nv;
   Nv = N>>2; //divide by 4;

   vec_float4 *vin1 = (vec_float4*)in1, *vin2 = (vec_float4*)in2;
   vec_float4 *vout = (vec_float4*)out;

   for (i=0; i<Nv; i++){
      vout[i] = spu_mul( vin1[i], vin2[i]);
   }
}

int main( )
{
   float in1[NN] __attribute__((aligned (16)));
   float in2[NN] __attribute__((aligned (16)));
   float out[NN];
   float vout[NN] __attribute__((aligned (16)));

   // init in1 and in2 vectors

   // scalar multiply ‘in1’ and ‘in2’ into ‘out’ array
   mult_(in1, in2, out, (int)NN);

   // SIMD multiply ‘in1’ and ‘in2’ into ‘vout’ array
   vmult_(in1, in2, vout, (int)NN);

   return 0;
}
```

Data organization - AOS versus SOA

This section discusses the two main data organization methods for SIMD programming and also how a scalar code may be converted to SIMD using the more common data organization method among the two (SOA).

Depending on the programmer’s performance requirements and code size restraints, advantages can be gained by properly grouping data in an SIMD vector. There are two main methods to organize the data as presented below.

The first method organizing data in SIMD vectors is called an *array of structures* (*AOS*) as demonstrated in Figure 4-8. This figure shows a natural way of using SIMD vectors to store the homogenous data values (x, y, z, w) for the three vertices (a, b, c) of a triangle in a 3D-graphics application. This organization has the name array of structures because the data values for each vertex are organized in a single structure, and the set of all such structures (vertices) is an array.

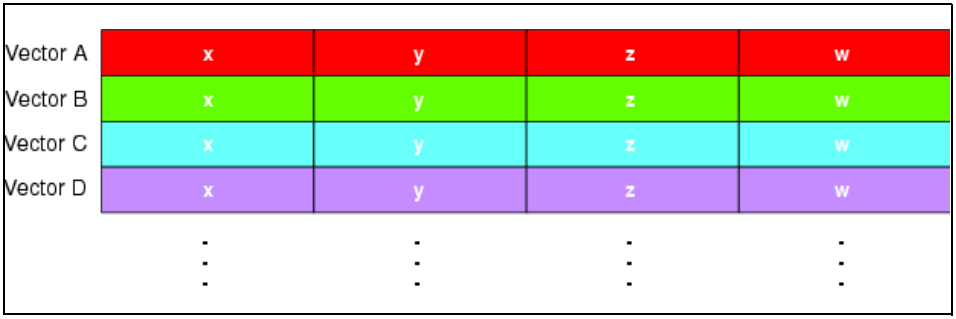


Figure 4-8 AOS (array of structures) organization

The second method is a *structure of arrays* (*SOA*) as demonstrated in Figure 4-9 which shows such SOA organization to represent the x, y, z vertices for four triangles. Not only are the data types the same across the vector, but now their data interpretation is the same. Each corresponding data value for each vertex is stored in a corresponding location in a set of vectors. This is different from the AOS case, where the four values of each vertex are stored in one vector.

Figure 4-9 SOA (structure of arrays) organization

The AOS data-packing approach often produces small code sizes, but it typically executes poorly and generally requires significant loop-unrolling to improve its efficiency. If the vertices contain fewer components than the SIMD vector can hold (for example, three components instead of four), SIMD efficiencies are compromised.

On the other hand, when using SOA it is usually very easy to perform loop unrolling or other SIMD programing on. The programer can think of the data as if

it were scalar, and the vectors are populated with independent data across the vector. The structure of a unrolled loop iteration should be similar to the scalar case but with one main difference of simply replacing the scalar operations with identical vectors operation the work simultaneously on few elements which are gathered in one element.

Example 4-57 illustrate a process of taking a scalar loop in which the elements are stored in AOS organization and the equivalent unrolled SOA based loop which has 4 times less iterations. Please notice that the scalar and the unrolled SOA loop are very similar and uses the same '+' operators. The only difference is how the indexing to the data structure is performed.

Source code: The code of Example 4-57 is included in the additional material that is provided with this book. See "SPE SOA loop unrolling" on page 616 for more information.

Example 4-57 SOA loop unrolling

```
#define NN 20

typedef struct{ // AOS data structure - stores one element
   float x;
   float y;
   float z;
   float w;
} vertices;

typedef struct{ // SOA structure - stores entire array
   vec_float4 x[NN];
   vec_float4 y[NN];
   vec_float4 z[NN];
   vec_float4 w[NN];
} vvertices;

int main( )
{
   int i, Nv=NN>>2;

   vertices  vers[NN*4];
   vvertices vvers __attribute__((aligned (16)));

   // init x, y, and z elements

   // original scalar loop - work on AOS which is difficult to SIMDized
```

```
   for (i=0; i<NN; i++){
      vers[i].w = vers[i].x + vers[i].y;
      vers[i].w = vers[i].w + vers[i].z;
   }

   // SOA unrolled SIMDized loop
   for (i=0; i<Nv; i++){
      vvers.w[i] = vvers.x[i] + vvers.y[i];
      vvers.w[i] = vvers.w[i] + vvers.z[i];
   }
   return 0;
}
```

The subject of different SIMD data organization is further discussed in *Converting Scalar Data to SIMD Data* chapter in Cell Broadband Engine Programming Handbook document.

4.6.5 Auto-SIMDizing by compiler

This chapter discuss the auto-SIMDizing support by Cell Be's GCC and XLC compilers. auto-SIMDizing is the process in which a compiler automatically merges scalar data into a parallel-packed SIMD data structure. The compiler perform this process by first identifies parallel operations in the scalar code, such as loops. The compiler then generates SIMD versions of them, for example by automatically performing loop unrolling. During this process, the compiler performs all analysis and transformations necessary to fulfill alignment constraint.

From the programer point of view, it means that in some cases there is not need to perform explicit translation of scalar code into a SIMD one as described in chapter "SIMD programming". Instead, the programer may write ordinary scalar code and instruct the compiler to perform auto-SIMDizing and translated the high level scalar code into SIMD data structures and SIMD assembly instructions.

However, at this point, there are limitations on the compilers' capabilities in translating a certain scalar code to a SIMD one and not any scalar code that theoretically can be translated into a SIMD will eventually be translated by the compilers.

Hence, a programmer knowledge of the compiler limitations is required, and which will enable the programer to choose in one of the two options:

- Write a code in a way that is supported by the compilers for auto-SIMDizing.

- Recognize the places in the code where auto-SIMDizing is not realistic and perform explicit SIMD programming in those places.

In addition, the programer should monitor the compiler auto-SIMDizing results in order to verify in which places the auto-SIMDizing was successful and in which cases it failed.
The programer may perform an iterative process of compiling with auto-SIMDizing option enabled, debugging the places where auto-SIMDizing failed, re-write the code in those cases, and then re-compile.

In order to activate the compilers auto-SIMDization, the programer should do the following:

- When using XLC: use optimization level `-O3 -qhot` or higher.
- When using GCC: use optimization level `–O2 –ftree-vectorize` or higher.

The next chapters discuss the following issues:

- “Coding for effective auto-SIMDization”- discuss how to wrote code which enables the compiler to perform effective auto-SIMDization, and what are the limitation of the compilers in auto-SIMDizing other types of code.
- “Debugging the compiler’s auto-SIMDization results” - discuss how the program may debug the compilers’ auto-SIMDization results in order to know wether is was successful or not. If it was not successful the compiler provide information of the potential problems which enables the programer to re-write the code.

Coding for effective auto-SIMDization

This chapter describes how to wrote code which enables the compiler to perform effective auto-SIMDization. The chapter also discuss the limitations of the compilers in auto-SIMDization of a scalar code. Knowing those limitation enables the programer to identify the places where auto-SIMDization is not possible. In those places the programer should then explicitly translate into a SIMD code.

Organize algorithms and loops

The programer should organize loops so that they can be auto-SIMDized and also structure algorithms to reduce dependencies:

- The inner-most loops are the ones that may be SIMDized.
- The programer should not manually unroll the loops.
- The programer should use ‘for’ loop construct since they are the only ones that can be auto-SIMDized. The ‘while’ construct on the other hand can not be auto-SIMDized.
- The number of iterations should be:

  - a constant (preferred using `#define` directive) and not a variable.
  - more than three times the number of elements per vector. Shorter loops might also be SIMDizable but it depends on their alignment and the number of statements in the loop.

- Using 'break' or 'continue' statement inside a loop should be avoided.
- The programer should avoid function calls within loops since they are not SIMDizable. Instead, either inline functions or macros may be used, or instead enable inlining by the compiler and possibly add an inlining directive to make sure that it happens. Another alternative is distributing the function calls into a separate loop.
- The programer should try to avoid operations that do not easily mapped onto vector operations. In general, all operations except branch, hint-for-branch, and load are capable of being mapped.
- The programer should use the select operation for conditional branches within the loop. Since loops that contain if-then-else statements might not always be SIMDizable, the programer should prefer using the C language :? (colon question-mark) operator which will cause the compiler to SIMDize this section using the select bits instruction.
- The programer should avoid aliasing problems, for example by using the `restrict` qualified pointers (illustrated in Example 4-60 on page 270). This qualifier when applied to a data pointer indicates that all access to this data are performed through this pointer an not through other pointer.
- Loops with inherent dependences are not SIMDizable, as illustrated in Example 4-58 on page 268.
- The programmer should keep the memory access-pattern simple:
  - Not using array of structures
    For example: `for (i=0; i<N; i++)  a[i].s = x;`
  - Should use constant increment.
    For example, do not use: `for (i=0; i<N; i+=incr)  a[i] = x;`

Organize data in memory

the programer should lay out data in memory so that operations on it can be easily SIMDize:

- The programer should use stride-one accesses (memory access patterns in which each element in a list is accessed sequentially). Non-stride-one accesses are less efficiently SIMDized, if at all. Random or indirect accesses are not SIMDizable.
- The programer should use arrays and not pointer arithmetic in the application to access large data structures.

- The programer should use global arrays that are statically declared.
- The programer should use global arrays that are aligned to 16B boundaries, for example using `aligned` attribute. As general, the programer should lay out the data to maximize 16B aligned accesses.
- If have more than a single misaligned store – the programer should distribute into a separate loop (currently the vectorizer peels the loop to align a misaligned store).
- If it is not possible to use aligned data, the programer should use the `alignx` directive to indicate to the compiler what the alignment is.
  For example: `#pragma alignx(16, p[i+n+1]);`
- If it is known that arrays are disjoint, the programer should use the `disjoint` directive to indicate to the compiler that the arrays specified by the pragma are not overlapping:
  For example:
```
#pragma disjoint(*ptr_a, b)
#pragma disjoint(*ptr_b, a)
```

Mix of data types

The mix of data types within code sections that may be potentially be SIMDized (i.e. loops with many iterations) may present problems. While the compiler may succeed in SIMDizing those sections, the programmer should try to avoid such mix of data types and try to keep a single data type within those section.

Scatter-gather

Scatter-gather refers to a technique for operating on sparse data, using an index vector. A gather operation takes an index vector and loads the data that resides at a base address added to the offsets in the index vector. A scatter operation stores data back to memory, using the same index vector.

The Cell BE processor's SIMD architecture does not directly support scatter-gather in hardware. therefore, the best way to extract SIMD parallelism is to combine operations on data in adjacent memory addresses into vector operations. This means that the programer may use scatter-gather to bring the data into a continuos area in the local store and then sequentially loop on the elements of this area variable. doing so may enable the compiler to SIMDize this loop.

Debugging the compiler's auto-SIMDization results

The XLC enables the programer to debug the compiler's auto-SIMDization results using the *-qreport* option. Doing so will produce a list of high level transformation performed by the compiler which includes everything from unrolling, loop interchange, and SIMD transformations. A transformed "pseudo source" will also be presented.

All loops considered for SIMDization are reported

- Successful candidates are reported
- If SIMDization was not possible, the reasons that prevented it are also provided

This feature is useful since it enables the programer to quickly identify opportunities for speedup. It provides feedback to the user explaining why loops are not vectorized. While those messages are not always trivial to understand, they may allow the programer to rewrites the relevant sections to allow SIMD vectorization.

Similarly, the GCC can also provide debug information about the auto-SIMDizing process using the following options:

- `-ftree-vectorizer-verbose=[X]` - Dumps information on which loops got vectorized, and which didn't and why (X=1 least information, X=6 all information). the information is dumped to stderr unless following flag is used:
- `-fdump-tree-vect` - Dumps information into <C file name>.c.t##.vect
- -fdump-tree-vect-details - Equivalent to setting the combination of the two flags: `-fdump-tree-vect -ftree-vectorizer-verbose=6`

The rest of the chapter illustrate how to debug a code which may not be SIMDized as well as another code which can be successfully SIMDized. We illustrate it using the XLC debug features (`-qreport` option enabled).

Example 4-58 shows a SPU code of a program named `t.c` which is hard to be SIMDized because of dependencies between sequential iterations:

Example 4-58 A non SIMDized loop

```
extern int *b, *c;

int main(){
      for (int i=0; i<1024; ++i)
           b[i+1] = b[i+2] - c[i-1];
}
```

The code is then compiled with `-qreport` option enabled using the command:

```
spuxlc -c -qhot -qreport t.c", in t.lst
```

Example 4-59 shows the `t.lst` file that is generated by the XLC compiler and contains the problems in SIMDizing the loop and also the transformed “pseudo source”:

Example 4-59 Reporting of SIMDization problems

```
1586-535 (I) Loop (loop index 1) at t.c <line 5> was not SIMD
vectorized because the aliasing-induced dependence prevents SIMD
vectorization.
1586-536 (I) Loop (loop index 1) at t.c <line 5> was not SIMD
vectorized because it contains memory references  with non-vectorizable
alignment.
1586-536 (I) Loop (loop index 1) at t.c <line 6> was not SIMD
vectorized because it contains memory references ((char *)b +
(4)*(($.CIV0 + 1))) with non-vectorizable alignment.
1586-543 (I) <SIMD info> Total number of the innermost loops considered
<"1">. Total number of the innermost loops SIMD vectorized <"0">.

     3 |  long main()
           {
     5 |    if (!1) goto lab_5;
           $.CIV0 = 0;
     6 |    $.ICM.b0 = b;
           $.ICM.c1 = c;
     5 |    do {   /* id=1 guarded */ /* ~4 */
             /* region = 8 */
             /* bump-normalized */
     6 |       $.ICM.b0[$.CIV0 + 1] = $.ICM.b0[$.CIV0 + 2] -
$.ICM.c1[$.CIV0 - 1];
     5 |       $.CIV0 = $.CIV0 + 1;
           } while ((unsigned) $.CIV0 < 1024u);    /* ~4 */
         lab_5:
           rstr = 0;
```

Other examples of messages that report problems with performing auto-SIMDization:

- `Loop was not SIMD vectorized because it contains operation which is not suitable for SIMD vectorization.`
- `Loop was not SIMD vectorized because it contains function calls.`
- `Loop was not SIMD vectorized because it is not profitable to vectorize.`
- `Loop was not SIMD vectorized because it contains control flow.`

- Loop was not SIMD vectorized because it contains unsupported vector data types
- Loop was not SIMD vectorized because the floating point operation is not vectorizable under -qstrict.
- Loop was not SIMD vectorized because it contains volatile reference

Example 4-60 shows a SPU code of which is similar to the previous example but with correcting SIMD Inhibitors:

Example 4-60 A SIMDized loop

```
extern int * restrict b, * restrict c;

int main()
{
   // __alignx(16, c);     Not strictly required since compiler
   // __alignx(16, b);     inserts runtime alignment check

    for (int i=0; i<1024; ++i)
        b[i] = b[i] - c[i];
}
```

Example 4-61 shows the output `t.lst` file after compiling with `-qreport` option enabled. The example report a successful auto-SIMDizing and also the transformed “pseudo source”:

Example 4-61 Reporting of SIMDization problems

```
1586-542 (I) Loop (loop index 1 with nest-level 0 and iteration count
1024) at t.c <line 9> was SIMD vectorized.
1586-542 (I) Loop (loop index 2 with nest-level 0 and iteration count
1024) at t.c <line 9> was SIMD vectorized.
1586-543 (I) <SIMD info> Total number of the innermost loops considered
<"2">. Total number of the innermost loops SIMD vectorized <"2">.
 4 |  long main()
          {
            $.ICM.b0 = b;
            $.ICM.c1 = c;
            $.CSE2 = $.ICM.c1 - $.ICM.b0;
            $.CSE4 = $.CSE2 & 15;
            if (!(! $.CSE4)) goto lab_6;
...
```

4.6.6 Using scalars and converting between different vector types

This chapter discuss how to convert between different types of vectors and how to work with scalars in the SIMD environment of the SPE. The three sections in this chapter covers the following issues:

- “Converting between different vector types” - describes how to perform correct and efficient conversion between vectors of different types.
- “Scalar overlay on SIMD instructions” - describes how to use scalars in SIMD instructions, include how to format them into vector data type and how to extract the results scalar from the vectors.
- “Casting between vectors and scalar” - describes how to cast vectors into equivalent array of scalars and vice versus.

Converting between different vector types

Casts from one vector type to another vector type has to be explicit and can be done using normal C-language casts. However, none of these casts performs any data conversion and the bit pattern of the result is the same as the bit pattern of the argument that is cast.

Example 4-62 shows an example of how we *do not* recommended casting between vectors. This is because the method shown usually does not provide the result expected by the programer since the integer variable `i_vector` will be assigned with a single precision float `f_vector` variable which has different format (i.e. the casting will not convert the bit pattern of the float to integer format).

Example 4-62 Not recommended casting between vectors

```
// BAD programming example
vector float f_vector;
vector int i_vector;

i_vector = (vector int)f_vector;
```

Instead, the recommended way to perform casting between vectors is using special intrinsics that convert between different data types of vectors including modify the bit pattern to the required type. The conversion intrinsics are:

- `spu_convtf`: convert signed or unsigned integer vector to float vector.
- `spu_convts`: convert float vector to signed integer vector.
- `spu_convtu`: convert float vector to unsigned integer vector.
- `spu_extend`: extend input vector to output vector whose elements have two times larger elements the input vector's (e.g short vector is extended to int vector, float vector is extended to double, etc.).

Scalar overlay on SIMD instructions

The SPU loads and stores one quadword at-a-time. When instructions use or produce scalar (sub quadword) l operands (including addresses), the value is kept in the preferred scalar slot of a SIMD register. The fact that the scalar should be located in the specific preferred slots requires extra instructions whenever a scalar is used as part of a SIMD instruction:

- When a scalar is loaded in order to be a parameter of some SIMD instruction it should be rotated to the preferred slot before being executed.
- When a scalar should be modified by some SIMD instruction it should be loaded, rotated to the preferred slot, modified by the SIMD instruction, rotated back to its original alignment and stored in to memory.

Obviously these extra rotating instructions reduce performance making vector operations on scalar data are not efficient.

The first technique in order to make such scalar operations more efficient is a static one:

- Use the `aligned` attribute and extra padding if needed in order to statically align the scalar to the preferred slot. Using this attribute is described in "aligned attribute" on page 251.
- Change the scalars to quadword vectors. This will eliminate the three extra instructions associated with loading and storing scalars which will reduce the code size and execution time.

In addition, the programer may use one of the SPU intrinsics to efficiently promote scalars to vectors, or vectors to scalars:

- `spu_insert`: Insert a scalar into a specified vector element.
- `spu_promote`: Promote a scalar to a vector containing the scalar in the element that is specified by the input parameter. Other elements of the vector are undefined.
- `spu_extract`: Extract a vector element from its vector and return the element as scalar.

- `spu_splats`: Replicate a single scalar value across all elements of a vector of the same type.

Since those instructions are very efficient, the programer can use them to eliminate redundant loads and stores. One example for using those instructions is to cluster several scalars into vectors, load multiple scalars at one instruction using a quadword memory, and perform SIMD operation that will operate on all the scalars at once.

There are two possible implementation for such mechanism:

1. Use extract or insert intrinsics: Cluster several scalars into vector using `spu_insert` intrinsics, perform some SIMD operations on them and extract them back to their scalar shape using `spu_extract` intrinsic.
   Example 4-63 show an SPU program that implements this mechanism. Even this simple case is more efficient then multiply the scalar vectors one by one using ordinary scalar operations. Obviously, if more SIMD operations are performed on the constructed vector, the performance overhead of creating the vector and extracting the scalars becomes negligible.
2. Another possible implementation is using the unions that perform casting between vectors and scalars arrays and are described in Example 4-64 on page 275 and the following Example 4-65.

Source code: The code of Example 4-63 is included in the additional material that is provided with this book. See “SPE scalar to vector conversion using insert and extract intrinsics” on page 617 for more information.

Example 4-63 Cluster scalars into vectors

```
#include <spu_intrinsics.h>
int main( )
{
   float a=10,b=20,c=30,d=40;
   vector float abcd;
   vector float efgh = {7.0,7.0,7.0,7.0};

   // initiate ‘abcd’ vector with the values of the scalars
   abcd = spu_insert(a, abcd, 0);
   abcd = spu_insert(b, abcd, 1);
   abcd = spu_insert(c, abcd, 2);
   abcd = spu_insert(d, abcd, 3);

   // SIMD multiply the vectors
   abcd = spu_mul(abcd, efgh);
```

```
   // do many other SIMD operations on 'abcd' and 'efgh' vectors

   // extract back the 'multiplied' scalar from the computed vector
   a = spu_extract(abcd, 0);
   b = spu_extract(abcd, 1);
   c = spu_extract(abcd, 2);
   d = spu_extract(abcd, 3);

   printf("a=%f, b=%f, c=%f, d=%f\n",a,b,c,d);
   return 0;
}
```

Casting between vectors and scalar

The SPU vector data types are kept in the memory in continuos 16 bytes area whose address is also 16 bytes aligned. Pointers to vector types and non-vector types may therefore be cast back and forth to each other. For the purpose of aliasing, a vector type is treated as an array of its corresponding element type. For example, a `vector float` can be cast to `float*` and vice versus.

If a pointer is cast to the address of a vector type, it is the programmer's responsibility to ensure that the address is 16-byte aligned.

Casts between vector types and scalar types are illegal. On the SPU, the spu_extract, spu_insert, and spu_promote generic intrinsics or the specific casting intrinsics may be used to efficiently achieve the same results.

In some cases it is essential to perform SIMD computation on some vectors but also perform some computations between different elements of the same vector. From convenient programming approach for that is define casting unions of either vectors or array of scalars as explained in Example 4-64.

Source code: The code of Example 4-64 and Example 4-65 is included in the additional material that is provided with this book. See "SPE scalar to vector conversion using unions" on page 617 for more information.

A SPU program that may uses those casting union is shown in the code of Example 4-65. The program uses those unions to perform a combination of SIMD operations on the entire vector and scalar operations between the vector elements.

It is important to know that while the scalar operation are easy to program that way they are not very efficient form performance point of view so the programer

should try to minimize the frequency in which they happen and use them only if there is not simple SIMD solution.

Example 4-64 Header file for casting between scalars and vectors

```
// vec_u.h file ================================================

#include <spu_intrinsics.h>

typedef union {
   vector signed char c_v;
   signed char c_s[16];

   vector unsigned char uc_v;
   unsigned char uc_s[16];

   vector signed short s_v;
   signed short s_s[8];

   vector unsigned short us_v;
   unsigned short us_s[8];

   vector signed int i_v;
   signed int i_s[4];

   vector unsigned int ui_v;
   unsigned int ui_s[4];

   vector signed long long l_v;
   signed long long l_s[2];

   vector unsigned long long ul_v;
   unsigned long long ul_s[2];

   vector float f_v;
   float f_s[4];

   vector double d_v;
   double d_s[2];

}vec128;
```

Example 4-65 Cluster scalars into vectors using casting union

```
#include <spu_intrinsics.h>

#include "vec_u.h" // code from Example 4-64 show

int main( )
{
   vec_float a __attribute__((aligned (16)));
   vec_float b __attribute__((aligned (16)));


   // do some SIMD operations on ‘a’ and ‘b’ vectors

   // perform some operations between scalar of specific vector
   a.s[0] = 10;
   a.s[1] = a.s[0] + 10;
   a.s[2] = a.s[1] + 10;
   a.s[3] = a.s[2] + 10;

   // initiate all ‘b’ elements to be 7
   b.v = spu_splats( (float)7.0 );

   // SIMD multiply the two vectors
   a.v = spu_mul(a.v, b.v);

   // do many other different SIMD operations on ‘a’ and ‘b’ vectors

   // extract back the scalar from the computed vector

   printf("a0=%f, a1=%f, a2=%f, a3=%f\n",a.s[0],a.s[1],a.s[2],a.s[3]);

   return 0;
}
```

4.6.7 Code transfer using SPU code overlay

This section provides a very brief overview on the SPU overlay facility which handles cases in which the entire SPU code is too big to fit the LS (taking into account that the 256 KB of LS should also store the data, stack and heap). 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 LS before the main program begins to execute, but is instead left in 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.

The overlay feature is supported on SDK3.0 for SPU programming but not for PPU programming.

Here are the main principles on which the overlay is base on:

- The linker generate the overlays as 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.
- 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 segment is already in LS. If not this segment is loaded dynamically using a DMA command, and may overlay another segment which had been loaded previously.
- XL compilers can assist in the construction of the overlays based upon the call graph of the application.

A detailed description of this facility including instructions how to use it and usage example is in *SPU code overlays* chapter in Programmer's Guide document.

4.6.8 Eliminating and predicting branches

The SPU hardware assumes sequential instruction flow means that unless explicitly defined otherwise assumes that all branches are not taken. Correctly predicted branches execute in one cycle, but a mispredicted branch (conditional or unconditional) incurs a penalty of 18 to 19 cycles, depending on the address of the branch target. Considering the typical SPU instruction latency of 2 to 7 cycles, mispredicted branches can seriously degrade program performance. The branch instructions also restrict a compiler's ability to optimally schedule instructions by creating a barrier on instruction reordering.

The most effective method of reducing the impact of branches is to eliminate them using three primary methods that are discuss in the next three chapters:

- “Function-inlining”: define functions as inline and avoid the branch when a function is called and another branch when it is returned.
- “Loop-unrolling”: remove loops or reduce the number of iterations in loop in order to reduce the number of branches (that appears in the end of the loop).
- “Branchless control flow statement”: use `spu_sel` intrinsics to replace simple control statement.

The second-most effective method of reducing the impact of branches is discuss on the last chapter:

- “Branch hint”: discuss the hint-for branch instructions. If software speculates that the instruction branches to a target path, a branch hint is provided. If a hint is not provided, software speculates that the branch is not taken (that is, instruction execution continues sequentially).

Function-inlining

Function-inlining technique can be used to increase the size of basic blocks (sequences of consecutive instructions without branches). This techniques eliminates the two branches associated with function-call linkage - the branch for function-call entry and the branch indirect for function-call return.

In order to use function inlining the programer can choose from one of the following techniques:

- Explicitly add the `inline` attribute to the declaration of any function that the programer would like to inline. One case when it is recommended to do so is for functions that are very short. Another case is for functions that have small number of instances in the code but are often executed in run time (for example when they appear inside a loop).
- Use the compiler options for automatic inlining the appropriate functions. Table 4-21 describes some of those options of the GCC compiler.

Over-aggressive use of inlining can result in larger code which reduces the LS space available for data storage or, in the extreme case, is too large to fit in the LS.

Table 4-21 GCC options for functions inlining

Option	Description
-finline-small-functions	Integrate functions into their callers when their body is smaller than expected function call code (so overall size of program gets smaller). The compiler heuristically decides which functions are simple enough to be worth integrating in this way.

Option	Description
-finline-functions	Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way. If all calls to a given function are integrated, and the function is declared static, then the function is normally not output as assembler code in its own right.
-finline-functions-called-once	Consider all static functions called once for inlining into their caller even if they are not marked inline. If a call to a given function is integrated, then the function is not output as assembler code in its own right.
-finline-limit=n	By default, GCC limits the size of functions that can be inlined. This flag allows the control of this limit for functions that are explicitly marked as inline.

Loop-unrolling

Loop-unrolling is another techniques that can be used to increase the size of basic blocks (sequences of consecutive instructions without branches), which increases scheduling opportunities. It eliminates branches by decreasing the number of loop iterations.

If the number of loop iterations is a small constant then it is usually recommended to remove the loop in order to eliminate brances in the code. Example 4-66 provide a similar code example.

Example 4-66 Remove short loop for eliminating branches

```
// original loop
for (i=0;i<3;i++) x[i]=y[i];

// can be removed and replces by
x[0]=y[0];
x[1]=y[1];
x[2]=y[2];
```

If the number of loops is bigger but the loop iteration are independent of each other the programer can reduce the number of loops and work on several items in each iterations as illustrate in Example 4-67 provide a similar code example. Another advantage of this technique is that it is usually improve the dual issue utilization. The loop unrolling techniques is often used when move from scalar to vector instructions.

Example 4-67 Long loop unrolling for eliminating branches

```
// original loop
for (i=0;i<300;i++) x[i]=y[i];

// can be unrolled to
for (i=0;i<300;i+=3){
   x[i]  =y[i];
   x[i+1]=y[i+1];
   x[i+2]=y[i+2];
}
```

An automatic loop unrolling can be performed by the compiler in case the optimization level is high enough or one of the appropriate options are set (e.g. `-funroll-loops`, `-funroll-all-loops`).

Typically, branches associated with loop with relatively large number of iteration are inexpensive because they are highly predictable. In this case non-predicted branch usually occur only in the first and last iterations.

Similar to function inlining, over-aggressive use of loop unrolling can result in code that reduces the LS space available for data storage or, in the extreme case, is too large to fit in the LS.

Branchless control flow statement

The select-bits (`selb`) instruction is the key to eliminating branches for simple control-flow statements such as if and if-then-else constructs. An if-then-else statement can be made branchless by computing the results of both the then and else clauses and using select bits intrinsics (`spu_sel`) to choose the result as a function of the conditional.

If computing both results costs less than a mispredicted branch, then a performance improvement is expected.

Example 4-66 demonstrate the use of `spu_sel` intrinsics to eliminate branches in simple if-then-else control block.

Example 4-68 Branchless if-then-else control block

```
// a,b,c,d are vectors

// original if-else control block
if (a>b) c +=1;
else     d = a+b;
```

```
// optimized spu_sel based code that eliminates branches but provides
// similar functionality.
select   = spu_cmpgt(a,b);
c_plus_1 = spu_add(c,1);
a_plus_b = spu_add(a,b);

c = spu_sel(c, c_plus_1, select);
d = spu_sel(a_plus_b, d, select);
```

Branch hint

The SPU supports branch prediction through a set of hint-for branch (HBR) instructions (`hbr`, `hbra`, and `hbrr`) and a branch-target buffer (BTB). These instructions support efficient branch processing by allowing programs to avoid the penalty of taken branches.

The hint-for branch instructions provide advance knowledge about future branches such as address of the branch target, address of the actual branch instruction, and prefetch schedule (when to initiate prefetching instructions at the branch target).

Hint-for branch instructions have no program-visible effects. They provide a hint to the SPU about a future branch instruction, with the intention that the information be used to improve performance by prefetching the branch target.

If software provides a branch hint, software is speculating that the instruction branches to the branch target. If a hint is not provided, software speculates that the branch is not taken. If speculation is incorrect, the speculated branch is flushed and prefetched. It is possible to sequence multiple hints in advance of multiple branches.

As with all programmer-provided hints, care must be exercised when using branch hints because, if the information provided is incorrect, performance might degrade. There are immediate and indirect forms for this instruction class. The location of the branch is always specified by an immediate operand in the instruction.

A common use to branch hint is in the end-of-loop branches when it is expected to be correct. Such hint will be correct for all loop iterations besides the last one.

A branching hint should be present soon enough in the code. A hint that precede the branch by at least eleven cycles plus four instruction pairs is minimal. Hints that are too close to the branch do not affect the speculation after the branch.

A common approach to generating static branch prediction is to use expert knowledge that is obtained either by feedback-directed optimization techniques or using linguistic hints supplied by the programmer.

There are many arguments against profiling large bodies of code, but most SPU code is not like that. SPU code tends to be well-understood loops. Thus, obtaining realistic profile data should not be time-consuming. Compilers should be able to use this information to arrange code so as to increase the number of fall-through branches (that is, conditional branches not taken). The information can also be used to select candidates for loop unrolling and other optimizations that tend to unduly consume LS space.

Programmer-directed hints can also be used effectively to encourage compilers to insert optimally predicted branches. Even though there is some anecdotal evidence that programmers do not use them very often, and when they do use them, the result is wrong, this is likely not the case for SPU programmers. SPU programmers generally know a great deal about performance and will be highly motivated to generate optimal code.

The SPU C/C++ Language Extension specification defines a compiler directive mechanism for branch prediction. The `__builtin_expect` directive allows programmers to predicate conditional program statements. Example 4-69 demonstrates how a programmer can predict that a conditional statement is false (a is not larger than b).

Example 4-69 Predict false conditional statement

```
if(__builtin_expect((a>b),0))
   c += a;
else
   d += 1;
```

Not only can the `__builtin_expect` directive be used for static branch prediction, it can also be used for dynamic branch prediction. The return value of `__builtin_expect` is the value of the `exp` argument, which must be an integral expression. For dynamic prediction, the value argument can be either a compile-time constant or a variable. The `__builtin_expect` function assumes that `exp` equals value. Example 4-70 show a code for a static-prediction.

Example 4-70 Static branch prediction

```
if (__builtin_expect(x, 0)) {
   foo(); /* programmer doesn't expect foo to be called */
}
```

A dynamic-prediction example might look like Example 4-71:

Example 4-71 Dynamic branch prediction

```
cond2 = .../* predict a value for cond1 */
...
cond1 = ...
if (__builtin_expect(cond1, cond2)) {
   foo();
}
cond2 = cond1;/* predict that next branch is the same as the previous*/
```

Compilers may require limiting the complexity of the expression argument because multiple branches can be generated. When this situation occurs, the compiler will issue a warning if the program's branch expectations are ignored.

4.7 Frameworks and domain-specific libraries

This chapter discuss some high level frameworks for development and execution of parallel applications on Cell BE and also some domain-specific libraries that are provided by SDK3.0.

The high level frameworks provides an alternative to using the lower level libraries. The lower level libraries enables the programer full control over the hardware mechanisms (e.g. DMA, mailbox, SPE thread) and are discussed in other chapters of "Cell BE programming" section.
The two main purposes of the high level frameworks are reducing the development time of programing an Cell BE application and creating an abstract layer which hides from the programer Cell BE's architecture specific features. In some cases, the performance of the application using those frameworks is similar to programing using the lower level libraries. Given the fact that development time is shorted and the code is more architecture independent using the framework in those case is preferred. However, as general using the low lever libraries can provide better performance since the programer can tune the program to the application specific requirements.

The first two chapters discuss the main frameworks that are provided with SDK3.0:

1. "DaCS - Data Communication and Synchronization" on page 284 discuss DaCS which is an API and a library of C callable functions that provides

communication and synchronization services amongst tasks of a parallel application running either on a Cell BE system. Another version of DaCS provides similar functionality for a hybrid system and is discussed in 7.1.1, “Hybrid DaCS” on page 443.

2. “ALF - Accelerated Library Framework” on page 291 ALF offers a framework for implementing the function off-load model on a Cell BE system using the PPE as the program control and SPEs as functions off-load accelerators. As in the DaCS case, hybrid version is also available and is discussed in 7.1.2, “Hybrid ALF” on page 456.

In addition to those SDK3.0 frameworks, a growing number of high level frameworks for Cell BE programming are being developed by companies other then IBM or by universities. Discussing those frameworks is out of the scope of this book. A brief description of some of those frameworks is in 3.1.4, “The Cell BE programming frameworks” on page 39.

The domain-specific libraries aim to assist Cell BE programmers by providing reusable functions that implement a set of common a algorithms and mathematical operators (e.g. FFT, monte carlo, BLAS, matrix and vector operators). Those libraries are discussed in the third chapter:

3. “Domain-specific libraries” on page 309 provide a brief description of the some of the main libraries which are provided by SDk3.0.

The functions that these libraries implement are optimized specifically to Cell BE and can reduce development time in cases where the developed application uses similar functions. In those cases the programer may use the corresponding library to implement those functions or use to as a reference and customized it to the specific requirement of the developed application (the libraries are open source).

4.7.1 DaCS - Data Communication and Synchronization

DaCS is an API and a library of C callable functions that provides communication and synchronization services amongst tasks of a parallel application running either on a Cell BE system or a hybrid system. Hybrid specific issues are discussed in 7.1.1, “Hybrid DaCS” on page 443. In the rest of this discussion, the actual implementation, Cell BE or hybrid, is of no importance as we only describe the concepts and the API calls.

DaCS can be used to implement various types of dialogs between parallel tasks using common parallel programming mechanisms like message passing, mailboxes, mutex and remote memory accesses to name a few. The only assumption is that there is a master task and slave tasks, a host element (HE) and accelerator elements (AE) in DaCS terminology. This is to be contrasted with

MPI which treats all tasks as equal. The aim of DaCS is to provide services for applications using the host/accelerator model, where one task subcontracts lower level tasks to perform a given piece of work. One model might be an application written using MPI communication at the global level with each MPI task connected to accelerators that communicate with DaCS. This is pictured below.

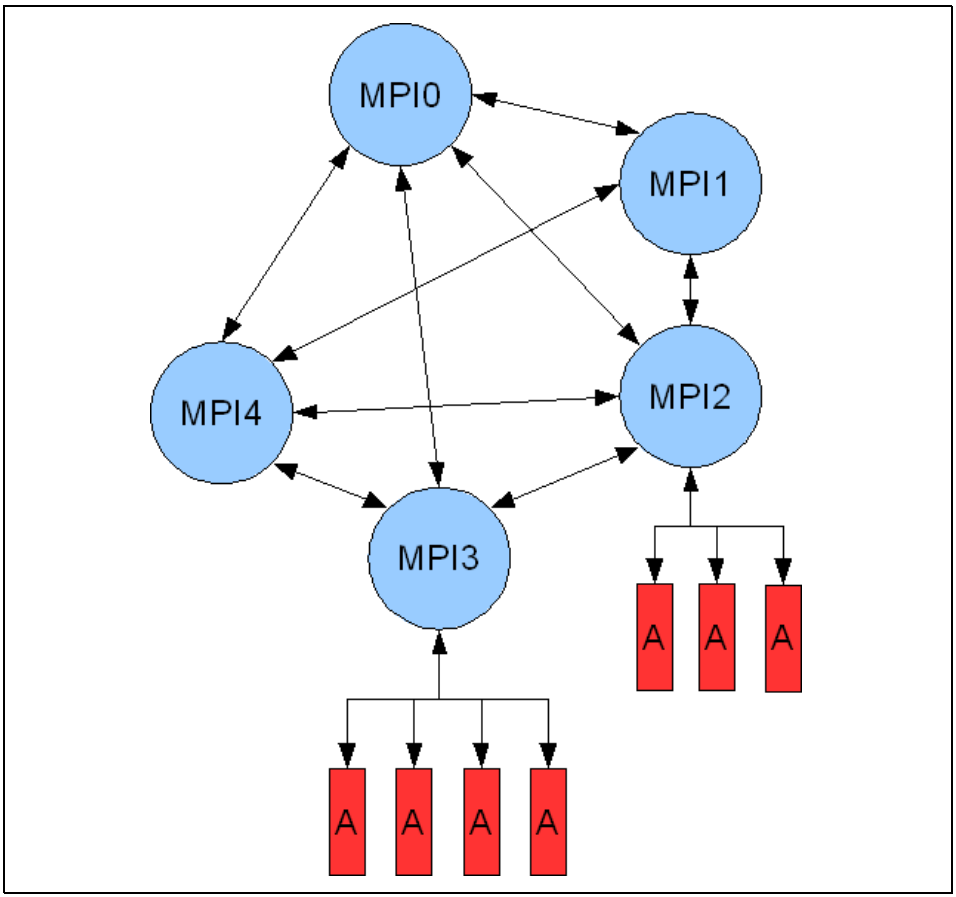


Figure 4-10 Possible arrangement for a MPI - DaCS application

Here, 5 MPI tasks will exchange MPI messages and use DaCS communication with their accelerators. No direct communication occurs between accelerators that report to a different MPI task.

DaCS also supports a hierarchy of accelerators. A task can be a an accelerator for a task higher up in the hierarchy and be a host element for lower level accelerators as shown below.

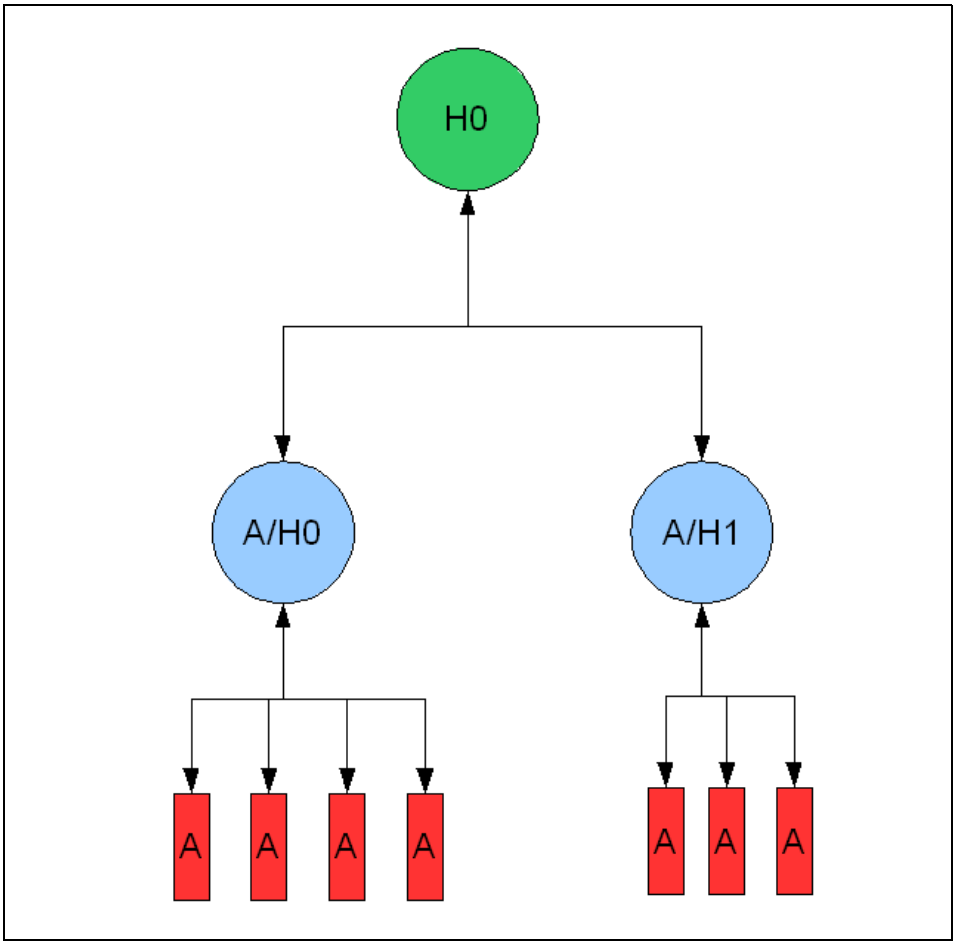


Figure 4-11 A two level hierarchy with DaCS

The host element H0 is accelerated by 2 accelerators (A/H0, A/H1), in turn accelerated by 4 and 3 accelerators.

A DaCS program need not be an MPI program nor use a complex multi-level hierarchy. DaCS can be used for an application that consists in a single host process and its set of accelerators.

DaCS main benefit is to offer abstractions (message passing, mutex, remote memory access) that are more common for application programmers than the DMA model that is probably better known by system programmers. It also hides the fact that the host element and its accelerators may not be running on the same system.

DaCS concepts

DaCS elements (HE/AE)

A task in a DaCS program is identified by a couple <DaCS Element, Process ID>. The DaCS Element (DE) identifies the accelerator or host element. This could be a PPE, a SPE or another system in a hybrid configuration. In general, a given DaCS Element could have multiple processes running, so we need a Process ID to uniquely identify a participating DaCS task.

A DE can be either a Host Element (HE) or an Accelerator Element (AE) or both in the case of a multi-level hierarchy. A HE will reserve a AE for its exclusive use and create and terminate processes to run on it.

DaCS groups

In DaCS, the groups are created by a HE and AE can only join a group previously created by their HE. The groups are used to enable synchronization (barrier) between tasks.

DaCS remote memory regions

A HE can create an area of memory that is to be shared by its AE. A region can be shared in read-only or read-write mode and once the different parties are set up to share a remote memory segment, the data is accessed by each DE using a put or get primitive to move data back and forth between its local memory and the shared region. The data movement primitives also support DMA lists to enable gather/scatter operations.

DaCS mutex

A HE can create a mutex that AE will agree to share. Once the sharing has been explicitly set up, the AE will be able to use lock-unlock primitives to serialize accesses to shared resources.

DaCS communication

Apart from the put/get primitives in remote memory regions, DaCS offers two other mechanisms for data transfer: mailboxes and message passing using basic send/recv functions. They are also asymmetric as the data in mailboxes and send/recv functions between HE and AE only.

DaCS wait identifiers

The data movement functions in DaCS: put/get and send/recv are asynchronous and return immediately. DaCS provides functions to wait for the completion of a previously issued data transfer request. These functions use a wait identifier that need to be explicitly reserved before being used. They should also be released when no longer in use.

DaCS services

The functions in the API are organized in groups. They are described below.

Resource and process management

A HE will use functions in this group to query the status and number of available AE and to reserve them for its future use. Once a AE has been reserved, it can be assigned some work with the `dacs_de_start()` function. In a Cell BE environment, the work given to an AE will be an embedded SPE program whereas in a hybrid environment, it will be a Linux binary.

Group management

Groups are required to operate synchronizations between tasks. Currently, only barriers are implemented.

Message passing

Two primitives are provided for sending and receiving data using a message passing model. The operations are non-blocking and the calling task must wait later for completion. The exchanges are point to point only and one of the end-point needs to be a HE.

Mailboxes

An efficient message notification mechanism for small 32-bit data between two separate processes. In the case of the Cell BE processor, these are implemented in the hardware using an interrupt mechanism for communication between the SPE and the PPE or other devices.

Remote memory operations

This is a mechanism for writing/reading directly from/to memory in remote processes address space. In MPI, this type of data movement is called one-sided communication.

Synchronization

Mutexes may be required to protect the remote memory operations and serialize accesses to shared resources.

Common patterns

The group, remote memory and synchronization services are implemented with a consistent set of create, share/accept, use and release/destroy primitives. In all three cases, the HE is the initiator and the AE are invited to share what the HE has prepared for them. This is shown below.

Table 4-22 Pattern for sharing resources in DaCS

HE side	AE side
Create the resource	
Invite each AE to share the resource and wait for confirmation from each AE, one by one	Accept the invitation to share the resource
Use the resource, the HA can take part to the sharing but it's not mandatory	Use the resource
Destroy the shared resource: wait for each AE to notify us that it does not use the resource anymore	Release: signal the HE that we do not use the resource anymore

For the remote memory regions, the table is as follows.

Table 4-23 Remote memory sharing primitives

HE side	AE side
dacs_remote_mem_create()	
dacs_remote_mem_share()	dacs_remote_mem_accept()
	dacs_put(), dacs_get()
dacs_remote_mem_destroy()	dacs_remote_mem_release()

As for the groups, we get:

Table 4-24 Group management primitives

HE side	AE side
dacs_group_init()	
dacs_group_add_member()	dacs_group_accept()
dacs_group_close(), this marks the end of the group creation	

HE side	AE side
	dacs_barrier_wait()
dacs_group_destroy()	dacs_groupe_leave()

And for mutexes:

Table 4-25 Mutexes primitives

HE side	AE side
dacs_mutex_init()	
dacs_mutex_share()	dacs_mutex_accept()
	dacs_mutex_lock(), dacs_mutex_unlock(), dacs_mutex_trylock()
dacs_mutex_destroy()	dacs_mutex_release()

An annotated DaCS example

We provide an example program that illustrates the use of most of the API functions. The program source code is very well documented such that reading through will give a very clear understanding of the DaCS API functions and how they need to be paired between the HE and the AE.

Source code: The DaCS code example that is mentioned above is part of the additional material that comes with the book. See “DaCS programming example” on page 611 for more details.

The DaCS libraries are fully supported by the debugging and tracing infrastructure provided by the IBM SDK for Multicore Acceleration. The sample code above can be built with the “debug” and “trace” flavors of the DaCS library.

Usage notes and current limitations

DaCS provides services for data communication and synchronization between a HE and its AE. It does not tie the application to a certain type of parallelism and any parallel programming structure can be implemented.

In the current release, no message passing between AE is allowed and complex exchanges will either require more HE intervention or will need to be implemented using the shared memory mechanisms (remote memory and mutexes). A useful extension could be to allow AE to AE messages. Some data movement patterns (pipeline, ring of tasks) would be easier to implement in

DaCS. Of course, we can always call directly libspe2 functions from within a DaCS task to implement custom task synchronizations and data communications but this technique is not a supported by the SDK.

4.7.2 ALF - Accelerated Library Framework

ALF offers a framework for implementing the function offload model. The functions that were previously running on the host are now being offloaded and accelerated by one or more accelerators. In ALF, the host and accelerator node types can be specified by the API and are general enough to allow various implementations. In the current implementations, the accelerator functions always run on the SPE of a Cell BE and the host applications run either on the PPE of a Cell BE on the offically supported Cell of ALF version or on an x86_64 node in the alpha version of the hybrid model.

ALF overview

With ALF, the application developer is required to divide the application in two parts : the control part and the computational kernels. The control part runs on the host and it sub-contracts accelerators to run the computational kernels. These kernels will take their input data from the host memory and will write back the output data to the host memory. ALF is an extension of the subroutine concept, with the difference that input arguments and output data have to move back and forth between the host memory and the accelerator memory, akin to the RPC (Remote Procedure Call) model. The input and output data may have to be further divided into blocks to be made small enough to fit the limited size of the accelerator's memory. The individual blocks are organized in a queue and are meant to be independent of each other. The ALF runtime manages the queue and balancing the work betweeen the accelerators. The application programmer only has to put the individual pieces of work in the queue.

Let us suppose we have an MPI application which we wish to accelerate by off-loading the computational routines onto a multitude of Cell BE SPEs. Once accelerated using ALF, each MPI task will still enjoy its life as a MPI task: working in sync with the other MPI tasks and performing the necessary message passing. But now, instead of running the computational parts between MPI calls, it will only orchestrate the work of the accelerator tasks that it will allocate for its own use. Each MPI task needs to know about the other MPI tasks for synchronization and message passing but an accelerator task does not need to know anything about its host task nor about its siblings and even less about the accelerators running on behalf of foreign MPI tasks. An accelerator task has no visibility to the outside world. It only answers to requests: it is fed with input data, does some processing and the output data it produces is sent backsends back back.

There is no need for an accelerator program to know about its host program as the ALF runtime takes care of all the data movement between the accelerator memory and the host memory on behalf of the accelerator task. The ALF runtime does the data transfer using all sorts of clever tricks, exploiting DMA, double or triple buffering and pipelining techniques that the programmer does not need to learn about. All the programmer has to do is to describe, generally at the host level, the layout of the input and output data in host memory that the accelerator task will work with.

ALF gives a lot of flexibility to manage accelerator tasks. It supports the MPMD (Multiple Program Multiple Data) model in two ways:

- a subset of accelerator nodes can run taskA providing the computational kernel ckA while another subset will run taskB providing the kernel ckB,
- while a single accelerator task can only perform a single kernel at an one time, there are ways the accelerator can load a different kernel after execution starts.

ALF can also express dependencies between tasks allowing for complex ordering of tasks when synchronization is required.

The ALF runtime and programmer's tasks

The ALF runtime provides the following services from the application programmer's perspective:

- at the host level:
  - work blocks queue management,
  - load balancing between accelerators,
  - dependencies between tasks,
- at the accelerator level:
  - optimized data transfers between the accelerator memory and the host memory, exploiting the data transfer list used to describe the input and output data.

On the host side, the application programmer will have to make calls to ALF API to:

- create the tasks
- split the work in work blocks, that is describing the input and output data for each block
- express the dependencies between tasks if needed
- put the work blocks in the queue

On the accelerator side, he will only have to write the computational kernels. As we will see later, this is slightly over-simplified as the separation of duties between the host programs and the accelerator program may become a bit blurred in the interest of performance.

ALF architecture

The picture below (from the ALF programming guide) summarizes how ALF works.

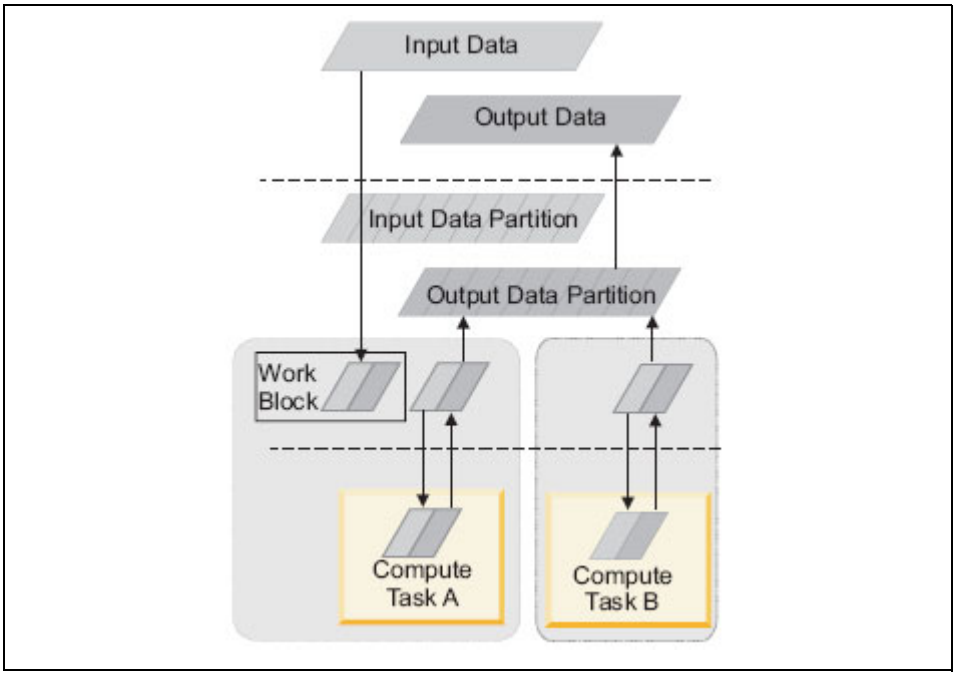


Figure 4-12 The ALF architecture

Near the top is the host view with presumably large memory areas figuring the input and output data for the function that is to be accelerated. In the middle, lies the data partitioning where the input and output are split into smaller chunks, small enough to fit in the accelerator memory, the so-called work blocks. At the bottom, the accelerator tasks are pictured, processing one work block at a time. The data transfer part between the host and accelerator memory is taken care of by the ALF runtime.

The picture below shows the split between the host task and the accelerator tasks. On the host side, we create the acelerator tasks, create the work blocks,

enqueue the blocks and wait for the tasks to collectively empty the work block queue. On the accelerator task, for each work block, the ALF runtime will fetch the data from the host memory, call the user provided computational kernel and send back to the host memory the output data.

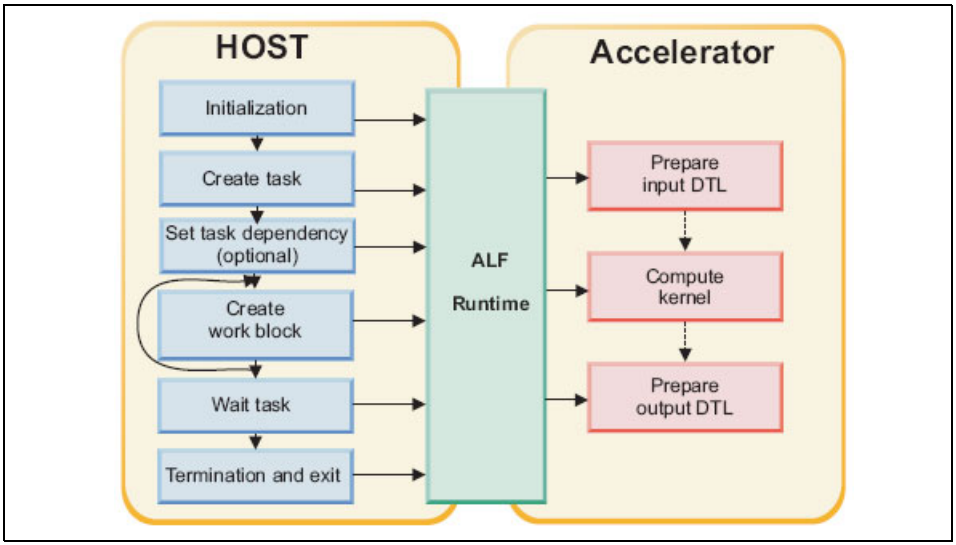


Figure 4-13 The host and accelerator sides

A simplified view of an accelerator task workflow

To illustrate how the ALF runtime works on the accelerator side, we present a simplified pseudo code of the accelerator task. This program is similar to what ALF provides. An application programmer only needs to register the routines that get called by the `call_user_routine`. This is obviously not the real ALF source code but this shows in a very simplistic way what it does.

Example 4-72 A pseudo code for the accelerator workflow

```
// This SPE program is started by the alf_task_create() call on the
// host program.
int main()
{
   // If the user provided a context setup routine, call it now
   call_user_routine(task_context_setup);

   // Enter the main loop, we wait for new work blocks
```

```
   while (more_work_blocks()) {
      while(multi_use_count) { // more on this later
         get_user_data(work_block_parm);

         // If the user wants to have the data partitioning on
         // the accelerator, call the routine he gave us to do so.
         if(accelerator_data_partitioning) {
            call_user_routine(input_prepare);
         }

         // The input data list is ready. Fetch the items from the
         // host into the task input buffer.
         get_input_data();

         // We have the input data. Let's call the user function.
         call_user_routine(compute_kernel);

         // If the output data partitioning is to be run on the
         // accelerator, call the user provided routine.
         if(accelerator_data_partitioning) {
            call_user_routine(output_prepare);
         }

         // The output data list is ready, scatteer the data back to
         // the host memory.
         put_output_data();
      }
   }

   // We are about to leave. If we were asked to merge the context,
   // do it now with the user provided routine.
   call_user_routine(task_context_merge);
}
```

Description of the ALF concepts

The entities that ALF manipulates are listed below:

- computational kernels,
- tasks,
- work blocks,

- data partitioning,
- datasets,.

Let's describe each topic with more details.

Computational kernel

This is the why we use ALF in the first place: we off-load the computation kernels of the application from the host task to accelrator tasks running on accelerator nodes, SPEs in our case. A computational kernel is a function that takes some input, does some processing and produces output data. Within ALF, a computational kernel function has a given prototype that application programmers have to conform to. In the simplest case, an application programmer needs to implement a single routine: the one that does thee computation. In the most general case, up to 5 functions may need to be written:

- the compute kernel,
- the input data transfer list prepare function,
- the output data transfer list prepare function,
- the task context setup function,
- the task context merge function.

The full prototypes are given below, taken from the `alf_accel.h` file which an accelerator program must include. The ALF runtime will fill in the buffers (input, output, context) before calling the user provided function. From an application programmer's perspective, our function gets called after the runtime has filled all the necessary data, transferring the data from the host memory to the accelerator memory, that is the local store. We do not have to care for that. We are just given pointers to where the data has been made available for us. This is similar to what the shell does for us when the `main()` function of a Linux program is called: the runtime system (the `exec()` system call that set us to run in this case) has filled for us the `char *argv[]` array for us to use.

Example 4-73 The accelerator functions prototypes

```
// This is the compute kernel. It is called for every work block.
// Some arguments may be NULL. For example, the inout buffer may
// not be used.

// The task context data pointed to by p_task_context is filled
// at task startup only, not for every work block, but we want to
// be able to use this state data from inside the compute kernel
// every time we get called.

// The current_count and total_count are 0 and 1 and can be ignored
```

```
// for single-use work blocks. See later for multi-use work blocks.

// The data pointed to by p_parm_context is filled every time we
// process a work block. I nthe case of a multi-use work block,
// it can be used to store data that is common to multi-use blocks
// invocations.
int (*compute_kernel) (
      void *p_task_context,
      void *p_parm_context,
      void *p_input_buffer,
      void *p_output_buffer,
      void *p_inout_buffer,
      int current_count,
      int total_count);

// The two routines below are used when we do the data
// partitioning on the accelerator side, possibly because
// this requires too much work for the PPE to keep up with
// the SPEs. If we stick to host data partitionning, we do
// not define these routines.

// The area pointed to by p_dtl is given to us by the ALF runtime.
// We will use this pointer as a handle and pass it as an argument to
// the functions we will call to add entries to the list of items
// that need to be brought in (resp. out) before (resp. after) the
// compute kernel is called. The parm_context data may contain
// information required to compute the data transfer lists.
int (*input_list_prepare or output_list_prepare) (
      void *p_task_context,
      void *p_parm_context,
      void *p_dtl,
      int current_count,
      int total_count);

// The task_context_setup function is called at task startup time.
// This function can be used to prepare the necessary environment
// to be ready when the work blocks will be sent to us.
int (*task_context_setup) (
      void *p_task_context);

// The task_context_merge function is called at task exit time. It can
// be used for reduction operations. We update our task context data
// (p_task_context) by applying a reduction operation between this
// data and the incoming context data that is filled for us by
// the runtime.
```

```
int (*task_context_merge) (
      void *p_context_to_merge,
      void *p_task_context);
```

The computational kernel functions need to be registered to the ALF runtime in order for them to be called when a work block is received. This is accomplished using export statements which usually come at the end of the accelerator source code. The listing below presents a typical layout for an accelerator task.

Example 4-74 Export statements for accelerator functions

```
...
#include <alf_accel.h>
....
int foo_comp_kernel(..)
{
// statements here...
}
int foo_input_prepare(...)
{
// statements here
}
int foo_output_prepare(...)
{
// statements here
}
int foo_context_setup(...)
{
// statements here
}
int foo_context_merge(...)
{
// statements here
}
ALF_ACCEL_API_LIST_BEGIN
   ALF_ACCEL_EXPORT_API(“foo_compute”,foo_comp_kernel);
   ALF_ACCEL_EXPORT_API(“foo_input_prepare”,foo_input_prepare);
   ALF_ACCEL_EXPORT_API(“foo_output_prepare”,foo_output_prepare);
   ALF_ACCEL_EXPORT_API(“foo_context_setup”,foo_context_setup);
   ALF_ACCEL_EXPORT_API(“foo_context_merge”,foo_context_merge);
ALF_ACCEL_API_LIST_END
```

It is important to understand that we do not write a `main()` program for the accelerator task. It's the ALF runtime that runs the `main()` function and which calls our functions upon receiving a work block.

Tasks and task descriptors

A task is a ready-to-be-scheduled instantiation of an accelerator program : an SPE here. The tasks are created and finalized on the host program. A task is created by the `alf_task_create()` call. Before calling the task creation routine, we need to describe it and this is done by setting the attributes of a task descriptor. We present the task descriptor Figure 4-75 below using a sample of pseudo C code.

Example 4-75 What is a task descriptor?

```
//

struct task_descriptor {

   // The task context buffer holds status data for the task.
   // It is loaded at task started time and can be copied back
   // at task unloading time. It's meant to hold data that is
   // kept across multiple invocations of the computational kernel
   // with different work blocks
   struct task_context {
      task_context_buffer [SIZE];
      task_context_entries [NUMBER];
   };

   // These are the names of the functions that this accelerator task
   // implements. Only the kernel function is mandatory. The context
   // setup and merge, if specified, get called upon loading and
   // unloading of the task. The input and output data transfer list
   // routines are called when the accelerator does the data
   // partitioning itself.
   struct accelerator_image {
      char *compute_kernel_function;
      char *input_dtl_prepare_function;
      char *output_dtl_prepare_function;
      char *task_context_setup_function;
      char *task_context_merge_function;
   };

   // Where is the data partitioning actually run ?
   enum data_partition
      {HOST_DATA_PARTITION,ACCEL_DATA_PARTITION};
```

```
   // Work blocks. A work block has an input and output buffer.
   // These can overlap if needed. A block can also have parameters
   // and a context when the block is a multi-use block
   struct work_blocks {
      parameter_and_context_size;
      input_buffer_size
      output_buffer_size;
      overlapped_buffer_size;
      number_of_dtl_entries;
   };

   // This is required so that the ALF runtime can work out
   // the amount of free memory space and therefore how much
   // multi-buffering can be done.
   accelerator_stack_size;
};
```

The task context is a memory buffer that is used for two purposes:

- store persistent data across work blocks. For example, we would load some state data that is to be read every time we process a work block. It contains work block “invariants”,
- store data that can be reduced between multiple tasks. The task context can be used to implement all-reduce (associative) operations like min, max or a global sum.

Work blocks

A work block is a single invocation of a task with a given set of input, output and parameter data. There are single-use work blocks, which are processed only once and multi-use work blocks which are processed up to `total_count` times.

The single-use work blocks input and output data description can be performed either at the host level or the accelerator level. For multi-use work blocks, the data partitioning is always performed at the accelerator level. The current count of the multi-use work block and the total count are passed as arguments every time we call the input list preparation routine, the compute kernel and the output data preparation routine.

With the multi-use blocks, the work block creation loop that was running on the host task is now performed jointly by all the accelerator tasks that this host has allocated. The only information that a given task is given to create, on the fly, the proper input and output data transfer lists is the work block context buffer, the current and total counts. The previous host loop is now parallelized across the

accelerator tasks which balance the work automatically between themselves. The purpose of the multi-use work blocks is to make sure that the PPE, which runs the host tasks, does not become a bottleneck, too busy creating work blocks for the SPEs.

The work blocks queue is managed by the ALF runtime which balances the work across the accelerators. There are API calls to influence the way the ALF work blocks are allocated if the default mechanism is not satisfactory.

Data partitioning

The purpose of data partitioning is to make sure each work block gets the right data to work with. The partitioning can be performed at the host level or at the accelerator level. We use a data transfer list, consisting of multiple entries of type <start address, type, count> that describe where from, in host memory, we need to gather data to be sent to the accelerators. The API calls differ whether you use the host or accelerator data partitioning.

Datasets

At the host level, we can create datasets which assign attributes to the data buffers that are to be used by the accelerator tasks. A memory region can be described as read-only, read-write or write-only. This information gives hints to the ALF runtime to help improve the data movement performance and scheduling.

The memory layout of an accelerator task

Memory on the accelerator is a limited resource. It is important to understand how the various data buffers are organized to be able to tune their definitions and usage. Also, the ALF runtime will have more opportunities to use clever multi-buffering techniques if it has more room left after the user data has been loaded into the accelerator memory. The picture below shows the memory map of an ALF program. The user code contains the computational kernels and optionally the input/output data transfer list functions and context setup/merge functions.

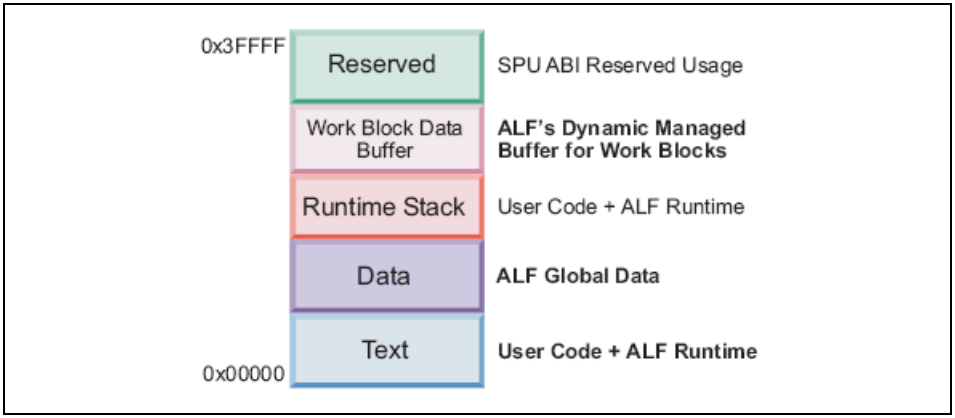


Figure 4-14 ALF accelerator memory map

As seen from the programmer's five pointers to data buffers are passed to the computational kernels (see Example 4-73 on page 296). Various combinations are possible, depending on the use of overlapped buffers for input and output. In the simplest case, no overlap exists between the input and output buffers.

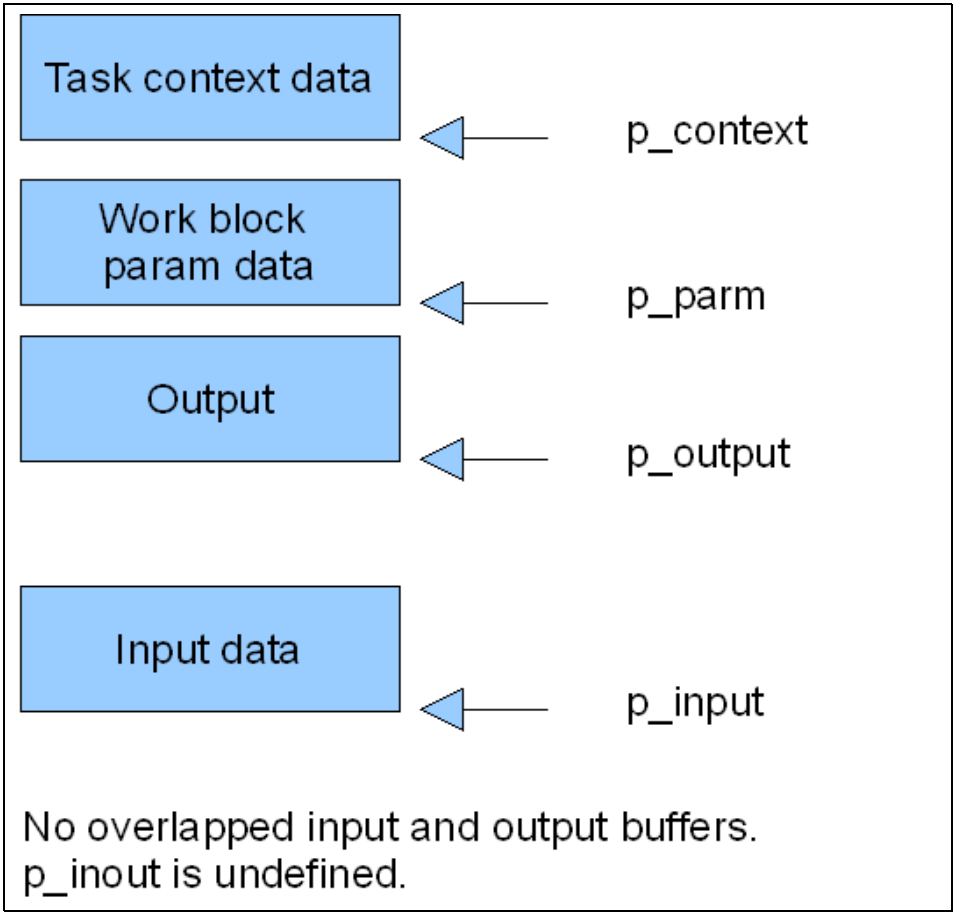


Figure 4-15 Memory map without overlapped input/output buffer

The input and output data buffers can overlap entirely.

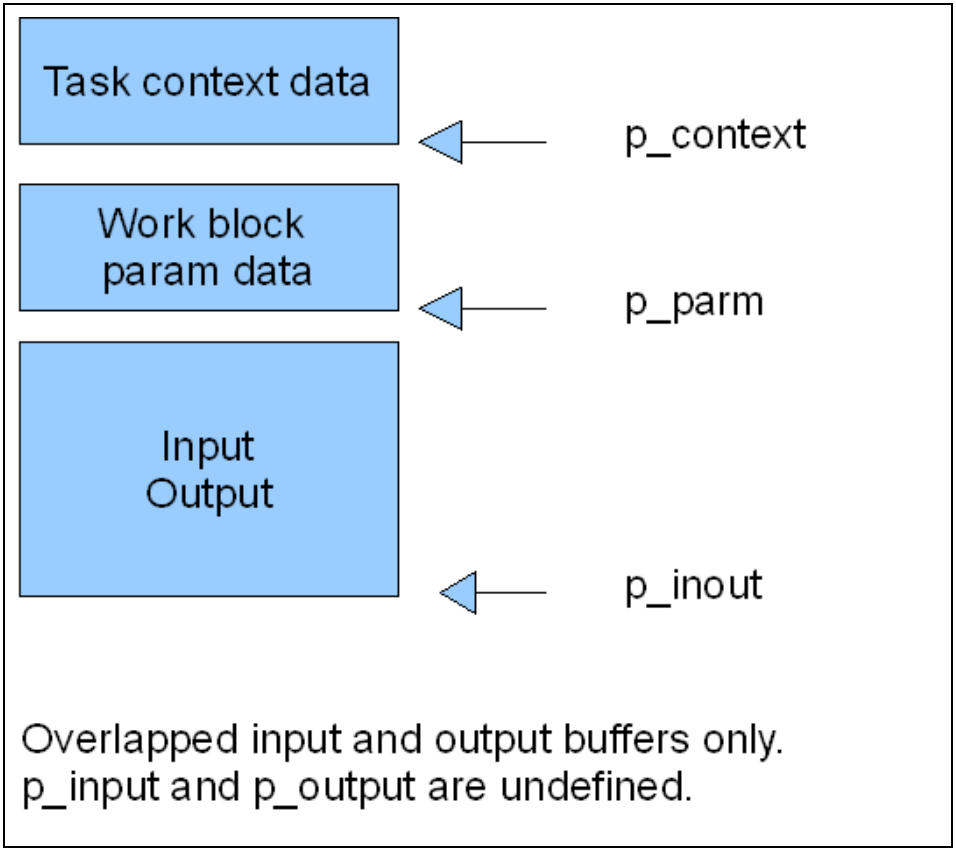


Figure 4-16 Memory map with a single input/output overlapped buffer

In the most general case three data buffer pointers can be defined.

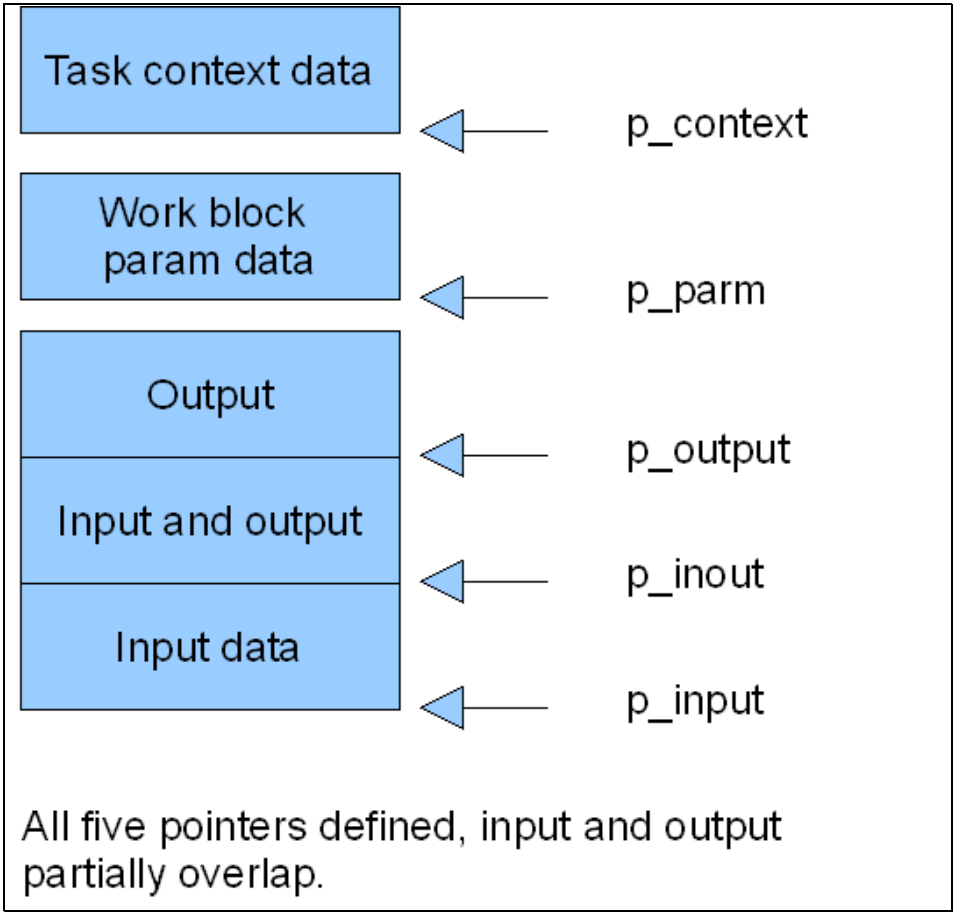


Figure 4-17 Memory map with all five data pointers defined

API description

The API has two components: the host API and the accelerator. A host program must include the alf.h file and an accelerator program must include the `alf_accel.h` file.

The main host API functions are listed in the table below.

Table 4-26 The host side of the ALF API

Groups	Functions	Description
Framework	alf_init alf_query_system_info alf_num_instances_set alf_exit	initialize ALF query various system infos set the max number of accelerators exit ALF
Tasks	alf_task_desc_create alf_task_desc_set_int32 alf_task_desc_set_int64 alf_task_desc_ctx_entry_add alf_task_desc_destroy alf_task_create alf_task_finalize alf_task_destroy alf_task_wait alf_taskdepends_on	create a task descriptor set a task descriptor parameter set a task descriptor parameter 64 bit add an entry in the task context destroy the context create a task make the task runnable terminate a task wait for a task terination express a task dependency
Work blocks	alf_wb_create alf_wb_parm_add alf_wb_dtl_begin alf_wb_dtl_entry_add alf_wb_dtl_end alf_wb_enqueue	create a work block add a work block parameter start a data transfer list add an entry to the list close the data transfer list queue the work block for execution
Datasets	al_dataset_create alf_dataset_buffer_add alf_task_dataset_associate alf_dataset_destroy	create a dataset structure add a buffer and type to the dataset associate the dataset with a task destroy the dataset structure

The accelerator API is much leaner as it only includes a few functions to perform the data partitioning.

Table 4-27 The accelerator side of the ALF API

Groups	Functions	Description
Framework	alf_accel_num_instances alf_accel_instance_id	number of accelerators my rank
Data partitioning	ALF_ACCEL_DTL_BEGIN ALF_ACCEL_DTL_ENTRY_ADD ALF_ACCEL_DTL_ENDt	start a data transfer list add to the list close the list

ALF optimization tips

Apart from tuning the computational kernel itself and making sure we maximize the amount of work per data communication, it can be beneficial to tune the data movement part. To do so, the following techniques should be explored:

- data partitioning on the accelerator side,
- multi-use work blocks.

These techniques will lower the workload on the host task which may otherwise not be able to keep up with the speed of the accelerators, thus becoming a bottleneck for the whole application. Also, using datasets on the host side and using overlapped input and output buffers whenever possible will give more flexibility to the ALF runtime to optimize the data transfers.

ALF application development notes

When designing an ALF strategy for an application, a trade-off will be necessary to decide on the granularity of the computational kernels. The forces are:

- the ability to extract independent pieces of work,
- the computation to communication ratio,
- the memory constraints imposed by the SPE

From an application development perspective, the host-accelerator model allows two different types of programmers to work on the same project. Developer Sam can concentrate on the high level view, implementing the application algorithm, managing MPI tasks, making sure they synchronize and communicate when needed. Developer Sam is also responsible for writing the “contract” between the task and the accelerator tasks, describing the input and output data as well as the required operation. Here comes developer Gordon who will focus on implementing the computational kernels according to the specs and tune these kernels to the metal.

The examples described below show the type of work that is involved when accelerating applications with ALF. Of particular interest in this respect are the `matrix_add` and `matrix_transpose` examples.

A guided tour of the ALF examples provided in the SDK

Before embarking on the acceleration of an application using ALF, it is highly advisable to take a look at the examples provided by the IBM SDK for Multicore Acceleration. The examples come with two rpms:

- alf-examples-source,
- alf-hybrid-examples-source

The hybrid version includes the non-hybrid ones too. The examples are described in the table below following a reading suggested order.

Table 4-28 ALF examples in the IBM SDK for Multicore Acceleration

Example	Description
hello_world	very simple, minimal ALF program
matrix_add	This example gives the steps that were taken to enable and tune this application with ALF. The successive versions are: ► scalar : the reference version ► host_partition : first ALF version, data partitioning on the host, ► host_partition_simd : the compute kernel is tuned using SIMD, ► accel_partition : data partitioning performed by the accelerators, ► dataset : use of the dataset feature, ► overlapped_io : use of overlapped input and output buffers
PI	shows the use of task context buffers for global parameters and reduction operations
pipe_line	shows the implementation of a pipeline using task dependencies and task context merge operations
FFT16M	shows multi-use work blocks and task dependencies
BlackScholes	pricing model: shows how to use multi-use work blocks
matrix_transpose	like matrix_add, shows the steps going from a scaler version to a tuned ALF version. The successive versions are: ► scalar : the reference version ► STEP1a : using ALF and host data partitioning ► STEP1b : using accelerator data partitioning ► STEP2 : using a tuned SIMD computational kernel
inout_buffer	shows the use of input/output overlapped buffers
task_context	shows the use of the task context buffer for associative reduction operations (min, max), a global sum and as a storage for a table lookup.
inverse_matrix_ovl	shows the use of function overlay, datasets

4.7.3 Domain-specific libraries

This chapter discuss few of the main domain specific libraries which are part of SDK 3.0. These libraries aim to assist Cell BE programmers by providing reusable functions that implement a set of common a algorithms and mathematical operators. The functions are optimized specifically to Cell BE by exploiting the unique architecture of this processor (e.g. run parallel on several SPEs, use SIMD instructions).

A software developer who start developing an application for Cell BE (or port an existing application) may first check whether some parts of its application are already implemented in one of the SDK's domain specific libraries. If yes, using the corresponding library can provide an easy solution for and save development efforts.

Most of those libraries are open source, so even if the exact functionality required by the developed application is not implemented, the programer can use those functions as a reference and be customized and tailored for developing the application specific functions.

The next four chapters provide a brief descriptions of the following libraries:

- "Fast Fourier Transform (FFT) library"
- "Monte Carlo libraries"
- "Basic linear algebra subprograms (BLAS) library"
- "Matrix, large matrix and vector libraries"

While those are the libraries that we found as the most useful, SDK3.0 provides also several other libraries. The document "Example Library API Reference" discuss the additional libraries and also a provide detailed description of some of the libraries which are discussed in this chapter (and are described only briefly).

Fast Fourier Transform (FFT) library

This prototype library handles a wide range of FFTs, and consists of the following:

1. API for the following routines used in single precision:
   - 1D of 2D FFT
   - 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 Cell BE FFT library by calling one of the streaming functions. An SPU version is also available.

2. Power-of-two-only 1D FFT code for complex-to-complex single and double precision processing. Supported on the SPU only.

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). The latter batch mode is more efficient in applications in which several distinct FFT operation may be executed one after the other since the initialization step and the termination step are done only once for all the FFT execution (initialization - before the first execution, termination - after the last execution).

Both FFT transform and inverse FFT transform are supported by this library.

In order to retrieve more information about this library the programer may:

- Enter the command `man /opt/cell/sdk/prototype/usr/include/libfft.3` on a system where the SDK is installed.
- Read *Fast Fourier Transform (FFT) library* chapter in “Example Library API Reference” document.

Another alternative library that implements FFT for Cell BE is the FFTW library. the Cell BE implementation of this library is currently available only as an alpha preview release.
More information can be find in this link: http://www.fftw.org/cell/

Monte Carlo libraries

The Monte Carlo libraries are a Cell BE 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 library contains 4 random number generation (RNG) algorithms (hardware-generated, Kirkpatrick-Stoll, Mersenne Twister, and Sobol), 3 distribution transformations (Box-Muller, Moro’s Inversion, and Polar Method), and two Monte Carlo simulation samples (calculations of pi and the volume of an n-dimensional sphere).

A detailed description of this library and how to use it is in “Monte Carlo Library API Reference Manual” document.

Basic linear algebra subprograms (BLAS) library

The BLAS (Basic Linear Algebra Subprograms) library is based upon a published standard interface (See the “BLAS Technical Forum Standard” document) 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 API is available as standard ANSI C and standard FORTRAN 77/90 interfaces. BLAS implementations are also available in open-source (netlib.org). Based on its functionality, BLAS is divided into 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.

Each routine has four versions – real single precision, real double precision, complex single precision and complex double precision. The BLAS library in SDK3.0 supports only real single precision and real double precision 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.

Some of theses routines have been optimized using the Synergistic Processing Elements (SPEs) and these show a marked increase in performance in comparison to the corresponding versions implemented solely on the PPE.

These optimized routines have an SPE interface in addition to the PPE interface; however, the SPE interface does not conform to the standard BLAS interface and provides a restricted version of the standard BLAS interface. The single precision versions of these routines have been further optimized for maximum performance using various features of the SPE (e.g. SIMD, Dual Issue, etc.):

Level 1:

- SSCAL, DSCAL
- SCOPY, DCOPY
- ISAMAX, IDAMAX
- SAXPY, DAXPY
- SDOT, DDOT

Level 2:

- SGEMV, DGEMV (TRANS='No Transpose' and INCY=1)

Level 3:

- SGEMM, DGEMM
- SSYRK, DSYRK (Only for UPLO='Lower' and TRANS='No transpose')
- STRSM, DTRSM (Only for SIDE='Right', UPLO='Lower', TRANS='Transpose' and DIAG='Non-Unit')

A detailed description of this library and how to use it is in "Basic Linear Algebra Subprograms Programmer's Guide and API Reference" document.

Matrix, large matrix and vector libraries

SDk3.0 provides three libraries that implement various linear operation on matrixes and vectors.

The first is matrix library which consists of various utility libraries that operate on 4x4 matrices as well as quaternions. The library is supported on both the PPE and SPE. In most cases, all 4x4 matrices are maintained as an array of 4 128-bit SIMD vectors, while both single precision and double precision operands are supported.

The second is large matrix The large matrix library which consists of various utility functions that operate on large vectors as well as large matrices of single precision floating-point numbers. The size of input vectors and matrices are limited by SPE local storage size. This library is currently only supported on the SPE.

The two libraries support different matrix operations such as multiplying, adding, transpose and inverse.

Similar to SIMDmath and MASS libraries the libraries can be used either as linkable library archive or as set of inline function headers. for more details see e Chapter , "SIMDmath library" on page 257 and Chapter , "MASS and MASSV libraries" on page 258.

The third is vector library which consists of a set of general purpose routines that operate on vectors. This library is supported on both the PPE and SPE.

A detailed description of those libraries and how to use them is in "Example Library API Reference" document, under *Matrix library* chapter, *Large matrix library* chapter, and *Vector library* chapter.

4.8 Programming guidelines

This chapter provide a general collection of programming guidelines and tips that covers different aspects of Cell BE programming.

This chapter is heavily rely on information from the following resources:

1. “Maximizing the power of the Cell Broadband Engine processor: 25 tips to optimal application performance”, Daniel A. Brokenshire, IBM Austin
2. “Cell BE Programming Gotchas! or “Common Rookie Mistakes”, Michael Perrone, IBM TJ Watson Research Center
3. Cell Broadband Engine Programming Tutorial, chapter *General SPE programming tips*.
4. Cell Broadband Engine Programming Handbook, chapter *SPE Programming Tips.*

Two other good sources of information for high performance programming in Cell BE are:

- Cell Be forum at developerworks: http://www.ibm.com/developerworks/forums/forum.jspa?forumID=739
- Cell Performance web site: http://www.cellperformance.com

4.8.1 General guidelines

- Off-load as much work onto the SPEs as possible. Use the PPE as the control processor and SPE to perform all the heavy computational lifting.
- Exploit Cell BE parallelism:
  - Use multithreading so as parallel task run on separate SPE threads.
  - Try to avoid using more threads then physical SPEs because context switching consumes a fair amount of time.
  - Do not spawn SPE threads for each scalar code hot spot since thread creation overhead reduces performance.
- Choose a partitioning and work allocation strategy that minimizes atomic operations and synchronization events.
- Use the appropriate programming model to your application. See for more in Chapter 3.3, “Which parallel programming model ?” on page 51.
- Choose the fixed point data types carefully.

  - Floating point: As most processors the SPE has better performance when performing SIMD operations on single precision floating point variables compare to double precision ones. On the other hand double precision operations are more accurate. It is therefore recommnded to use double precision types only in cases the accuracy of single precision is not sufficient.
  - Fixed point: Similarly to floating point, 32 bit integers have better performance then 64 bit ones. In addition, specifically for multiply 16 bit fixed point integers have better performance then 32 bit ones.

- Use the `volatile` keyword for the declaration of DMA buffers in order to instruct the compiler not to reorder software memory access to those buffers and DMA requests and waiting for completion. Please notice that using the `volatile` keyword can significantly impact the compilers ability to order buffer accesses and coalesce multiply loads.

4.8.2 SPE programming guidelines

This section contains a short summary of programming guidelines for optimizing the performance of SPE programs. The intention here for programing issues related only to programing the SPU itself an without interacting with external components (e.g. PPE, other SPEs, main storage).

Since almost any SPE program does need to interact with external component, it is recommended to be familiar with the programming guidelines in the other chapters in “Programming guidelines” section.

General

- Try to avoid over usage of 128 bytes alignment. Consider on which cases such alignment is indeed essential (e.g. for data transfer that are performed often) and use redundant alignment (e.g. 16 bytes) for other cases. There two main reasons why 128 alignment can reduce the performance:
  - 128 bytes alignment requires the definition of the variables as global which cause the program to use more registers and reducing the number of free registers which will also reduce the performance (e.g. increased loads/stores, increases stack size, reduced loop unrolling). therfore, if only a redundant alignment is required (e.g. 16 bytes alignment), you can use the variables as local which may significantly increase the performance.
  - 128 bytes alignment increase the code size because of the padding that is added by the compiler to make the data aligned.
- Try to avoid writing recursive SPE code that uses lots of stack variables which may cause stack overflow errors. The compilers provides support for runtime

stack overflow checking that can be enabled during application debug such errors.

Intrinsics

- Use intrinsics to achieve machine-level control without the need to write assembly language code.
- Understand how the intrinsics map to assembly instructions and what the assembly instructions do.

Local Store

- Design for the LS size. The LS holds up to 256 KB for the program, stack, local data structures, heap, and DMA buffers. One can do a lot with 256 KB, but be aware of this size.
- In case the code is too big too fit into the LS (taking into account that data should also reside in LS) use the overlays mechanism (see Chapter 4.6.7, “Code transfer using SPU code overlay” on page 276).
- Use code optimization carefully since they may increased the code size (e.g. function inline, loop unrolling).

Loops

- If the number of loop iterations is a small constant, then consider removing the loop altogether.
- If the number of loop iterations is variable, consider unrolling the loop as long as the loop is relatively independent (that is, an iteration of the loop is not dependent upon the previous iteration).
  Unrolling the loops reduce dependencies and increase dual-issue rates, and by that will let the compiler optimization to exploits the large SPU register file.

SIMD programming

- Exploit SIMD programing as much as possible. Doing so may increase the performance of your application in several factors, especially in computation bounded programs.
- Consider using the compiler auto-SIMDizing feature which can convert ordinary scalar code into a SIMD one. Be aware of the compiler limitations and on the code structures that are supported for auto-SIMDizing and try to code according to those limitations and structures. See more in “Auto-SIMDizing by compiler” on page 264.
- Another alternative is to explicitly doing SIMD programming. In order to do so you can use intrinsics, SIMD libraries, and supported data type. See more in “SIMD programming” on page 253.

- Not all the SIMD operations are supported by all data types. Review the operation which are critical to your application and verify in which data types they are supported.
- Choose an SIMD strategy appropriate for the application under development. The two common ones:
  - Array-of-structure (AOS) organization. From programming point of view can have more-efficient code size and simpler DMA needs, but SIMDize is more difficult. From computation point of view it may be less efficient, but it depends on the specific application.
  - Structure-of-arrays (SOA) organization. From programming point of view this organization is usually easier to SIMDize, but the data must be maintained in separate arrays or the SPU must shuffle AOS data into an SOA form.
  - If the data is in AOS, consider runtime converting the AOS data to SOA, performing the calculations, and converting the results back
  - For more details, see “Data organization - AOS versus SOA” on page 261.
- As general:
  - Using the auto-SIMDizing requires less development time but in most cases the performance are inferior compare to explicit SIMD programming (unless the program can perfectly fit into the code structures that are supported by the compiler for auto-SIMDizing).
  - On the contrary, correct SIMD programming will provide the best performance but requires not negligible development effort.

Scalars

- Because SPUs only support quadword loads and stores, scalar loads and stores (less then quadword) are slow, with long latency.
- Aligning the scalar that are often used to quadword address to improve the performance of operations that are done on those scalars.
- Cluster scalars into groups and load multiple scalars at a time using a quadword memory access. Later use extract or insert intrinsics to explicitly move between scalars and vector data types. This will eliminate redundant loads and stores.
- For more details, see “Using scalars and converting between different vector types” on page 271

Branches

- Eliminate nonpredicted branches using select bits intrinsics (`spu_sel`).

- For branches who are highly predicted, use the `__builtin_expect` directive to explicitly direct branch prediction. Compiler optimization may add the corresponding branch hint in this case.
- Inline functions that are often called by explicitly define them as inline in the program, or use compiler optimization.
- Use feedback-directed optimization, for example using FDPRPro tool.
- For more details, see “Eliminating and predicting branches” on page 277

Multiplies

- Try to avoid integer multiplies on operands greater than 16 bits in size. The SPU supports only a “16-bit x16-bit multiply”. A “32-bit multiply” requires five instructions (three 16-bit multiplies and two adds).
- Keep array elements sized to a power-of-2 to avoid multiplies when indexing
- When multiply or dividing by a factor which is power-of-2, instead of using the ordinary operators use the shift operation (corresponding intrinsics for vectors and ‘<<’ and ‘>>’ operator for scalars).
- Cast operands to unsigned short prior to multiplying. Constants are of type int and also require casting. Use a macro to explicitly perform 16-bit multiplies. This can avoid inadvertent introduction of signed extends and masks due to casting.

Dual-Issue

- Choose intrinsics carefully to maximize dual-issue rates or reduce latencies.
- Dual issue will occur if a pipe-0 instruction is even-addressed, a pipe-1 instruction is odd-addressed, and there are no dependencies (operands are available).
- Manually insert nops to align instructions for dual-issue in case writing non-optimizing assembly programs. In other cases the compilers automatically insert nops when needed.
- Use software pipeline loops to improve dual-issue rates (described in Loop Unrolling and Pipelining chapter in Cell Broadband Engine Programming Handbook).
- Understand the fetch and issue rules to maximize dual-issue rate. Those rules are described in chapter SPU Pipelines and Dual-Issue Rules in Cell Broadband Engine Programming Handbook).
- Avoid over usage of the odd pipeline for load instructions which may cause instruction starvation. This can happen for example for large matrix transpose on SPE when there are lots of loads on odd pipeline and minimal usage of even pipeline for computation. Similar case can happen for dot product of

large vectors.In order to solve it the programer may add more computation on each data being loaded.

4.8.3 Data transfers and synchronization guidelines

This section contains a short summary of programming guidelines for performing efficient data transfer and synchronization on Cell BE program.

- Choose the transfer mechanism that fits your application data access pattern:
  - If pattern is predictable (e.g. sequentially access array or matrix) use explicit DMA requests to transfer data (may be implemented with SDK core libraries functions).
  - If pattern is random or unpredictable (e.g. sparse matrix operation) consider using the software manages cache, especially if the is high ratio of data re-use (e.g. the same data or cache line is used in different iterations of the algorithm).
- When the core libraries for explicitly intuiting DMA transfer:
  - Follow the supported and recommended value for DMA parameters (see “Supported and recommended values for DMA parameters” on page 115 and “Supported and recommended values for DMA-list parameters” on page 116)
  - DMA throughput is maximized if transfers are at least 128 bytes, and transfers greater than or equal to 128 bytes should be cache-line aligned (aligned to 128 bytes). This stands for the data transfer size, as well as to the source and destination addresses.
  - Overlap DMA data transfer with computation using double buffering or multibuffering mechanism (see “Efficient data transfers by overlapping DMA and computation” on page 157).
  - Minimize small transfers. Transfers of less than one cache line consume bus bandwidth equivalent to a full cache-line transfer.
- When explicitly using software managed cache try to exploit it unsafe asynchronous mode as it can provide significant better results then the unsafe synchronous mode. A double mechanism may also implemented using this safe mode.
- Uniformly distribute memory bank accesses. The Cell BE memory subsystem has 16 banks, interleaved on cache line boundaries. Addresses 2KB apart access the same bank. System memory throughput is maximized if all memory banks are uniformly accessed
- Use SPE-initiated DMA transfers rather than PPE-initiated DMA transfers. There are more SPEs than the one PPE, and the PPE can enqueue only

eight DMA requests whereas each SPE can enqueue 16. In addition, the SPE is also much more efficient at enqueuing DMA requests.

- Use of a kernel with large 64KB base pages to reduce page table and TLB thrashing. If significantly large data set are accessed, consider using huge pages instead (see “Improving page hit ratio using huge pages” on page 163).
- For applications that are memory bandwidth-limited consider using NUMA (see “Improving memory access using NUMA” on page 168). Following are two common cases when using NUMA is recommended in a system of more then one Cell BE node (such as QS20 and QS21):
  - Only one Cell BE node is used. Since the access latency is slightly lower on node 0 (Cell BE 0) as compared to node 1 (Cell BE 1) it is advised to use NUMA to allocate memory and processor on this node.
  - More then one Cell BE node is used (e.g. use the two nodes of QS21) and the data and tasks execution can be perfectly divided between nodes. In that case NUMA can be used to allocate memory on both nodes and exploit the aggregated memory bandwidth (processor on node 0 primarily access memory on this node, and the same for node 1).
- DMA transfers from main storage have high bandwidth with moderate latency, whereas transfers from the L2 have moderate bandwidth with low latency. For that reason, considerate the effect of whether the updated data is stored in the L2 versus on the main memory:
  - When SPEs access large data set make sure that it is not on the L2. This can be done for example by making sure the PPE does not access the data set before the SPEs do so.
  - When the SPEs and PPE need to share short messages (e.g. notification or status) it is recommended to do so on the L2. Doing so can be done for example by the PPE accessing the data before the SPEs which will ensure that the system will create a copy of this data on the L2.
  - You can also have control over the L2 behavior using the `__dcbf` function to flush a data cache block and `__dcbst` function to store data cache block in cache.
  - However, most applications are better off not trying to over manage the PPE cache hierarchy.
- Exploit the on-chip data transfer and communication mechanism by using LS to LS DMA transfers when sharing data between SPEs and utilizing mailboxes, signal notification registers for small data communications and synchronization. The reason for that is that the EIB provides significantly more bandwidth than system memory (in the order of 10 or more).
- Be aware the when the SPEs receive a DMA ‘put’ data transfer completion it only means that the local MFC completed the transaction from its side but not

unnecessarily that the data is already stored in memory. So it may not be accessible yet for other processors.

- Use explicit command to force data ordering when sharing data between SPEs and PPE and between SPEs to themselves because Cell BE architecture does not guarantees such ordering between the different storage domain (while coherency is guaranteed on each of the memory domain separately):
  - Be aware that the DMA can be re-ordered compare to the order in which the SPE program initiate the corresponding DMA commands. Explicit DMA ordering command must be issued to force ordering.
  - Use fenced or barriered DMA command to order DMA transfers within a tag group.
  - Use barrier command to order DMA transfers within the queue.
  - Minimize the use of such ordering command as they have negative effect on the performance.
  - See more at "Shared storage synchronizing and data ordering" on page 213 with some practical scenarios at "Practical examples using ordering and synchronization mechanisms" on page 235
- Use affinity to improve the communication between SPEs (e.g. LS to LS DMA data transfer, mailbox, signals). See "Creating SPEs affinity using gang" on page 93 for more.
- Minimize the use of atomic, synchronizing, and data-ordering commands as they may add significant overhead.
- Atomic operations operate on reservation blocks corresponding to 128-byte cache lines. As a result, synchronization variables should be placed in their own cache line so that other non-atomic loads and stores do not cause inadvertent lost reservations.

4.8.4 Inter-processor communication

- Some recommended methods for inter-processor communication:
  - PPE-to-SPE:
    - PPE writes to SPE inbound mailbox
    - SPE perform blocking read of its inbound mailbox
  - SPE-to-PPE
    - SPE writes to system memory (which will also invalid the corresponding cache line in L2).
    - PPE polls L2

  - SPE-to-SPE
    - SPE writes to a remote SPE's inbound mailbox or signal notification registers or LS.
    - SPE polls it inbound mailbox or signal notification registers or LS.
- Avoid PPE waiting for SPEs to complete by polling the SPE outbox mailbox.

5

Programming Tools and Debugging Techniques

In this chapter we introduce and explore the plethora of available development tools for the Cell/B.E.™ architecture, with special attention to the potential tool management issues that an heterogeneous architecture may arise.

For the exact same reasons, we dedicate a great portion of this chapter to the debugger and available debugging techniques and/or aids, with focus on both error detection and performance analysis.

The following topics are discussed:

5.1 Tools Taxonomy and basic Time line approach.

The design of the Cell Broadband Engine (CBE) presents many challenges to software development. With nine cores, multiple ISA's and non-coherent memory, the CBE imposes challenges to compiling, debugging and performance tuning.

For those reasons, understanding a few basic concepts on how tools interlock with the Operating System, can become key to succeed.

5.1.1 Dual Toolchain

The Cell Broadband Engine (CBE) processor is a heterogeneous multiprocessor not only because the SPEs and the PPE have different architectures but also because they have disjoint address spaces and different models of memory and resource protection. The PPE can run a virtual-memory operating system, so it can manage and access all system resources and capabilities. In contrast, the synergistic processor units (SPUs) are not intended to run an operating system, and SPE programs can access the main-storage address space, called the effective address (EA) space, only indirectly through the DMA controller in their memory flow controller (MFC). The two processor architectures are different enough to require two distinct tool chains for software development.

ABI

Application Binary Interface establishes the set of rules and conventions to ensure portability of code and compatibility between code generators, linker and runtime libraries. Typically, the ABI states about data types, register usage, calling conventions and object formats.

The tool chains for both the PPE and SPE processor elements produce object files in the ELF format. ELF is a flexible, portable container for re-locatable, executable, and shared object (dynamically linkable) output files of assemblers and linkers. The terms PPE-ELF and SPE-ELF are used to differentiate between ELF for the two architectures. CESOF is an application of PPE-ELF that allows PPE executable objects to contain SPE executables.

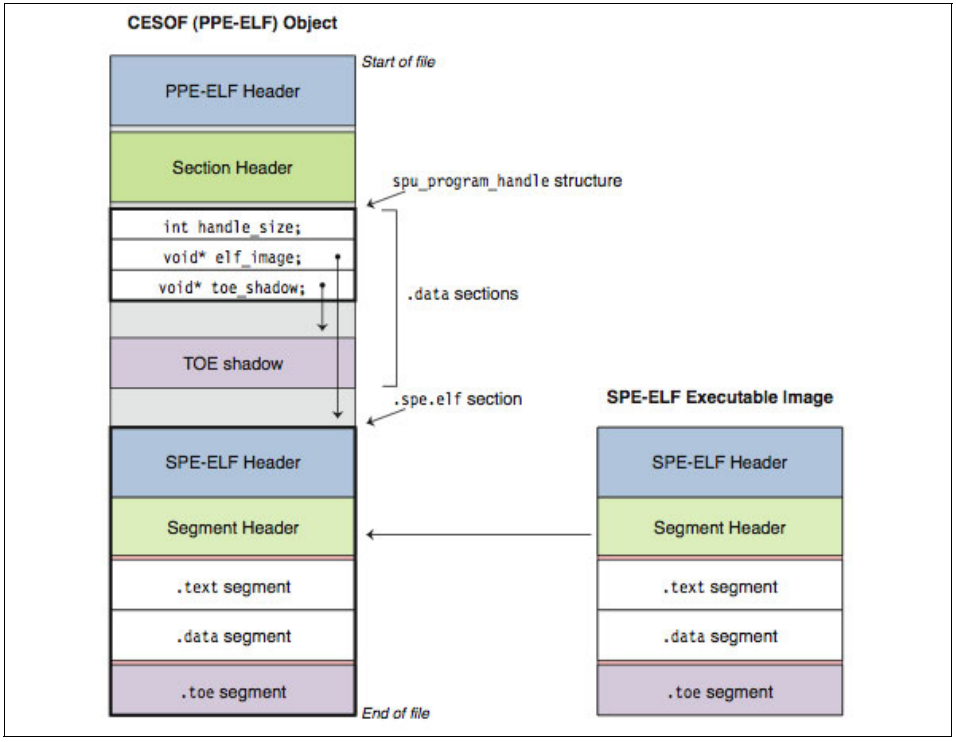


Figure 5-1 CESOF layout

To ease the development of combined PPE-SPE multiprocessor programs, the CBE operating- system model uses CESOF and provides SPE process-management primitives. Though programmers often keep in mind a heterogeneous model of the CBE processor when dividing an application program into concurrent threads, the CESOF format and, for example, the Linux operating-system thread application programming interfaces (APIs) allow programmers to focus on application algorithms instead of managing basic tasks such as SPE process creation and global variable sharing between SPE and PPE threads. It's important to observe that, from an application developer's point of view, such mechanism also enables the access of PPE variables from SPE code.

5.1.2 Typical Tools Flow

The typical tools usage pattern should be similar to the following:

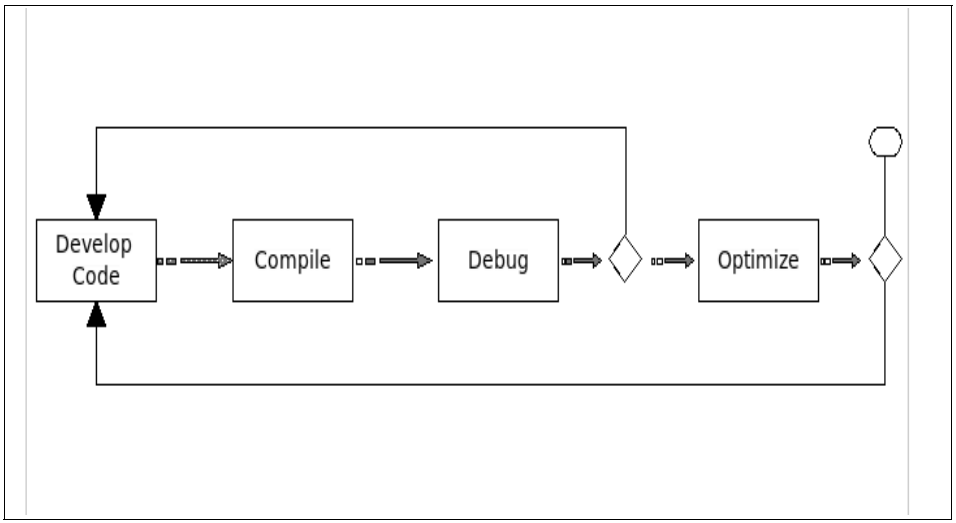


Figure 5-2 Typical development cycle

Throughout the remainder of this chapter, besides exploring each tools' relevant characteristics, you will be presented with similar flows as above, relating the development cycle and where each tool fits on it. The intent is to guide you where and when to use each tool.

5.2 Compiling and Building

On the next few sections we explore the Cell BE SDK 3.0 capabilities in compiling and optimizing executables and managing the build process.

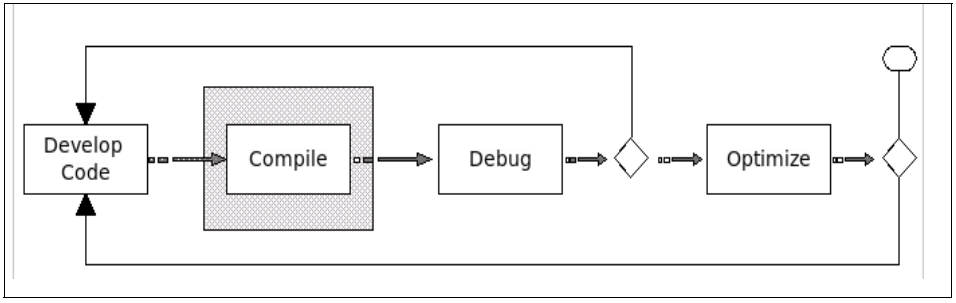


Figure 5-3 Compile

5.2.1 Compilers: gcc

The GNU tool chain contains the GCC C-language compiler (GCC compiler) 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.

This release of the GNU tool chain includes a GCC compiler and utilities that optimize code for the Cell BE processor. These are:

- The spu-gcc compiler for creating an SPU binary
- The ppu-embedspu (and ppu32-embedspu) tool which enables an SPU binary to be linked with a PPU binary into a single executable program
- The ppu-gcc (and ppu32-gcc) compiler

Basics

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.

Example 5-1 Compiling SPE code

```
#/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.

Example 5-2 Embedding SPE code

```
#/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.

Example 5-3 Linking

```
#/usr/bin/ppu32-gcc -g -o simple simple.c simple_spu-embed.o -lspe2
```

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.

Example 5-4 Implicitly linking

```
/usr/bin/ppu32-gcc -g -o simple simple.c simple_spu -lspe2
```

Fine Tuning: Command line options

Each of the GNU compilers offers Cell BE relevant options, whether to enable architecture specific options, or to further optimize the generated binary code.

ppu-gcc:

-mcpu=cell	Selects the instruction set architecture to generate code for, Cell/B.E. or PowerXCell. Code compiled with -march=celledp may make use of new instructions, and will not run on Cell/B.E. -march=celledp also implies -mtune=celledp.
-m32	Selects 32bit option. The ppu32-gcc defaults to 32bit.
-m64	Selects 64bit option. The ppu-gcc defaults to 64bit.
-maltivec	Enables generation of code that uses Altivec vector instructions (default in ppu-gcc)

spu-gcc:

-march=cell | celledp	Selects between the Cell BE architecture and the PowerXCell architecture, as well as its registers, mneumonics and instruction scheduling parameters
-mtune=cell | celledp	Tune the generated code for either Cell BE or PowerXCell architecture. Mostly affects instruction scheduling strategy.
-mfloat=accurate | fast	
-mdouble=accurate | fast	
	Selects whether to use the fast fused-multiply operations for floating point operations or, enable calls to library routines that implement more precise operations).
-mstdmain	Provides standard argv/argc calling convention for the main SPU function
-fpic	
-mwarn-reloc	
-merror-reloc	Generates positions independent code, warning if the resulting code requires load-time relocations.
-msafe-dma	
-munsafe-dma	Controls whether loads and stores instructions are not moved past the DMA operations, by the compiler optimizations.
-ffixed-<reg>	

-mfixed-range=<reg>-<reg>

Reserve specific registers for user application.

Language Options

The GNU gcc offers a few language extensions in order to provide access to specific Cell BE architectural features, from a programming point of view.

Vectors	The GNU compiler language support offers vectors data types for both PPU and SPU, as all as arithmetic operations with those. Please refer to 4.6.4, “SIMD programming” on page 253 for more details.
Intrinsics	The full set of AltiVec and SPU intrinsics are available. Please refer to 4.6.2, “SPU instruction set and C/C++ language extensions (intrinsics)” on page 244 for more details.

Optimizations

The GNU compiler offers a few mechanisms to optimize code generation specifically to the underlying architecture.

For a complete reference of all optimization-related options, consult the GCC manual, in particular:

http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Optimize-Options.html

Basics

The -O family of options offer *predefined* generic set of optimizations at hand:

O0 (default):	no optimization shortest compilation time, best results when debugging
-O1 (-O):	default optimization moderately increased compilation time
-O2:	heavy optimization significantly increased compilation time no optimizations with potentially adverse effects
-O3:	optimal execution time may increase code size, may make debugging difficult
-Os:	optimal code size may imply slower execution time than -O3

By default, GCC generates completely unoptimized code. To switch on optimization, use one of the -O, -O2, or -O3 flags above. Usually, -O3 is the best

choice, however it activates some optimizations like automatic inlining that may be undesirable under certain circumstances. In those cases fall back to -O2. There may be cases (very few) where source code was heavily hand-optimized for the SPU, in which even -O2 would generate worse code than just -O. While these cases should be rare, when in doubt it would be good to simply first try that option as well.

Tip: When in doubt, use the following set of compiler options to generate optimized code for SPE:

-O3 -funroll-loops -fmodulo-sched -ftree-vectorize -ffast-math

Flags for special optimization passes

GCC supports a number of specific optimization passes that are not implemented by default at any optimization level (-O...). These can be selected manually where appropriate. The following is a list of some of these options that might be of particular interest on the SPE.

-funroll-loops	"Unroll" loops by duplicating the loop body multiple times. Can be helpful on the SPE as it reduces the number of branches. On the other hand, the option will increase code size.
-fmodulo-sched	A special scheduling pass (Swing Modulo Scheduling, also known as Software Pipelining) that attempts to arrange instructions in loops so as to minimize pipeline stalls. Usually beneficial on SPE.
-ffast-math	Allows the compiler to optimize floating-point expressions without preserving exact IEEE semantics. For example, allows the vectorizer to change the order of computation of complex expressions.

Compiler directives: Function Inlining

The function inlining is an optimization technique where performance is achieved by replacing function calls by their explicit set of instructions (or body). The actual formal parameters are replaced with arguments.

The benefits of such technique are basically avoiding the function call overhead and keeping the function as a whole for combined optimization. The usual disadvantages are code size increase and compilation time increase.

The compiler offers two choices of function inlining:

- By explicit declaration, with both the “inline” keyword in function declarations and by defining C++ member functions inside class body

Example 5-5 Inline keyword usage

```
...
static inline void swap(int a, int b) {
   int tmp;
   tmp = a;
   a = b;
   b = tmp;
}
```

- Activated by compiler option -finline-functions, where the compiler will apply heuristics over the function call, considering size and complexity.

Compiler directives: Others

See 4.6.3, “Compiler directives” on page 251, for more on the usage of other available compiler directives like volatile, aligned, builtin_expect, align_hint and restrict.

Auto-Vectorization

This feature, enabled by the *-ftree-vectorize* switch, will automatically detect situations in your source code where a loop over scalar instructions can be transformed into a loop over vector instructions. Should be usually beneficial on SPE; in cases where the compiler manages to transform a loop that is performance-critical to the overall application, significant speedup can be observed.

To help the vectorizer detect loops that are safe to transform, you should follow some general rules when writing loops: Use countable loops (known number of iterations), avoid function calls or "break"/"continue" in the loop; Avoid aliasing problems by using the C99 "restrict" keyword where appropriate; Keep memory access patterns simple; Operate on properly aligned memory addresses whenever possible.

If your code has loops that you think should get vectorized, but aren't, you can use the *-ftree-vectorizer-verbose=[X]* option to get some information why this is so. X=1 is least amount of output, X=6 largest.

Please refer to 4.6.5, “Auto-SIMDizing by compiler” on page 264 for more on this topic.

Profile-directed feedback optimization

Although considered an advanced optimization technique, the “Profile Directed Feedback Optimization” allows the compiler to tune generated code according to behavior measured during execution of trial runs.

To use this approach, you should perform the following general steps:

1. First the application needs to be built with the *-fprofile-generate* switch. This will generate an instrumented executable.
2. Next, the generated instrumented application needs to be ran on a sample input data set, resulting in a profile data file.
3. In a second compile step using the *-fprofile-use* switch (instead of -fprofile-generate), the compiler will incorporate feedback from the profile run to generate an optimized final executable.

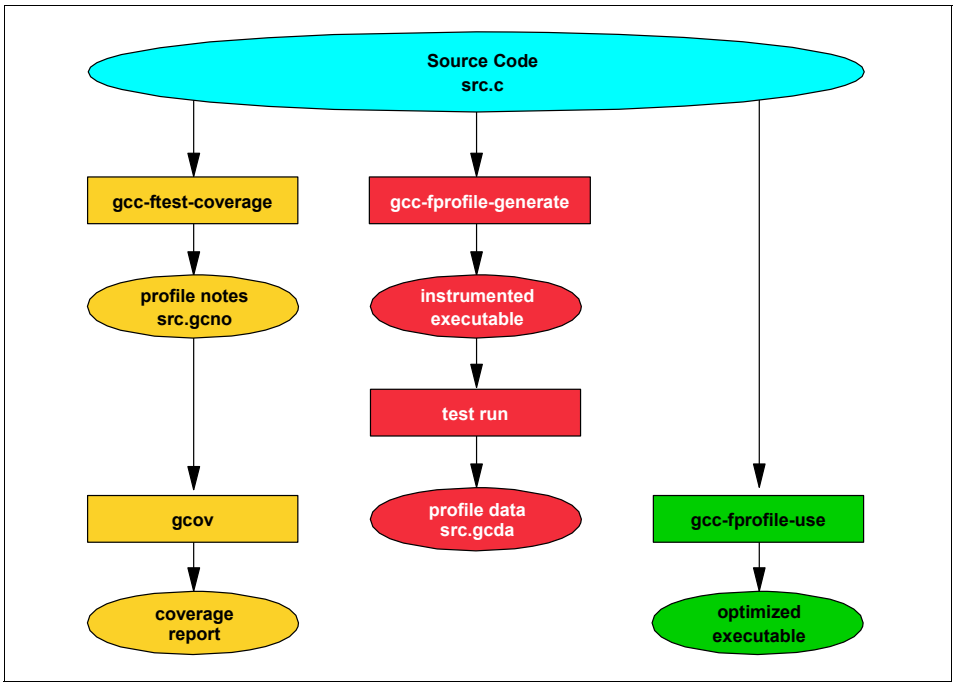


Figure 5-4 Profile Directed Optimization process

5.2.2 Compilers: xlc

IBM XL C/C++ for Multicore Acceleration for Linux is an advanced, high-performance cross-compiler that is tuned for the Cell Broadband Engine Architecture (CBEA). The XL C/C++ compiler, which is hosted on an x86, IBM PowerPC technology-based system, or a BladeCenter QS21, 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.

The full documentation for the IBM XL C/C++ compiler can be found at:

http://www.ibm.com/software/awdtools/xlcpp/library/

Optimization

The IBM XL C/C++ introduces several innovations, specially with regard to the optimization options. Let's go over the general concepts involved and a few useful tips.

Overview

The XL compiler also offers the "predefined" optimizations options as follows:

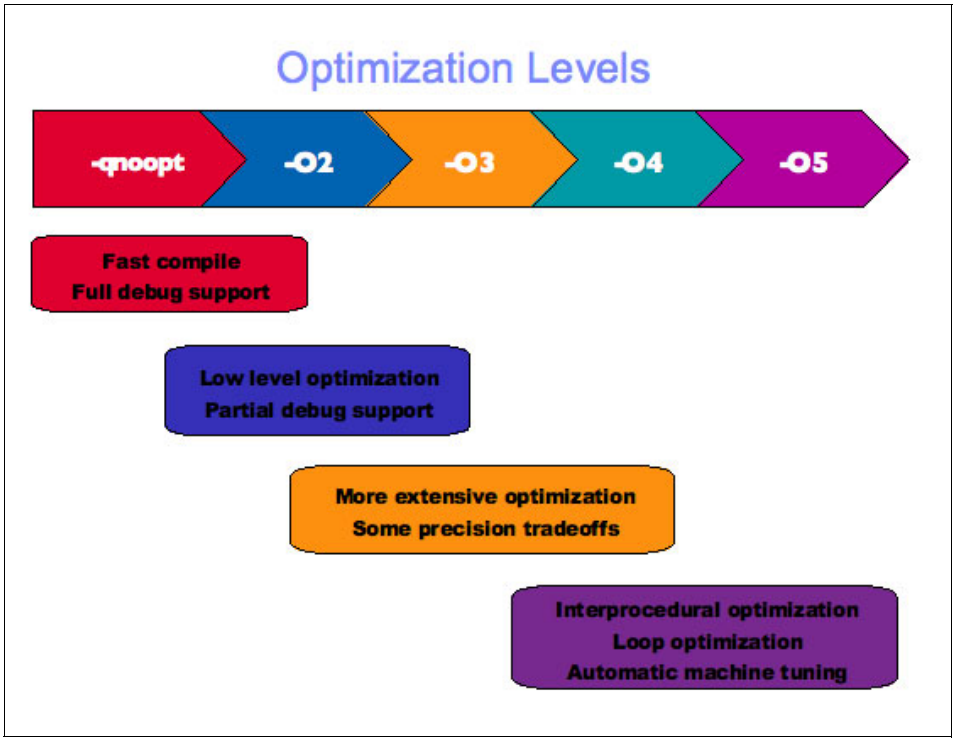


Figure 5-5 XL Optimization levels

Levels 2 and 3

- -O2 level brings comprehensive low level optimizations while keeping partial support for debugging
  - Global assignment of user variables to registers
  - Strength reduction and effective usage of addressing modes

- Elimination of unused or redundant code
- Movement of invariant code out of loops
- Scheduling of instructions for the target machine
- Some loop unrolling and pipelining
- Externals and parameter registers visible at procedure boundaries
- Snapshot™ pragma/directive creates additional program points for storage visibility
- -qkeepparm option forces parameters to memory on entry so that they can be visible in a stack trace

- -O3 level has extensive optimization but may introduce some precision trade-offs
  - Deeper inner loop unrolling
  - Loop nest optimizations such as unroll-and-jam and interchange (-qhot subset)
  - Better loop scheduling
  - Additional optimizations allowed by -qnostrict
  - Widened optimization scope (typically whole procedure)
  - No implicit memory usage limits (-qmaxmem=-1)
  - Reordering of floating point computations
  - Reordering or elimination of possible exceptions (e.g. divide by zero, overflow)

In order to get the most of level 2 and 3 optimizations, there are some general attention points the we need to focus on.

First of all ensure that your code is standard-compliant and also, if possible, test and debug your code without optimization before using -O2.

With regard to the C code, ensure that pointer use follows type restrictions (generic pointers should be char* or void*) and verify if all shared variables and pointers to same are marked volatile.

Try to be uniform, compiling as much of your code as possible with -O2. If you encounter problems with -O2, consider using -qalias=noansi or -qalias=nostd rather than turning off optimization.

Next, use -O3 on as much code as possible. If you encounter problems or performance degradations, consider using –qstrict, -qcompact, or -qnohot along

with -O3 where necessary. If you still have problems with -O3, switch to -O2 for a subset of files/subroutines but consider using -qmaxmem=-1 and/or -qnostrict.

High order transformations (-qhot)

High order transformations are supported for all languages. The usage is specified as:

```
-qhot[=[no]vector | arraypad[=n] | [no]simd]
```

The general optimization gain involve the following areas:

- High level transformation (e.g. interchange, fusion, unrolling) of loop nests to optimize:
  - memory locality (reduce cache/TLB misses)
  - usage of hardware prefetch
  - loop computation balance (typically ld/st vs. float)
- Optionally transforms loops to exploit MASS vector library (e.g. reciprocal, sqrt, trig) - may result in slightly different rounding
- Optionally introduces array padding under user control - potentially unsafe if not applied uniformly
- Optionally transforms loops to exploit VMX/SIMD unit

The -qhot option is designed to be used with other optimization levels, like -O2 and -O3, since it will have neutral effect if no optimization opportunities exist.

Some times you may encounter a long unacceptable compilation time or even performance degradation, which could be solved by the combined use of -qhot=novector, or -qstrict or -qcompact along with -qhot.

As with any other optimization option, try disabling it selectively, if needed.

Link-time Optimization (-qipa)

The XL compiler also offers a “link-time” optimization option:

```
-qipa[=level=n | inline= | fine tuning]
```

The link-time optimization can be enabled per compile unit (compile step) or on the whole program (compile and link), where it expands the reach to the whole final artifact (executable or library).

The following options can be explored by this feature:

- level=0: Program partitioning and simple inter procedural optimization
- level=1: Inlining and global data mapping

- level=2: Global alias analysis, specialization, inter procedural data flow
- inline=: Precise user control of inlining
- fine tuning: Specify library code behavior, tune program partitioning, read commands from a file

Although -ipa works when building executables or shared libraries, make sure you compile main and exported functions also with -qipa. Again, try to apply it as much as possible.

Levels 4 and 5

Optimization levels 4 (-O4) and 5 (-O5) automatically applies all previous optimization level techniques (-O3). Additionally, it includes its own “packages” options:

- -qhot
- -qipa
- -qarch=auto
- -qtune=auto
- -qcache=auto
- in -O5 only,-qipa=level=2

Vectorization (VMX and SIMD)

The XL support two modes for exploitation for the vector features in the Cell BE architecture:

- User driven, where the code is explicitly ported to make use of vector types and intrinsics, as well as alignment constrains.
- Automatic Vectorization (SIMDization), where the compiler tries to automatically identify parallel operations across the scalar code and generates Vector versions of them. The compiler also performs all necessary transformations to resolve any alignment constrains. Requires, at least, optimization level -O3 -qhot.

Although the compiler carries a through analyzes to produce the best fit auto-vectorized code, still the programmer may influence the overall process, making it more efficient. The more relevant tips are:

- Loop structure
  - Inline function calls inside innermost loops
  - Automatically (-O5 more aggressive, use inline pragma/directives)
- Data alignment

- Align data on 16-byte boundaries

  ```
  __attribute__((aligned(16))
  ```

- Describe pointer alignment, which can be placed anywhere in the code, preferably close to the loop

  ```
  _alignx(16, pointer)
  ```

- Use -O5 (enables inter-procedural alignment analysis)

- Pointer aliasing
  - Refine pointer aliasing

    ```
    #pragma disjoint(*p, *q) or restrict keyword
    ```

Obviously, if you already manually unrolled any of the loops, it becomes more difficult to the simdization process. Even in that case, you can manually instruct the compiler to skip those:

```
#pragma nosimd (right before the innermost loop)
```

5.2.3 Building

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.

Table 5-1 make.footer example configurable features

Environment Variable	Description	Example Value
CFLAGS_[gcc|xlc]	Pass additional compilation flags to the compiler	-g -DLIBSYNC_TRACE
LDFLAGS_[gcc|xlc]	Pass additional linker flags to the linker	-Wl,-q -L/usr/lib/trace
CC_OPT_LEVEL	Overrides specifically the compiler optimization level	-O0
INCLUDE	Additional include paths to the compiler	-I/usr/spu/lib
IMPORTS	Additional libraries to be imported by the linker	-lnuma -lpthread

Changing the Environment

Environment variables in the /opt/cell/sdk/buildutils/make.* files are used to determine which compiler is used to build the examples. The /opt/cell/sdk/buildutils/cellsdk_select_compiler script can be used to switch the compiler. The syntax of this command is:

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

where the xlc flag selects the XL C/C++ compiler and the gcc flag selects the GCC compiler. The default, if unspecified, is to compile the examples with the GCC compiler. 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, SPU_COMPILER, PPU_COMPILER, PPU32_COMPILER, or PPU64_COMPILER.

5.3 Debugger

The debugger is a tool to help finding and removing problems in your code. Besides fixing problems, the debugger can also help you understand the program, as it typically gives you memory and registers contexts, stack call traces and step-by-step execution.

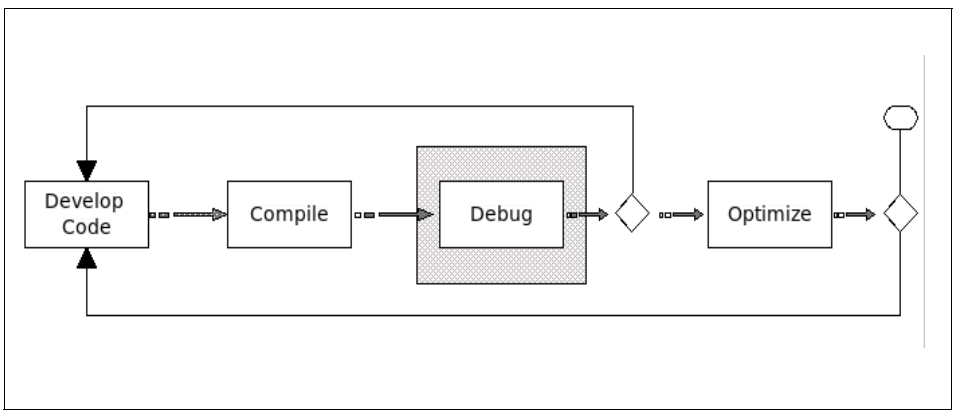


Figure 5-6 Debug

5.3.1 Debugger: 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 BE processor environment and this section describes how to debug Cell BE

software using the new and extended features of the GDBs which are supplied with SDK 3.0. Debugging in a Cell BE processor environment is different from debugging in a multi threaded 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 BladeCenter QS21:

- gdb which is installed with the Linux operating system for debugging PowerPC applications. You should NOT use this debugger for Cell BE 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.

Setup and Considerations

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 information. When you use the top level Makefiles of the SDK, you can specify the -g option on compilation commands by setting the CC_OPT_LVL 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_LVL Makefile variable to -g.

Debugging PPE code

There are several ways to debug programs designed for the Cell BE processor. If you have access to Cell BE 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. 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.

Debugging multi-threaded 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 multi threaded 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.

Debugging architecture

On the Cell BE 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 following command:

```
show architecture
```

Using scheduler-locking

Scheduler-locking is a feature of GDB that simplifies multi thread 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 multi threaded 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

Threads and Per-Frame architecture

The most significant design change introduced by the combined debugger is the switch from a per-thread architecture selection to a per-frame selection. The new combined debugger for SDK 3.0 eliminates GDB's notion of a "current architecture", that is, remove the global notion of a current architecture per thread. In practice, the architecture will depend on the current frame. The frame is another fundamental GDB notion; it refers to a particular level in the function call sequence (stack back-trace) on a specific thread.

The architecture selection per-frame notion allow the Cell/B.E. back-ends to represent flow of control that switches from PPE code to SPE code and back. The full back-trace can thus represent e.g. the following program state:

1. (current frame) PPE libc printf routine.
2. PPE libspe code implementing a PPE-assisted call
3. SPE newlib code that issued the PPE-assisted call
4. SPE user application code that called printf
5. SPE main
6. PPE libspe code that started SPE execution
7. PPE user application code that called spe_context_run
8. PPE main

Therefore, any thread of a combined Cell BE 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.

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.

Breakpoints

Generally speaking, you can use the same procedures to debug code for Cell BE as you would for PPC code. However, some existing features of GDB and one

new command can help you to debug in the Cell BE processor multi threaded environment. These features are described below.

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 specifies that GDB should set a pending breakpoint if the code for the breakpoint location is not loaded.
- 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 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

Multiple Defined Symbols

When debugging a combined Cell BE application consisting of a PPE program and more or more SPE programs, it may happen that multiple definitions of a global function or variable with the same name exist. For example, both the PPE and SPE programs will define a global main function. If you run the same SPE executable simultaneously within multiple SPE contexts, all its global symbols will show multiple instances of definition. This might cause problems when attempting to refer to a specific definition from the GDB command line, for example when setting a break point. It is not possible to choose the desired instance of the function or variable definition in all cases.

To catch the most common usage cases, GDB uses the following rules when looking 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.

Architecture specific commands

The Cell/B.E. SDK 3.0, besides promoting changes to some of the common gdb debugger commands behavior, also introduces a set of new commands to better accommodate both PPE and SPE code needs.

Stop on SPU load

The new 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 command:

```
show spu stop-on-load
```

Info SPU commands

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 sub command name. This displays full documentation. Command name abbreviations are allowed if unambiguous.

info spu event

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

Example 5-6

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

info spu signal

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

Example 5-7

```
(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:

Example 5-8

```
(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. The output is similar to:

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

Opcode  Tag TId RId EA                 LSA     Size    LstAddr LstSize E
get     1   2   3   0x000000000ffc0001 0x02a80 0x00020                 *
putllc  0   0   0   0xd000000000230080 0x00080 0x00000
get     4   1   1   0x000000000ffc0004 0x02b00 0x00004                 *
mfcsync 0   0   0                     0x00300 0x00880
get     0   0   0   0xd000000000230900 0x00e00 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
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
```

Figure 5-7 info spu dma output

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
```

Figure 5-8 info spu proxydma output

5.4 Simulator

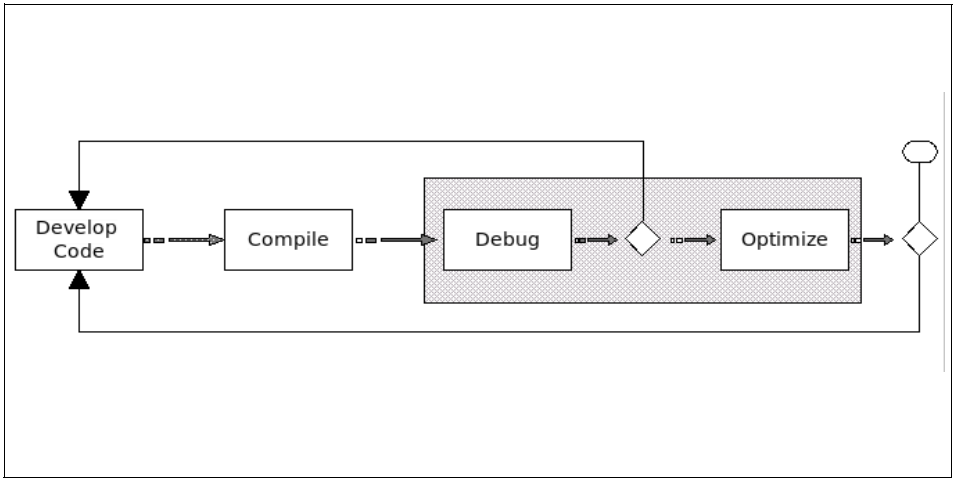


Figure 5-9 Simulate

The IBM Full-System Simulator is a software application that emulates the behavior of a full system that contains a Cell BE processor. You can start a Linux operating system on the simulator and run applications on the simulated operating system. The simulator also supports the loading and running of statically-linked executable programs and standalone tests without an underlying operating system. The simulator infrastructure is designed for modeling processor and system-level architecture at levels of abstraction, which vary from

functional to performance simulation models with a number of hybrid fidelity points in between:

- Functional-only simulation: Models the program-visible effects of instructions without modeling the time it takes to run these instructions. Functional-only simulation assumes that each instruction can be run in a constant number of cycles. Memory accesses are synchronous and are also performed in a constant number of cycles. This simulation model is useful for software development and debugging when a precise measure of execution time is not significant. Functional simulation proceeds much more rapidly than performance simulation, and so is also useful for fast-forwarding to a specific point of interest.
- Performance simulation: For system and application performance analysis, the simulator provides performance simulation (also referred to as timing simulation). A performance simulation model represents internal policies and mechanisms for system components, such as arbiters, queues, and pipelines. Operation latencies are modeled dynamically to account for both processing time and resource constraints. Performance simulation models have been correlated against hardware or other references to acceptable levels of tolerance. The simulator for the Cell BE processor provides a cycle-accurate SPU core model that can be used for performance analysis of computational-intense applications. The simulator for SDK 3.0 provides additional support for performance simulation. This is described in the IBM Full-System Simulator Users Guide.

The simulator can also be configured to fast-forward the simulation, using a functional model, to a specific point of interest in the application and to switch to a timing-accurate mode to conduct performance studies. This means that various types of operational details can be gathered to help you understand real-world hardware and software systems.

See the /opt/ibm/systemsim-cell/doc subdirectory for complete documentation including the simulator user's guide.

5.4.1 Setting up and Bringing up

To verify that the simulator is operating correctly and then run it, issue the following commands:

```
export PATH=/opt/ibm/systemsim-cell/bin:$PATH

systemsim -g
```

The systemsim script found in the simulators bin directory launches the simulator.

The -g parameter starts the graphical user interface.

You can use the simulator's GUI to get a better understanding of the Cell BE architecture. For example, the simulator shows two sets of PPE state. This is because the PPE processor core is dual-threaded and each thread has its own registers and context. You can also look at the state of the SPEs, including the state of their Memory Flow Controller (MFC).

Access to the Simulator Image

By default the simulator does not write changes back to the simulator system root (sysroot) image. This means that the simulator always begins in the same initial state of the sysroot image. When necessary, you can modify the simulator configuration so that any file changes made by the simulated system to the sysroot image are stored in the sysroot disk file so that they are available to subsequent simulator sessions.

To specify that you want update the sysroot image file with any changes made in the simulator session, change the newcow parameter on the mysim bogus disk init command in .systemsim.tcl to rw (specifying read/write access) and remove the last two parameters. The following is the changed line from .systemsim.tcl:

- mysim bogus disk init 0 $sysrootfile rw

When running the simulator with read/write access to the sysroot image file, you must ensure that the file system in the sysroot image file is not corrupted by incomplete writes or a premature shutdown of the Linux operating system running in the simulator. In particular, you must be sure that Linux writes any cached data out to the file system before exiting the simulator. To do this, issue

- sync ; sync

in the Linux console window just before you exit the simulator.

Selecting Architecture

Many of the tools provided in SDK 3.0 support multiple implementations of the CBEA. These include the Cell BE processor and a future processor. This future 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 future processor also supports improved issue and latency for all double precision instructions.

The simulator also supports simulation of the future processor. The simulator installation provides a tcl run script to configure it for such simulation. For example, the following sequence of commands start the simulator configured for the future processor with a graphical user interface.

- export PATH=$PATH:/opt/ibm/systemsim-cell/bin
- systemsim -g -f config_edp_smp.tcl

5.4.2 Operating the GUI

The simulators GUI offers a visual display of the state of the simulated system, including the PPE and the eight SPEs.

You can view the values of the registers, memory, and channels, as well as viewing performance statistics. The GUI also offers an alternate method of interacting with the simulator.

The main GUI window has two basic areas:

- The vertical panel on the left.
- The rows of buttons on the right.

The vertical panel represents the simulated system and its components. The rows of buttons are used to control the simulator.

To start the GUI from the Linux run directory, enter:

- PATH=/opt/ibm/systemsim-cell/bin:$PATH; systemsim -g

The simulator will then configure the simulator as a Cell Broadband Engine and display the main GUI window, labeled with the name of the application program. When the GUI window first appears, click the Go button to boot the Linux operating system.

The simulation panel

When the main GUI window first appears, the vertical panel contains a single folder labeled mysim.

To see the contents of mysim, click on the plus sign (+) in front of the folder icon. When the folder is expanded, you can see its contents. These include

- a PPE (labelled PPE0:0:0 and PPE0:0:1,
- the two threads of the PPE),
- eight SPEs (SPE0... SPE7).

The folders representing the processors can be further expanded to show the viewable objects and the options and actions available.

PPE Components

There are five PPE components visible in the expanded PPE folder. The five visible PPE components are:

- PCTrack
- PCCCore
- GPRegs
- FPRegs
- PCAddressing

The general-purpose registers (GPRs) and the floating-point registers (FPRs) can be viewed separately by double-clicking on the GPRegs and the FPRegs folders respectively.

As data changes in the simulated registers, the data in the windows is updated and registers that have changed state are highlighted.

The PPE Core window (PPCCore) shows the contents of all the registers of the PPE, including the Vector/SIMD Multimedia Extension registers.

SPE components

The SPE folders (SPE0 ... SPE7) each have ten sub-items. Five of the sub-items represent windows that show data in the registers, channels, and memory:

- SPUTrack
- SPUCore
- SPEChannel
- LS_Stats
- SPUMemory

Two of the sub-items, and, represent windows that show state information on the MFC:

- MFC
- MFC_XLate

The last three sub-items represent actions to perform on the SPE:

- SPUStats
- Model
- Load-Exec

The last three items in an SPE folder represent actions to perform, with respect to the associated SPE. The first of these is SPUStats. When the system is stopped and you double-click on this item, the simulator displays program performance statistics in its own pop-up window. These statistics are only collected when the Model is set to pipeline mode.

The next item in the SPE folder is labelled either:

- Model: instruction,
- Model: pipeline, or
- Model: fast.

The label indicates whether the simulation is in:

- instruction mode for checking and debugging the functionality of a program,
- pipeline mode for collecting performance statistics on the program, or
- fast mode for fast functional simulation only.

The model can be toggled by double-clicking the item. The Perf Models button on the GUI can also be used to display a menu for setting the simulator model modes of all of the SPEs simultaneously.

The last item in the SPE folder, Load-Exec, is used for loading an executable onto an SPE. When you double-click the item, a file-browsing window is displayed, allowing you to find and select the executable file to load.

Simulation control buttons

On the right side of the GUI screen are five rows of buttons. These are used to manipulate the simulation process. The five rows of buttons do the following:

- Advance Cycle: Advances the simulation by a set number of cycles. The default value is 1 cycle, but it can be changed by entering an integer value in the textbox above the buttons, or by moving the slider next to the textbox. The drop-down menu at the top of the GUI allows the user to select the time domain for cycle stepping. The time units to use for cycles are expressed in

terms of various system components. The simulation must be stopped for this button to work; if the simulation is not stopped, the button is inactive.

- Go: Starts or continues the simulation. In the SDK\u2019s simulator, the first time the Go button is clicked it initiates the Linux boot process. (In general, the action of the Go button is determined by the startup tcl file located in the directory from which the simulator is started.)
- Stop: Pauses the simulation.
- Service GDB: Allows the external gdb debugger to attach to the running program. This button is also inactive while the simulation is running.
- Triggers/Breakpoints: Displays a window showing the current triggers and breakpoints.
- Update GUI: Refreshes all of the GUI screens. By default, the GUI screens are updated automatically every four seconds. Click this button to force an update.
- Debug Controls: Displays a window of the available debug controls and allows you to select which ones should be active. Once enabled, corresponding information messages will be displayed.
- Options: Displays a window allowing you to select fonts for the GUI display. On a separate tab, you can enter the gdb debugger port.
- Emitters: Displays a window with the defined emitters, with separate tabs for writers and readers.
- Fast Mode: Toggles fast mode on and off. Fast mode accelerates the execution of the PPE at the expense of disabling certain system-analysis features. It is useful for quickly advancing the simulation to a point of interest. When fast mode is on, the button appears depressed; otherwise it appears normal. Fast mode can also be enabled with the mysim fast on command and disabled with the mysim fast off command.
- Perf Models: Displays a window in which various performance models can be selected for the various system simulator components. Provides a convenient means to set each SPU\u2019s simulation mode to either cycle accurate pipeline mode or instruction mode or fast functional-only mode. The same capabilities are available using the Model:instruction, Model:pipeline, Model:fast toggle menu sub-item under each SPE in the tree menu at the left of the main control panel.
- SPE Visualization: Plots histograms of SPU and DMA event counts. The counts are sampled at user defined intervals, and are continuously displayed. Two modes of display are provided: a \u201cscroll\u201d view, which tracks only the most recent time segment, and a \u201ccompress\u201d view, which accumulates samples to provide an overview of the event counts during the

time elapsed. Users can view collected data in either detail or summary panels.

  - The detailed, single-SPE panel tracks SPU pipeline phenomena (such as stalls, instructions executed by type, and issue events), and DMA transaction counts by type (gets, puts, atomics, and so forth).
  - The summary panel tracks all eight SPEs for the CBE, with each plot showing a subset of the detailed event count data available.

- Process-Tree and Process-Tree-Stats: This feature requires OS kernel hooks that allow the simulator to display process information. This feature is currently not provided in the SDK kernel.
- SPU Modes: Provides a convenient means to set each SPU\u2019s simulation mode to either cycle accurate pipeline mode or fast functional-only mode. The same capabilities are available using the Model:instruction or Model:pipeline toggle menu sub-item under each SPE in the tree menu at the left of the main control panel.
- Event Log: Enables a set of predefined triggers to start collecting the log information. The window provides a set of buttons that can be used to set the marker cycle to a point in the process.
- Exit: Exits the simulator and closes the GUI window.

5.5 IBM Multi core Acceleration Integrated Development Environment

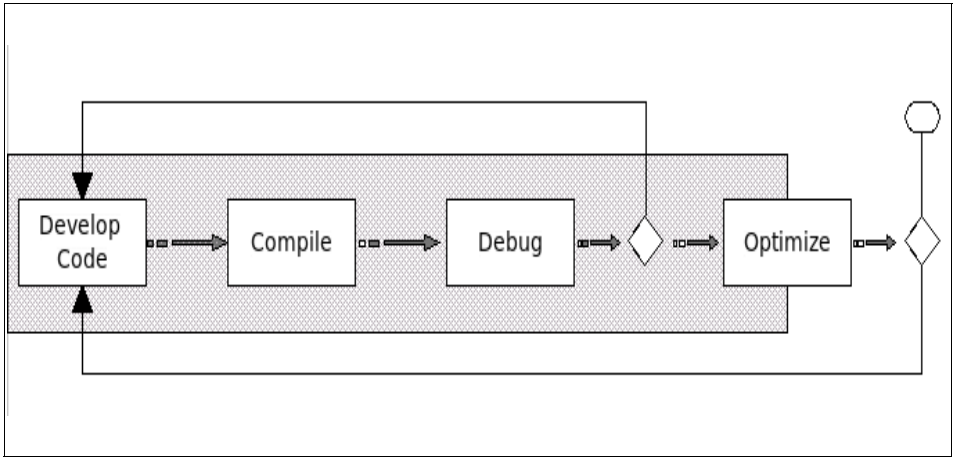


Figure 5-10 IDE supports all parts of the process

The IBM SDK for Multi core Acceleration Integrated Development Environment (Cell/B.E. IDE), which is built upon the Eclipse and C Development Tools (CDT) platform, integrates the Cell BE GNU tool chain, compilers, IBM Full-System Simulator for the Cell BE, and other development components in order to provide a comprehensive, user-friendly development platform that simplifies Cell BE development. 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 rich visual interface for PPE (Power Processing Element) and SPE (Synergistic Processing Element) GDB (GNU debugger)
- seamless integration of simulator into Eclipse
- automatic builder, performance tools, and several other enhancements

The Cell/B.E. IDE offers developers a complete solution for developing an application. The IDE is capable of managing all artifacts involved in the development, as well as deploying and testing them on the target environment. Typically, the developer goes from projects creation, including multiple build configurations, target environment setup and application launching/debugging.

5.5.1 Step 1: Projects

The underlying Eclipse Framework architecture offers the concept of projects as units of agglomeration for your application artifacts. A project is responsible not

only for holding a file system structure, but also for binding your application code with build, launch, debug configurations and even non-source code artifacts.

Defining Projects

The Cell/B.E. IDE leverages the Eclipse CDT framework, which is the tooling support for developing C/C++ applications. As well as the CDT framework, there's a choice whether yourself manages the build structure, or Eclipse automatically generates for you. This is, respectively, the difference between Standard Make and Managed Make options for project creation.

Table 5-2 Project Management Options

Project Management Style	Description
Standard Make C/C++	Requires you to provide a Makefile.
Managed Make C/C++	Eclipse will auto-create and manage the makefiles for you.

If you chose **File** → **New** → **Project,** the New Project Wizard should pop-up, offering the choices of Standard Make or Managed Make projects under both C and C++ project groups.

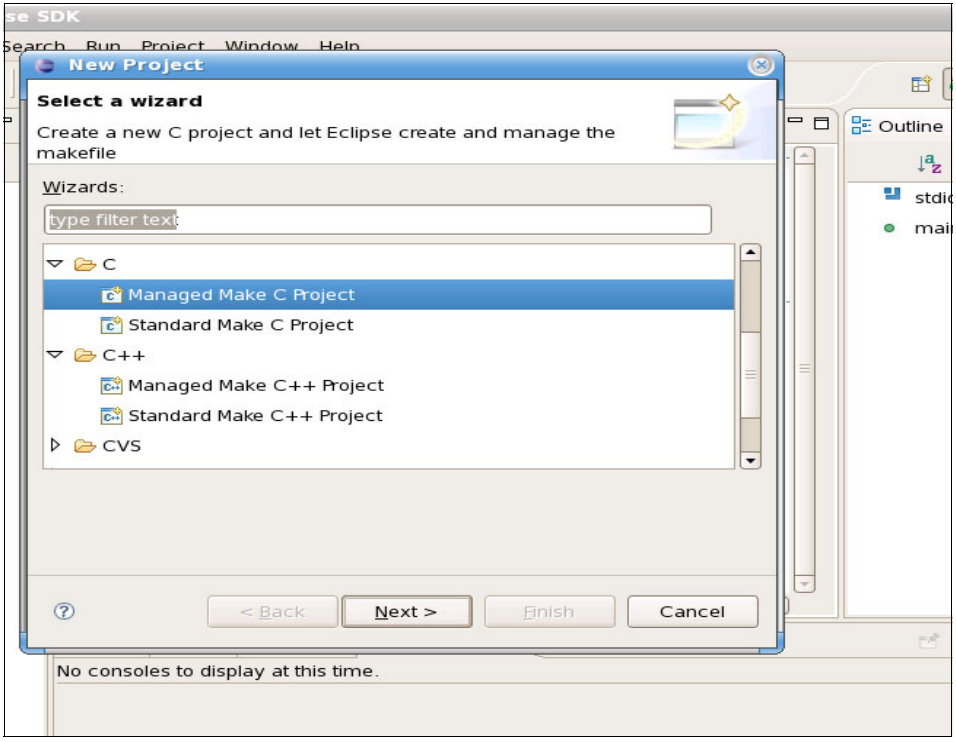


Figure 5-11 Available project creation wizards

Creating Projects

Once you defined which style of projects better suits your needs, give your project a name. The next step is to choose among the available project types for Cell/B.E. development:

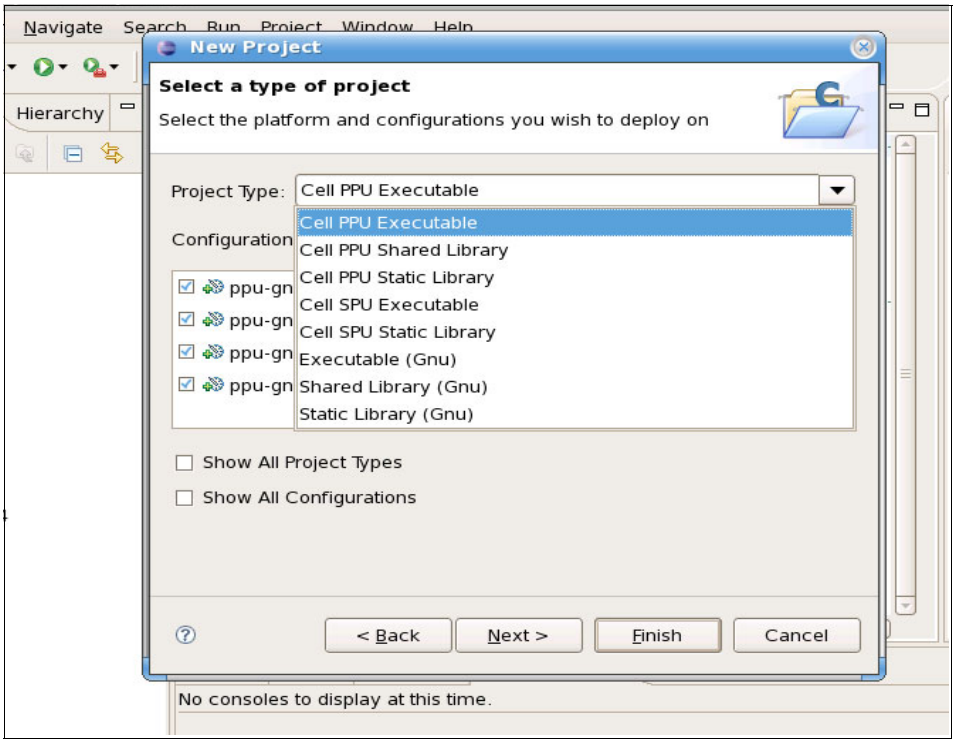


Figure 5-12 Project Types

Table 5-3 Project Type Options

Project Type	Description
Cell PPU Executable	Creates a PPU executable binary. This project has the capability of referencing any other SPU project binary, in order to produce a CBE combined binary.
Cell PPU Shared Library	Creates a PPU shared library binary. This project has the capability of referencing any other SPU project binary, in order to produce a CBE combined library.
Cell PPU Static Library	Creates a PPU static library binary. This project has the capability of referencing any other SPU project binary, in order to produce a CBE combined library.
Cell SPU Executable	Creates a SPU binary. The resulting binary can be executed as an spulet or embedded in a PPU binary.

Project Type	Description
Cell SPU Static Library	Creates a SPU static library. The resulting library can be linked together with other spu libraries and also be embedded in an PPU binary.

Project Configuration

The newly created project should appear in the C/C++ view, on the left side of the screen. The next remaining task is to configure the project's build options.

Note: If you choose the Standard Make style of project management, that implies you are responsible for maintaining and updating your makefiles. Eclipse will only offer a thin run wrapper, where you are able to define which are the relevant targets within your makefile (so when you hit build in Eclipse, it will invoke the desired makefile target, instead of the default 'all').

Select the desired project and right click on the "**Properties**" option. As soon as the properties dialog should appear on the screen locate and select the "**C/C++ Build***"* on the left portion of the dialog.

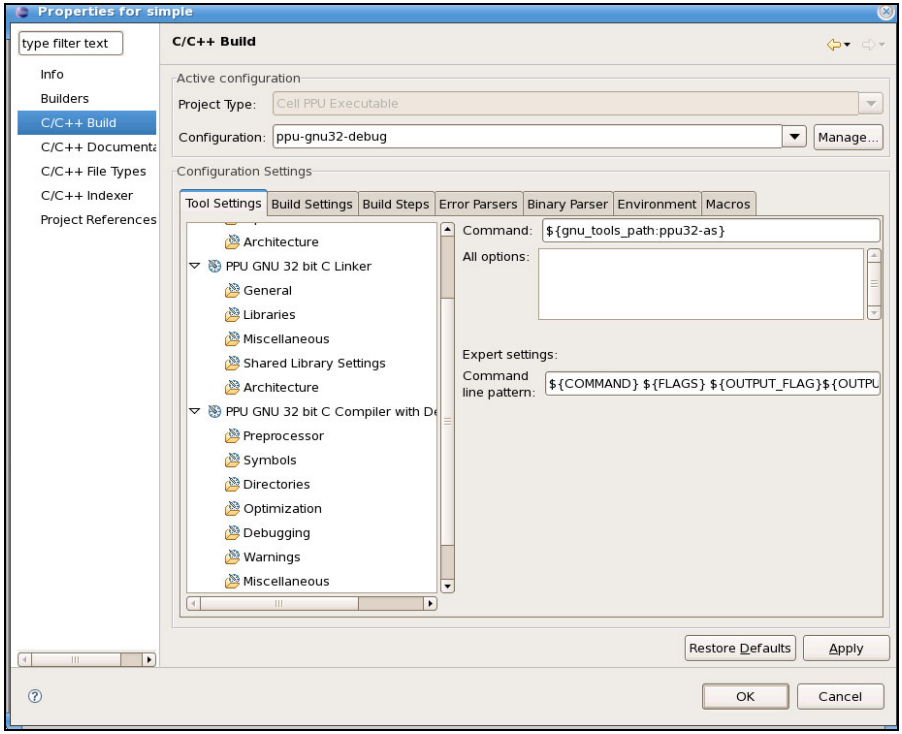


Figure 5-13 C/C++ Build options

The “C/C++ Build Options” carries all needed build and compile configuration entry points, so we are able to customize the whole process. Each tool that is part of the toolchain has its configuration entry point in the “Tools Settings” tab. As soon as you finish altering/adding any of the values, you need to hit “Apply” and “OK” buttons, which will trigger IDE’s automatic makefile generation process, so you project’s build is updated immediately.

5.5.2 Step 2: Choosing Target Environments with Remote Tools

Now that the projects are created and properly configured, we can test our program, but first, a Cell environment must be created and started. The Cell IDE integrates IBM's full-system simulator for the Cell BE processor and Cell BE Blades into Eclipse, so that your only a few clicks away from testing your application on a Cell environment.

Simulator Integration

In the Cell Environments view at the bottom, right click on Local Cell Simulator and select Create.

This is the Local Cell Simulator properties window, which you can use to configure the simulator to meet any specific needs that you might have. You can modify an existing Cell environment's configuration at any time (as long as its not running) by right clicking on the environment and selecting Edit. Enter a name for this simulator (e.g. My Cell Simulator), then click on Finish.

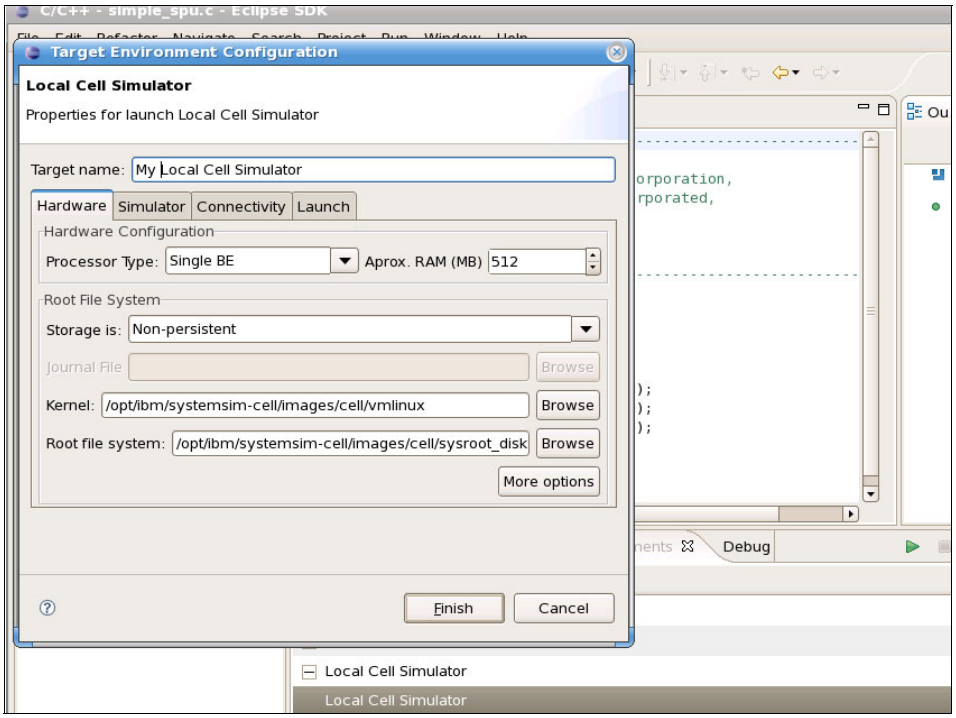


Figure 5-14 Simulator Environment Dialog

Cell BE Blade Integration

In the Cell Environments view at the bottom, right click on Cell Box and select Create.

This is the Cell Box properties window, which you can use to configure remote access to your Cell BE Blade to meet any specific needs that you might have. You can modify an existing Cell environment's configuration at any time (as long as its not running) by right clicking on the environment and selecting Edit. Enter a name for this configuration (e.g. My Cell Blade), then click on Finish.

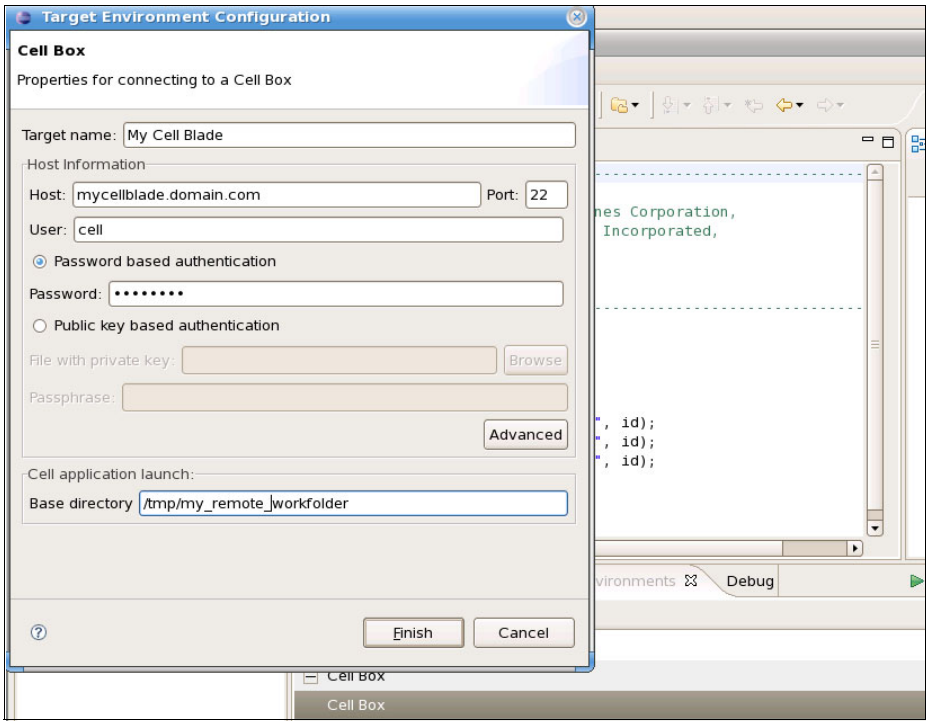


Figure 5-15 Cell Blade Environment Dialog

Starting the Environments

We need to have, at least, one of the environment configurations started in order to proceed to the launch configuration options. For that, highlight the desired environment configuration and locate the green “arrow shaped” button at the left corner of environments view. When you press this button, you are activating the connection between the chosen environment and IDE.

5.5.3 Step 3: Debugger

Next, a C/C++ Cell Application launch configuration needs to be created and configured for the application debugging.

Creating the launch configuration

Locate the Run menu. Choose **Run** → **Debug...** Next, in the left pane, locate and right click on C/C++ Cell Target Application, and select New

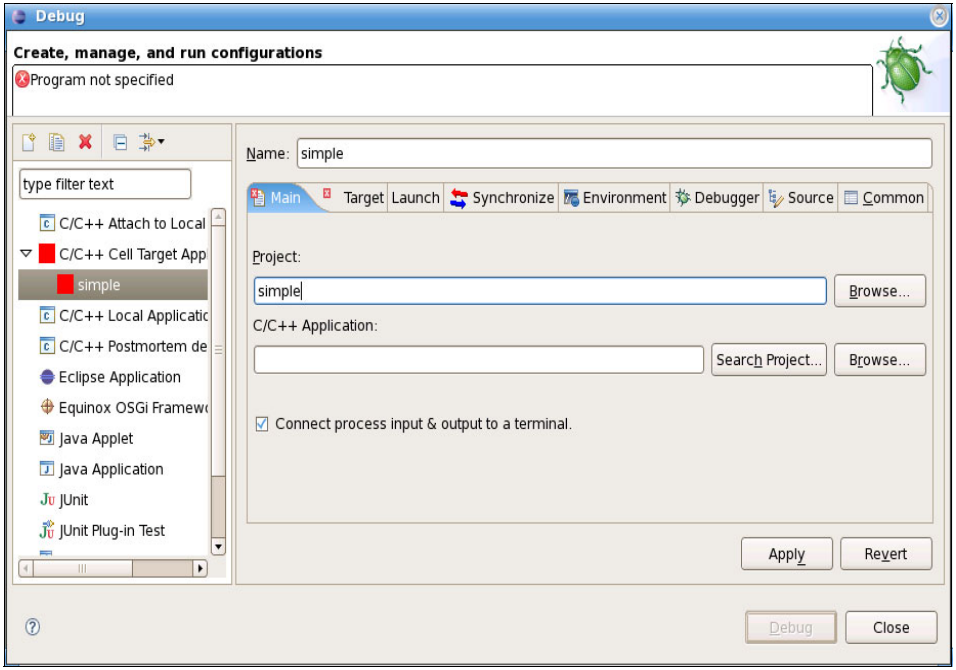


Figure 5-16 Launch configurations dialog

Configuring the Launch

With the launch configuration dialog opened, locate the Project field (in the Main tab) and specify which project you are going to debug. You must also specify which application, within the project

Now, navigate to the Target tab.

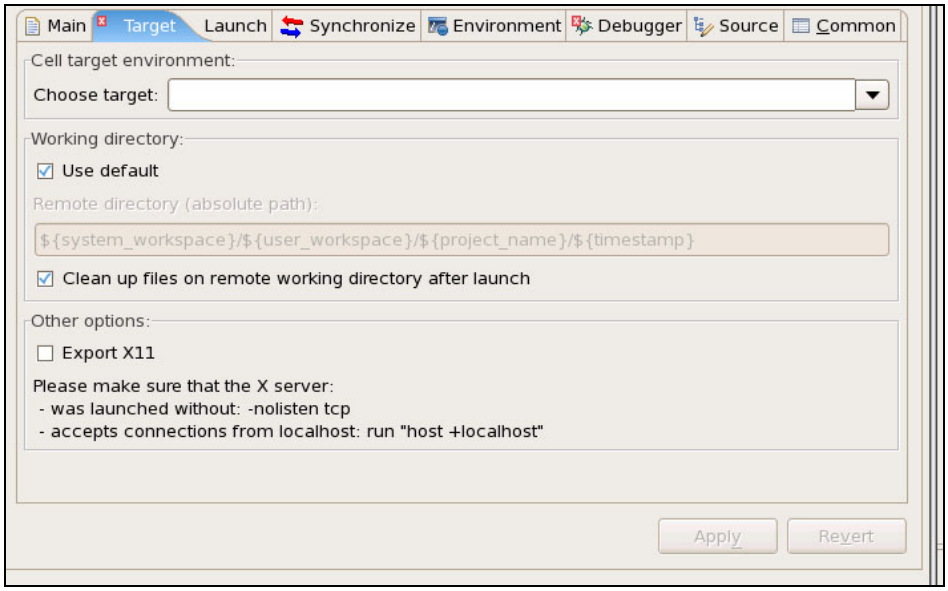


Figure 5-17 Target tab

The target tab allows you to select which remote environment you are about to debug your application with. It is possible to select any of the previously configured environments, as long as they are active (i.e. started).

The next tab, launch, allows you to specify any command line arguments as well as shell commands that needs to be executed before and/or after your application.

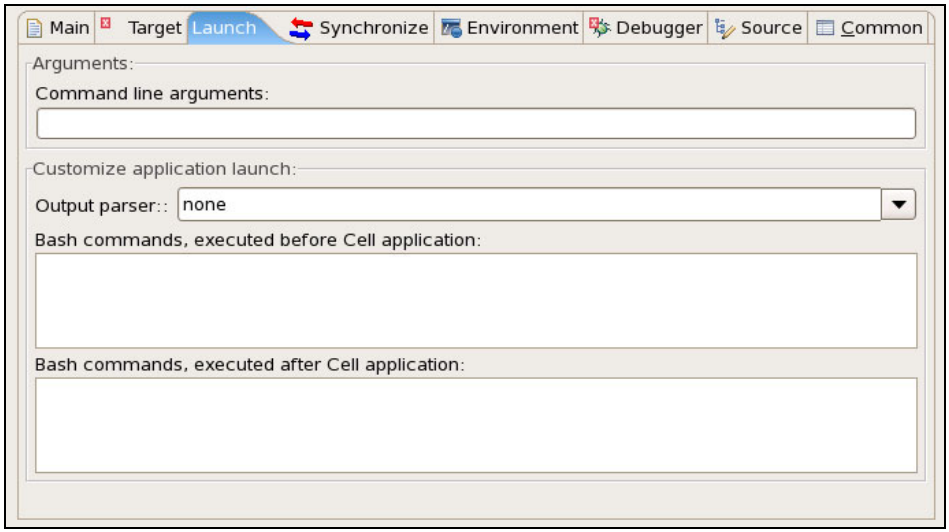


Figure 5-18 Launch tab

Moving on the Synchronize tab, you can specify resources (such as input/output files) that need to be synchronized with the Cell environment's file system before and/or after the application executes. Use New upload rule to specify resource(s) to copy to the Cell environment before the application executes, and use New download rule to copy resource(s) back to your local file system after execution. Don't forget to check the Upload rules enabled and/or Download rules enabled boxes after adding any upload/download rules.

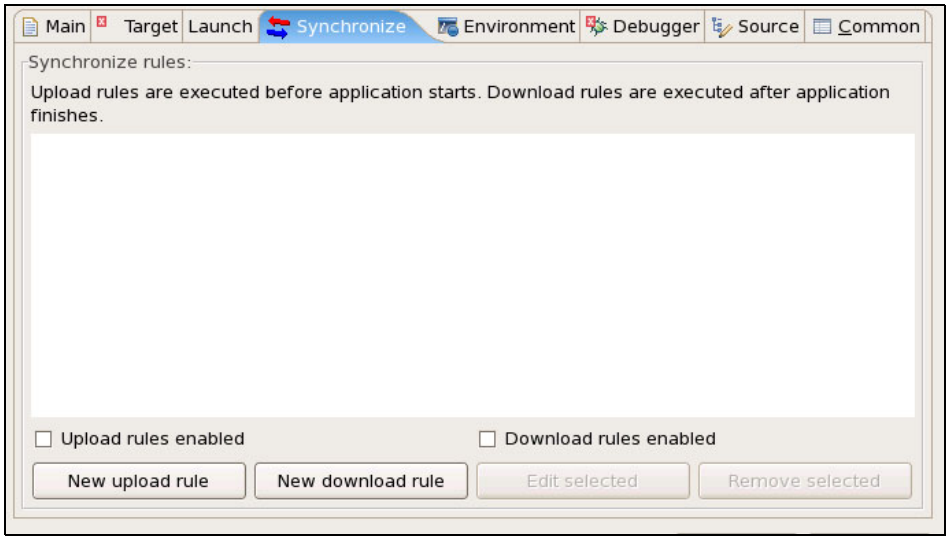


Figure 5-19 Synchronize tab

The remaining step is to actually configure the debugger parameters. Go to the Debugger tab.

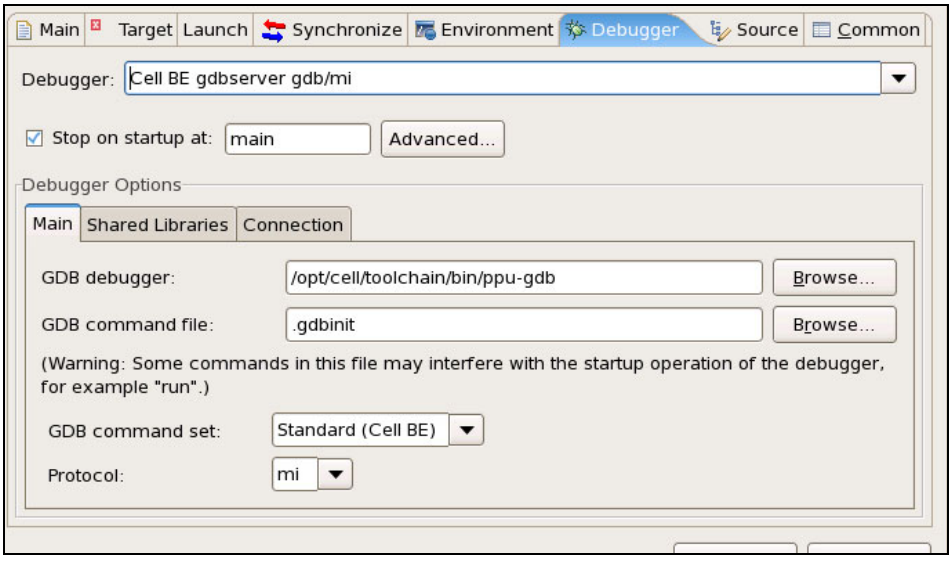


Figure 5-20 Debugger Main tab

Locate the Debugger field, where you can choose from Cell PPU gdbserver, Cell SPU gdbserver, or Cell BE gdbserver. To debug only PPU or SPU programs, select Cell PPU gdbserver or Cell SPU gdbserver, respectively. The Cell BE gdbserver option is the combined debugger, which allows for debugging of PPU and SPU source code in one debug session.

Since this is a remote debugging session, it's also important to select the correct remote side debugger (gdbserver) type, according to your application. Go to the Connection tab, inside the Debugger tab.

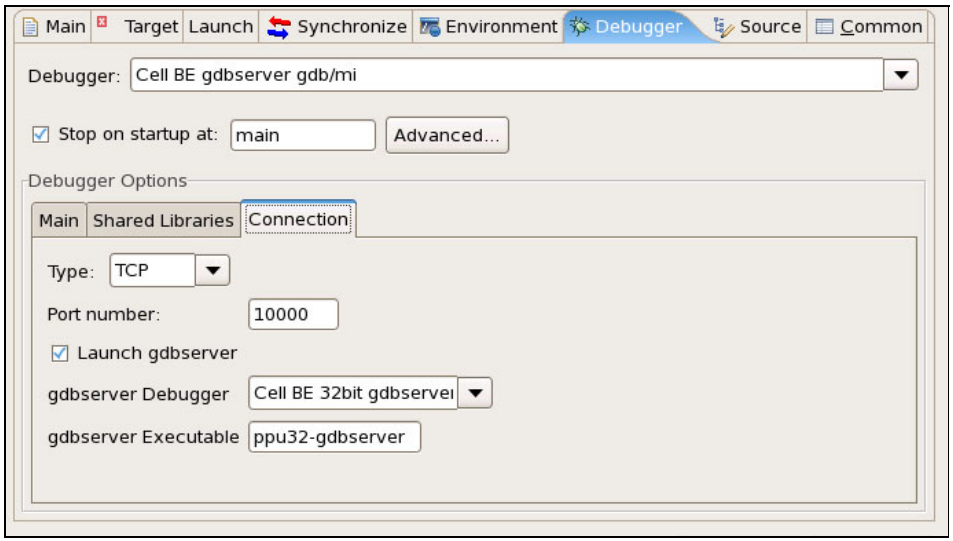


Figure 5-21 Debugger Connection tab

If your application is 32-bit, choose the Cell BE 32bit gdbserver option at the "gdbserver Debugger" combo. Otherwise, if you have a 64bit application, select the "Cell BE 64bit gdbserver".

If there aren't any pending configuration problems, all is left is to hit the "Apply" button and the "Debug" button. The IDE should switch to the Debug perspective.

Debug Perspective

Once in Eclipse's "Debug Perspective", you are able to carry on any of the regular debugging features available there, like setting breakpoints, inspecting variables, registers and memory.

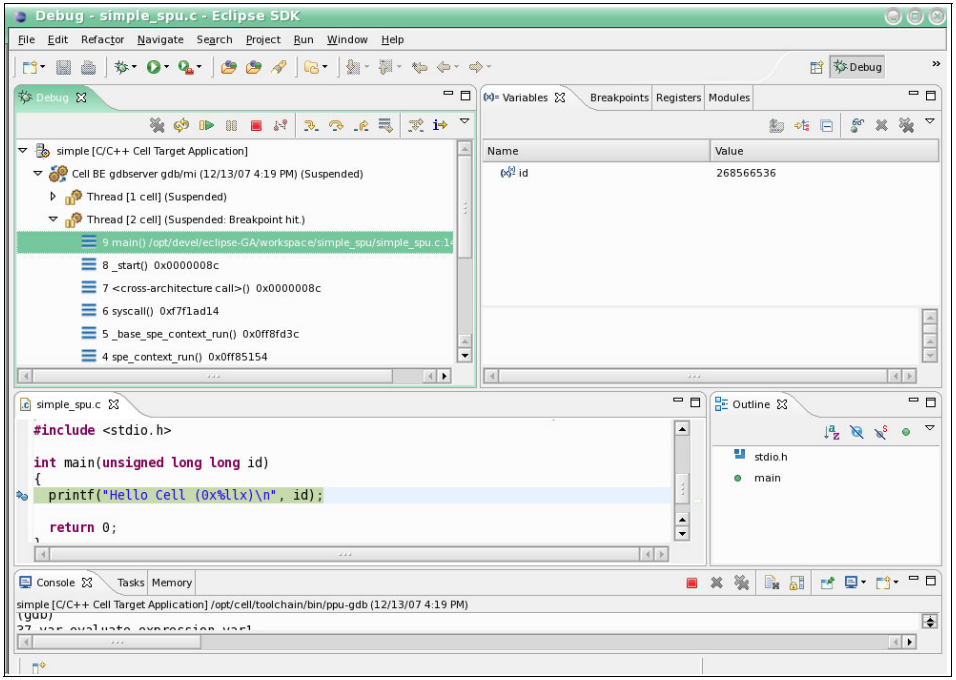


Figure 5-22 Debug perspective

Additionally, the SPU architecture info debug commands (see Info SPU commands) are also available thorough their respective views. Go to **Window** → **Show View** to locate them:

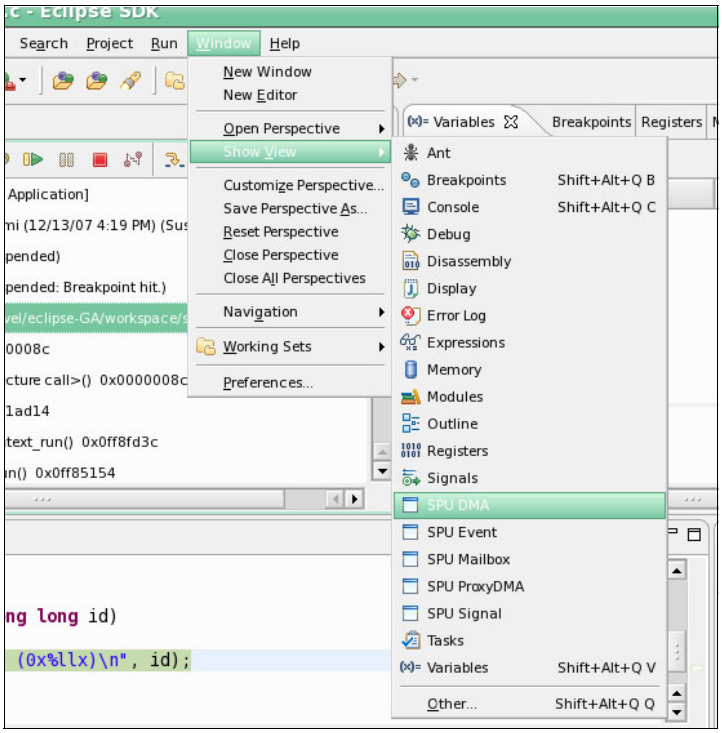


Figure 5-23 "Info SPU" available views.

Tip: As explained in "Threads and Per-Frame architecture" section above, the **`info spu`** commands will only work when actually debugging in the SPE architecture. Use the stack frame view of Eclipse, at the upper left corner (under the name "Debug") to precisely determine in which architecture your code is executing.

5.6 Performance Tools

As we are about to introduce the set of performance tools in the Cell BE SDK 3.0, it is crucial to understand how each one relates to the other, according to a time flow perspective.

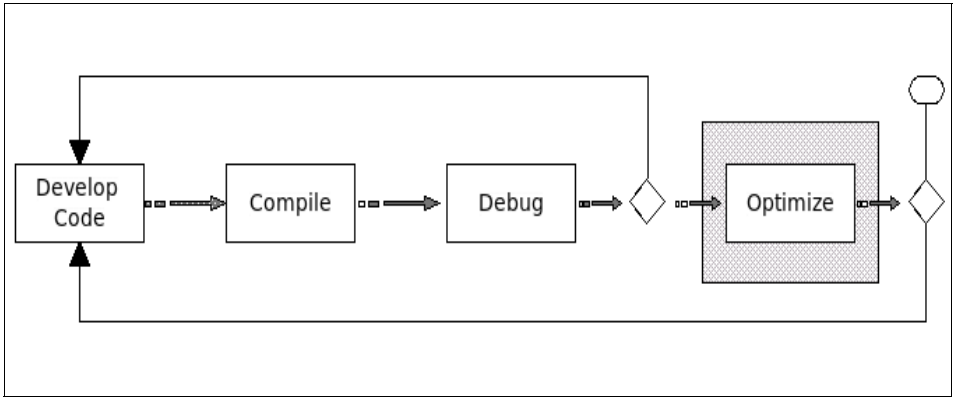


Figure 5-24 Optimize

5.6.1 Typical Performance Tuning Cycle

The major tasks to achieve a complete development cycle can be organized as following:

- Develop/Port the application for Cell/B.E.
  - Programming best practices
  - Knowledge from the architecture
- Compile, Link and Debug
  - Compiler (gcc or xlc)
  - Linker (ld)
  - Debugger (gdb)
- Performance tuning
  - oprofile, PDT, PDTR, VPA, FDPR-Pro, CPC

The following graph tries to summarizes the Cell BE tools interlock:

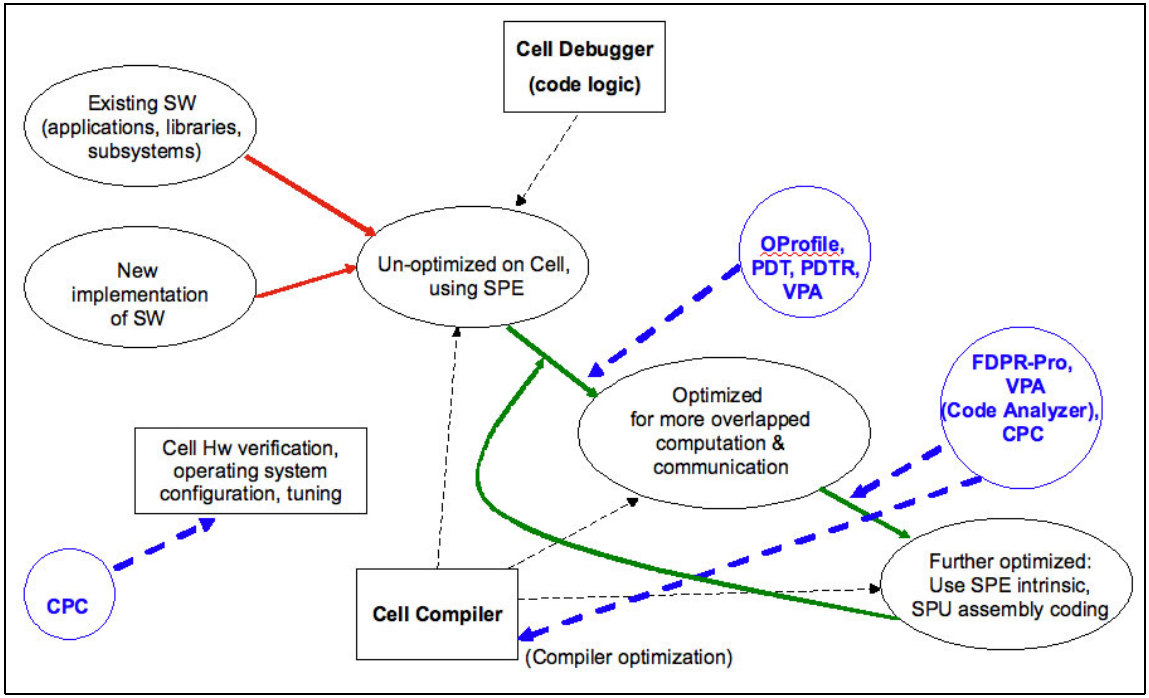


Figure 5-25 Typical Tools flow

5.6.2 CPC

The cell-perf-counter (cpc) tool purposes are for setting up and using the hardware performance counters in the Cell BE processor. These counters allow you to see how many times certain hardware events are occurring, which is useful if you are analyzing the performance of software running on a Cell BE system.

Hardware events are available from all of the logical units within the Cell BE processor including:

- the PPE,
- the SPEs,
- the interface bus,
- memory
- and I/O controllers.

The Cell BE performance monitoring unit (PMU) provides four 32-bit counters, which can also be configured as pairs of 16-bit counters, for counting these

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

Operation

The tool offers two modes of execution:

Workload mode	PMU counters active only during complete execution of a workload, providing a very accurate view of the performance of a single process.
System-wide mode	PMU counters monitor all processes running on specified CPUs for specified duration

The results are grouped according to seven logical blocks - PPU, PPSS, SPU, MFC, EIB, MIC, and BEI, where each block has signals (hardware events) organized in to groups. The PMU can monitor any number of signals within one group, with a maximum of two signal groups at a time.

Hardware Sampling

The Cell BE PMU provides a mechanism for the hardware to periodically read the counters and store the results in a hardware buffer. This allows the cpc tool to collect a large number of counter samples while greatly reducing the number of calls that cpc has to make into the kernel.

The following steps are performed in the Hardware Sampling mode:

- Specify initial counter values and sampling time interval
- PMU record and reset counter values after each interval
- Samples are available in hardware trace-buffer

As the default behavior, hardware buffers will contain the total number of each monitored signal's hit for the specified interval, which is called count mode. Besides simply sampling the counters and accumulating to the buffers, the Cell BE PMU also offers other sampling modes:

Occurrence mode	Monitors one or two entire groups of signals, allowing the specifying of any signal within the desired group. Indicates whether each event occurred at least once during each sampling interval.
Threshold mode	Each event is assigned a "threshold" value. Indicates whether each event occurred at least the specified number of times during each sampling interval.

PPU Bookmarks

The CPC tool offers a feature that allows finer grained tracing method for signals' sampling. The PPU Bookmark mode is provided as an option to start and/or stop the counters when a value is written to what is called the "bookmark" register. The triggering write can be issued from both command line or within your application, achieving the desired sampling scope narrowing.

The chosen bookmark register must be specified as a command line option for CPC, as well as the desired action to be performed on the counters' sampling. The registers can be reached as files in the sysfs filesystem:

`/sys/devices/system/cpu/cpu*/pmu_bookmark`

Overall Usage

The typical command line syntax for CPC is as follows:

`cpc [options] [workload]`

where the presence of the **`[workload]`** parameter controls whether CPC must be ran against a single application (workload mode) or against the whole running system (system-wide mode). In system-wide mode, the following options control both the duration and broadness of the sampling:

`--cpus <CPUS>` Controls which CPUs to use, where CPUS should be a comma separated list of CPUs, or the keyword *all*.

`--time <TIME>` Controls the sampling duration time (in seconds).

Typical Options

These are the cpc options that enable the features described above:

`--list-events` Returns the list of all possible events, grouped by logic units within the Cell BE (see Example 5-9).

Example 5-9

```
Performance Monitor Signals

Key:
    1) Signal Number:
           Digit 1   = Section Number
           Digit 2   = Subsection Number
           Digit 3,4 = Bit Number
           .C (Cycles) or .E (Events) if the signal can record
either
    2) Count Type
           C = Count Cycles
```

```
            E = Count Event Edges
            V = Count Event Cycles
            S = Count Single-Cycle Events
        3) Signal Name
        4) Signal Description


*******************************************************************
* Unit 2: PowerPC Processing Unit (PPU)
*******************************************************************

2.1PPU Instruction Unit - Group 1 (NClk)
-----
2100V  Branch_Commit_t0Branch instruction committed. (Thread 0)

2101E  Branch_Flush_t0Branch instruction that caused a misprediction
flush is committed. Branch misprediction includes: (1) misprediction
of taken or not-taken on conditional branch, (2) misprediction of
branch target address on bclr[1] and bcctr[1]. (Thread 0)

2102C  Ibuf_Empty_t0Instruction buffer empty. (Thread 0)

2103E  IERAT_Miss_t0Instruction effective-address-to-real-address
translation (I-ERAT) miss. (Thread 0)

2104.CC
2104.EE  IL1_Miss_Cycles_t0L1 Instruction cache miss cycles. Counts
the cycles from the miss event until the returned instruction is
dispatched or cancelled due to branch misprediction, completion
restart, or exceptions (see Note 1). (Thread 0)

2106C  Dispatch_Blocked_t0Valid instruction available for dispatch,
but dispatch is blocked. (Thread 0)

2109E  Instr_Flushed_t0Instruction in pipeline stage EX7 causes a
flush. (Thread 0)

2111V  PPC_Commit_t0Two PowerPC instructions committed. For
microcode sequences, only the last microcode operation is counted.
Committed instructions are counted two at a time. If only one
instruction has committed for a given cycle, this event will not be
raised until another instruction has been committed in a future
cycle. (Thread 0)

2119V  Branch_Commit_t1Branch instruction committed. (Thread 1)
```

```
2120E  Branch_Flush_t1Branch instruction that caused a misprediction
flush is committed. Branch misprediction includes: (1) misprediction
of taken or not-taken on conditional branch, (2) misprediction of
branch target address on bclr[1] and bcctr[1]. (Thread 1)

2121C  Ibuf_Empty_t1Instruction buffer empty. (Thread 1)
.....
```

--event <ID> Specifies the event to be counted.

--event <ID.E> Some events allow counting of either (E) events or cycles (C).

--event <ID:SUB> When specifying SPU or MFC events, the desired subunit can be given (see Example 5-10).

Example 5-10 Specifying subunits for events.

```
cpc -event 4103:1 ...# Count event 4103 on SPU 1.
```

--event <ID[.E][:SUB],...

> Specifies multiple coma-separated events (in all of the above forms) to be counted.

--event <ID[.E][:SUB],...

--event <ID[.E][:SUB],...

--switch-timeout <ms>

Multiple specification of the **event** option, allows events to be grouped in sets, with the kernel cycling through them at the interval defined by the **switch-timeout** option (see Example 5-11).

Example 5-11 Multiple sets of events

```
cpc --event 4102:0,4103:0 --event 4102:1,4103:1 \
--switch-timeout 150m ....
```

--interval <TIME> Specifies the interval time for the *Hardware Sampling* mode. Suffix the value with 'n' for nanoseconds, 'u' for microseconds, or 'm' for milliseconds.

--sampling-mode <MODE>

Used in conjunction with the **interval** option, defines the behavior of the hardware sampling mode (as explained in Hardware Sampling). The "threshold" mode has one

particularity, with regard to the option syntax, which requires the specification of the desired threshold value for each event (see Example 5-12)

Example 5-12 Hardware Sampling mode

```
cpc --event 4102:0=1000,4103:0=1000 --interval 10m \
--sampling-mode threshold ....
```

--start-on-ppu-th0-bookmark

Start counters upon PPU hardware-thread 0 bookmark start.

--start-on-ppu-th1-bookmark

Start counters upon PPU hardware-thread 1bookmark start.

--stop-on-ppu-th0-bookmark

Stop counters upon PPU hardware-thread 0 bookmark stop.

--stop-on-ppu-th0-bookmark

Stop counters upon PPU hardware-thread 1bookmark stop.

Example 5-13 PPU Bookmarks with command line trigger

```
cpc --events 4103 --start-on-ppu-th0-bookmark app # set-up bookmark
```

There are two choices for triggering the counters, as shown below:

Example 5-14 Command line trigger

```
echo 9223372036854775808 > /sys/devices/system/cpu/cpu0/pmu_bookmark
```

Example 5-15 Program embedded trigger

```
#define PMU_BKMK_START (1ULL << 63)
char str[20] ;
fd = open("/sys/devices/system/cpu/cpu0/pmu_bookmark", O_WRONLY);
sprintf(str, "%llu", PMU_BKMK_START);
write(fd, str, strlen(str) + 1);
```

5.6.3 OProfile

OProfile for Cell BE is a system-level profiler for Cell Linux systems, capable of profiling all running code at low overhead.

It consists of a kernel driver and a daemon for collecting sample data, and several post-profiling tools for turning data into information.

OProfile leverages the hardware performance counters of the CPU to enable profiling of a wide variety of interesting statistics, which can also be used for basic time-spent profiling. All code is profiled: hardware and software interrupt handlers, kernel modules, the kernel, shared libraries, and applications.

The OProfile tool can be used in a number of situations, like:

- low overhead is a requirement and cannot use other highly intrusive profiling methods
- profiling of an application and its shared libraries
- profiling of interrupt handlers
- performance behavior of entire system
- there's a need to exam the hardware effects
- requires instruction-level profiles
- requires call-graph profiles

Operation

The OProfile requires root privileges and exclusive access to the Cell BE PMU. For those reasons, it only supports one session at a time and no other PMU related tool (like CPC) simultaneously running.

Tools

The tool is actually composed by a few utilities, which controls the profiling session and reporting. The most relevant here are:

Table 5-4 OProfile relevant utilities

Tool	Description
opcontrol	Configures and control the profiling system. Sets the performance counter event to be monitored and the sampling rate.

Tool	Description
opreport	Generates the formatted profiles according to symbols or file names. Supports text and XML output
opannotate	Generates annotated source or assembly listings. The listings are annotated with hits per each source code line. Requires compilation with debug symbols (-g).

Considerations

The current SDK 3.0 version of OProfile for Cell BE supports profiling on the POWER processor events and SPU cycle profiling. These events include cycles as well as the various processor, cache and memory events. It is possible to profile on up to four events simultaneously on the Cell BE system. There are restrictions on which of the PPU events can be measured simultaneously. (The tool now verifies that multiple events specified can be profiled simultaneously. In the previous release it was up to the user to verify that.). When using SPU cycle profiling, events must be within the same group due to restrictions in the underlying hardware support for the performance counters. You can use the following command to view the events and which group contains each event:

Example 5-16 Listing available events

```
opcontrol --list-events
```

Overall Process

The **opcontrol** is the utility that drives the profiling process, which can be initiated even at compilation time, if source annotation is desired. Both the **opreport** and **opannotate** tool are deployed at the end of the process to collect, format and co-relate sampled information.

Step1: Compiling (optional)

Usually, OProfile does not require any changes to the compilation options, even with regard to the optimization flags. However, in order to achieve a better annotation experience, it is necessary to leave relocation and debugging symbols in the application binary. The required flags are as follows:

Table 5-5

Flag	Tool	Description
-g	compiler	Instructs the compiler to produce debugging symbols in the resulting binary.
-Wl,q	linker	Preserve the relocation and the line number information in the final integrated executable

Step 2: Initializing

As previously explained, the opcontrol utility drive the process from initialization to the end of sampling. Since the OProfile solution comprehends a kernel module, a daemon and a collection of tools, in order to properly operate the session, we must assure that both the kernel module and the daemon are operational, and that no other stale session information is present:

Example 5-17 Initialization steps

```
opcontrol --deinit # recommended to clear any previous Oprofile data
opcontrol --start-daemon --no-vmlinux # start the Oprofile daemon
opcontrol --init # perform the initializaton procedure
opcontrol --reset # sanity reset
```

Note: The **--no-vmlinux** option shown above controls whether kernel profiling should also be carried on. The example here disables such profiling, which is always true if sampling for the SPUs (there aren't kernels running on them). In case such option is needed (for PPUs only), replace the former with **--vmlinux=/path/to/vmlinux**, where *vmlinux* is the running kernel's uncompressed image.

Step 3: Running

After properly cleaning up and assuring the kernel module and daemon are present, we are able to proceed with the sampling, after properly adjusting how samples should be organized, the desired event group to monitor and the number of events per sampling:

Example 5-18 Adjusting event group and sampling interval

```
opcontrol –separate=all --event=SPU_CYCLES:100000
```

--separate	Organizes samples based on the given separator. The “lib” separates dynamically linked library samples per application. ’kernel’ separates kernel and kernel module samples per application; ’kernel’ implies ’library’.’thread’ gives separation for each thread and task. ’cpu’ separates for each CPU. ’all’ implies all of the above options and ’none’ turns off separation.
--event	Defines the events to be measured as well as the interval (in number of events) to sample the Program Counter. The **--list-events** option gives a complete list of the available events, where the commonly used are: CYCLES which measures PPU; SPU_CYCLES which measures the SPUs.

At this point, OProfile environment is ready to initiate the sampling. We may proceed with the following commands:

Example 5-19 Initiating Profiling

```
opcontrol --start # Fires profiling
app # start application ‘app’ (replace with the desired one)
```

Step 4: Stopping

As soon as the application returns, we should stop the sampling and dump the results:

Example 5-20 Stopping and collecting results

```
opcontrol --stop
opcontrol --dump
```

Step 5: Reports

After sampling data is collected, OProfile offers a utility, **opreport**, to format and generate profiles. The opreport tool allows the output to be created based on symbols and files names, as well as references back to the source code (e.g. how many times a function did perform). The following example contains commonly used options:

Example 5-21 Commonly used options for opreport

```
opreport -X -g -l -d -o output.opm
```

-X	Output generated in XML format. Required when used in conjunction with the VPA tool (see Visual Performance Analyzer).

-g	Maps each symbol to its corresponding source file and line.
-l	Organize sampling information per symbol.
-d	For each symbol, shows per-instruction details.
-o	Specifies the output file name.

5.6.4 Performance Debugging Tool (PDT)

The PDT provides tracing means for recording significant events during program execution and maintaining the sequential order of events. The main objective of the PDT is to provide the ability to trace events of interest, in real time, and record relevant data from the SPEs and PPE.

This objective is achieved by instrumenting the code that implements key functions of the events on the SPEs and PPE and collecting the trace records. This instrumentation requires additional communication between the SPEs and PPE as trace records are collected in the PPE memory. The traced records can then be viewed and analyzed using additional SDK tools.

Operation

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

Prior to each application run, the user can configure the PDT to trace events of interest. The user can also use the PDT API to dynamically control the tracing.

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

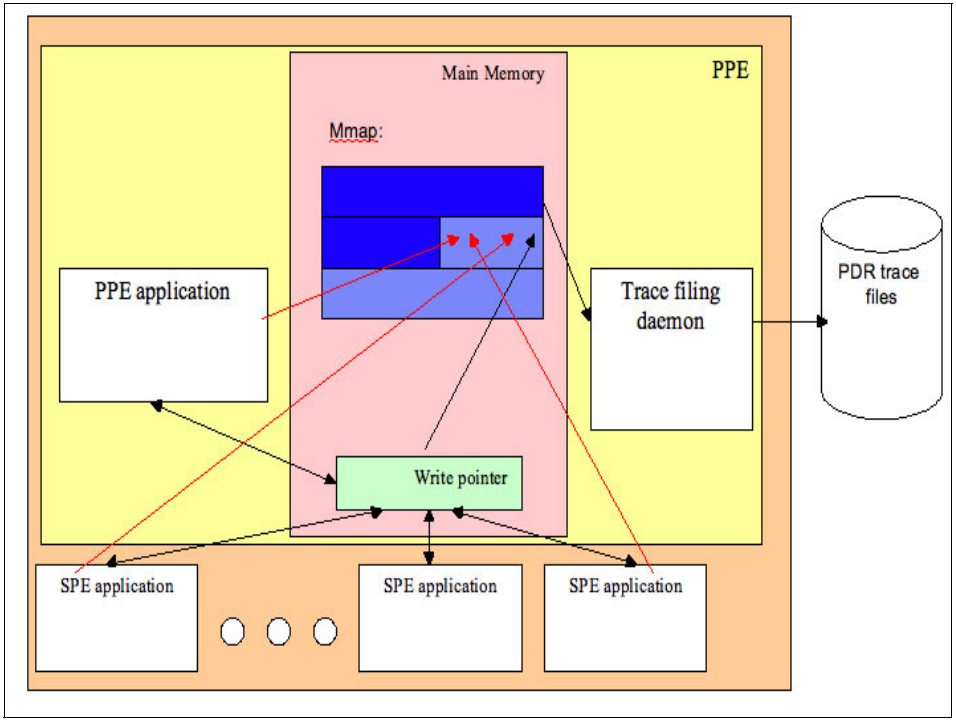


Figure 5-26 Tracing Architecture

Considerations

Tracing 16 SPEs using one central PPE might lead to a heavy load on the PPE and the bus, and therefore, might influence the application performance. The PDT is designed to reduce the tracing execution load and provide a means for throttling the tracing activity on the PPE and each SPE. In addition, the SPE tracing code size is minimized so that it fits into the small SPE local store.

Events tracing is enabled by instrumenting selected function of the following SDK libraries:

- on the PPE: DaCS, ALF, libspe2, and libsync
- on the SPE: DaCS, ALF, libsync, the spu_mfcio header file, and the overlay manager.

Performance events are captured by the SDK functions that are already instrumented for tracing. These functions include; SPEs activation, DMA transfers, synchronization, signaling, user-defined events, etc. Statically linked applications should be compiled and linked with the trace-enabled libraries. Applications using shared libraries are not required to be rebuilt.

Overall Process

The PDT tracing facility is designed to minimize the effort that is needed to enable the tracing facility for a given application. The process includes compiling (in most cases on the SPU code needs to be compiled since it is statically linked), linking with trace libraries, setting environment variables, (optionally) adjusting the trace configuration file and running the application.

Step 1: Compilation

The compilation part involves the specification of a few tracing flags and the tracing libraries. There two sets of procedures (one for PPE and one for SPE) involved:

SPE

1. Addition of the following compilation flags (*CFLAGS* variable in SDK Makefile):

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

2. Addition of the trace headers to the beginning of the include path (*INCLUDE* variable in SDK Makefile):

   ```
   -I/usr/spu/include/trace
   ```

3. Addition of the instrumented libraries in /usr/spu/lib/trace (e.g. libtrace.a) to the linker process (*LDFLAGS* variable in SDK Makefile for the library path and *IMPORTS* variable in SDK Makefile for the library) :

   ```
   -L/usr/spu/lib/trace -ltrace
   ```

4. (OPTIONAL) To enable the correlation between events and the source code, for the analysis tools, the application should be rebuilt using the linking relocation flags (*LDFLAGS* variable in SDK Makefile):

   ```
   -Wl,q
   ```

PPE

1. (Optional, if using libsync) Addition of the following compilation flag (*CFLAGS* variable in SDK Makefile):

   ```
   -DLIBSYNC_TRACE
   ```

2. Addition of the trace headers to the beginning of the include path (*INCLUDE* variable in SDK Makefile):

   ```
   -I/usr/include/trace
   ```

3. Addition of the instrumented libraries (e.g. libtrace.a) in /usr/lib/trace (or /usr/lib64/trace for 64bit applications) to the linker process (*LDFLAGS* variable in SDK Makefile for the library path and *IMPORTS* variable in SDK Makefile for the library):

```
-L/usr/lib/trace -ltrace
```

or

```
-L/usr/lib64/trace -ltrace
```

4. (OPTIONAL) To enable the correlation between events and the source code, for the analysis tools, the application should be rebuilt using the linking relocation flags (*LDFLAGS* variable in SDK Makefile):

```
-Wl,q
```

Step 2: Preparing the Run Environment

After the build process is complete, PDT requires a few environment variables to be set, prior to the application run:

Table 5-6 PDT Environment Variables

Variable	Definition
LD_LIBRARY_PATH	The full path to the traced library location: /usr/lib/trace (or -L/usr/lib64/trace for 64bit applications)
PDT_KERNEL_MODULE	PDT kernel module installation path. Should be /usr/lib/modules/pdt.ko
PDT_CONFIG_FILE	The full path to the PDT configuration file for the application run. The PDT package contains a pdt_cbe_configuration.xml file in the /usr/share/pdt/config directory that can be used "as is" or copied and modified for each application run.
PDT_TRACE_OUTPUT	(Optional) The full path to the PDT output directory (must exist prior to the application run)
PDT_OUTPUT_PREFIX	Optional variable is used to add a prefix to the PDT output files names.

Step 2a: (OPTIONAL - Recommended) Preparing Configuration Files

A configuration XML file is used to configure the PDT. The PDT tracing facility that is built into the application at run time reads the configuration file that is defined by the PDT_CONFIG_FILE environment variable. The /usr/share/pdt/config directory contains a reference configuration file (pdt_cbe_configuration.xml). This file should be copied and then specifically modified for the requirements of each application.

The /usr/share/pdt/config directory also contains reference configuration files for applications that are using the DaCS and ALF libraries: pdt_dacs_config_cell.xml for DaCS and pdt_alf_config_cell.xml for ALF. In addition, a pdt_libsync_config.xml reference file is provided for applications that are using the libsync library.

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

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

The SDK provides instrumentation for the following libraries (events are defined in the XML files):

GENERAL (pdt_general.xml)

These are the general trace events such as trace start, trace stop, etc. Tracing of these events is always active.

LIBSPE2 (pdt_lbspe2.xml)

These are the libspe2 events.

SPU_MFCIO (pdt_mfcio.xml)

These are the spu_mfcio events that are defined in the spu_mfcio.h header file.

LIBSYNC (pdt_libsync.xml)

These are the mutex events that are part of the libsync library.

DACS (pdt_dacs.xml, pdt_dacs_perf.xml, and pdt_dacs_spu.xml)

These are the DaCS events (separated into three groups of events).

ALF (pdt_alf.xml, pdt_alf_perf.xml, and pdt_alf_spu.xml)

These are the ALF events (separated into three groups of events).

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

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

It is highly recommended that tracing be enabled only for those events that are of interest. It is specially recommended to turn off tracing of non-stalling events, since the relative overhead there is higher. Depending on the number of processors involved, programs might produce events at a high rate. If this scenario occurs, the number of traced events might also be very high.

Step 3: Run

After both the environment variables and configuration files are set, simply execute the application. The PDT will produce trace files in a directory that is defined by the environment variable PDT_TRACE_OUTPUT. If this environment variable is not defined, the output location is taken from the definition provided by the output_dir attribute in the PDT configuration file. If neither is defined, the current path will be used. The output directory must exist prior to the application run, and the user must have a write access to this directory. The PDT creates the following files in that output directory at each run.

Table 5-7 Output files

File	Contents
<prefix>-<app_name>-yyyymmddhhmmss.pex	Meta file of the trace.

File	Contents
<prefix>-<app_name>-yyyymmddhhmmss.maps	Maps file from /proc/<pid>/ for the address-to-name resolution performed by the PDTR tool (or pdtr command).
<prefix>-<app_name>-yyyymmddhhmmss.<N>.trace	Trace file. An application may produce multiple trace files where N is the index.

Notes:

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

PDTR

The PDTR tool (pdtr command) is a command-line tool that provides both viewing and post processing of PDT traces on the target (client) machine. The other alternative would be the graphical Trace Analyzer, part of VPA (explained further on).

To use this tool, you must instrument your application by building with the PDT. After the instrumented application has run and created the trace output files, the pdtr command can be run to show the trace output.

for example, given a PDT trace fileset:

```
app-20071115094957.1.trace
app-20071115094957.maps
app-20071115094957.pex
```

run pdtr as follows:

```
pdtr [options] app-20071115094957
```

which produces the output file:

```
app-20071115094957.pep
```

PDTR Produces various summary output reports with lock statistics, DMA statistics, Mailbox usage statistics and overall event profiles The tool is also capable of producing sequential reports with time-stamped event and its parameters per line.

The following are examples of reports produced by PDTR. See the PDTR man page for additional output examples and usage details.

Example 5-22 General Summary Report Example

```
General Summary Report
======================================================================

1.107017 seconds in trace

Total trace events: 3975
Count EvID  Event                                  min     avg     max        evmin,evmax
-------------------------------------------------------------------------------------------
  672 1202  SPE_MFC_READ_TAG_STATUS         139.7ns 271.2ns 977.8ns           26, 3241
  613 0206  _DACS_HOST_MUTEX_LOCK           349.2ns   1.6us  20.3us          432, 2068
  613 0406  _DACS_HOST_MUTEX_UNLOCK
  336 0402  SPE_MFC_GETF
  240 1406  _DACS_SPE_MUTEX_UNLOCK
  239 1206  _DACS_SPE_MUTEX_LOCK           279.4ns   6.6us 178.7us          773, 3152
  224 0102  SPE_MFC_PUTF
   99 0200  HEART_BEAT
   96 0302  SPE_MFC_GET
   64 1702  SPE_READ_IN_MBOX               139.7ns   3.7us  25.0us           21,  191
   16 0002  SPE_MFC_PUT
   16 2007  _DACS_MBOX_READ_ENTRY
   16 2107  _DACS_MBOX_READ_EXIT_INTERVAL   11.7us  15.2us  25.9us          557,  192
   16 0107  _DACS_RUNTIME_INIT_ENTRY
   16 2204  _DACS_MBOX_WRITE_ENTRY
   16 0207  _DACS_RUNTIME_INIT_EXIT_INTERVAL    6.6us   7.4us   8.2us           29, 2030
   16 2304  _DACS_MBOX_WRITE_EXIT_INTERVAL    4.1us   6.0us  11.3us         2011,  193
   16 0601  SPE_PROGRAM_LOAD
   16 0700  SPE_TRACE_START
   16 0800  SPE_TRACE_END
   16 2A04  _DACS_MUTEX_SHARE_ENTRY
...
```

Example 5-23 Sequential Trace Output Example

```
----- Trace File(s) ----------------------------------------------
Event type size: 0
Metafile version: 1
```

```
    0    0.000000      0.000ms  PPE_HEART_BEAT           PPU 00000001 TB:0000000000000000
    1    0.005025      5.025ms  PPE_SPE_CREATE_GROUP     PPU F7FC73A0 TB:000000000001190F  *** Unprocessed event ***
    2    0.015968     10.943ms  PPE_HEART_BEAT           PPU 00000001 TB:0000000000037D13
    3    0.031958     15.990ms  PPE_HEART_BEAT           PPU 00000001 TB:000000000006FB69
    4    0.047957     15.999ms  PPE_HEART_BEAT           PPU 00000001 TB:00000000000A7A3B
    5    0.053738      5.781ms  PPE_SPE_CREATE_THREAD    PPU F7FC73A0 TB:00000000000BBD90
    6    0.053768      0.030ms  PPE_SPE_WRITE_IN_MBOX    PPU F7FC73A0 TB:00000000000BBF3F  *** Unprocessed event ***
    :
   20           0      0.000us  SPE_ENTRY                SPU 1001F348 Decr:00000001
   21         163     11.384us  SPE_MFC_WRITE_TAG_MASK   SPU 1001F348 Decr:000000A4  *** Unprocessed event ***
   22         170      0.489us  SPE_MFC_GET              SPU 1001F348 Decr:000000AB Size: 0x80 (128), Tag: 0x4 (4)
   23         176      0.419us  SPE_MFC_WRITE_TAG_UPDATE SPU 1001F348 Decr:000000B1  *** Unprocessed event ***
   24         183      0.489us  SPE_MFC_READ_TAG_STATUS_ENTRY SPU 1001F348 Decr:000000B8
   25         184      0.070us  SPE_MFC_READ_TAG_STATUS_EXIT SPU 1001F348 Decr:000000B9   >>>  delta tics:1 (   0.070us)
rec:24 {DMA done[tag=4,0x4] rec:22   0.978us 130.9MB/s}
   26         191      0.489us  SPE_MUTEX_LOCK_ENTRY     SPU 1001F348 Lock:1001E280 Decr:000000C0
    :
   33        4523      0.210us  SPE_MUTEX_LOCK_EXIT      SPU 1001F348 Lock:1001E280 Decr:000011AC   >>>  delta tics:3 (
0.210us) rec:32
   34           0        0.000us  SPE_ENTRY                SPU 1001F9D8 Decr:00000001
   35          96      6.705us  SPE_MFC_WRITE_TAG_MASK   SPU 1001F9D8 Decr:00000061  *** Unprocessed event ***
   36         103      0.489us  SPE_MFC_GET              SPU 1001F9D8 Decr:00000068 Size: 0x80 (128), Tag: 0x4 (4)
   37         109      0.419us  SPE_MFC_WRITE_TAG_UPDATE SPU 1001F9D8 Decr:0000006E  *** Unprocessed event ***
   38         116      0.489us  SPE_MFC_READ_TAG_STATUS_ENTRY SPU 1001F9D8 Decr:00000075
   39         117      0.070us  SPE_MFC_READ_TAG_STATUS_EXIT SPU 1001F9D8 Decr:00000076   >>>  delta tics:1 (   0.070us)
                rec:38 {DMA done[tag=4,0x4] rec:36 0.978us 130.9MB/s}
```

Example 5-24 Lock Report Example

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


     Accesses
               Hits           Misses       Hit hold time (uS)   Miss wait time (uS)

Acount
  %Total          Count %Acount  Count %Acount   min,   avg,    max   min,   avg,    max     Name
--------------------------------------------------------------------------------------------------------------------------------

*
   600 (100.0 )           3 (  0.5 )   597 ( 99.5 )    100.8, 184.6, 402.4   13.3, 264.4,  568.0   shr_lock (0x10012180)
                           2
 ( 66.7 )   298 ( 49.9 )    100.8, 101.1, 101.5  181.8, 249.7,  383.6   main (0x68c)(lspe=1)
                           1
 ( 33.3 )   199 ( 33.3 )    200.7, 201.3, 202.5   13.3, 315.2,  568.0   main (0x68c)(lspe=2)
                           0
 (  0.0 )   100 ( 16.8 )      0.0,   0.0,   0.0  205.0, 206.8,  278.5   main (0x68c)(lspe=3)
*
-Implicitly initialized locks (used before/without mutex_init)
```

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

5.6.5 FDPR-Pro

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

Operation

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

1. Instrumentation phase, where The optimizer creates an instrumented executable program and an empty template profile file
2. Training phase, where the instrumented program is executed with a representative workload and as it runs it updates the profile file.
3. Optimization phase, where the optimizer generates the optimized executable program file. You can control the behavior of the optimizer with options specified on the command line.

Considerations

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

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

Overall process

The typical FDPR-Pro process encompasses the three steps as mentioned above: Instrumentation, Training and Optimization.

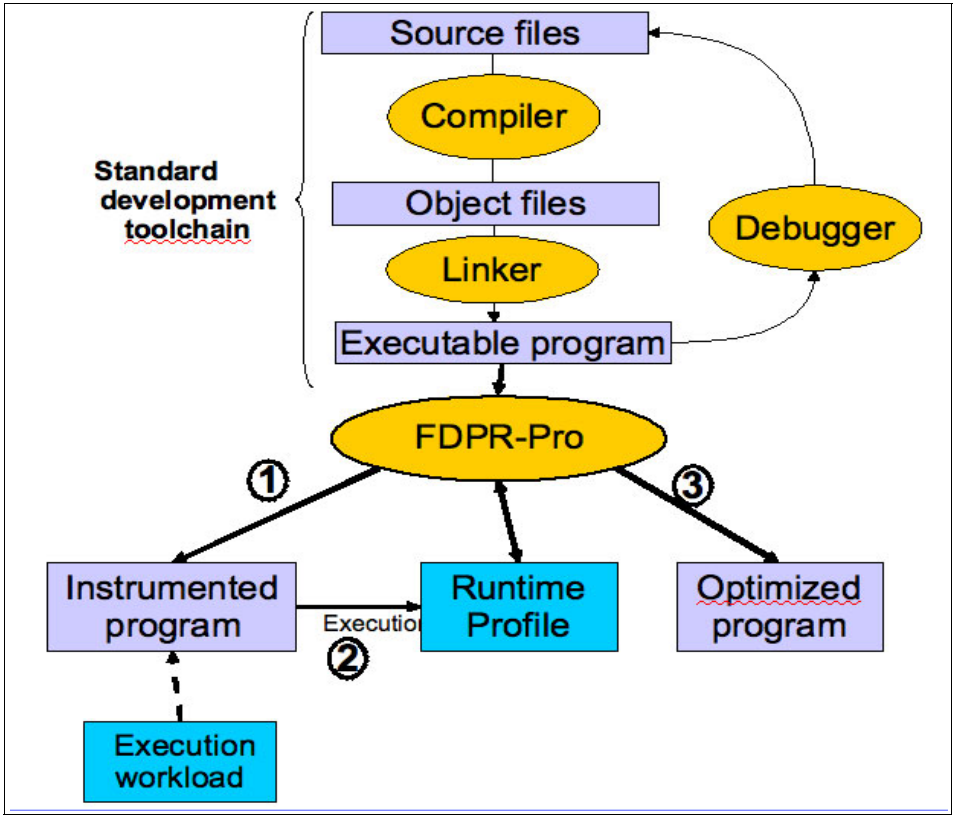


Figure 5-27 FDPR-Pro process

Step 0: Prepare input files

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

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

If you are using the SDK Makefiles structure (with make.footer), set the following variables (depending on the compiler) in the Makefile:

Example 5-25 make.footer variables

```
LDFLAGS_xlc += -Wl,q # for XLC
LDFLAGS_gcc += -Wl,q # for GCC
```

Step 1: Instrumentation Phase

PPE and SPE executables are instrumented differently. For PPE executable, the fdprpro command creates an instrumented file and a profile file. The profile file is later populated with profile information while the instrumented program runs with a specified workload. In contrast, for SPE executable, the profile (with extension .mprof) is created only when the instrumented program runs.

FDPR-Pro processes (instruments or optimizes) the PPE program and the embedded SPE images. Processing the SPE images is done by extracting them to external files, processing each of them, then encapsulating them back into the PPE executable. Two modes are available in order to fully process the PPE/SPE combined file: *integrated mode*, and *standalone mode*.

Integrated Mode

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

Example 5-26 Instrumentation in integrated mode

```
fdprpro -cell -a instr myapp -o myapp.instr
```

Standalone Mode

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

1. Extraction

   SPE images are extracted from the input program and written as executable files in the specified directory.

Example 5-27 Extraction in standalone mode

```
fdprpro -a extract -spedir somedir myapp
```

2. Processing

   The SPE images are processed one by one. You should place all of the output files in a distinct directory.

Example 5-28 Processing in standalone mode

```
fdprpro -a (instr|opt) somedir/spe_i [-f prof_i] [opts ...]
outdir/spe_i
```

Note: Replace <spe_i> by the actual name of the spu file obtained in the extraction process. Profile and optimization options need to be specified only in the optimization phase (see below).

3. Encapsulation and PPE processing

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

Example 5-29 Encapsulation in standalone mode

```
fdprpro -a (instr|opt) --encapsulate -spedir outdir [ opts ...] myapp
-o myapp.instr
```

SPE Instrumentation

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

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

Step 2: Training Phase

The training phase consists of running the instrumented application with a representative workload (one that fully exercises the desired code areas). While the program runs with the workload, the PPE profile (by default with .nprof extension) is populated with profile information. Simultaneously, SPE profiles, one for each executed SPE thread, are generated in the current directory.

If an old profile exists before instrumentation starts, fdprpro accumulates new data into it. In this way you can combine the profiles of multiple workloads. If you do not want to combine profiles, remove the old SPE profiles (the .mprof files) and replace the PPE profile (by default .nprof file) with its original copy, before starting the instrumented program.

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

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

Step 3: Optimization Phase

The optimization phase takes all profiling information generated during the test phase in order to optimize the application.

Example 5-30 Typical optimization

```
fdprpro -a opt -f myapp.nprof [ opts ... ] myapp -o myapp.fdpr
```

The same considerations shown in the instrumentation phase, with regard to *integrated mode* and *standalone mode*, are still valid during the optimization phase. The notable exceptions are the action command being performed (here *-a opt*) and the options (here related to optimization phase). The routine is: Extract, Process and Encapsulate

Optimization Options

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

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

5.6.6 Visual Performance Analyzer

The Visual Performance Analyzer (VPA) is an Eclipse based tool set, currently including six plug-in applications working cooperatively: Profile Analyzer, Code

Analyzer, Pipeline Analyzer, Counter Analyzer, Trace Analyzer and the experimental Control Flow Analyzer. It aims to provide a platform independent, easy to use integrated set of graphical application performance analysis tools, leveraging existing platform specific non-GUI performance analysis tools to collect a comprehensive set of data.

In doing so, VPA creates a consistent set of integrated tools to provide a platform independent drill down performance analysis experience.

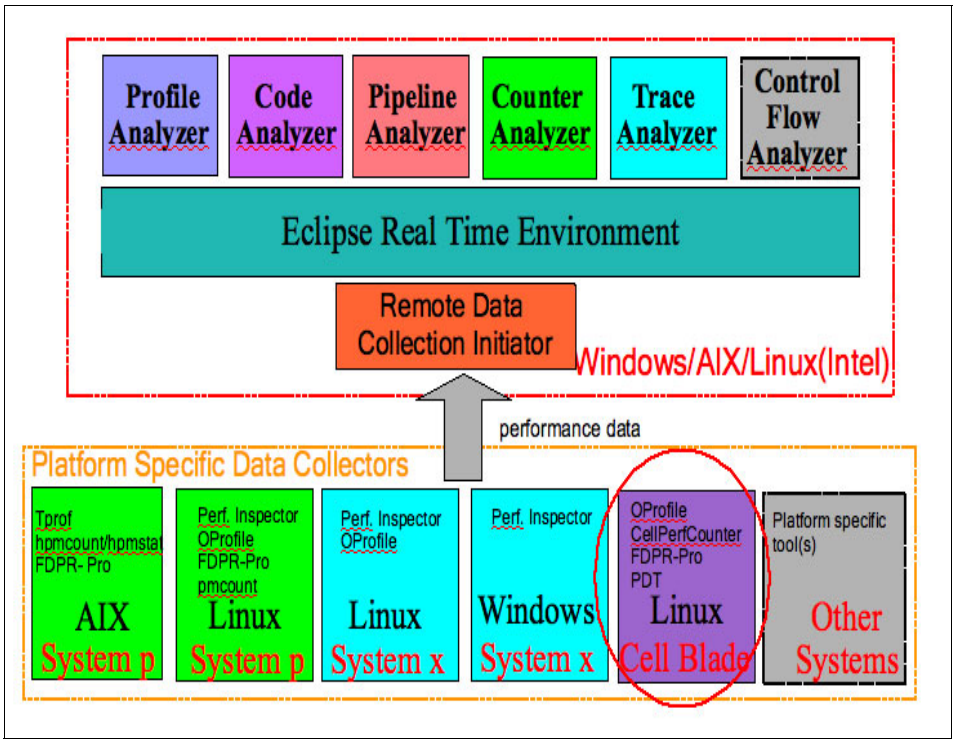


Figure 5-28 VPA Architecture

However VPA does not supply performance data collection tools. Instead, it relies on platform specific tools, like OProfile and PDT, to collect the performance data.

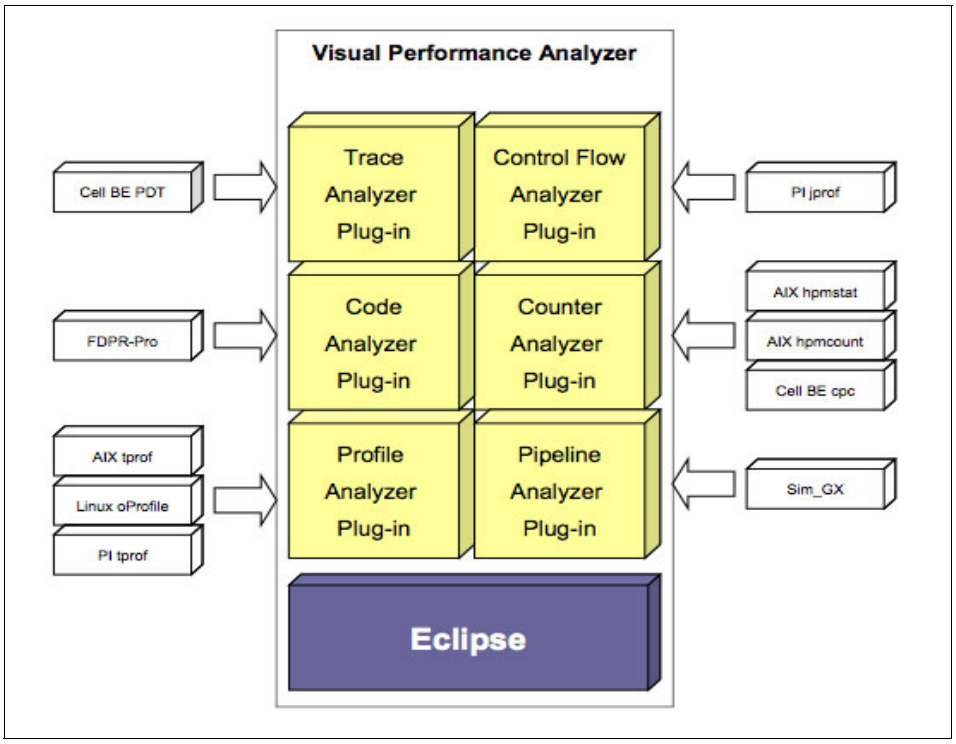


Figure 5-29 Relationship between tools

In general, each visualization tool acts complementary to each other:

Profile Analyzer Is a system profile analysis tool. This plug-in obtains profile information from various platform specific tools, and provide analysis views for user to identify performance bottle necks.

Pipeline Analyzer Gets pipeline information of Power processors, and provides two analysis views; scroll mode and resource mode.

Code Analyzer Reads XCOFF (AIX® binary file format) files or ELF files running on Linux on Power, and displays program structure with block information. With related profile information, it can provide analysis views on hottest program block as well as some optimization suggestions.

Counter Analyzer Reads counter data files generated by AIX hpmcount/hpmstat, and it provides multiple views to help users identify and eliminate performance bottlenecks by examine the hardware performance counter values,

computed performance metrics and also CPI breakdown models.

Trace Analyzer Reads in traces generated by the Performance Debugging Tool for Cell BE, and displays time-based graphical visualization of the program execution as well as a list of trace contents and the event details for selection.

Control Flow Analyzer Reads the call trace data file, and display execution flow graph and call tree to help user analyze when and where one method invocation happens, and how long it runs.

The VPA is better described by its own individual tools. In the Cell BE particular case, we have the following available relevant tools:

- oProfile
- CPC
- PDT
- FDPR-Pro

Considering the list of Cell BE performance tools above, we will be focusing on each respective visualization tool, namely: Profile Analyzer, Counter Analyzer, Trace Analyzer and Code Analyzer.

Note: In the following sections you will find a brief overview on each Cell BE relevant VPA tool. For more in depth coverage of tool usage, please see Chapter 6, “Using Performance Tools” on page 411.

Profile Analyzer

Profile Analyzer is a tool that allows you to navigate through a system profile, looking for performance bottlenecks. It provides a powerful set of graphical and text-based views to allow users to narrow down performance problems to a particular process, thread, module, symbol, offset, instruction or source line. It supports profiles generated by the Cell BE OProfile.

Overall Process

The Profile Analyzer works with properly formatted data, gathered by OProfile. The initial step consists in running the desired application with OProfile, followed by formatting its data. After that, we load the information in the Profile Analyzer and explore its visualization options.

Step 1: Collect profile data

As outlined by the OProfile section above (see OProfile), initiate the profile session as usual, adding the selected events to be measured and, as a required

step for VPA, configure the session to properly separate the samples with "**opcontrol --separate=all**"

Next, as soon as the measuring is ended, prepare the output data for VPA with the following tool:

Example 5-31 Formatting OProfile output for VPA

```
opreport -X –g –d –l myapp.opm
```

Note: Consult the OProfile section above for an explanation on the options.

Step 2: Load Data

Start the VPA tool and select the Profile Analyzer (**Tools** → **Profile Analyzer**).

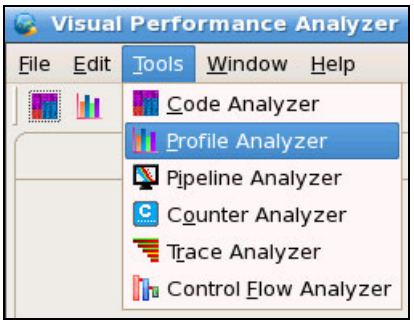


Figure 5-30 Selecting Profile Analyzer

Since we already have profile data from the previous step, simply load the information by going to **File** → **Open FIle,** and selecting the .opm file generated in the previous step.

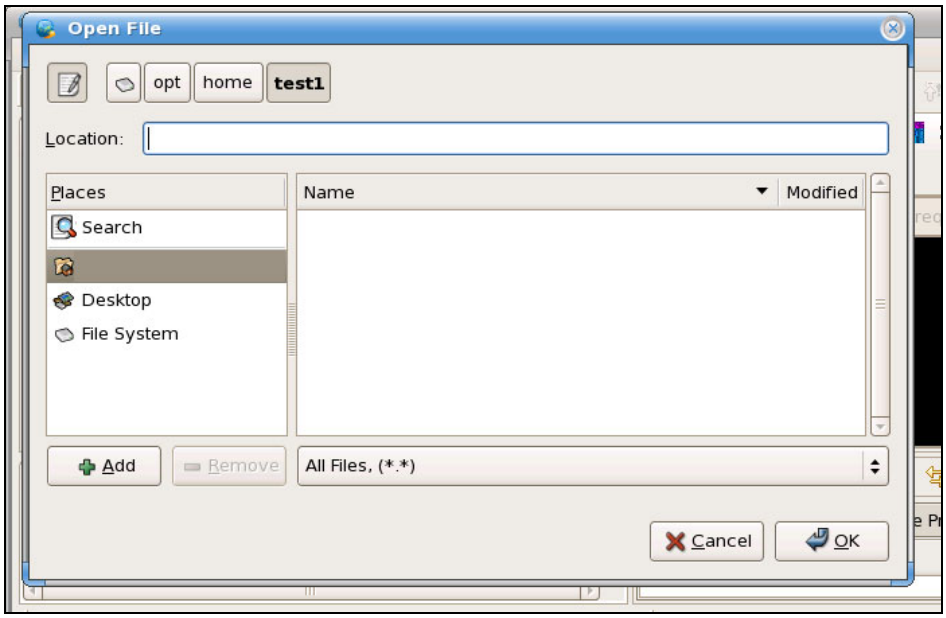


Figure 5-31 Open file screen

In VPA a profile data file loading process is able to run as a background runnable job. When VPA is loading a file, you can click a button to put the loading job to run in the background. While the loading job is running in the background, you can use Profile Analyzer to view already loaded profile data files, or event start another loading job at the same time.

The Process hierarchy view appears by default in the top center pane. It shows an expandable list of all processes within the current profile. You can expand a process to view its module, later thread and etc. You can also view the profile in the form of thread or module and etc. Actually, you can define the hierarchy view by right- click profile and choose Hierarchy Management.

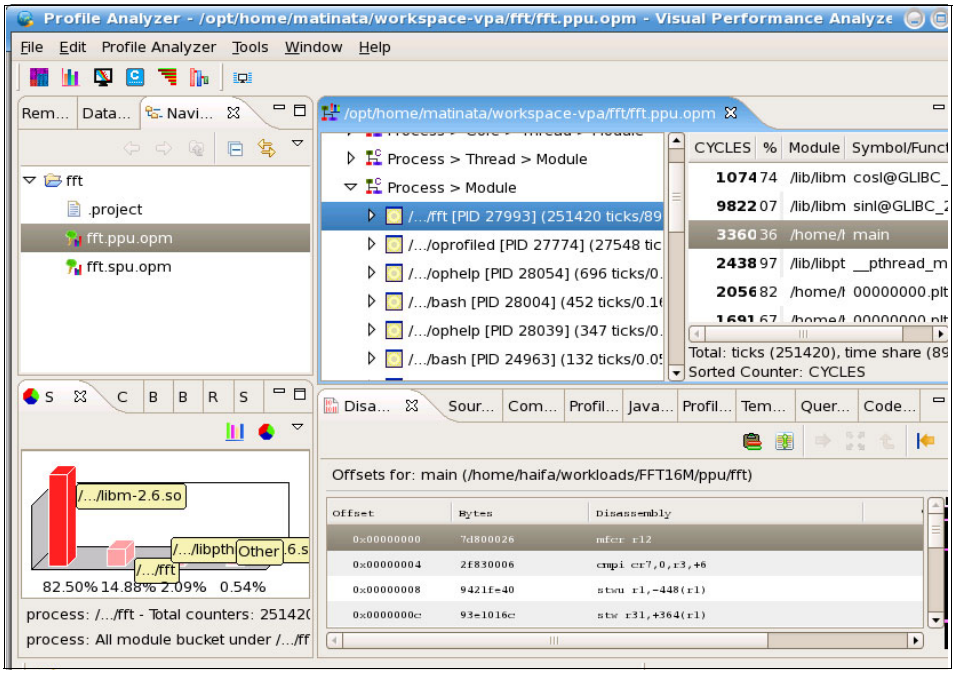


Figure 5-32 Process Hierarchy view

Code Analyzer

Code Analyzer displays detailed information on basic blocks, functions and assembly instructions of executable files and shared libraries. It is built on top of FDPR-Pro (Feedback Directed Program Restructuring) technology and allows adding of FDPR-Pro and tprof profile information. Code Analyzer is able to show statistics, to navigate the code, to display performance comment and grouping information on the executable and to map back to source code.

Overall Process

The Code Analyzer works with the artifacts generated from the FDPR-Pro session on your executable. Initially the original executable is loaded, followed by the .nprof files generated. After that, you should be able to visualize the information

Step 1: Collect profile data

Initially, we should run at least one profiling session (without optimization) with your application. As previously explained at the FDPR-Pro section (see FDPR-Pro), the first step is to instrument the desired application followed by the actual training (i.e. profile information generation). The expected results are profiles files for both PPU (*.nprof) and SPU codes (*.mprof).

Example 5-32 Typical FDPR-Pro session

```
rm *.mprof # remove old data
mkdir ./somespudir# create temp folder
fdprpro -a instr -cell -spedir somespudir myapp # instrument
myapp ... # run your app with a meaninful workload
```

Note: Consult the FDPR-PRO section above for an explanation on the options.

Step 2: Load Data

Start the VPA tool and select the Code Analyzer (**Tools** → **Code Analyzer**).

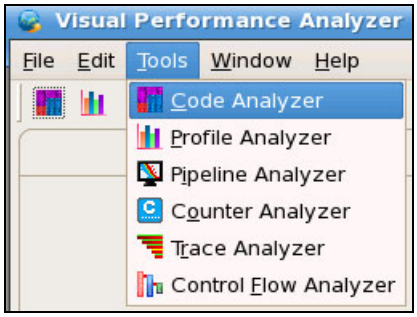


Figure 5-33 Selecting Code Analyzer

First, locate the original application binary add it to the Code Analyzer choosing **File** → **Code Analyzer** → **Analyze Executable.** Select the desired file and press **Open.**

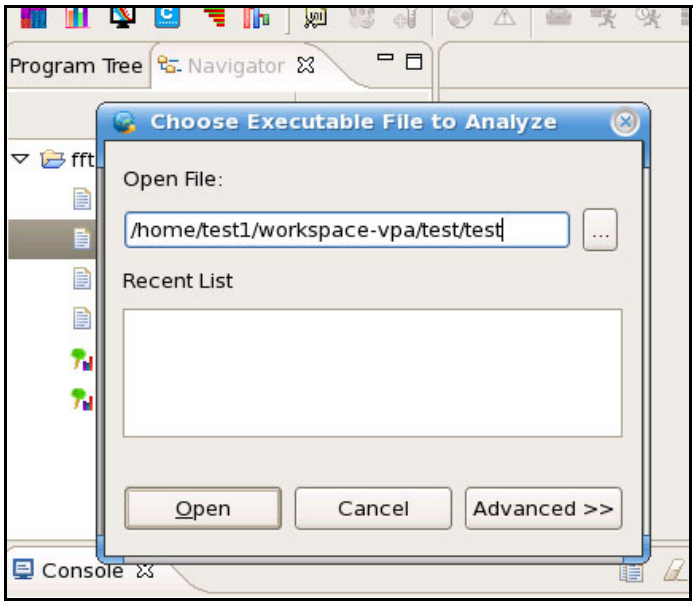


Figure 5-34 Open file screen

The executable will be loaded, and information tabs should appear for both ppu code and spu code.

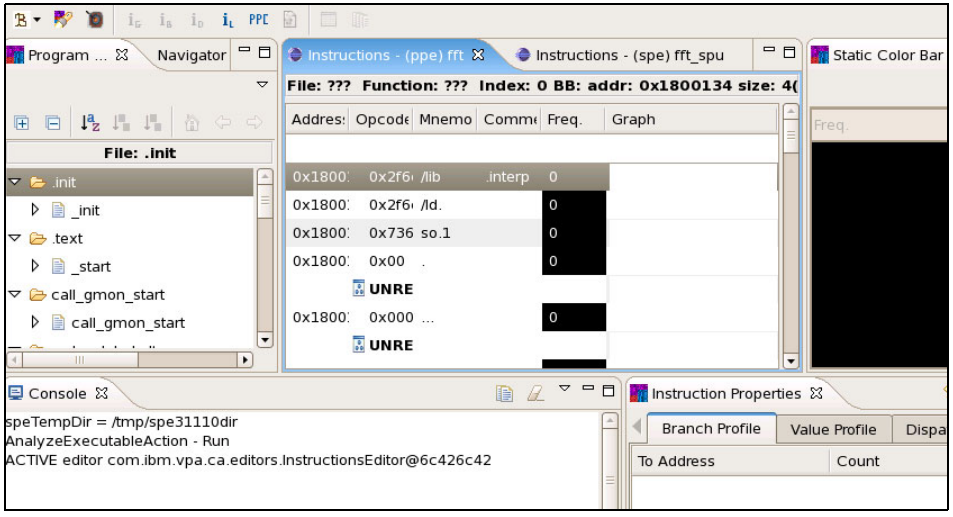


Figure 5-35 Executable information

To enhance the views with profile information for the loaded executable, you can either an instrumentation profile file or a sampling profile. For that, choose **File** → **CodeAnalyzer** → **Add Profile Information** for each of the executable tabs available in the center of the screen. There needs to be matching between the profile information added and the executable tab selected: for ppu tabs, add the *.nprof profiles and for spus, add the respective *.nprof file.

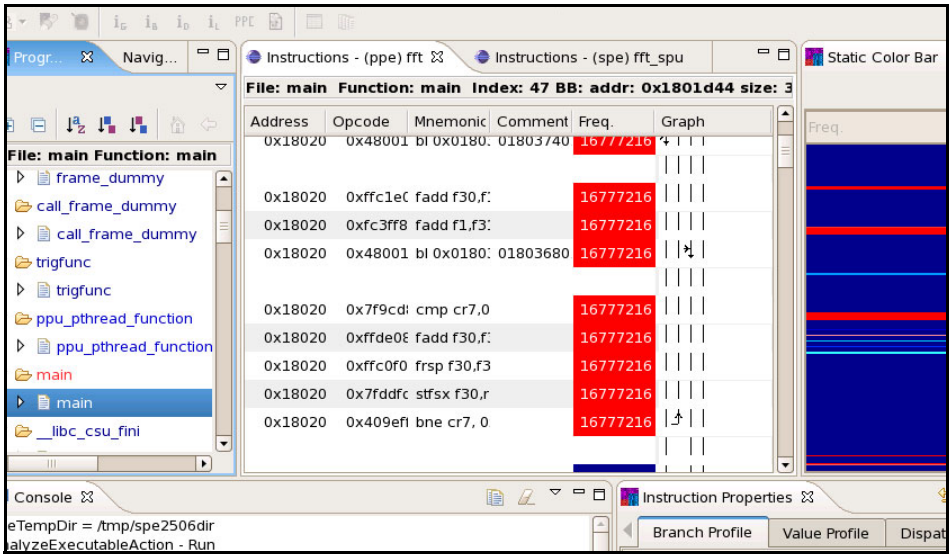


Figure 5-36 Added profile information

Trace Analyzer

Trace Analyzer visualizes Cell BE traces containing information such as DMA communication, locking and unlocking activities, mailbox messages, etc. Trace Analyzer shows this data organized by core along a common time line. Extra details are available for each kind of event: for example, lock identifier for lock operations, accessed address for DMA transfers and etc.

The tool introduces a few concepts that we should be familiar with:

Events — Events are records that have no duration, for example, records describing non-stalling operations, such as releasing a lock. Events' input on performance is normally insignificant, but they may be important for understanding the application and tracking down sources of performance problems.

Intervals — Intervals are records that may have non-zero duration. They normally come from stalling operations, such as acquiring a lock. Intervals are often a very significant

performance factor, and identifying long stalls and their sources is an important task in performance debugging. A special case of an interval is live interval, that starts when an SPE thread begins to execute and ends when the thread exits.

Overall Process

The Trace Analyzer tool work with trace information generated by the PDT. The tool then processes the trace for analysis and visualization. Most importantly, this processing adds context parameters (e.g., estimated wall clock time, unique SPE thread ids, etc.) to individual records.

Step 1: Collect trace data

As already shown in section 5.6.4, “Performance Debugging Tool (PDT)”, the PDT tool require a few steps to produce the trace files needed by the Trace Analyzer. To summarize, they are:

- Re-compile and linking with instrumented libraries
- Run time environment variables setup.
- Application execution.

If everything is properly configured, we should have three tracing related files in the configured output directory: .pex, .maps and .trace.

Step 2: Loading the trace files

Let’s start the VPA tool and select the Trace Analyzer (**Tools** → **Trace Analyzer**).

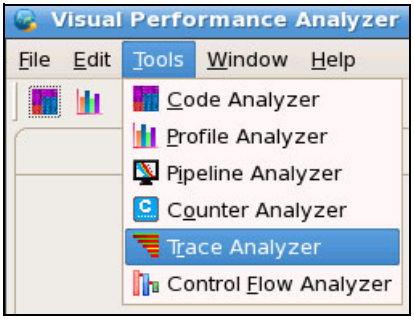


Figure 5-37 Selecting Trace Analyzer

Go to **File** → **Open File** and locate the .pex file generated during the tracing session. After loading in the trace data, the Trace Analyzer Perspective displays the data in its views and editors.

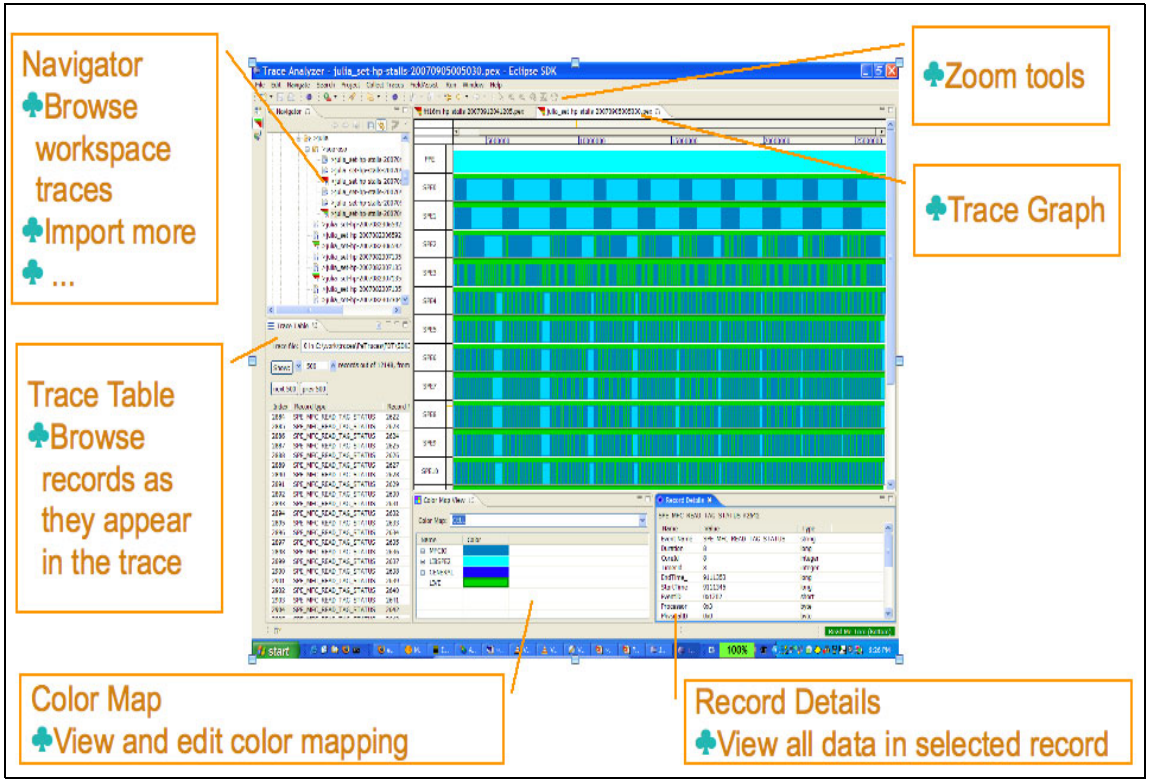


Figure 5-38 Trace Perspective

Going from the top left clockwise, we see:

- Navigator View
- Trace Editor, showing the trace visualization by core, where data from each core is displayed in a separate row, and each trace record is represented by a rectangle. Time is represented on the horizontal axis, so that the location and size of a rectangle on the horizontal axis represent the corresponding event's time and duration. The color of the rectangle represents the type of event, as defined by the Color Map View.

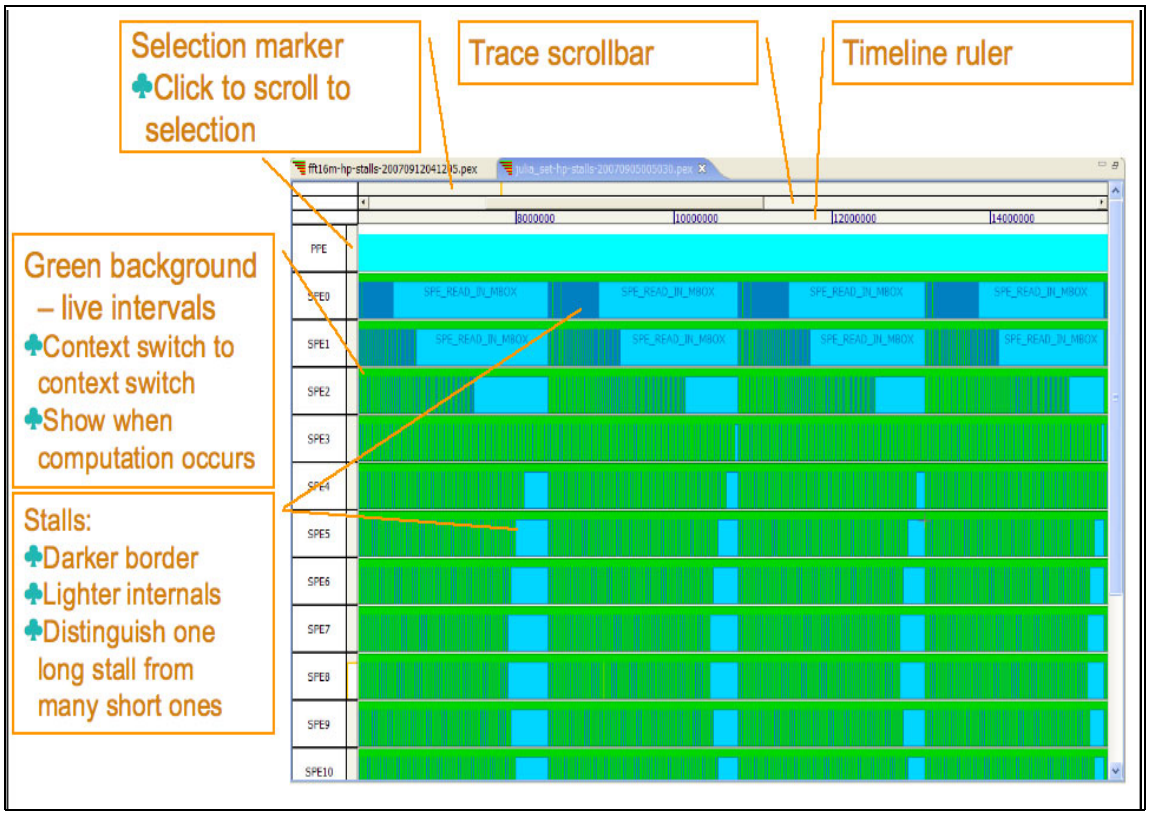


Figure 5-39 Trace Editor details

- Details View, showing the details of the selected record (if any)
- Color Map View, allowing the user to view and modify color mapping for different kinds of events
- Trace Table View, which shows all the events on the trace in the order of their occurrence

Counter Analyzer

The Counter Analyzer tool is a common tool to analyze hardware performance counter data among many IBM eServer™ platforms, which includes systems running on Linux on Cell BE.

The Counter Analyzer tool accepts hardware performance counter data in the form of a cross-platform XML file format. The tool provides multiple views to help user identify the data. The views can be divided into two categories: one category is the “table” views, which are basically two-dimension tables displaying

data. The data could be raw performance counter values, derived metrics, counter comparison results and so on. Another category is the “plot” views. In these views data are represented by different kind of plots. The data could also be raw performance counter values, derived metrics, and comparison results and so on. Besides these “table” views and “plot” views, there are also some “utility” views to help user configure and customize the tool.

Overall Process

In the particular case of Cell BE, the Counter Analyzer work with count data produced by the CPC tool, in .pmf XML format files. After loading in the counter data of .pmf file, the Counter Analyzer Perspective displays the data in its views and editors.

Step 1: Collecting Count Data

In order to obtain the required counter data, you should proceed as explained in the CPC section (see CPC), making sure that the output is generated in XML format.

Typically, you may use CPC as follows:

Example 5-33 Collecting count data

```
cpc -e EVENT,... --sampling-mode -c CPUS -i 100000000 –X file.pmf \
-t TIME # for system wide mode
cpc -e EVENT,EVENT,... -i INTERVAL –X file.pmf some_workload # for
workload mode
```

Step 2: Loading the count data

Start the VPA tool and select the Counter Analyzer (**Tools** → **Counter Analyzer**).

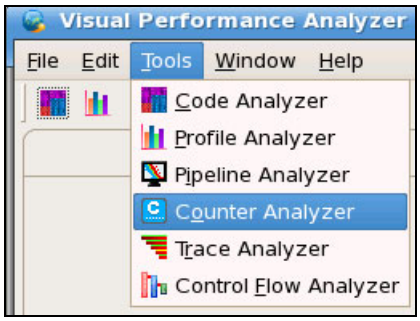


Figure 5-40 Selecting Counter Analyzer

Load the count data by choosing **File** → **Open File** and locating the .pmf file generated by CPC.

After loading in the counter data of .pmf file, the Counter Analyzer Perspective displays the data in its views and editors. Primary information of details, metrics and CPI breakdown is displayed in Counter Editor. Resource statistics information of the file (if available) will be showed in tabular view Resource Statistics. The View Graph illustrates the details, metrics and CPI breakdown in a graphic way.

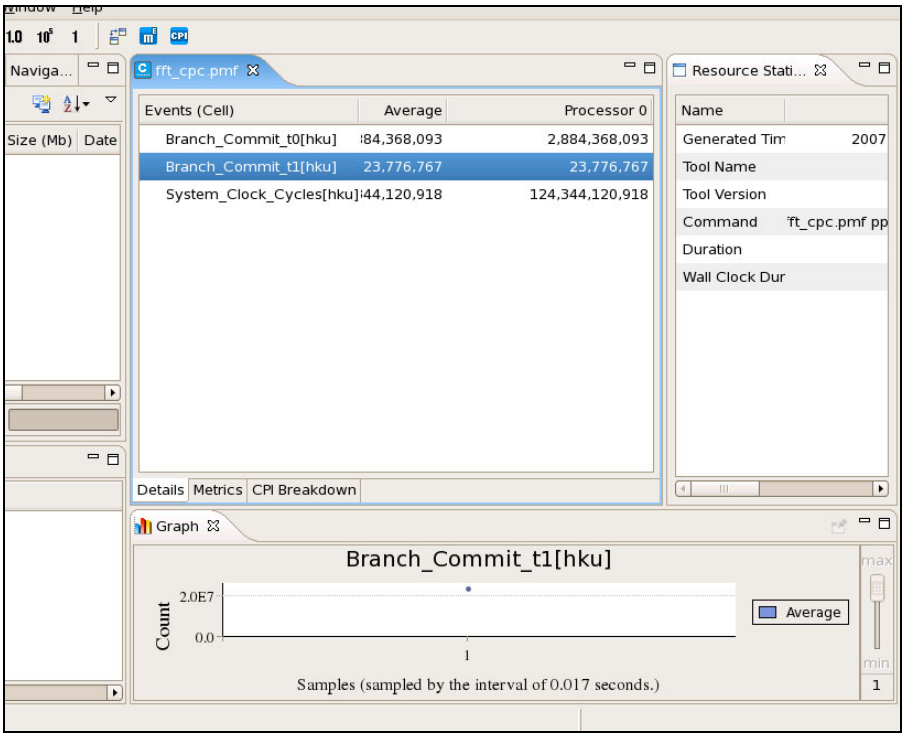


Figure 5-41 Counter Analyzer Perspective

The Counter Analyzer organizes the information according to a few concepts:

- Performance Monitoring

  Counter Performance monitor counter provides comprehensive reports of events that are critical to performance on systems. It is able to gather critical hardware events, such as the number of misses on all cache levels, the number of floating point instructions executed, the number of instruction loads that cause TLB misses.

- Metrics

The metric information is calculated with user-defined formula and event count from performance monitor counter. It's used to provide performance information like CPU utilization rate, million instructions per second. This helps the algorithm designer or programmer identify and eliminate performance bottlenecks.

- CPI Breakdown Model

  Cycles per instruction (CPI) is the measurement for analyzing the performance of a workload. CPI is simply defined as the number of processor clocked cycles needed to complete an instruction. It is calculated as CPI = Total Cycles / Number of Instructions Completed. A high CPI value usually implies under utilization of machine resources.

For more information, consult the VPA manual or go to its homepage in:

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

6

Using Performance Tools

In this chapter we explore a practical “hands-on”example on the use of performance tools, explaining how to collect proper information and how to access relevant visualization features.

6.1 Practical case: FFT16M Analysis

In the section, we present a full example on how to explore the Cell BE performance tools and, specially, how to visualize the results.

6.1.1 The FFT16M

The chosen target sample application for analysis is the FFT16M application that can be found in the Cell SDK 3.0 demos bundle:

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

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

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

6.1.2 Prepare and Build for profiling

For the sake of flexibility, let's setup a "sandbox" styled project tree structure, so we have more flexibility while modifying and generating files:

Step 1: Copy the application from SDK tree

To work on a "sandbox" tree means we are going to have our own copy of the project, on an accessible location (for example your home dir):

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

Step 2:Prepare the Makefile

Go to your recently created project structure and locate the Makefiles. You should be able to find the three of them:

~/FFT16M/Makefile

~/FFT16M/ppu/Makefile

~/FFT16M/ppu/Makefile

Next, let's introduce a few modifications to the Makefiles so we prevent them from trying to install executables back to the SDK tree, and we introduce the required compilation flags for profiling data:

Note 1: No further Makefile modifications, beyond these, are required.

Note 2: There are specific changes depending whether you use gcc our xlc as the compiler

Modifying the ~/FFT16M/ppu/Makefile

In Example 6-1 we comment out install directives. In Example 6-2 we introduce the -g and -Wl,-q compilation flags in order to preserve the relocation and the line number information in the final integrated executable.

Example 6-1 Changing ~/FFT16M/ppu/Makefile for gcc

```
##########################################################################
#
#        Target
##########################################################################
#

PROGRAM_ppu= fft

##########################################################################
#
#        Objects
##########################################################################
#

IMPORTS = ../spu/fft_spu.a -lspe2 -lpthread -lm -lnuma

#INSTALL_DIR= $(EXP_SDKBIN)/demos
#INSTALL_FILES= $(PROGRAM_ppu)
LDFLAGS_gcc    = -Wl,-q
CFLAGS_gcc = -g

##########################################################################
#
#        buildutils/make.footer
##########################################################################
#

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

```
   include ../../../../buildutils/make.footer
endif
```

Example 6-2 Changing ~/FFT16M/ppu/Makefile for gcc

```
##########################################################################
#
#        Target
##########################################################################
#

PROGRAM_ppu= fft

##########################################################################
#
#        Objects
##########################################################################
#
PPU_COMPILER = xlc

IMPORTS = ../spu/fft_spu.a -lspe2 -lpthread -lm -lnuma

#INSTALL_DIR= $(EXP_SDKBIN)/demos
#INSTALL_FILES= $(PROGRAM_ppu)
LDFLAGS_xlc = -Wl,-q
CFLAGS_xlc = -g

##########################################################################
#
#        buildutils/make.footer
##########################################################################
#

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

- ~/FFT16M/spu/Makefile
  - Introduce the -g and -Wl,-q compilation flags in order to preserve the relocation and the line number information in the final integrated executable.

Example 6-3 Modifying ~/FFT16M/spu/Makefile for gcc

```
##########################################################################
#
#        Target
##########################################################################
#

PROGRAMS_spu:= fft_spu
LIBRARY_embed:= fft_spu.a

##########################################################################
#
#        Local Defines
##########################################################################
#

CFLAGS_gcc:= -g --param max-unroll-times=1 # needed to keep size of
program down
LDFLAGS_gcc    = -Wl,-q -g

##########################################################################
#
#                        buildutils/make.footer
##########################################################################
#

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

Example 6-4 Modifying ~/FFT16M/spu/Makefile for xlc

```
##########################################################################
#
#        Target
##########################################################################
#
SPU_COMPILER = xlc
PROGRAMS_spu:= fft_spu
LIBRARY_embed:= fft_spu.a
```

```
##########################################################################
#
#        Local Defines
##########################################################################
#

CFLAGS_xlc:= -g -qnounroll -O5
LDFLAGS_xlc:= -O5 -qflag=e:e -Wl,-q -g


##########################################################################
#
#                         buildutils/make.footer
##########################################################################
#

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

Before the actual build, make sure you set the default compiler accordingly by issuing:

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

Now we are ready for the build:

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

6.1.3 Creating and working with profile data

Assuming that we already have a proper set-up project tree and a successful build, let's collect and work with profile data.

Step 1: Collecting data with CPC

Before collecting the application data, execute a small test in order to verify that CPC is properly work. Type the following command for measuring clock-cycles and branch instructions committed on both hardware threads for all processes on all CPUs for 5 seconds, and you should immediately see counter statistics:

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

Given that CPC is properly behaving, let's collect counter data for the FFT16M application. The following example count PPC instructions committed in one event-set, and L1 cache load misses in a second event-set and write the output in the xml format (suitable for counter analyzer) to the file fft_cpc.pmf:

```
cd ~/FFT16M

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

As the result, you should have the following file:

~/FFT16M/fft_cpc.pmf

Step 2: Counter Analyzer

The generated counter information can now be visualized with the Counter Analyzer tool in VPA. For that, proceed as following:

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

The result will be something similar to that shown in Figure 6-1 on page 418, exhibiting the collected counter information:

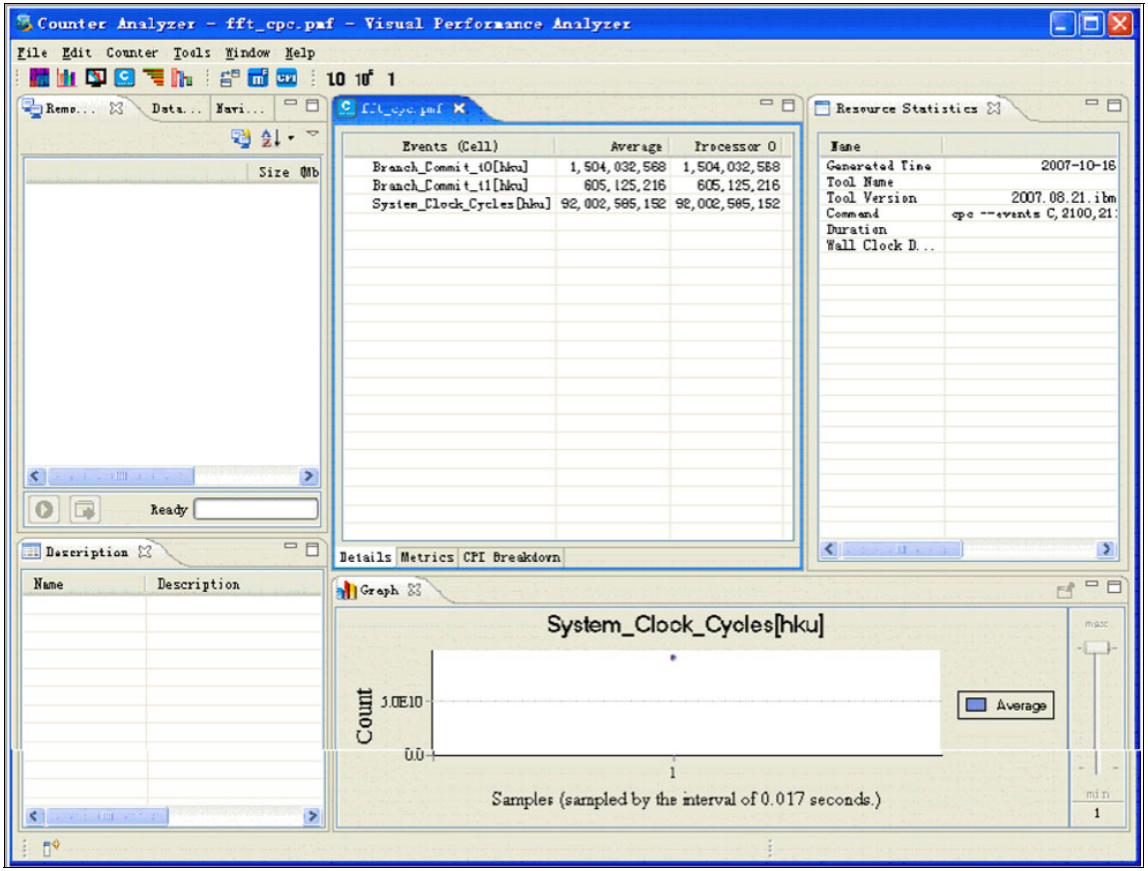


Figure 6-1 Counter Analyzer screen

Step 2: Collecting data with OProfile

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

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

Example 6-5 OProfile initialization and run for SPU profiling

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

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

For generating the report:

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

- Repeat the steps for PPU:

Example 6-6 OProfile initialization and run for PPU profiling

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

For generating the report:

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

Step 3: Profile Analyzer

Let's load the generated profile information with Profile Analyzer:

1. Open VPA and select **Tools** → **Profile Analyzer**
2. Choose **File** → **Open File**
3. Locate the fft.spu.opm file and select it

You should get the following screen:

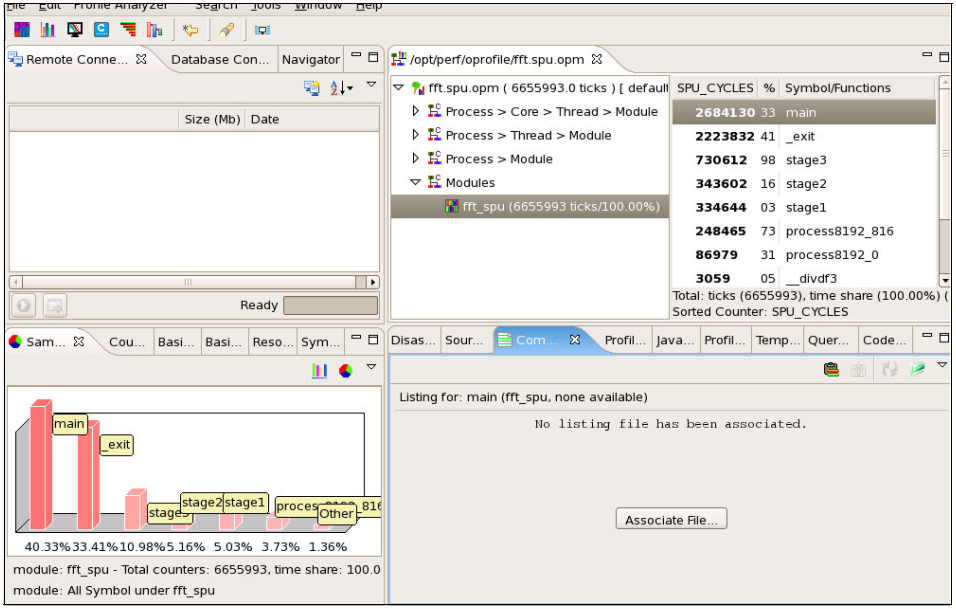


Figure 6-2 fft.spu.opm in Profile Analyzer

Next, let's examine the disassembly information by selecting the fft_spu entry contained inside the Modules section at the center of the screen (see above) and double-clicking the "main" symbol at the right sided Symbol/Functions view. The result should appear in the Disassembly view at the bottom center position.

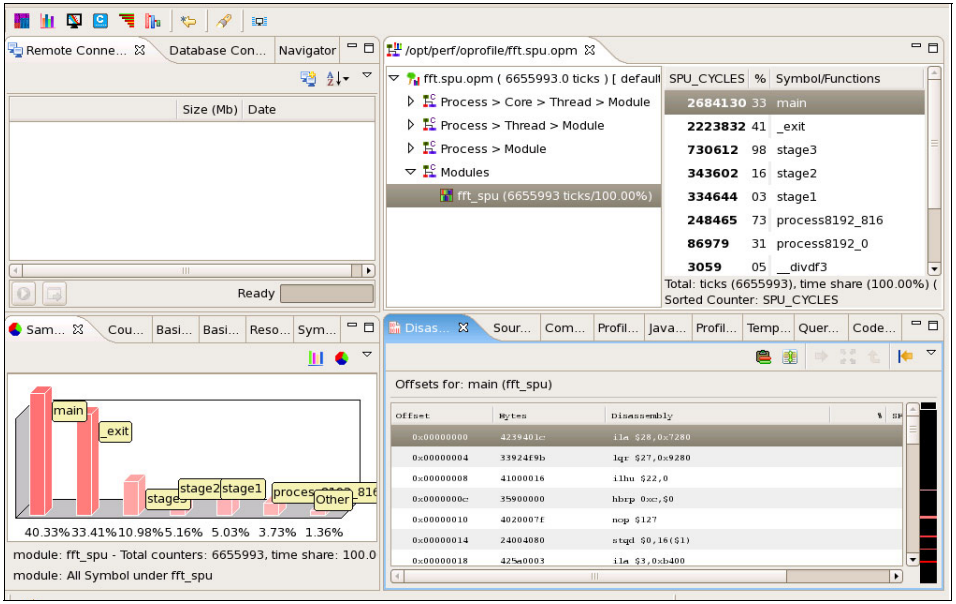


Figure 6-3 Disassembly view for fft_spu.opm

After double-clicking the symbol, the tool may ask you for that particular symbol's source code. In case you do have that, you may also switch to the "Source Code" tab at the bottom center portion of the screen.

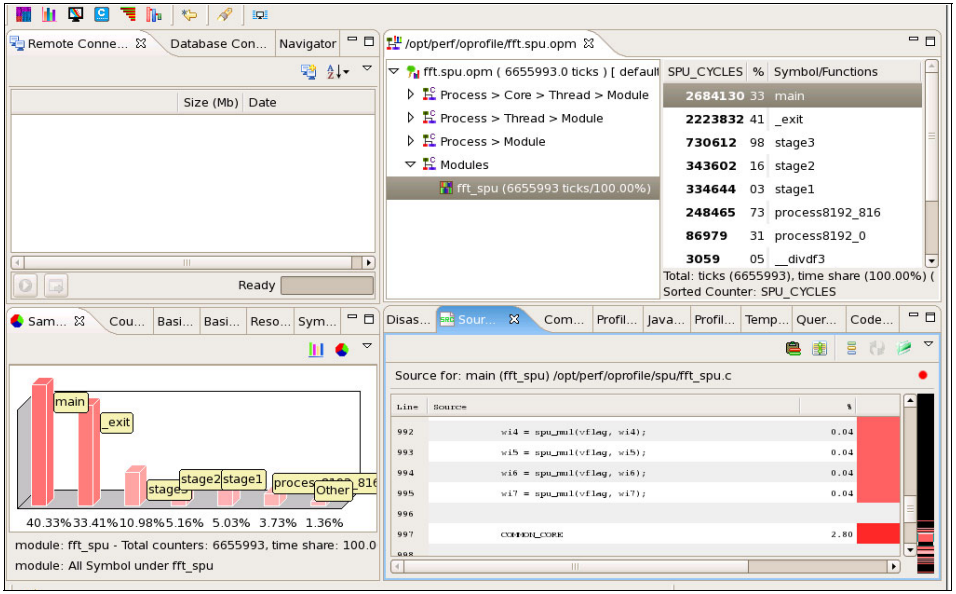


Figure 6-4 Source view for fft_spu.opm

If desired, you may repeat the exact same procedure for analyzing fft_ppu.opm profile results.

Step 4: Gathering profile information with FDPR-Pro

The FDPR-Pro tool, in addition to its principal optimization function, allows us to investigate the application performance, while mapping back to the source code, when used in combination with the Code Analyzer.

Initially, we need to proceed with the FDPR-Pro instrumentation for collecting the profiling data:

1. Clean-up old profile information and create a temporary working dir for FDPR-Pro:

   ```
   cd ~/FFT16M/ppu ; rm -f *.mprof *.nprof ; mkdir sputmp
   ```

2. Instrument the fft executable with the following command:

   ```
   fdprpro fft -cell  -spedir sputmp -a instr
   ```

Example 6-7 Sample output from FDPR-Pro instrumentation

```
FDPR-Pro Version 5.4.0.16 for Linux (CELL)
fdprpro ./fft -cell -spedir sputmp -a instr
```

```
> spe_extraction -> ./fft ...
...
> processing_spe_file -> sputmp/fft_spu ...
...
> reading_exe ...
> adjusting_exe ...
...
> analyzing ...
> building_program_infrastructure ...
@Warning: Relocations based on section .data -- section may not be
reordered
> building_profiling_cfg ...
> spe_encapsulation -> sputmp/out ...
>> processing_spe -> sputmp/out/fft_spu ...
> instrumentation ...
>> throw_&_catch_fixer ...
>> adding_universal_stubs ...
>> running_markers_and_instrumenters ...
>> linker_stub_fixer ...
>> dynamic_entries_table_sections_bus_fixer ...
>> writing_profile_template -> fft.nprof ...
> symbol_fixer ...
> updating_executable ...
> writing_executable -> fft.instr ...
bye.
```

3. Run the generated instrumented profile:

   ./fft.instr 20 1

4. There should be two relevant generated files:

   ~/FFT16M/ppu/fft.nprof # PPU profile information

   ~/FFT16M/ppu/fft_spu.mprof # SPU profile information

Step 5: Use profile data with Code Analyzer

The Code Analyzer tool imports information from the FDPR-Pro, and creates a visualization for it. In order to work with that, we need the following procedures:

1. With VPA open, select **Tools** → **Code Analyzer.**
2. Go to **Tools** → **Code Analyzer** → **Analyze Executable** and locate the original fft executable file. At this point, we should get two opened editor tabs at the center view of the screen: one for PPU and one for SPU.

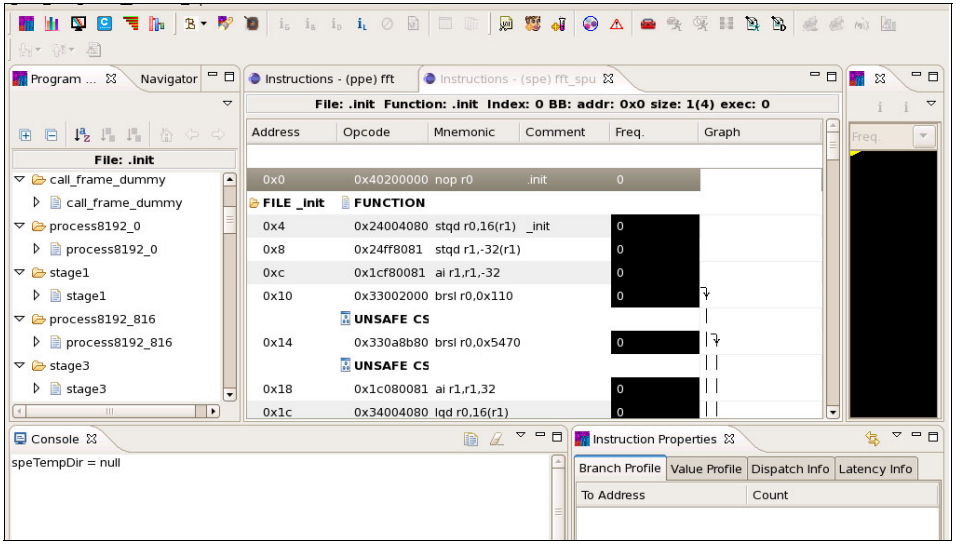


Figure 6-5 PPU and SPU editors in Code Analyzer

3. Associate the PPU profile information by selecting the PPU editor tab view and going to **File** → **Code Analyzer** → **Add Profile Info** and locating the fft.nprof file.

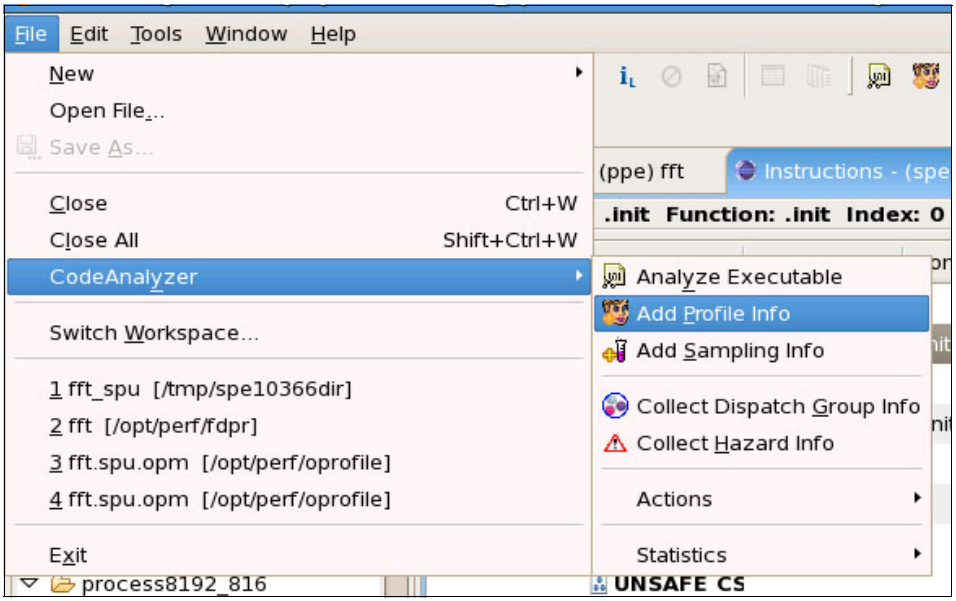


Figure 6-6 Adding profile info

4. Repeat the same procedure for the SPU part, by selecting the SPU editor tab, going to **File** → **Code Analyzer** → **Add Profile Info** and locating the fft_spu.mprof file.

The immediate effect, after loading the profile information, is the coloring of the instructions on both editor's tab, showing red for highly frequent executed instructions.

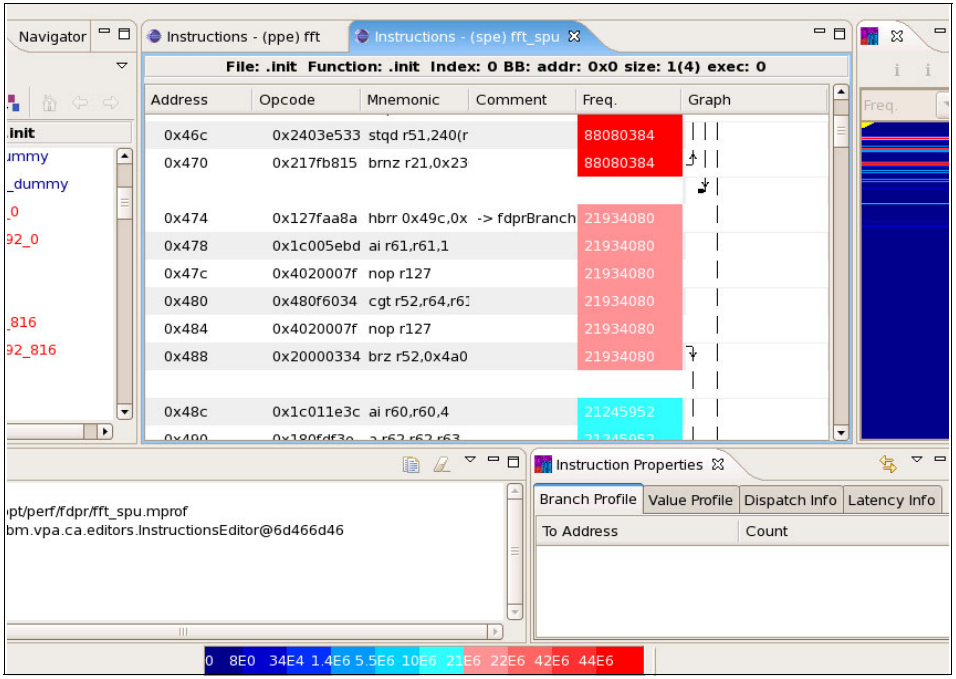


Figure 6-7 Code Analyzer showing rates of execution

Instead of only instructions, we can also associate the source code by selecting symbols in the Program Tree, right clicking, choosing “Open Source Code” and locating the proper source code. The result should be the addition of the “Source Code” tab at the center of the screen, where you will be able to see rates of execution per line of source code.

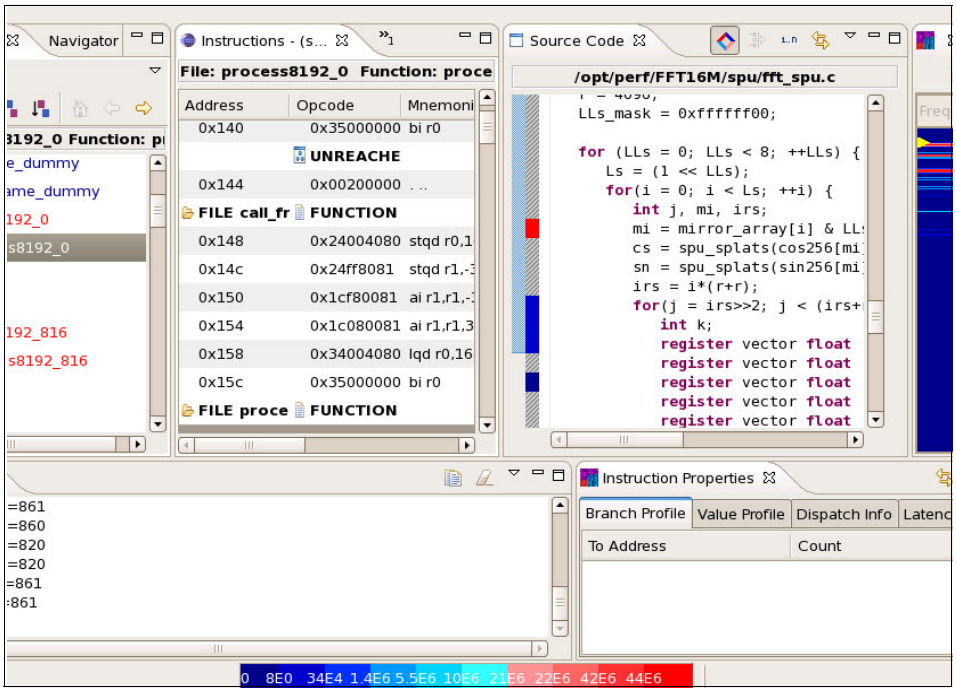


Figure 6-8 Code Analyzer source code tab

Calculate dispatch grouping boundaries for both fft PPE and fft SPU tabs by selecting each tab and pressing the "Collect display information about dispatch groups" button. You can also simultaneously select “Collect hazard info” button in order to collect comments about performance bottlenecks, right above source lines that apply.

Figure 6-9 “Collect ...” buttons in code analyzer

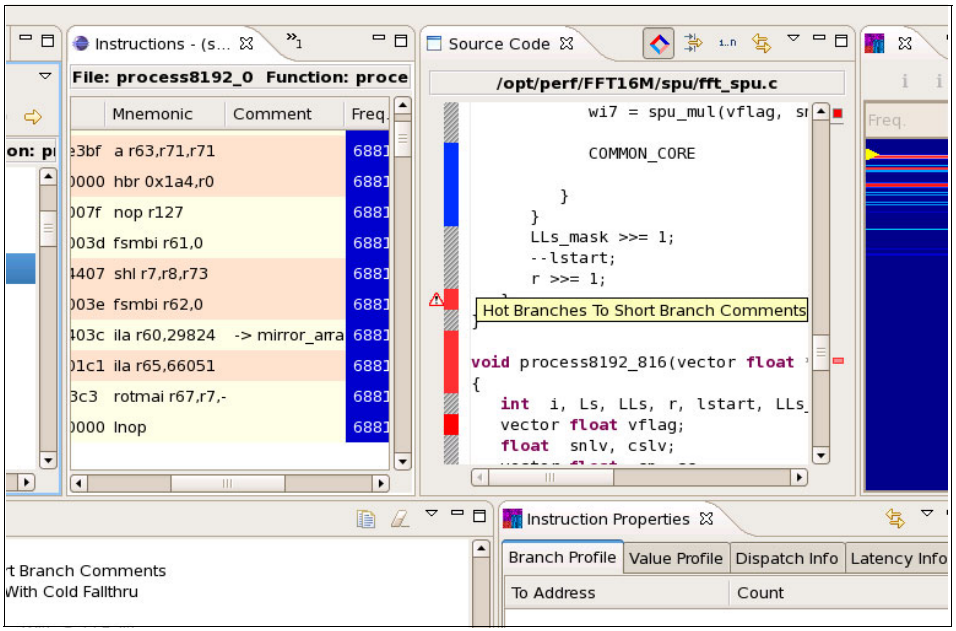


Figure 6-10 Hazards Info commented source code

Display pipeline population for each dispatch group, by choosing the "Dispatch Info" tab on right lower corner view (inside “Instruction Properties tab”) and pressing its "Link with table" button.

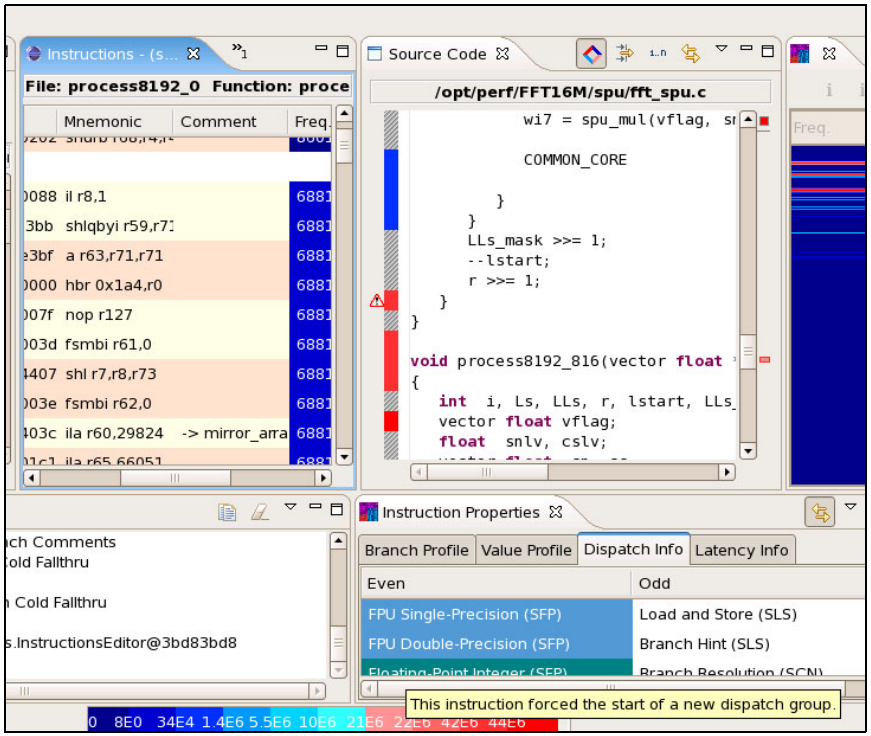


Figure 6-11 Dispatch Info tab with “Link with Table” option

The Latency Info tab, on right lower corner view, display latencies for each selected instruction

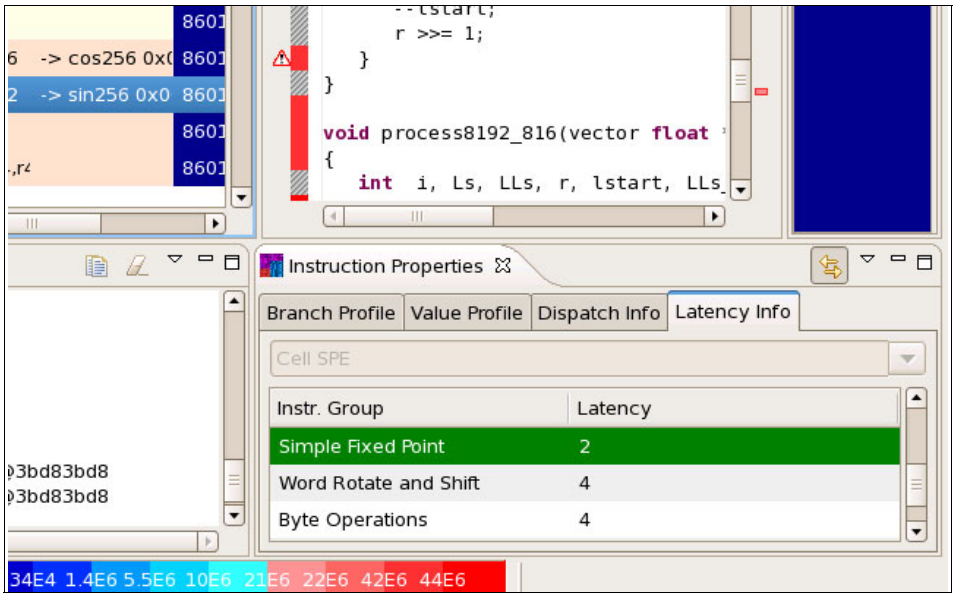


Figure 6-12 Latency Info view

The Code Analyzer also offers the possibility of inspecting SPU Timing information, at the pipeline level, with detailed stages of the Cell pipeline population. For that, select the fft SPU editor tab, locate the desired symbol at the Program Tree, right-click and choose “Show SPU-Timing”.

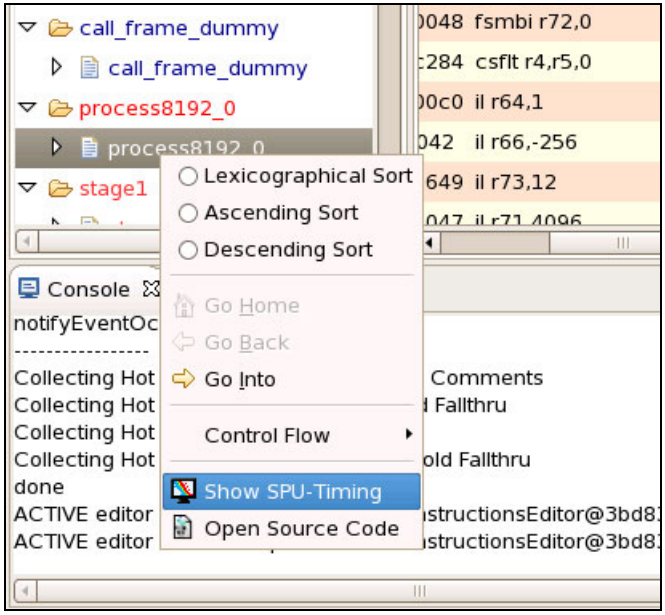


Figure 6-13 Selecting SPU-Timing information

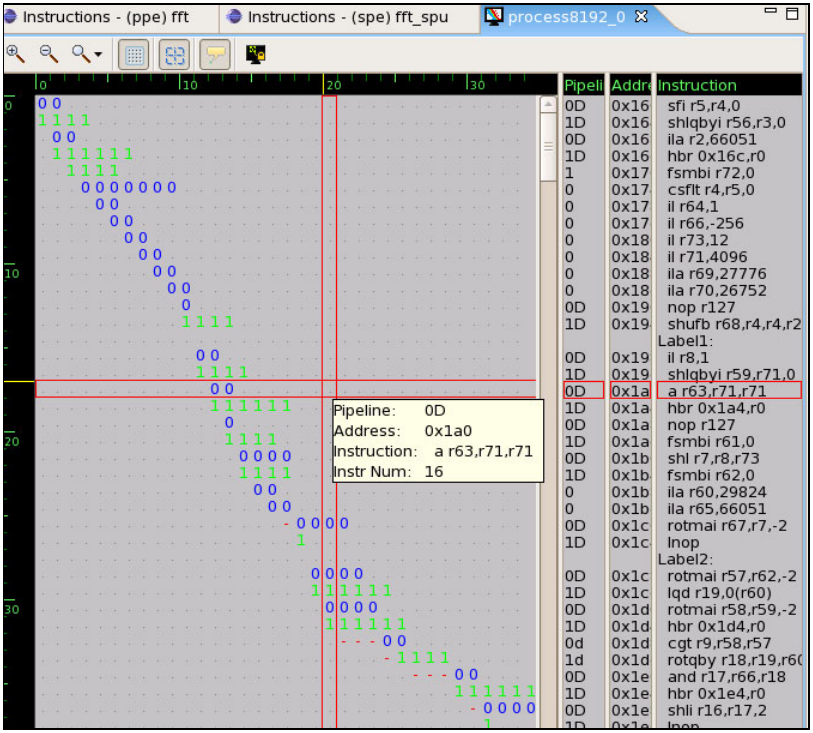


Figure 6-14 Cell Pipeline tab

6.1.4 Creating and working with trace data

The PDT tool produces tracing data, which can be viewed and analyzed in the Trace Analyzer tool. In order to properly collect trace data, we need to recompile the fft application according to the required PDT procedures:

Step 1: Collecting trace data with PDT

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

Example 6-8 Modifying ~/FFT16M/spu/Makefile for gcc compiler

```
##########################################################################
#
#        Target
##########################################################################
#

PROGRAMS_spu:= fft_spu
```

```
LIBRARY_embed:= fft_spu.a

##############################################################################
#
#        Local Defines
##############################################################################
#

CFLAGS_gcc:= -g --param max-unroll-times=1 -Wall -Dmain=_pdt_main
-Dexit=_pdt_exit -DMFCIO_TRACE -DLIBSYNC_TRACE
LDFLAGS_gcc    = -Wl,-q -g -L/usr/spu/lib/trace
INCLUDE          = -I/usr/spu/include/trace
IMPORTS          = -ltrace

##############################################################################
#
#                          buildutils/make.footer
##############################################################################
#

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

Example 6-9 Modifying ~/FFT16M/spu/Makefile for xlc compiler

```
##############################################################################
#
#        Target
##############################################################################
#
SPU_COMPILER = xlc
PROGRAMS_spu:= fft_spu
LIBRARY_embed:= fft_spu.a

##############################################################################
#
#        Local Defines
##############################################################################
#
```

CFLAGS_xlc:= -g -qnounroll -O5
CPP_FLAGS_xlc := -I/usr/spu/include/trace -Dmain=_pdt_main
-Dexit=_pdt_exit -DMFCIO_TRACE -DLIBSYNC_TRACE
LDFLAGS_xlc:= -O5 -qflag=e:e -Wl,-q -g -L/usr/spu/lib/trace -ltrace

```
########################################################################
#
#                        buildutils/make.footer
########################################################################
#

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

2. Re-build the fft application:

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

3. As it is strongly recommended in order to focus on stalls here, let's setup a configuration file with only the relevant stalls (mailboxes and read tag status for SPE):

   a. Copy the default xml to the place the FFT is ran, so we can modify it.

   cp /usr/share/pdt/config/pdt_cbe_configuration.xml ~/FFT16M

   b. Open the copied file for editing. At the first line, change the application name value to "fft".

   c. Next, search for <configuration name="SPE">, and below that line you will find the MFCIO group tag. Set it to active="false". And finally delete the SPE_MFC group. This should be sufficient to trace only the *stalls* in the SPE.

4. Prepare the environment by setting the following variables:

   export LD_LIBRARY_PATH=/usr/lib/trace

   export PDT_KERNEL_MODULE=/usr/lib/modules/pdt.ko

   export PDT_CONFIG_FILE=~/FFT16M/pdt_cbe_configuration.xml

5. Run the fft application at least three times, so we have better sampling:

   cd ~/FFT16M/ppu ; ./fft 1 1 4 1 0

6. You should have the three trace files - .pex, .map and .trace - available right after the execution.

Note 1: The default PDT_CONFIG_FILE for the SDK establishes the trace files prefix as “test”. If you haven’t modified the file, you should look for the trace files with “test” as the prefix.

Note 2: Remember to unset LD_LIBRARY_PATH environment variable, before running the original (non-PDT) binary later.

Step 7: Importing PDT data into Trace Analyzer

The Trace Analyzer allows the visualization of the application’s stages of execution. It works with data generated from the PDT tool, more specifically it reads information available in the generated .pex file. Execute the following procedure to visualize the data on the Trace Analyzer:

1. With VPA open, select **Tools** → **Trace Analyzer.**
2. Go to **File** → **Open File** and locate the .pex file, generated in the previous steps. The following screen should appear:

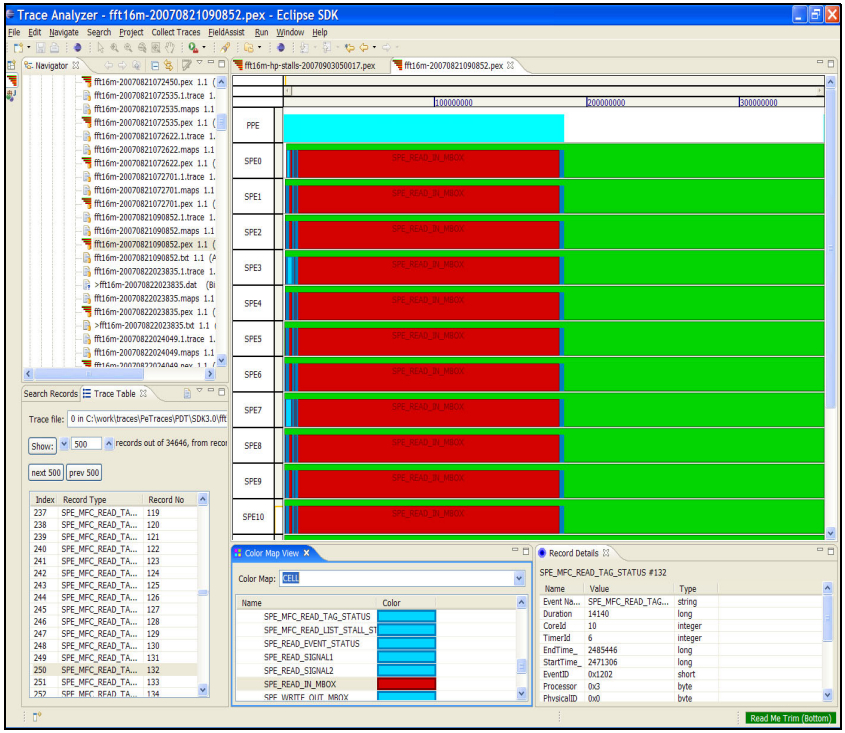


Figure 6-15 Trace Analyzer screen

The screen shown above corresponds to FFT16M application ran with 16 SPEs and no huge pages. As we can observe, a less intensive blue has been selected for the MFC_IO group, and we now see the difference between the borders and the internals of the interval. Additionally we've used the color map to change the color of read_in_mbox to be red rather than its group's default blue. You see a huge stall in the middle. This is where the benchmark driver verifies the result of the test run to make sure the benchmark computes correctly. The timed run is the thin blue strip after the stall. So next we zoom into this area, which is all that interests us in this benchmark.

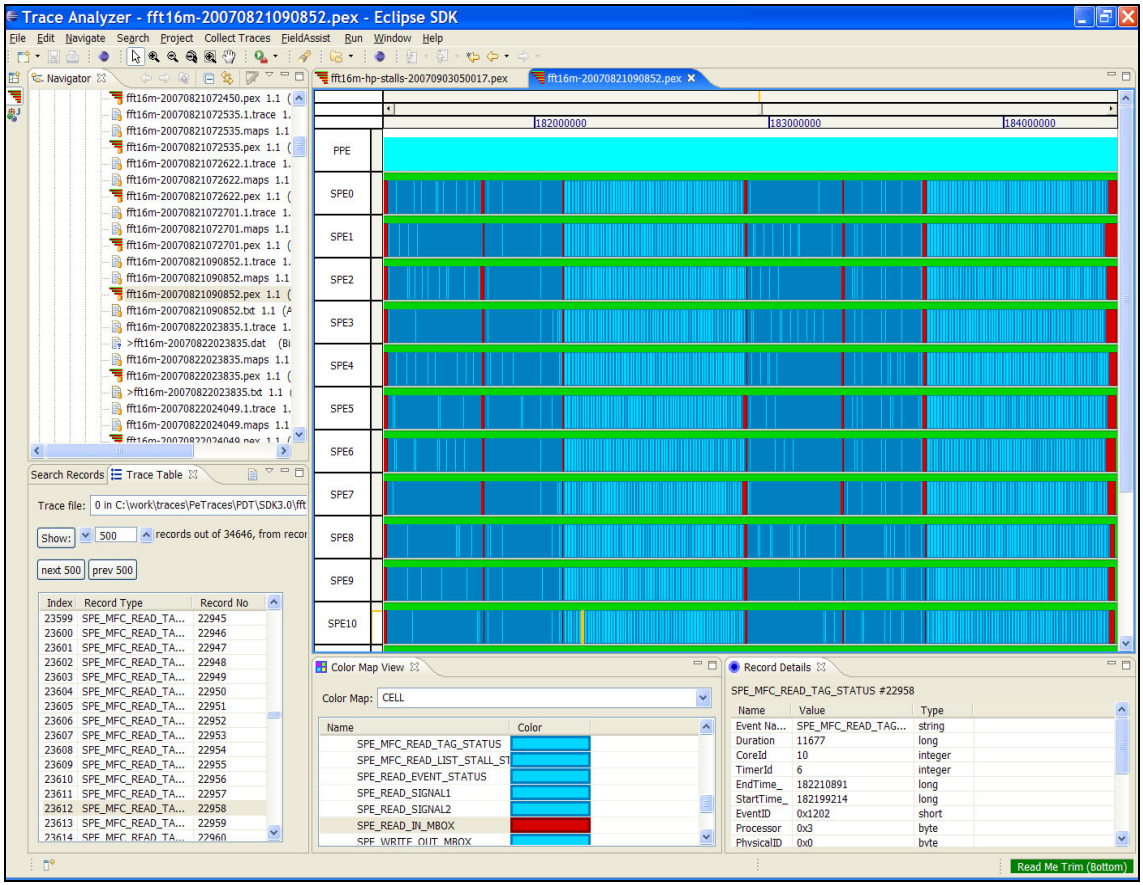


Figure 6-16 Zoomed trace view

As we can observe above, the mailboxes (red bars) break the execution into 6 stages. Different stages have different behavior, for example, the 3rd and 6th stages are much longer than the rest and have a lot of massive stalls. Trace Analyzer allows the selection of a stall, to obtain further details, by simply clicking on it (as shown in the picture by the yellow highlight). The selection marker rulers on the left and top show the location of the selected item (and can be used to get back to it if you scroll away). The data collected by the PDT for the selected stall is shown in the record details window. We see the stall is huge – almost 12K ticks. We can now check Cell BE performance tips for a possible cause of the stall, and arrive at TLB misses as a possible culprit as well as huge pages as a possible fix.

This is a sample on how trace visualization allows us to discover a significant amount of information regarding the potential application problems.

It's possible to observe how well balanced the application is, by looking at execution and start/stop time for each SPU. Since It breaks down, by kind, which are the causes of stalls in the code, we can identify synchronization problems.

7

Programming in distributed environments

The intent of this chapter is to provide an overview on a few of the distributed programming techniques available on Cell BE. We introduce the Hybrid Programming Model, as well as its libraries in SDK 3.0, and the Dynamic Application Virtualization (DAV) case.

The topics available here are:

7.1 Hybrid Programming Models in SDK 3.0

The Cell BE architecture is one of the answers for the problems being faced by the computer industry with regard to the performance degradation on traditional single threaded styled solutions. The power, frequency and memory wall problems lead us to the multi/many core solutions, and the exploitation of memory hierarchies enforcing the data locality.

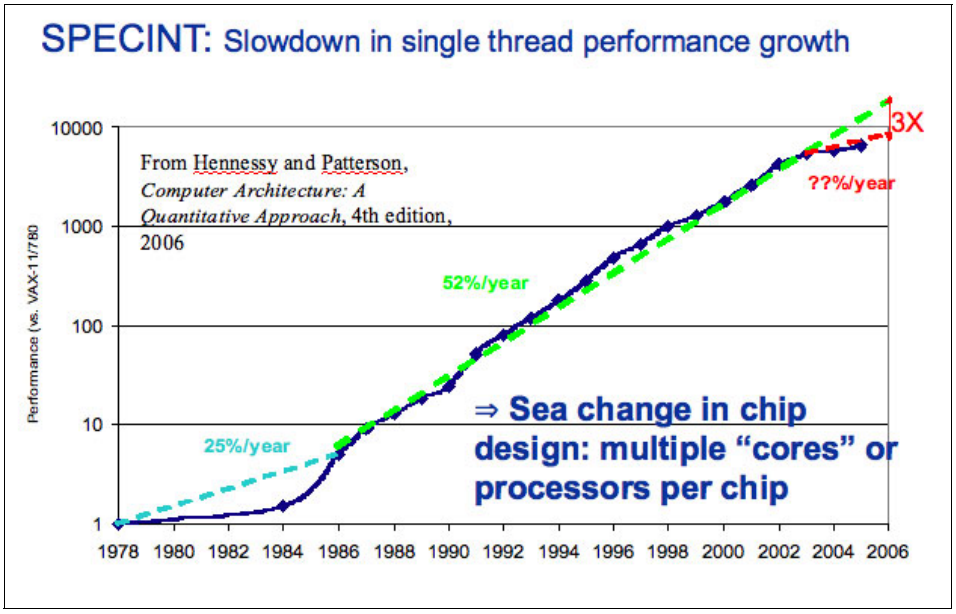


Figure 7-1 Single thread performance

Although Cell BE has been performing in an outstanding manner for certain types of applications, there's a need to consider other requirements into this picture, like power balance, legacy application integration and larger cluster scenarios, where network latency has a great influence on the performance.

The Hybrid Model Architecture proposal is a combination of characteristics from traditional superscalar multi-core solutions and Cell BE accelerated features.

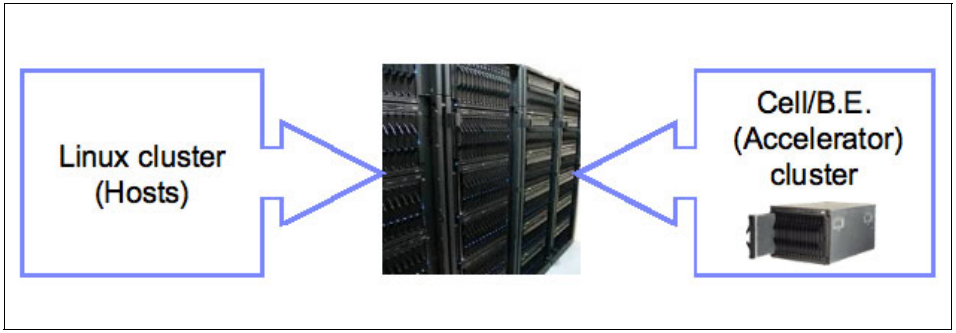


Figure 7-2 Hybrid Model System

The basic idea is to leverage traditional general purpose surperscalar cluster of machines as “processing masters” (Hosts), handling large data partitioning and I/O bound computations (e.g message passing), while off-loading well characterized computational intensive functions to computing kernels running on Cell BE accelerator nodes.

The solution enables a finer grain control over the applications’ parallelism and a more flexible offering, as it easily accommodates MPMD (Multiple Process Multiple Data) tasks with SPMD tasks (Single Process Multiple Data).

Motivations

There’s a well known balance between generality and performance. While Cell BE is definitely not an all general purpose solution, it still maintain a strong general purposes capabilities, specially with the presence of the PPE.

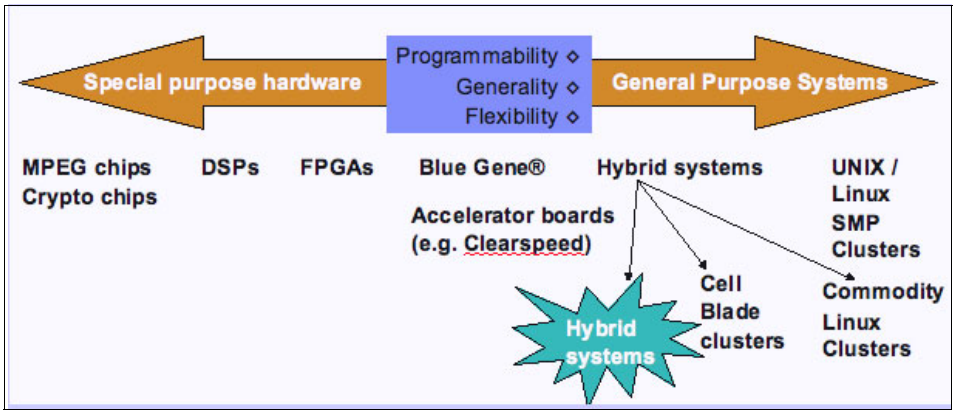


Figure 7-3 Balance between special and general purpose solutions

To understand the motivations behind the Hybrid architecture, we need to understand a few high performance scenario requirements:

- When moving from single-threaded to multi-core solutions, although there's a significant performance boost (throughput), there may be a power (energy) demand increase.
- There exists legacy solutions based on homogenous architectures.
- There are applications which need finer grain parallel computation control, since different components have different computational profiles.

The Hybrid Model system architecture address those requirements by:

- Increasing the throughput performance with the addition of the accelerators (Cell BE), in a more energy efficient way. Since Cell BE has outstanding performance for computation specific parallel tasks, given the same energy footprint, the whole system perform better as if simply kept as a homogeneous solution.

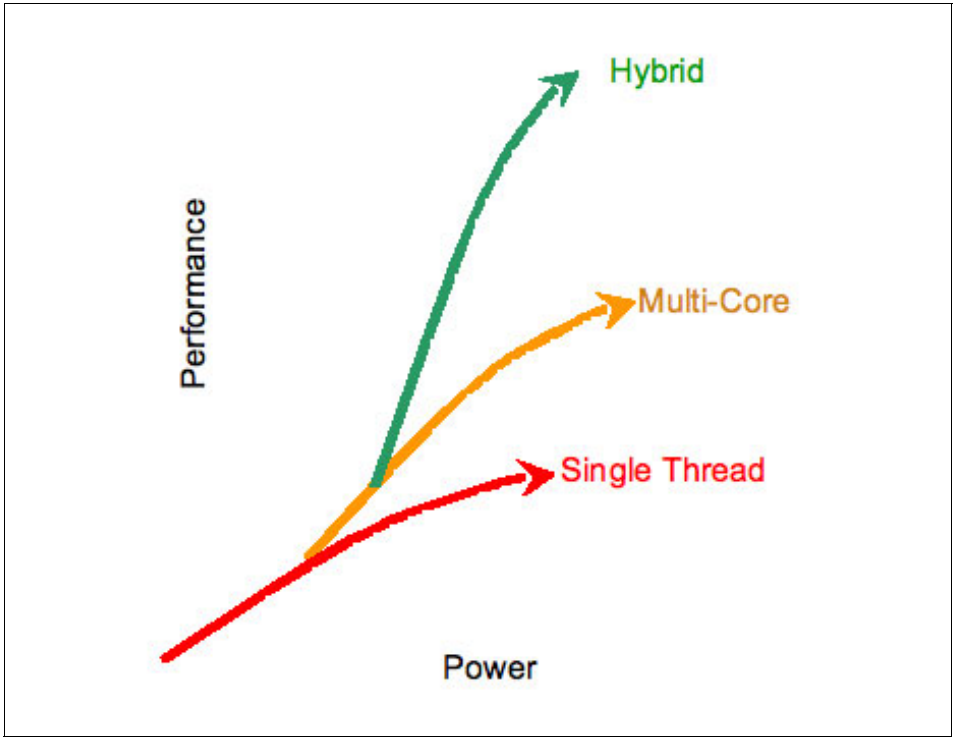


Figure 7-4 Performance curves: single thread, multi-core, hybrid

- Accommodating legacy solutions, since includes general purpose units.

- Achieving finer grain parallelism, since it is capable of mixing MPMD and SPMD at different levels (Host and Accelerator).

7.1.1 Hybrid DaCS

The Hybrid version of the Data Communication and Synchronization (DaCS) library provides a set of services which ease the development of applications and application frameworks in a heterogeneous multi-tiered system (for example a 64 bit x86 system (x86_64) and one or more Cell BE systems). The DaCS services are implemented as a set of APIs providing an architecturally neutral layer for application developers on a variety of multi-core systems. One of the key abstractions that further differentiates DaCS from other programming frameworks is a hierarchical topology of processing elements, each referred to as a DaCS Element (DE). Within the hierarchy each DE can serve one or both of the following roles:

A general purpose processing element, acting as a supervisor, control or master processor. This type of element usually runs a full operating system and manages jobs running on other DEs. This is referred to as a Host Element (HE).

A general or special purpose processing element running tasks assigned by an HE. This is referred to as an Accelerator Element (AE).

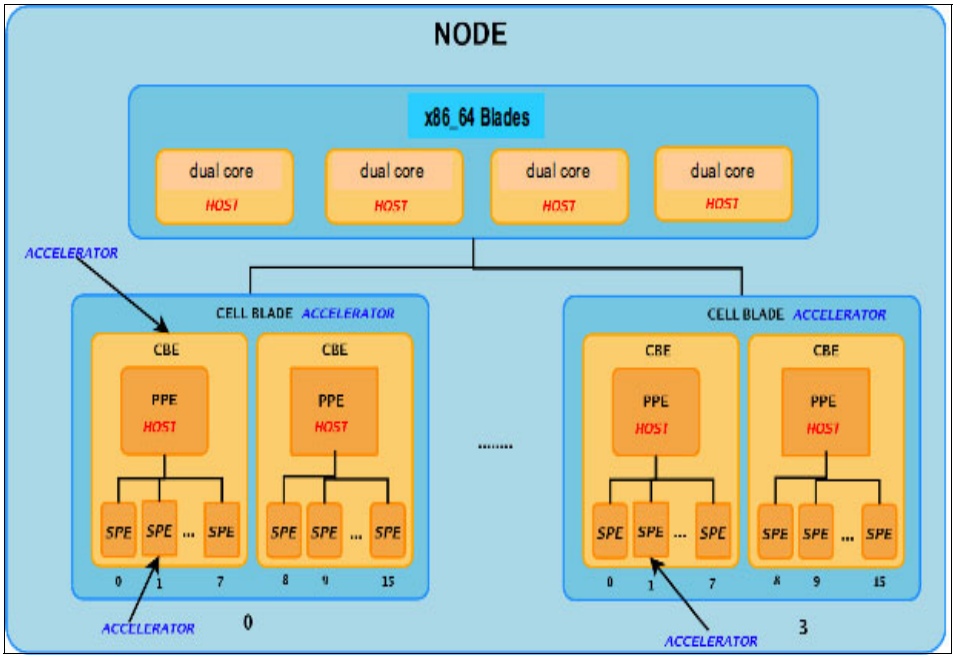


Figure 7-5 Hybrid DaCS System

DaCS for Hybrid Implementation

DaCS for Hybrid (DaCSH) is an implementation of the DaCS API specification which supports the connection of an HE on an x86_64 system to one or more AEs on Cell Broadband Engines (CBEs). In SDK 3.0, DaCSH only supports the use of sockets to connect the HE with the AEs. DaCSH provides access to the PowerPC Processor Element (PPE), allowing a PPE program to be started and stopped and allowing data transfer between the x86_64 system and the PPE. The SPEs can only be used by the program running on the PPE.

A PPE program that works with the SPEs can also be a DaCS program. In this case the program will use DaCS for Cell (DaCSC - see DaCS Programmers Guide and API Reference for Cell BE); the PPE will act as an AE for DaCSH (communicating with the x86_64 system) and as an HE for DaCSC (communicating with the SPEs). The DaCS API on the PPE is supported by a combined library which, when the PPE is being used with both DaCSH and DaCSC, will automatically use the parameters passed to the API to determine if the PPE is an AE talking to its HE (DaCSH) or an HE talking to its AEs (DaCSC).

In order to manage the interactions between the HE and the AEs DaCSH starts a service on each of them. On the host system the service is the Host DaCS daemon (hdacsd) and on the accelerator the service is the Accelerator DaCS daemon (adacsd). These services are shared between all DaCSH processes for an operating system image. For example, if the x86_64 system has multiple cores that each run a host application using DaCSH, only a single instance of the hdacsd service is needed to manage the interactions of each of the host applications with their AEs via DaCSH. Similarly, on the accelerator, if the CBE is on a Cell Blade (which has two CBEs), a single instance of the adacsd service is needed to managed both of the CBEs acting as AEs, even if they are used by different HEs.

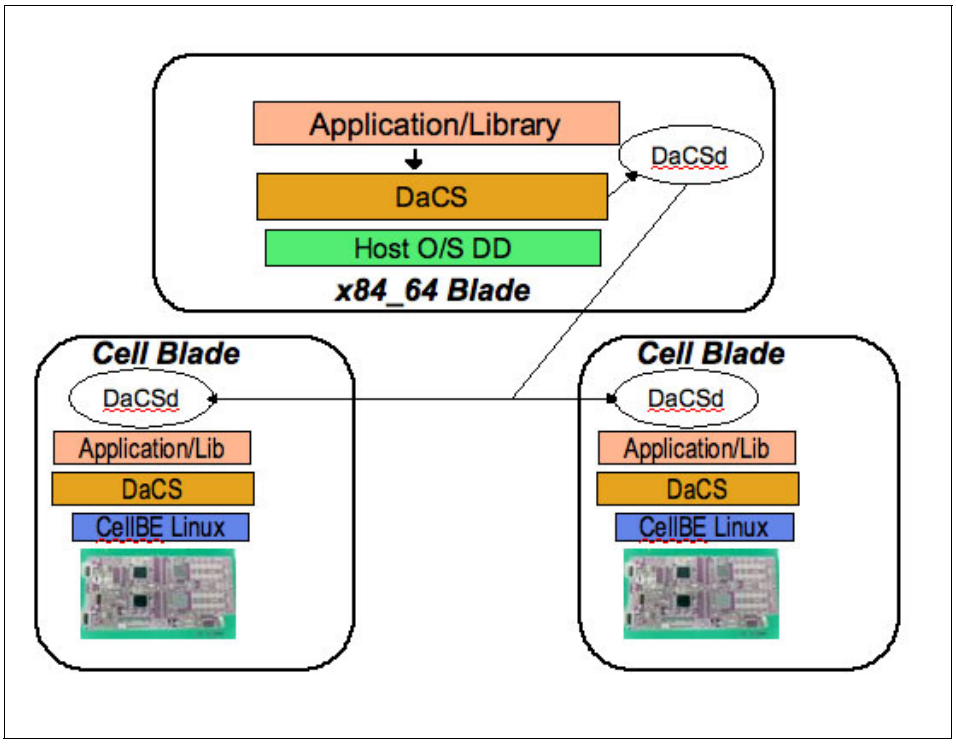


Figure 7-6 Hybrid DaCS architecture

When a host application starts using DaCSH this connects to the hdacsd service. This service manages the system topology from a DaCS perspective (managing reservations) and starts the accelerator application on the AE. Only process management requests will use the hdacsd and adacsd services. All other interactions between the host and accelerator application will flow via a direct socket connection.

Services

The DaCS services can be divided into the following categories:

Resource reservation The resource reservation services allow an HE to reserve AEs below itself in the hierarchy. The APIs abstract the specifics of the reservation system (O/S, middleware, etc.) to allocate resources for an HE. Once reserved, the AEs can be used by the HE to execute tasks for accelerated applications.

Process management The process management services provide the means for an HE to execute and manage accelerated applications on AEs, including, but not limited to, remote process

	launch and remote error notification. The host system DaCSd (hdacsd) provides services to the HE applications. The accelerator DaCSD (adacsd) provides services to the hdacsd and HE application, including the launching of the AE applications on the accelerator for the HE applications.
Group management	The group management services provide the means to designate dynamic groups of processes for participation in collective operations. In SDK 3.0 this is limited to process execution synchronization (barrier).
Remote memory	The remote memory services provide the means to create, share, transfer data to, and transfer data from a remote memory segment. The data transfers are performed using a one-sided put/get remote direct memory access (rDMA) model. These services also provide the ability to scatter/gather lists of data, and provide optional enforcement of ordering for the data transfers.
Message passing	The message passing services provide the means for passing messages asynchronously, using a two-sided send/receive model. Messages are passed point-to-point from one process to another.
Mailboxes	The mailbox services provide a simple interface for synchronous transfer of small (32-bit) messages from one process to another.
Process Synchronization	
	The process synchronization services provide the means to coordinate or synchronize process execution. In SDK 3.0 this is limited to the barrier synchronization primitive.
Data Synchronization	The data synchronization services provide the means to synchronize and serialize data access. These include management of wait identifiers for synchronizing data transfers, as well as mutex primitives for data serialization.
Error Handling	The error handling services enable the user to register error handlers and gather error information.

Programming Considerations

The DaCS library API services are provided as functions in the C language. The protocols and constants required are made available to the compiler by including the DaCS header file dacs.h as:

```
#include <dacs.h>
```

In general the return value from these functions is an error code. Data is returned within parameters passed to the functions.

Process management model

When working with the host and accelerators there has to be a way to uniquely identify the participants that are communicating. From an architectural perspective, each accelerator could have multiple processes simultaneously running, so it is not enough simply to identify the accelerator. Instead the unit of execution on the accelerator (the DaCS Process) must be identified using its DaCS Element Id (DE id) and its Process Id (Pid). The DE Id is retrieved when the accelerator is reserved (using dacs_reserve_children()) and the Pid when the process is started (using dacs_de_start()). Since the parent is not reserved, and no process is started on it, two constants are provided to identify the parent: DACS_DE_PARENT and DACS_PID_PARENT. Similarly, to identify the calling process itself, the constants DACS_DE_SELF and DACS_PID_SELF are provided.

Resource sharing model

The APIs supporting the locking primitives, remote memory access and groups follow a consistent pattern of creation, sharing, usage and destruction:

- Creation: An object is created which will be shared with other DEs, for example with dacs_remote_mem_create().
- Sharing: The object created is then shared by linked share and accept calls. The creator shares the item (for instance with dacs_remote_mem_share()), and the DE it is shared with accepts it (in this example with dacs_remote_mem_accept()). These calls must be paired. When one is invoked it waits for the other to occur. This is done for each DE the share is action with.
- Usage: This may require closure (such as in the case of groups) or the object may immediately be available for use. For instance remote memory can immediately be used for put and get.
- Destruction: The DEs that have accepted an item can release the item when they are done with it (for example by calling dacs_remote_mem_release()). The release does not block, but notifies the creator that it is not longer being used and cleans up any local storage. The creator does a destroy (in this case dacs_remote_mem_destroy()) which will wait for all of the DEs it has shared with to release the item, and then destroy the shared item.

API environment

To make these services accessible to the runtime code each process must create a DaCS environment. This is done by calling the special initialization

service dacs_runtime_init(). When this service returns the environment is set up so that all other DaCS function calls can be invoked. When the DaCS environment is no longer required the process must call dacs_runtime_exit() to free all resources used by the environment.

Important: DaCS for Hybrid and DaCS for Cell BE share the same API set, although they are two different implementations. For more on DaCS API, please refer to 4.7.1, “DaCS - Data Communication and Synchronization” on page 284

Building and Running Hybrid DaCS application

Three versions of the DaCS libraries are provided with the DaCS packages: optimized, debug and traced. The optimized libraries have minimal error checking and are intended for production use. The debug libraries have much more error checking than the optimized libraries and are intended to be used during application development. The traced libraries are the optimized libraries with performance and debug trace hooks in them. These are intended to be used to debug functional and performance problems that might be encountered. The traced libraries use the interfaces provided by the Performance Debug Tool (PDT) and require that this tool be installed.

Both static and shared libraries are provided for the x86_64 and PPU. The desired library is selected by linking to the chosen library in the appropriate path. The static library is named libdacs.a, and the shared library is libdacs.so.

DaCS Configuration

The host daemon service is named hdacsd and the accelerator daemon service is named adacsd. Both daemons are configured using their respective /etc/dacsd.conf files.

Default versions of these files are automatically installed with each of the daemons. These default files contain comments on the parameters and values currently supported.

When one of these files is changed the changes will not take affect until the respective daemon is restarted as described above.

DaCS Topology

The topology configuration file /etc/dacs_topology.config is only used by the host daemon service. Back up this file before changing it. Changes will not take effect until the daemon is restarted.

The host DaCS daemon might stop if there is a configuration error in the dacs_topology.config file. Check the log file specified by the dacsd.conf file (default is /var/log/hdacsd.log) for configuration errors.

The topology configuration file identifies the hosts and accelerators and their relationship to one another. The host can contain more than one CPU core, for example a multi core x86_64 Blade. The host can be attached to one or more accelerators, for example Cell BE Blade. The topology configuration file allows you to specify a number of configurations for this hardware. For example, it can be configured such that each core is assigned one Cell Broadband Engine or it might be configured so that each core can reserve any (or all) of the Cell Broadband Engines.

The default topology configuration file is for a host that has four cores and is attached to a single Cell BE Blade:

Example 7-1 Default topology file

```
<DaCS_Topology
 version="1.0">
  <hardware>
    <de tag="OB1" type="DACS_DE_SYSTEMX" ip="192.168.1.100">
      <de tag="OC1" type="DACS_DE_SYSTEMX_CORE"></de>
      <de tag="OC2" type="DACS_DE_SYSTEMX_CORE"></de>
      <de tag="OC3" type="DACS_DE_SYSTEMX_CORE"></de>
      <de tag="OC4" type="DACS_DE_SYSTEMX_CORE"></de>
    </de>
    <de tag="CB1" type="DACS_DE_CELLBLADE" ip="192.168.1.101">
      <de tag="CBE11" type="DACS_DE_CBE"></de>
      <de tag="CBE12" type="DACS_DE_CBE"></de>
    </de>
  </hardware>
  <topology>
    <canreserve he="OC1" ae="CB1"/>
    <canreserve he="OC2" ae="CB1"/>
    <canreserve he="OC3" ae="CB1"/>
    <canreserve he="OC4" ae="CB1"/>
  </topology>
</DaCS_Topology>
```

The <hardware> section identifies the host system with its four cores (OC1-OC4) and the Cell BE BladeCenter (CB1) with its two Cell Broadband Engines (CBE11 and CBE12).

The <topology> section identifies what each core (host) can use as an accelerator. In this example, each core can reserve and use either the entire Cell

BE BladeCenter (CB1) or one or more of the Cell Broadband Engines on the BladeCenter. The ability to use the Cell BE is implicit in the <canreserve> element. This element has an attribute only which defaults to false. When it is set to true, only the Cell BE BladeCenter can be reserved. If the fourth <canreserve> element was changed to <canreserve he="OC4" ae="CB1" only="TRUE"></canreserve>, then OC4 can only reserve the Cell BE BladeCenter. The usage can be made more restrictive by being more specific in the <canreserve> element. If the fourth <canreserve> element is changed to <canreserve he="OC4" ae="CBE12"></canreserve>, then OC4 can only reserve CBE12 and can not reserve the Cell BE BladeCenter.

Modify the topology configuration file to match your hardware configuration. Make a copy of the configuration file before changing it. At a minimum, update the IP addresses of the ip attributes to match the interfaces between the host and the accelerator

DaCS Daemons

The daemons can be stopped and started using the shell service command in the sbin directory. For example, to stop the host daemon type:

```
/sbin/service hdacsd stop
```

and to restart the host daemon type:

```
/sbin/service hdacsd start
```

The accelerator daemon (adacsd) may be restarted in like manner. See the man page for service for more details on the service command.

Running an Application

A hybrid DaCS application on the host (x86_64) must have processor affinity to start, which can be done on the command line. Here is a command line example to set affinity of the shell to the first processor:

```
taskset -p 0x00000001  $$
```

The bit mask, starting with 0 from right to left, is an index to the processor affinity setting. Bit 0 is on or off for CPU 0, bit 1 for CPU 1, and bit number x is CPU number x. $$ means the current process gets the affinity setting.

```
taskset -p$$
```

will return the mask setting as an integer. Using the -c option makes the taskset more usable. For example,

```
taskset -pc 3 $$
```

will set the processor CPU affinity to CPU 3for the current process. The -pc parameter sets by process and CPU number.

```
taskset -pc  $$
```

will return the current CPU setting for affinity for the current process. According to the man page for taskset a user must have CAP_SYS_NICE permission to change CPU affinity. See the man page for taskset for more details.

To launch a DaCS application use a taskset call, for example:

```
taskset 0x00000001 MyDaCSApp arg1
```

where the application program is MyDaCSApp and is passed an argument of “arg1”.

Step-by-Step Example

Let’s create, next, a simple Hybrid DaCS Hello World application, from building to deploying.

Step 0: Verify the configuration

In order to properly build and run this example, we need to verify e few requirements on the configuration:

- You will need one x86_64 Blade server and one QS21 Cell Blade, both configured with the SDK 3.0
- Verify if you have a properly configured DaCS topology files in your x86_64 node (see above).
- Make sure hdacsd is started on the x86_64 and adacsd is started on the QS21 Blade.

Step 1: Create the build structure

Although the build process will occur on the host (x86_64) machine, create the following directory structure, under the root of your user home dir, on both host and accelerator machines:

Example 7-2 Directory structure

```
hdacshello
hdacshello/ppu
hdacshello/ppu/spu
```

We will also need a Makefile in each of the created folders (only on the host machine):

Example 7-3 hdacshello/Makefile

```
DIRS := ppu
INCLUDE := -I/opt/cell/sdk/prototype/usr/include
IMPORTS := /opt/cell/sdk/prototype/usr/lib64/libdacs_hybrid.so
CC_OPT_LEVEL := -g
PROGRAM := hdacshello
include $(CELL_TOP)/buildutils/make.footer
```

Example 7-4 hdacshello/ppu/Makefile

```
DIRS := spu
INCLUDE := -I/opt/cell/sdk/sysroot/usr/include
IMPORTS :=
/opt/cell/sysroot/opt/cell/sdk/prototype/usr/lib64/libdacs_hybrid.so
spu/hdacshello_spu
LDFLAGS += -lstdc++
CC_OPT_LEVEL = -g
PROGRAM_ppu64 := hdacshello_ppu
include $(CELL_TOP)/buildutils/make.footer
```

Example 7-5 hdacshello/ppu/spu/Makefile

```
INCLUDE := -I/opt/cell/sdk/sysroot/usr/include
IMPORTS := /opt/cell/sysroot/usr/spu/lib/libdacs.a
CC_OPT_LEVEL := -g
PROGRAM_spu := hdacshello_spu
include $(CELL_TOP)/buildutils/make.footer
```

Step 2: Create the source files

The following are the source files for each part of the application (only on the Host machine):

Example 7-6 hdacshello/hdacshello.c

```
#include <dacs.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

de_id_t cbe[2];
dacs_process_id_t pid;
```

```
int main(int argc __attribute__ ((unused)), char* argv[] __attribute__
((unused)))
{
    DACS_ERR_T dacs_rc;
    dacs_rc  = dacs_runtime_init(NULL,NULL);
    printf("HOST: %d : rc = %s\n",__LINE__,dacs_strerror(dacs_rc));
fflush(stdout);

    uint32_t num_cbe = 1;
    dacs_rc = dacs_reserve_children(DACS_DE_CBE,&num_cbe,cbe);
    printf("HOST: %d : rc = %s\n",__LINE__,dacs_strerror(dacs_rc));
fflush(stdout);
    printf("HOST: %d : num children = %d, cbe =
%08x\n",__LINE__,num_cbe,cbe[0]); fflush(stdout);

    char const * argp[] = {0};
    char const * envp[] = {0};
    char program[1024];
    getcwd(program,sizeof(program));
    strcat(program,"/ppu/hdacshello_ppu");

    dacs_rc =
dacs_de_start(cbe[0],program,argp,envp,DACS_PROC_REMOTE_FILE,&pid);
    printf("HOST: %d : rc = %s\n",__LINE__,dacs_strerror(dacs_rc));
fflush(stdout);

    int32_t status = 0;
    dacs_rc = dacs_de_wait(cbe[0],pid,&status);
    printf("HOST: %d : rc = %s\n",__LINE__,dacs_strerror(dacs_rc));
fflush(stdout);

    dacs_rc = dacs_release_de_list(num_cbe,cbe);
    printf("HOST: %d : rc = %s\n",__LINE__,dacs_strerror(dacs_rc));
fflush(stdout);

    dacs_rc = dacs_runtime_exit();
    printf("HOST: %d : rc = %s\n",__LINE__,dacs_strerror(dacs_rc));
fflush(stdout);

    return 0;
}
```

Example 7-7 hdacshello/ppu/hdacshello_ppu.c

```
#include <dacs.h>
```

```
#include <libspe2.h>
#include <malloc.h>
#include <inttypes.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

extern spe_program_handle_t hdacshello_spu;

de_id_t spe[2];
dacs_process_id_t pid;

int main(int argc __attribute__ ((unused)), char* argv[] __attribute__
((unused)))
{
    DACS_ERR_T dacs_rc;
    dacs_rc  = dacs_runtime_init(NULL,NULL);
    printf("PPU: %d : rc = %s\n",__LINE__,dacs_strerror(dacs_rc));
fflush(stdout);

    uint32_t num_spe = 1;
    dacs_rc = dacs_reserve_children(DACS_DE_SPE,&num_spe,spe);
    printf("PPU: %d : rc = %s\n",__LINE__,dacs_strerror(dacs_rc));
fflush(stdout);
    printf("PPU: %d : num children = %d, spe =
%08x\n",__LINE__,num_spe,spe[0]); fflush(stdout);

    char const * argp[] = {0};
    char const * envp[] = {0};
    void * program;
    program = &hdacshello_spu;
    DACS_PROC_CREATION_FLAG_T flags = DACS_PROC_EMBEDDED;
    dacs_rc = dacs_de_start(spe[0],program,argp,envp,flags,&pid);
    printf("PPU: %d : rc = %s\n",__LINE__,dacs_strerror(dacs_rc));
fflush(stdout);

    int32_t status = 0;
    dacs_rc = dacs_de_wait(spe[0],pid,&status);
    printf("PPU: %d : rc = %s\n",__LINE__,dacs_strerror(dacs_rc));
fflush(stdout);

    dacs_rc = dacs_release_de_list(num_spe,spe);
    printf("PPU: %d : rc = %s\n",__LINE__,dacs_strerror(dacs_rc));
fflush(stdout);
```

```
    dacs_rc = dacs_runtime_exit();
    printf("PPU: %d : rc = %s\n",__LINE__,dacs_strerror(dacs_rc));
fflush(stdout);

    return 0;
}
```

Example 7-8 hdacshello/ppu/spu/hdacshello_spu.c

```
#include <dacs.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc __attribute__ ((unused)), char* argv[] __attribute__
((unused)))
{
    DACS_ERR_T dacs_rc;

    dacs_rc = dacs_runtime_init(NULL,NULL);
    printf("SPU: %d : rc = %s\n",__LINE__,dacs_strerror(dacs_rc));
fflush(stdout);

    printf("Hello !\n"); fflush(stdout);

    dacs_rc = dacs_runtime_exit();
    printf("SPU: %d : rc = %s\n",__LINE__,dacs_strerror(dacs_rc));
fflush(stdout);

    return 0;
}
```

Step 3: Build and deploy the files

Change to the topmost folder (hdacshello), and build the application:

Example 7-9 Build

```
cd ~/hdacshello
CELL_TOP=/opt/cell/sdk make
```

If everything proceeded as expected, you should have the following binaries available:

```
hdacshello/hdacshello

hdacshello/ppu/hdacshello_ppu
```

`hdacshello/ppu/spu/hdacshello_spu`

Assure that you have permission to execute each of then. The following command may be of help:

```
chmod a+x ~hdacshello/hdacshello # repeat on the other executables
```

Next, we need to deploy the CBE binary `hdacshello/ppu/hdacshello_ppu` to the matching location on the accelerator (QS21 Cell Blade) machine. You may use scp, for instance:

```
scp ~/hdacshello/ppu/hdacshello_ppu user@qs21:~/hdacshello/ppu
```

Step 4: Run

Since Hybrid DaCS requires a few variable and commands to be run, create a helper script like the following:

Example 7-10 hdacshello/run.sh

```
# Set the environment
export
LD_LIBRARY_PATH=/opt/cell/sdk/prototype/usr/lib64:$LD_LIBRARY_PATH
export
DACS_START_ENV_LIST="LD_LIBRARY_PATH=/opt/cell/sdk/prototype/usr/lib64:
$LD_LIBRARY_PATH"
# Set the shell's cpu affinity
taskset -pc 1 $$

# Launch the target program
~/hdacshello/hdacshello
```

Make sure all daemons are properly running on the Host and the Accelerator, and execute the helper script:

~/hdacshello/run.sh

7.1.2 Hybrid ALF

There are two implementations of ALF: the ALF Cell implementation and the ALF Hybrid implementation. The ALF Cell implementation executes on the Cell PPU host and schedules tasks and work blocks on Cell SPUs. The ALF Hybrid implementation executes on an x86_64 host and schedules tasks and work blocks on an associated set of Cell SPUs through a communications mechanism, which in this case uses the Data Communications and Synchronization (DaCS) library

ALF for Hybrid-x86 Implementation

ALF for Hybrid-x86 is an implementation of the ALF API specification in a system configuration with an Opteron x86_64 system connected to one or more Cell BE processors. In this implementation, the Opteron system serves as the host, the SPEs on the Cell BE BladeCenters act as accelerators, and the PPEs on the Cell BE processors act as facilitators only. From the ALF application programmer's perspective, the application interaction, as defined by the ALF API, is between the Hybrid-x86 host and the SPE accelerators.

This implementation of the ALF API uses the Data Communication and Synchronization (DaCS) library as the process management and data transport layer. Refer to the DaCS for Hybrid section above for more information about how to set up DaCS in this environment.

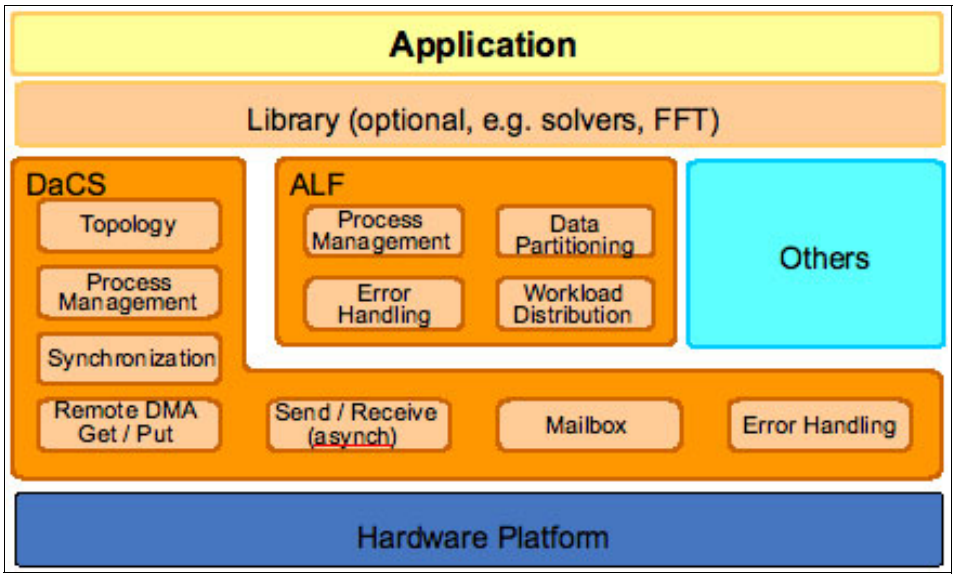


Figure 7-7 Hybrid ALF stack

To manage the interaction between the ALF host runtime on the Opteron system and the ALF accelerator runtime on the SPE, this implementation starts a PPE process (ALF PPE daemon) for each ALF runtime. The PPE program is provided as part of the standard ALF runtime library.

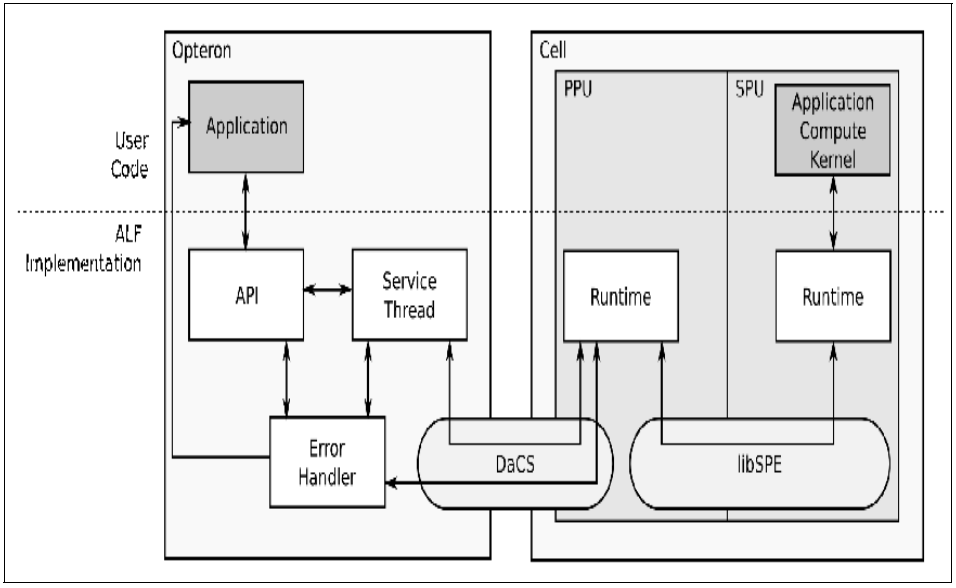


Figure 7-8 Hybrid ALF flow

Programming Considerations

As occurs with DaCS and Hybrid DaCS, ALF and Hybrid ALF are two different implementations of the same API set. For more on the ALF programming model, please refer to 4.7.2, “ALF - Accelerated Library Framework” on page 291.

Building and Running a Hybrid ALF application

Three versions of the ALF for Hybrid-x86 libraries are provided with the SDK:

- Optimized: This library has minimal error checking on the SPEs and is intended for production use.
- Error-check enabled: This version has a lot more error checking on the SPEs and intended to be used for application development.
- Traced: These are the optimized libraries with performance and debug trace hooks in them. These are intended for debugging functional and performance problems associated with ALF.

Additionally, both static and shared libraries are provided for the ALF host libraries. The ALF SPE runtime library is only provided as static libraries.

An ALF for Hybrid-x86 application must be built as two separate binaries as follows:

- The first binary is for the ALF host application, and you need to do the following:
  a. Compile the x86_64 host application with the -D_ALF_PLATFORM_HYBRID define variable, and specify the /opt/cell/sdk/prototype/usr/include include directory.
  b. Link the x86_64 host application with the ALF x86_64 host runtime library, alf_hybrid, found in the /opt/cell/sdk/prototype/usr/lib64 directory and the DaCS x86_64 host runtime library, dacs_hybrid, also found in the /opt/cell/sdk/prototype/usr/lib64 directory.
- The second binary is for the ALF SPE accelerator computational kernel, and you need to do the following:
  a. Compile the application's SPE code with the -D_ALF_PLATFORM_HYBRID define variable, and specify the /opt/cell/sysroot/usr/spu/include and the /opt/cell/sysroot/opt/cell/sdk/prototype/usr/spu/include include directories.
  b. Link the application's SPE code with the ALF SPE accelerator runtime library, alf_hybrid, found in the /opt/cell/sysroot/opt/cell/sdk/prototype/usr/spu/lib directory.
  c. Use the ppu-embedspu utility to embed the SPU binary into a PPE ELF image. The resulting PPE ELF object needs to be linked as a PPE shared library.

Running an Hybrid ALF application

The following steps describe how to run an ALF application.

Note: You need to ensure that the dynamic libraries libalf_hybrid and libdacs_hybrid are accessible. You can set this through LD_LIBRARY_PATH. For example:

export LD_LIBRARY_PATH=/opt/cell/sdk/prototype/usr/lib64

To run an application, do the following:

1. Build the ALF for Hybrid-x86 application, both the host application as an executable, my_appl, and the accelerator computational kernel as a PPE shared library, my_appl.so.
2. Copy the PPE shared library with the embedded SPE binaries from the host where it was built to a selected directory on the Cell BE where it is to be executed. For example:

   scp my_appl.so <CBE>:/tmp/my_directory

3. Set the environment variable ALF_LIBRARY_PATH to the above selected directory on the Cell BE. For example:

   export ALF_LIBRARY_PATH=/tmp/my_directory

4. Set the processor affinity on the Hybrid-x86 host. For example:

   taskset –p 0x00000001 $$

5. Run the x86_64 host application in the host environment. For example:

   ./my_appl

Step-by-Step Example

The following is a simple Hybrid ALF Hello World application, from building to deploying.

Step 1: Create the build structure

The build process will occur on the host (x86_64) machine. Start by creating the following directory structure, under the root of your user home dir for instance:

Example 7-11 Directory structure

```
halfhello
halfshello/host
halfhello/spu
```

We will also need a Makefile in each of the created folders:

Example 7-12 halfhello/Makefile

```
DIRS := spu host
include $(CELL_TOP)/buildutils/make.footer
```

Example 7-13 halfhello/spu/Makefile

```
INCLUDE := -I/opt/cell/sysroot/opt/cell/sdk/prototype/usr/spu/include
IMPORTS := -lalf_hybrid
LDFLAGS := -L/opt/cell/sysroot/opt/cell/sdk/prototype/usr/spu/lib
CPPFLAGS := -D_ALF_PLATFORM_HYBRID_
OBJS_alf_hello_world_spu := main_spu.o
PROGRAMS_spu := alf_hello_world_spu
SHARED_LIBRARY_embed64 := alf_hello_world_hybrid_spu64.so
include $(CELL_TOP)/buildutils/make.footer
```

Example 7-14 halfhello/host/Makefile

```
TARGET_PROCESSOR := host
```

```
INCLUDE := -I/opt/cell/sdk/prototype/usr/include
IMPORTS := -lalf_hybrid -lpthread -ldl -ldacs_hybrid -lnuma -lstdc++
-lrt
LDFLAGS := -L/opt/cell/sdk/prototype/usr/lib64
CPPFLAGS := -D_ALF_PLATFORM_HYBRID_
PROGRAM := alf_hello_world_hybrid_host64
include $(CELL_TOP)/buildutils/make.footer
```

Step 2: Create the source files

The following are the source files for each part of the application:

Example 7-15 Host source code (~/halfhello/host/main.c)

```
#include <stdio.h>
#include <alf.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#define DEBUG

#ifdef DEBUG
#define debug_print(fmt, arg...) printf(fmt,##arg)
#else
#define debug_print(fmt, arg...) { }
#endif

#define IMAGE_PATH_BUF_SIZE 1024    // Max length of path to PPU image
char spu_image_path[IMAGE_PATH_BUF_SIZE];   // Used to hold the
complete path to SPU image
char library_name[IMAGE_PATH_BUF_SIZE];     // Used to hold the name of
spu library
char spu_image_name[] = "alf_hello_world_spu";
char kernel_name[] = "comp_kernel";
char input_dtl_name[] = "input_prep";
char output_dtl_name[] = "output_prep";

int main()
{
  int ret;
  alf_handle_t handle;
  alf_task_desc_handle_t task_desc_handle;
  alf_task_handle_t task_handle;
```

```
  alf_wb_handle_t wb_handle;
  void *config_parms = NULL;

  sprintf(library_name, "alf_hello_world_hybrid_spu64.so");

  debug_print("Before alf_init\n");

  if ((ret = alf_init(config_parms, &handle)) < 0) {
    fprintf(stderr, "Error: alf_init failed, ret=%d\n", ret);
    return 1;
  }

  debug_print("Before alf_num_instances_set\n");
  if ((ret = alf_num_instances_set(handle, 1)) < 0) {
    fprintf(stderr, "Error: alf_num_instances_set failed, ret=%d\n",
ret);
    return 1;
  } else if (ret > 0) {
    debug_print("alf_num_instances_set returned number of SPUs=%d\n",
ret);
  }

  debug_print("Before alf_task_desc_create\n");
  if ((ret = alf_task_desc_create(handle, 0, &task_desc_handle)) < 0) {
    fprintf(stderr, "Error: alf_task_desc_create failed, ret=%d\n",
ret);
    return 1;
  } else if (ret > 0) {
    debug_print("alf_task_desc_create returned number of SPUs=%d\n",
ret);
  }

  debug_print("Before alf_task_desc_set_int32\n");
  if ((ret = alf_task_desc_set_int32(task_desc_handle,
ALF_TASK_DESC_MAX_STACK_SIZE, 4096)) < 0) {
    fprintf(stderr, "Error: alf_task_desc_set_int32 failed, ret=%d\n",
ret);
    return 1;
  }

  debug_print("Before alf_task_desc_set_int32\n");
  if ((ret = alf_task_desc_set_int32(task_desc_handle,
ALF_TASK_DESC_WB_PARM_CTX_BUF_SIZE, 0)) < 0) {
    fprintf(stderr, "Error: alf_task_desc_set_int32 failed, ret=%d\n",
ret);
```

```
    return 1;
  }

  debug_print("Before alf_task_desc_set_int32\n");
  if ((ret = alf_task_desc_set_int32(task_desc_handle,
ALF_TASK_DESC_WB_IN_BUF_SIZE, 0)) < 0) {
    fprintf(stderr, "Error: alf_task_desc_set_int32 failed, ret=%d\n",
ret);
    return 1;
  }

  debug_print("Before alf_task_desc_set_int32\n");
  if ((ret = alf_task_desc_set_int32(task_desc_handle,
ALF_TASK_DESC_WB_OUT_BUF_SIZE, 0)) < 0) {
    fprintf(stderr, "Error: alf_task_desc_set_int32 failed, ret=%d\n",
ret);
    return 1;
  }

  debug_print("Before alf_task_desc_set_int32\n");
  if ((ret = alf_task_desc_set_int32(task_desc_handle,
ALF_TASK_DESC_WB_INOUT_BUF_SIZE, 0)) < 0) {
    fprintf(stderr, "Error: alf_task_desc_set_int32 failed, ret=%d\n",
ret);
    return 1;
  }

  debug_print("Before alf_task_desc_set_int32\n");
  if ((ret = alf_task_desc_set_int32(task_desc_handle,
ALF_TASK_DESC_TSK_CTX_SIZE, 0)) < 0) {
    fprintf(stderr, "Error: alf_task_desc_set_int32 failed, ret=%d\n",
ret);
    return 1;
  }

  debug_print("Before alf_task_desc_set_int64\n");
  if ((ret = alf_task_desc_set_int64(task_desc_handle,
ALF_TASK_DESC_ACCEL_IMAGE_REF_L, (unsigned long long)spu_image_name)) <
0) {
    fprintf(stderr, "Error: alf_task_desc_set_int64 failed, ret=%d\n",
ret);
    return 1;
  }

  debug_print("Before alf_task_desc_set_int64\n");
```

```
  if ((ret = alf_task_desc_set_int64(task_desc_handle,
ALF_TASK_DESC_ACCEL_LIBRARY_REF_L, (unsigned long long)library_name)) <
0) {
    fprintf(stderr, "Error: alf_task_desc_set_int64 failed, ret=%d\n",
ret);
    return 1;
  }

  debug_print("Before alf_task_desc_set_int64\n");
  if ((ret = alf_task_desc_set_int64(task_desc_handle,
ALF_TASK_DESC_ACCEL_KERNEL_REF_L, (unsigned long long)kernel_name)) <
0) {
    fprintf(stderr, "Error: alf_task_desc_set_int64 failed, ret=%d\n",
ret);
    return 1;
  }

  debug_print("Before alf_task_desc_set_int64\n");
  if ((ret = alf_task_desc_set_int64(task_desc_handle,
ALF_TASK_DESC_ACCEL_INPUT_DTL_REF_L, (unsigned long
long)input_dtl_name)) < 0) {
    fprintf(stderr, "Error: alf_task_desc_set_int64 failed, ret=%d\n",
ret);
    return 1;
  }

  debug_print("Before alf_task_desc_set_int64\n");
  if ((ret = alf_task_desc_set_int64(task_desc_handle,
ALF_TASK_DESC_ACCEL_OUTPUT_DTL_REF_L, (unsigned long
long)output_dtl_name)) < 0) {
    fprintf(stderr, "Error: alf_task_desc_set_int64 failed, ret=%d\n",
ret);
    return 1;
  }

  debug_print("Before alf_task_create\n");
  if ((ret = alf_task_create(task_desc_handle, NULL, 1, 0, 0,
&task_handle)) < 0) {
    fprintf(stderr, "Error: alf_task_create failed, ret=%d\n", ret);
    return 1;
  }

  debug_print("Before alf_task_desc_destroy\n");
  if ((ret = alf_task_desc_destroy(task_desc_handle)) < 0) {
```

```
    fprintf(stderr, "Error: alf_exit alf_task_desc_destroy, ret=%d\n",
ret);
    return 1;
  }

  debug_print("Before alf_wb_create\n");
  if ((ret = alf_wb_create(task_handle, ALF_WB_SINGLE, 1, &wb_handle))
< 0) {
    fprintf(stderr, "Error: alf_wb_create failed, ret=%d\n", ret);
    return 1;
  }

  debug_print("Before alf_wb_enqueue\n");
  if ((ret = alf_wb_enqueue(wb_handle)) < 0) {
    fprintf(stderr, "Error: alf_wb_enqueue failed, ret=%d\n", ret);
    return 1;
  }

  debug_print("Before alf_task_finalize\n");
  if ((ret = alf_task_finalize(task_handle)) < 0) {
    fprintf(stderr, "Error: alf_task_finalize failed, ret=%d\n", ret);
    return 1;
  }

  debug_print("Before alf_task_wait\n");
  if ((ret = alf_task_wait(task_handle, -1)) < 0) {
    fprintf(stderr, "Error: alf_task_wait failed, ret=%d\n", ret);
    return 1;
  } else if (ret > 0) {
    debug_print("alf_task_wait returned number of work blocks=%d\n",
ret);
  }

  debug_print("In main: alf_task_wait done.\n");
  debug_print("Before alf_exit\n");
  if ((ret = alf_exit(handle, ALF_EXIT_POLICY_FORCE, 0)) < 0) {
    fprintf(stderr, "Error: alf_exit failed, ret=%d\n", ret);
    return 1;
  }

  debug_print("Execution completed successfully, exiting.\n");

  return 0;
}
```

Example 7-16 SPU source code (~/halfhello/host/main_spu.c)

```
#include <stdio.h>
#include <alf_accel.h>
int debug = 1;                  // set to 0 to turn-off debug

int comp_kernel(void *p_task_context, void *p_parm_context,
                          void *p_input_buffer, void *p_output_buffer,
void *p_inout_buffer,
                          unsigned int current_count, unsigned int
total_count)
{
  if (debug)
    printf
        ("Entering alf_accel_comp_kernel, p_task_context=%p,
p_parm_context=%p, p_input_buffer=%p, p_output_buffer=%p,
p_inout_buffer=%p, current_count=%d, total_count=%d\n",
         p_task_context, p_parm_context, p_input_buffer,
p_output_buffer, p_inout_buffer, current_count, total_count);
  printf("Hello World!\n");
  if (debug)
    printf("Exiting alf_accel_comp_kernel\n");
  return 0;
}

int input_prep(void *p_task_context, void *p_parm_context, void *p_dtl,
                                  unsigned int current_count, unsigned
int total_count)
{
  if (debug)
    printf
        ("Entering alf_accel_input_list_prepare, p_task_context=%p,
p_parm_context=%p, p_dtl=%p, current_count=%d, total_count=%d\n",
         p_task_context, p_parm_context, p_dtl, current_count,
total_count);
  if (debug)
    printf("Exiting alf_accel_input_list_prepare\n");
  return 0;
}

int output_prep(void *p_task_context, void *p_parm_context, void
*p_dtl, unsigned int current_count,
                                   unsigned int total_count)
{
  if (debug)
```

```
    printf
        ("Entering alf_accel_output_list_prepare, p_task_context=%p,
p_parm_context=%p, p_dtl=%p, current_count=%d, total_count=%d\n",
         p_task_context, p_parm_context, p_dtl, current_count,
total_count);
  if (debug)
    printf("Exiting alf_accel_output_list_prepare\n");
  return 0;
}

ALF_ACCEL_EXPORT_API_LIST_BEGIN
   ALF_ACCEL_EXPORT_API("", comp_kernel);
   ALF_ACCEL_EXPORT_API("", input_prep);
   ALF_ACCEL_EXPORT_API("", output_prep);
ALF_ACCEL_EXPORT_API_LIST_END
```

Step 3: Build and deploy the files

Change to the topmost folder and build the application:

Example 7-17 Build and deploy the files

```
cd ~/halfhello
CELL_TOP=/opt/cell/sdk make
```

If everything proceeded as expected, you should have the following binaries available:

```
halfhello/alf_hello_world_hybrid_host64

halfhello/spu/alf_hello_world_hybrid_spu64.so
```

Next, we need to deploy the ALF spu shared library `halfhello/spu/alf_hello_world_hybrid_spu64.so` to the matching location on the accelerator (QS21 Cell Blade) machine. You may use scp, for instance:

```
scp ~/halfhello/spu/alf_hello_world_hybrid_spu64.so user@qs21:/tmp
```

Step 4: Run

In order to properly run the application, execute the following sequence:

Example 7-18 Running the application

```
# Set the environment
export LD_LIBRARY_PATH=/opt/cell/sdk/prototype/usr/lib64
export ALF_LIBRARY_PATH=/tmp
```

```
taskset -pc 1 $$
~/halfhello/alf_hello_world_hybrid_host64
```

Make sure all Hybrid DaCS daemons are properly running on the Host and the Accelerator, before running

7.1.3 DAV - Dynamic Application Virtualization

IBM DAV, Dynamic Application Virtualization, is a technology offering available from the IBM Alphaworks web site[1]. DAV implements the function offload programming model. Using DAV, applications running under Microsoft Windows can transparently tap on the compute power of the Cell BE. The originality lies in the fact that the application that we wish to accelerate does not require any source code changes. Of course, the offloaded functions will have to be written to exploit the accelerator, the Cell BE in this case, but the main application remains unaware. It will just benefit from an increased level of performance.

The technology was developped initially for the financial services sector but can be used in other areas. The ideas are applicable wherever an application exhibits computational kernels with a high computational intensity (ratio of computation over data transfers) and cannot be re-written using other Cell BE frameworks, possibly because we do not have its source code or because we do not wish to port the whole application to the Cell BE environment.

This opens up a whole new world of opportunities to exploit the Cell BE architecture for applications which did not initially target the Cell BE. It also offers an increased flexibility as an application may have its front-end run on a laptop or desktop with nice GUI bells and whistles and have the hardcore, number crunching part run on specialized hardware like the Cell BE based blade servers.

From the accelerator perspective, DAV is a way to gain grounds into more application areas without a need to port the necessary middleware to run the full application. This is true for every type of acceleration model. The accelerator platform, the Cell BE here, does not need to support many database clients, external filesystems, job schedulers or grid middleware. This is taken care of at the client level. This separation of work lets the accelerator have a very light operating system and middleware layer as it is only providing raw computing power.

DAV target applications

In its current implementation, DAV can be used to speed up Microsoft Windows applications written in Visual C/C++ or Visual Basic (Excel spreasheets). DAV

[1] http://www.alphaworks.ibm.com/tech/dav

supports accelerator platforms (the server) running Linux. They can be any x86 or ppc64 server, including Cell BE blade servers. DAV refers to the Windows part of the application as the client side and the accelerator part as the server side.

The best candidate functions for offloading to a Cell BE platform are the ones that show a high ratio of computation over communication. The increased levels of performance obtained by running on the Cell BE should not be offset by the communication overhead between the client and the server. The transport mechanism currently used by DAV is TCP/IP sockets. Other options may be explored in the future, to lower the latency and increase the network bandwidth. Clearly, any advance on this front will increase, for a given application, the number of functions that could be accelerated.

Workloads that have a strong affinity for parallelization are optimal. A good example is option pricing using Monte-Carlo methods, like we describe in Chapter 8, “Case study: Monte Carlo Simulation” on page 493.

DAV architecture

The DAV trick is to make a clever use of DLL or (Dynamically Loaded Libraries)[2]. In fact, only functions that reside in a DLL can be offloaded to the server. What DAV does is to fake the Microsoft Windows DLL that contains the client code with a new one which communicates with the server to implement the offloaded functions. Enabling an application with DAV means the following steps.

On the client side

First identify the functions to be offloaded : their C prototypes, what data they need on input and what they produce on output. These functions need to reside in a DLL.

On the server side

Then, write an implementation of the exact same functions, exploiting all the Cell BE strengths. The implementation can use any Cell BE programming techniques. The functions need to be made into 32bit DLLs (the equivalent of shared libraries in Linux.)

Back to the client side

We then need to fake the application with a DAV DLL that will replace the original DLL. It is called the stub library. This DLL will make the application happy but will in fact interface with the DAV infrastructure to ship data back and forth between the client and the server. We will show how to use the IBM DAV Tooling component to create the stub library from the C prototypes of the offloaded functions

[2] See http://en.wikipedia.org/wiki/Dynamic-link_library

The whole process is described in the (Figure 7-9 on page 470). It shows the unchanged application linked with the stub library on the client side (left) and the actual implementation of the accelerated library on the server side (right).

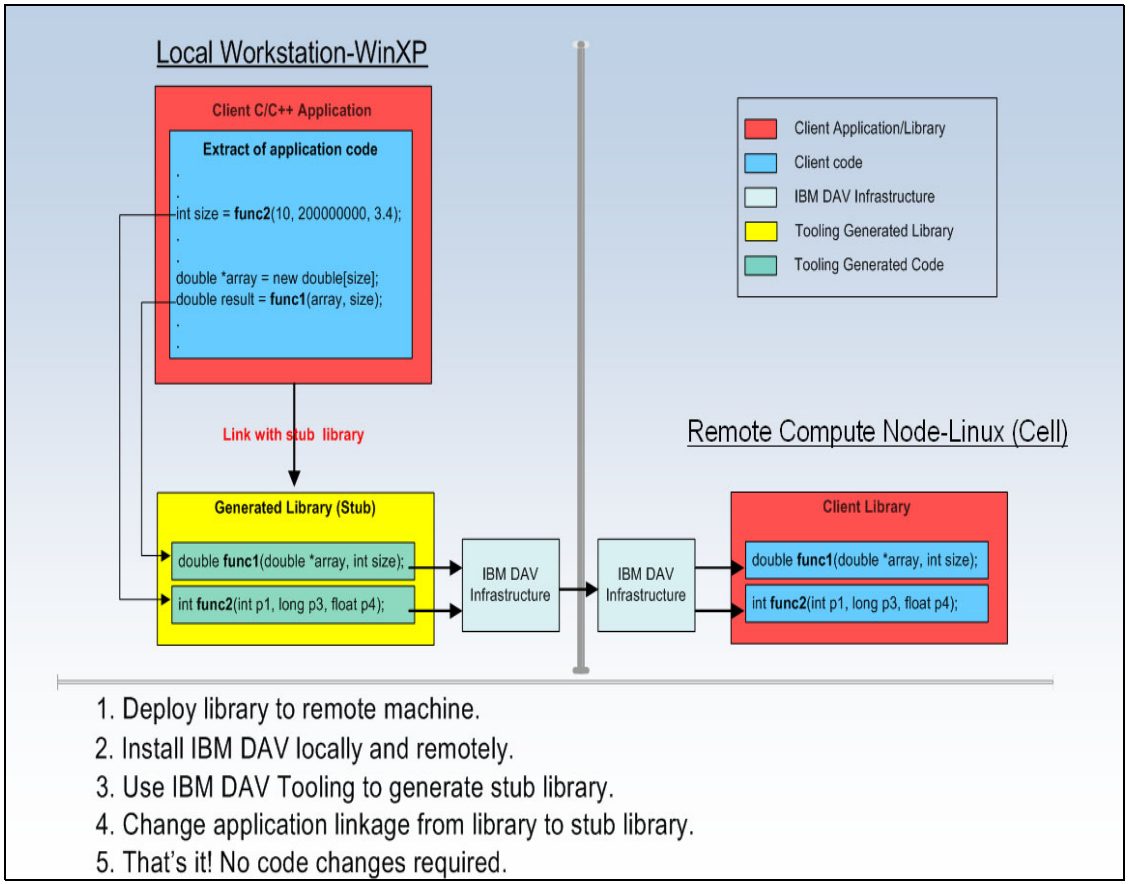


Figure 7-9 How DAV works

Running a DAV enabled application

To run the application, we need to make sure the DAV runtime on the client side knows how to contact the server and that on the server side that the DAV server is running, waiting for being called by the client DAV infrastructure. We will show how to start the DAV service on the DAV server and how to set the various DAV runtime configuration parameters.

The client application is then run as usual. Only will it be faster, thanks to the acceleration provided by the Cell BE.

System requirements

To install the DAV components, you need Microsoft Windows XP SP2 on the client side and RHEL5 or Fedora7 on the Cell BE side. After downloading the DAV software, you will get 3 files.

- DAVClientInstall.exe : run this executable to install the client part of the DAV infrastructure. This needs to be installed on any machine where a DAV application will run.
- dav-server.ppc64.rpm : install this RPM on every server that is to become a DAV accelerator node.
- DAVToolingInstall.exe : run this executable to install the DAV Tooling part : the one that creates the stub DLLs from the C prototypes of the functions to be offloaded. This needs to be installed only on the machine where the stub DLLs are to be created. The DAV Tooling requires the DAV client package to be installed too.

A C compiler is required to create the DAV stub library. Although a Cygwin environment with gcc would probably work, we have used Microsoft Visual C++® 2005 Express Edition as this is likely to be more common among Microsoft developers. This is a trial version of Microsoft Visual C++ 2005. The installation is described in the DAV user guide available at : http://dl.alphaworks.ibm.com/technologies/dav/IBM_DAV_User_Guide.pdf.

The DAV client package comes with a few samples located under the `C:\Program Files\IBM\DAV\sample` directory. We will describe the use of DAV following one of the examples.

A Visual C++ application example

We will show the steps needed to DAV enable a Visual C++ application. The application is very simple. It initializes two arrays and calls two functions to perform some computations on the arrays. Here is the C source code for the main program.

Example 7-19 The main() function

```
#include "Calculate.h"
int main(int argc, char * argv[])
{
   #define N 100
   int i;
   double in[N],out[N];

   for(i=0;i<N;i++)
      in[i]=1.*i;
```

```
   printf("calculate_Array sent %f\n",calculate_Array(in,out,N));
   printf("calculate_Array2 sent %f\n",calculate_Array2(in,out,N));


   return 0;
}
```

The two functions `calculate_Array` and `calculate_Array2` are listed here. They are the ones that we wish to offload to the accelerator. They take the `in` array as input and compute the `out` array and the `ret` result.

Example 7-20 The computational functions

```
#include “Calculate.h”
double calculate_Array(double *in,double *out,int size)
{
   int i;
   double ret=0.;
   for(i=0;i<size;i++) {
      out[i]=2.*in[i];
      ret+=in[i];
   }
   return ret;
}
double calculate_Array2(double in[],double out[],int size)
{
   int i;
   double ret=0.;
   for(i=0;i<size;i++) {
      out[i]=0.5*in[i];
      ret+=out[i];
   }
   return ret;
}
```

The function prototypes are defined in a header file. This file is very important as this is the input for the whole DAV process. In real situations, the C source code for the main program, the functions and the header may not be available. But you should find a way to create a prototype for each function that you wish to offload. This is the only source file that is absolutely reauired to get DAV to do what it needs to do. You may have to ask the original writer of the functions or do some reverse engineering to figure out the parameters that are passed to the function.

Here is our header file for this example.

Example 7-21 The Calculate.h header file.

```
// Calculate.h

#if defined(__cplusplus)
extern "C" {
#endif

double calculate_Array(double *in,double *out,int size);
double calculate_Array2(double in[],double out[],int size);

#if defined(__cplusplus)
}
#endif
```

We will now go through the steps required to enable DAV acceleration for this application. The steps are listed below :

- build the original application : this will create an exe file and a DLL file for the functions,
- do the DAV tooling using the header file for the functions. This will create the stub DLL that will replace the original DLL,
- instruct the original application to use the stub DLL,
- on the Cell BE, compile the functions and put them in a shared library
- start the DAV server on the Cell BE
- change the DAV parameters on the client machine so that it points to the right accelerator node
- run the application with DAV acceleration

Build the original application

Figure 7-10 shows how the original application is built. The .c and .h files are the compute functions source code.

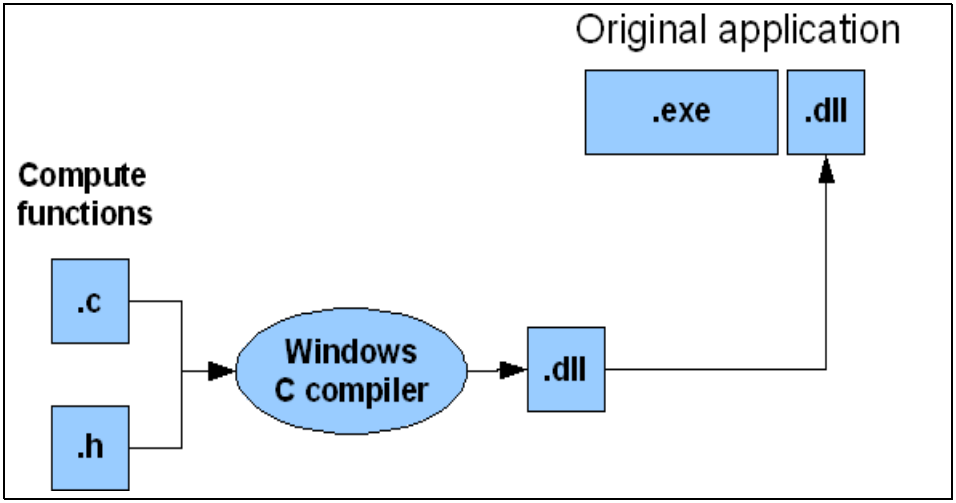


Figure 7-10 The original application

We have used the Visual C++ 2005 Express Edition for this. Here is a picture showing the two projects : CalculateApp, the main program and Calculate, the functions.

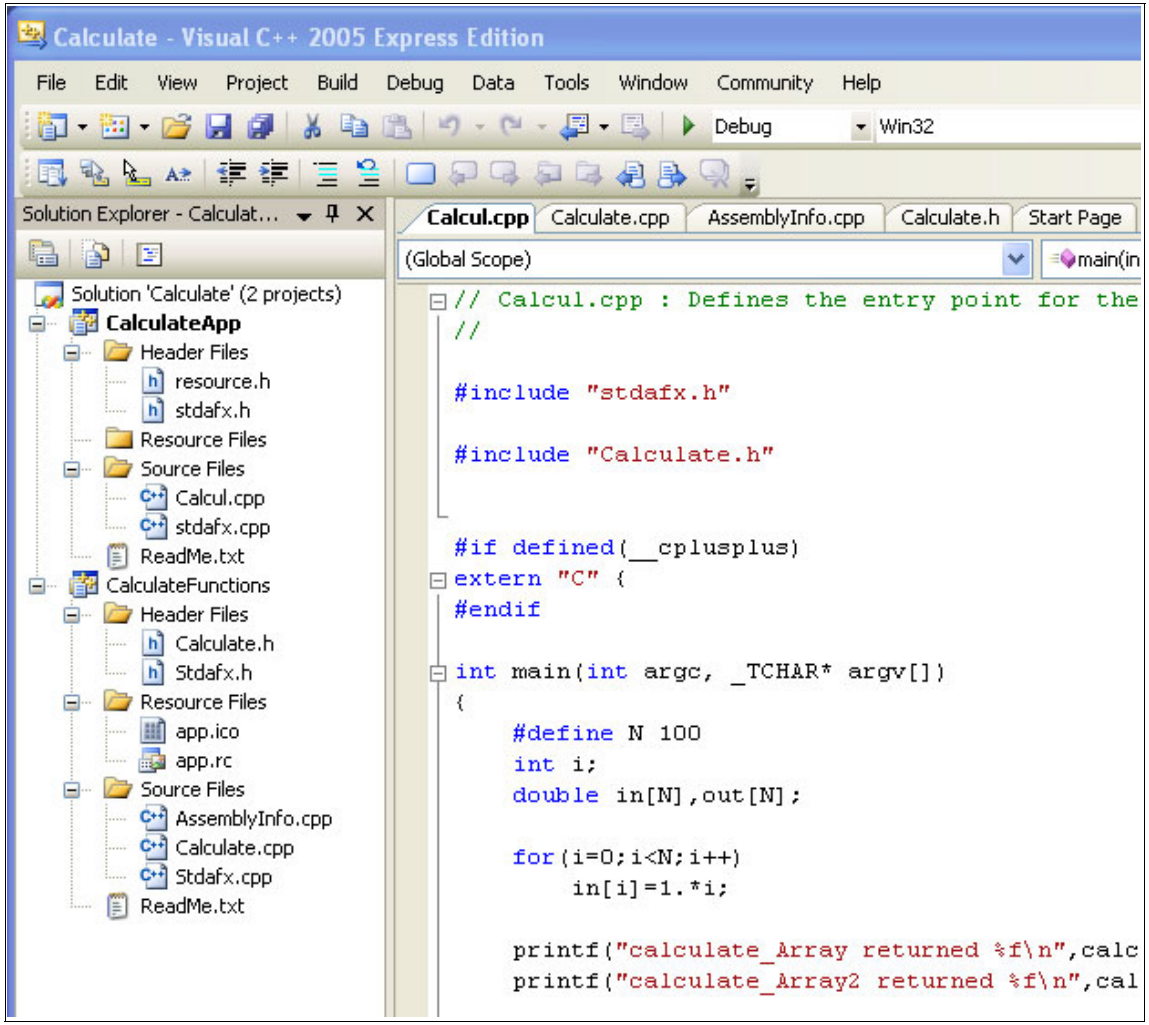


Figure 7-11 The CalculateApp and Calculate projects in Visual C++ 2005 Express Edition

DAV tooling

Next, we apply the DAV Tooling which produces the stub DLLs and the source code for the server side data marshalling. This is depicted in (Figure 7-12). This step only requires the .h file.

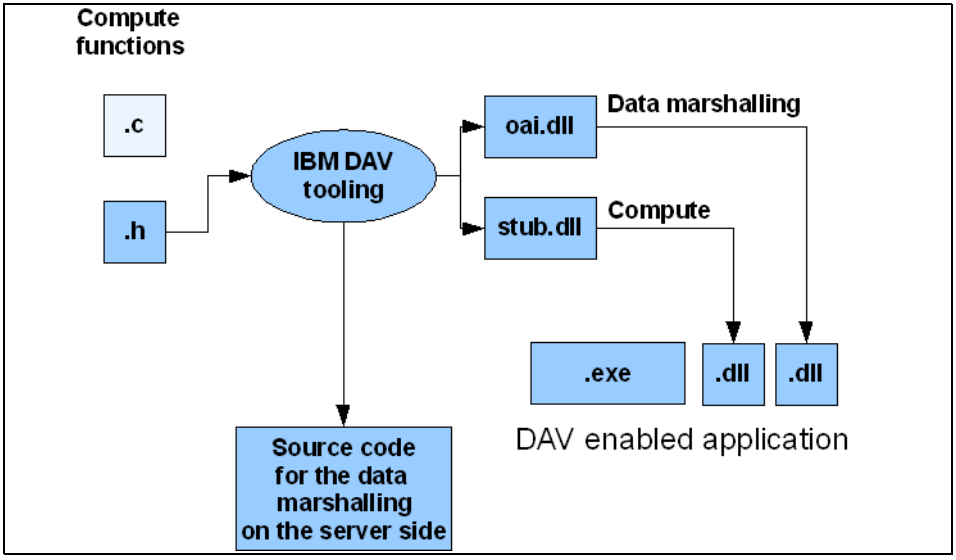


Figure 7-12 The IBM DAV Tooling step

The DAV Tooling component comes bundled with Eclipse 3.3. You should have your header file ready before starting the tooling. After starting the IBM DAV Tooling, open a new project as shown on (Figure 7-13 on page 477) and choose the IBM DAV Tooling wizard.

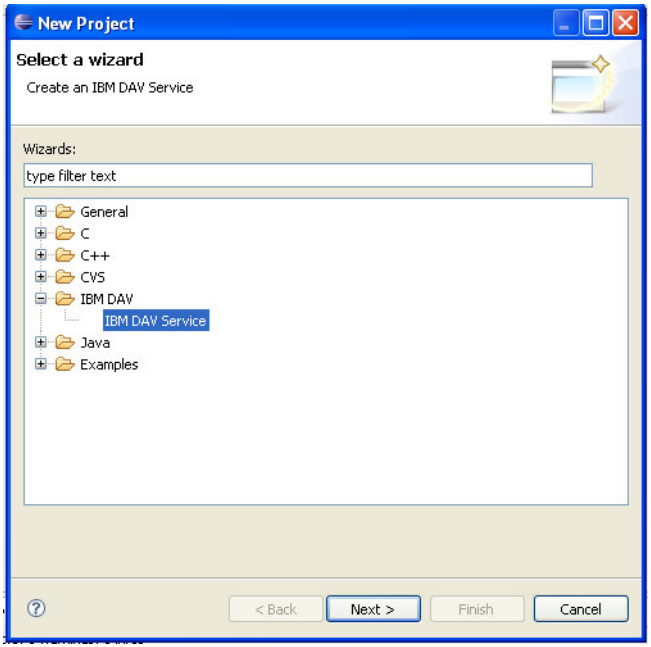


Figure 7-13 Open a new project

Choose a name for the project and open the header file. We used the one supplied in the IBM DAV client package, located under `C:\Program Files\IBM\DAV\sample\Library\Library.h`. See (Figure 7-14 on page 478). You must check the syntax using the “Check syntax” button. The “Finish” button will not be active until you check the syntax of the header file.

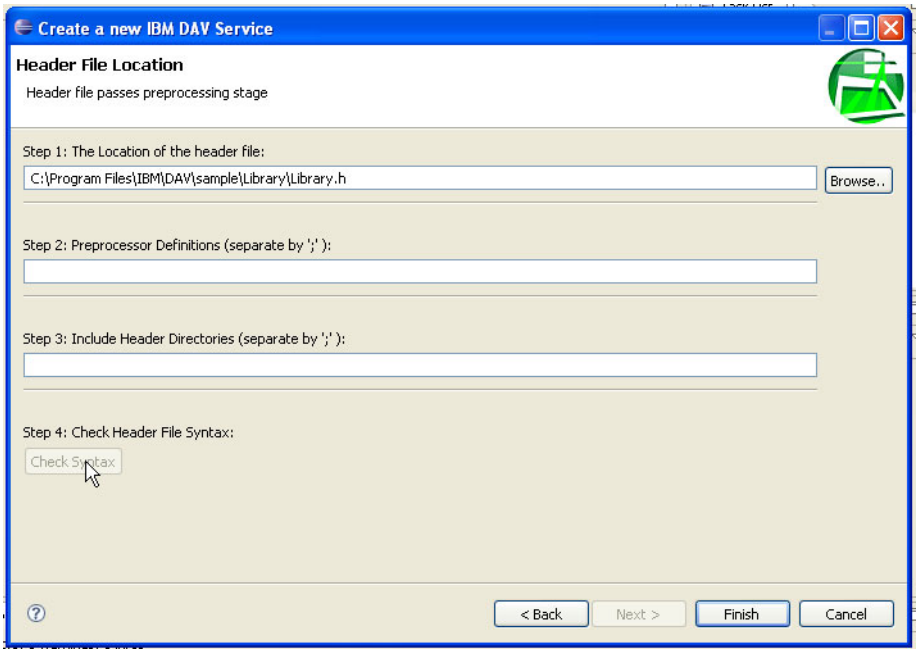


Figure 7-14 Choose the header file

Next, you will then be taken to the DAV Tooling itself. For each function in the list, double click the function prototype and you will then fill the semantic information, describing for each argument, the type of data that is behind. This information will be stored in DAV to generate the necessary code to send data between the client and the server. This is shown below in (Figure 7-15 on page 479).

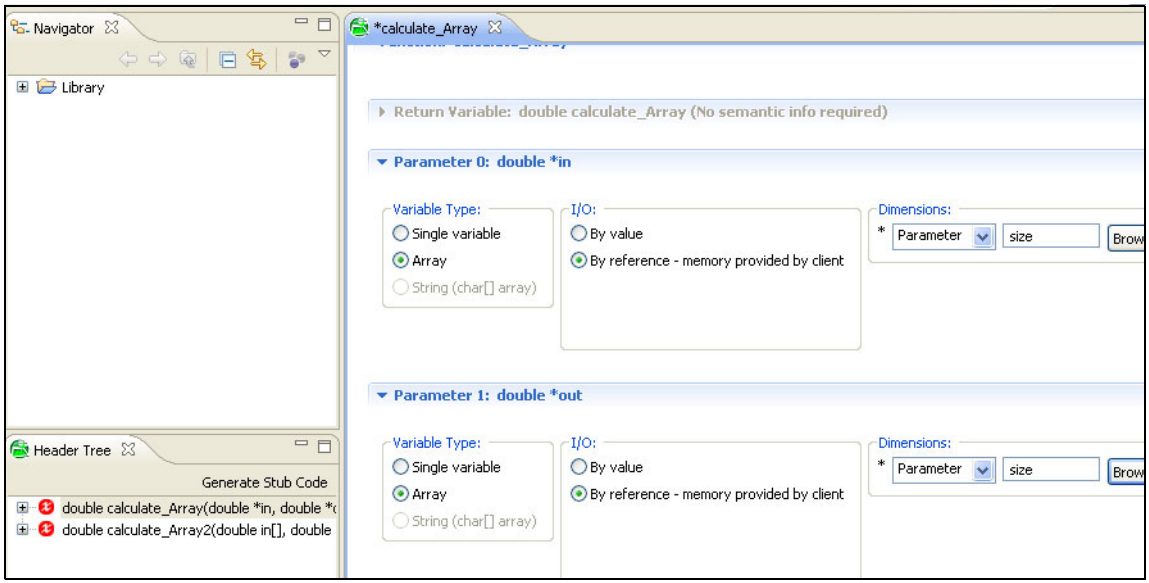


Figure 7-15 Creating the semantic information for the functions

Once every function has been completely described, it's time to generate the stub DLL. This is accomplished by pressing the "Generate stub code" button as shown in (Figure 7-16).

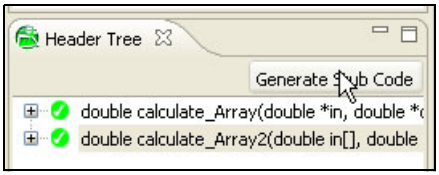


Figure 7-16 This will create the stub DLL

We experienced a slight problem with the Visual C++ 2005 Express Edition here. The free version lacks some libraries that the linker tries to bring in when creating the stub DLL. These libraries (odbc32.lib and odbccp32.lib) are not needed and we just changed the following DAV script : C:\Program Files\IBM\IBM DAV Tooling\eclipse\plugins\com.ibm.oai.appweb.tooling_1.0.1\SDK\compilers\vc so that it would not try to link them in. This step creates many important files for the client and the server side.

The files are located under the C:Documents and settings\<username>\workspace\<projectname> directory. We find there :

Example 7-22 List of files created by the DAV Tooling

```
client directory :
<libraryname>_oai.dll
<libraryname>_oai.lib
<libraryname>_stub.dll
<libraryname>_stub.lib

server directory :
<libraryname>_oai.cpp
<libraryname>_oai.h
makefile
changes_to_server_config_file
```

The client files will be needed to run the DAV enabled application. They must be available in the search path for shared libraries. The `_stub` tagged files contain the fake functions. The `_oai` tagged files contain the client side code for the input and output data marshalling between the client and the server. We copied the four files to the C:\Program Files\IBM\DAV\bin directory. We will point the executable file to this path later on so that it can find the DLLs when it needs to load them.

The server files (tagged `_oai`) contain the server side code for the input and output data marshalling between the client and the server. These files will have to be copied over and compiled on the server node. The `makefile` is provided to do so. The `changes_to_server_config_file` file contains the instructions to make the changes to the DAV configuration files on the server to be able to serve our now offloaded functions.

Build the server side shared libraries

The next step is to implement the offloaded functions on the Cell BE server. In our case, we just compiled them using the gcc compiler. In the real world, we would implement the functions using the compute power of the SPE. The process is shown in (Figure 7-17).

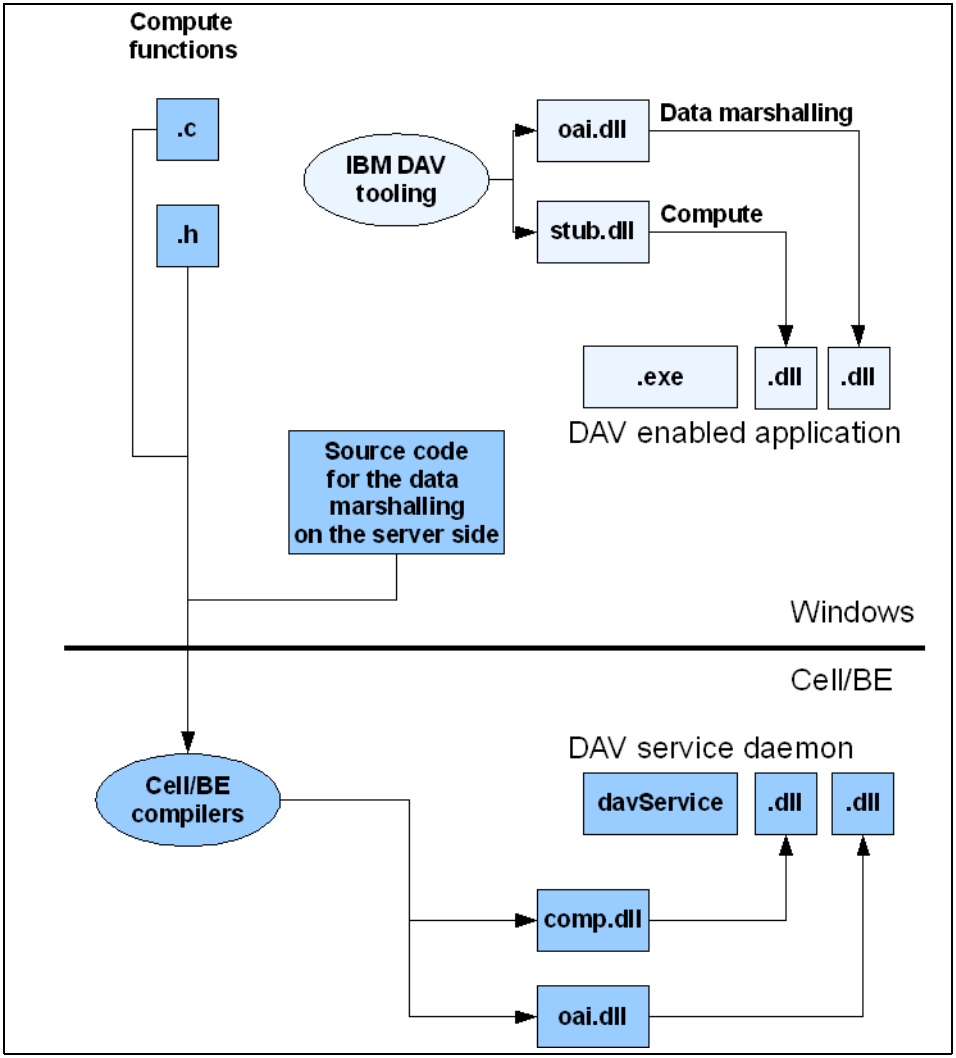


Figure 7-17 Create the shared libraries on the Cell BE

Just like on the client side, we build the lib<libraryname>_oai.so library containing the data marshalling library and the lib<libraryname>.so library containing the actual computational functions. This last one is built using the Calculate.cpp and Calculate.h files listed in Example 7-20 on page 472 and Example 7-21 on page 473 using a command like :

Example 7-23 Build the offloaded functions as a shared library

```
$ gcc -shared -fPIC -o libLibrary.so Calculate.cpp
```

We then have two libraries called `libLibrary.so` and `libLibrary_oai.so` that we decided to put under `/usr/dav/services/Library`. This path can be changed to some other location provided you also change the server config file accordingly.

Configure the server

The last step on the server is to configure it to be able to receive requests from the client. There are a few things to do.

First we need to set a system wide `IBM_DAV_PATH` environment variable to point at the location where DAV has been installed. The default is `/usr/dav`. We added a `dav.sh` file under `/etc/profile.d`.

Next, we need to change the server DAV config file located under `$IBM_DAV_PATH/IBM_DAV.conf`. We need to incorporate the changes that were suggested in the `changes_to_server_config_file` file when we did the tooling step. They basically tell the name by which the service will be called as well as some path information regarding where the shared libraries for the application and the data marshalling libraries are to be found. We can also adjust the logging settings and the port number that the DAV server will be listening to. The exact same port number will have to be specified in the client config file.. The contents of our server file is shown in (Example 7-24)

Example 7-24 The contents of the IBM_DAV.conf file

```
#Logging level - 2 = normal
dav.log.level=2

#Log file location - change as desired
#ensure that the location specified has write access for the user
running dav
dav.log.directory=/var/log
dav.log.filename=ibm_dav.log

dav.listen.port=12456
dav.listen.max=20


#Services
dav.server.service.restarts=1
dav.server.service.restart.interval=3
```

```
dav.server.service.root=services
#Relatve path specified resulting in path of "$IBM_DAV_PATH/services"
#If this entry is missing it defaults to "$IBM_DAV_PATH/bin"


#Service Entries
#These entries will be automatically generated by IBM DAV Tooling in a
"changes_to_server_config_file.txt" file.
#The entries should be inserted here

#This is the sample library and its library path resulting in a
location of "$IBM_DAV_PATH/services/Library"
dav.server.service.Library=Library get_Library_oai Library_oai
dav.server.service.Library.root=Library
```

The last step on the server is to start the DAV server.

Example 7-25 Starting the DAV server

```
# $IBM_DAV_PATH/bin/davStart -t Library
```

`Library` is the name of our service here. The Cell BE server is ready to receive requests from the client application. We now get back to the client side.

Link the client application with the stub library

The last step before running the accelerated application is to relink the application with the stub DLL. This is done using Visual C++ 2005 by changing the linker parameters in the project properties dialog putting the `C:\Program Files\IBM\DAV\bin` directory at the top of the list of searched paths.

Setting client side parameters

We need to tell the DAV client side where to get its acceleration from. This is done by editing the `IBM_DAV.conf` file on the client. The file is located under `C:\Program Files\IBM\DAV`. The contents of this file is listed on (Example 7-26 on page 483). Our Cell BE blade address is listed there together with the port number where the DAV server is listening to.

Example 7-26 The client IBM_DAV.conf file

```
#Logging level - 2 = normal
dav.log.level=2

#Log file location - change as desired
dav.log.directory=c:\
```

```
dav.log.filename=ibm_dav.log

#Connections to servers
#If sample1 is not available, sample2 will be tried
dav.servers=qdc221
qdc221.dav.server.ip=9.3.84.221
qdc221.dav.server.port=12456
```

The server is ready. The application has been instructed to load its computational DLL from the path where the stud DLL has been stored. The DAV client know which accelerator it can work with. We are ready to run the application using Cell BE acceleration.

Run the application

We can now run the application, which although we have not changed it at all, will enjoy Cell BE acceleration. (Figure 7-18 on page 485) shows how this all works. Here are the steps :

1. the application calls a function which now lives in the stub library,
2. the control is passed to the data marshalling library that send input data to the Cell BE server,
3. the data is shipped to the Cell BE server,
4. the data is received by the data marshalling library on the server side which in turn calls the compute functions,
5. the results are handed back to the data marshalling library,
6. the output data is transferred back to the client side,
7. the data movement library handles the output data received,
8. the control is passed back to the stub compute library which will resume the main application.

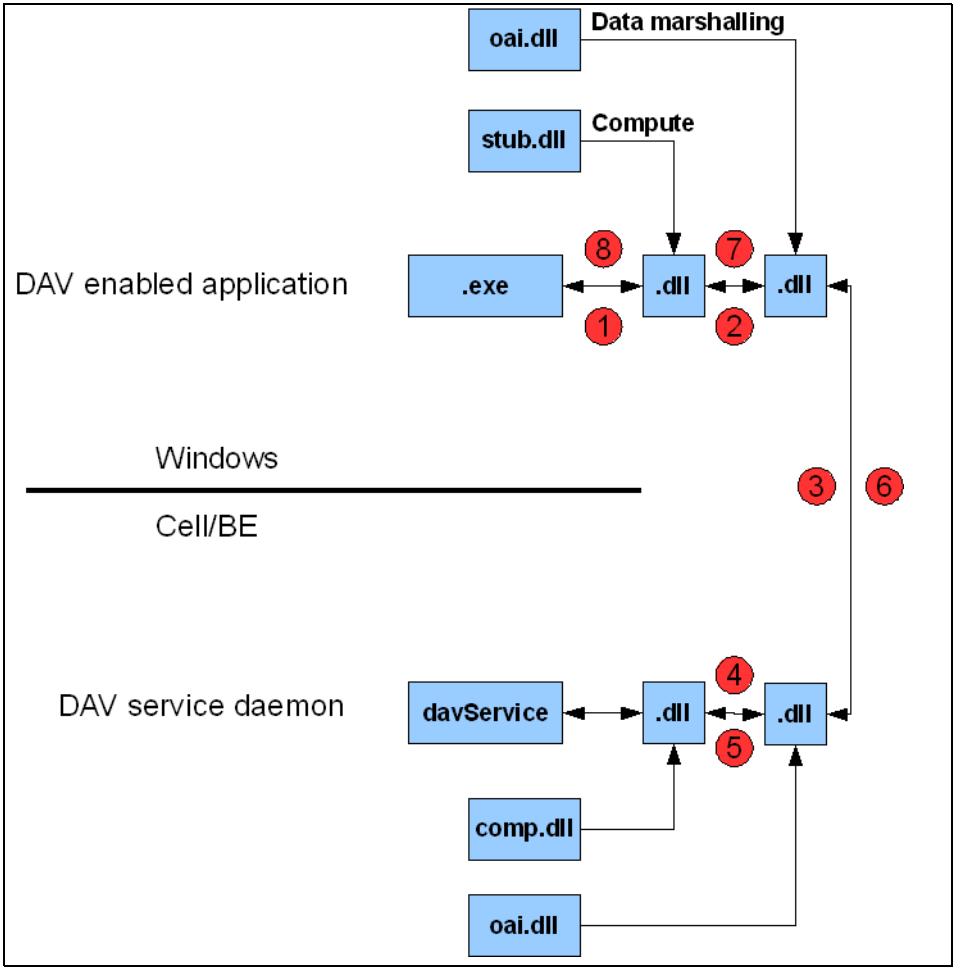


Figure 7-18 The flow of the Cell BE accelerated application

Let's start the application. See (Figure 7-19 on page 486).

Figure 7-19 Launching the application

Quickly after being started, the application will call the `CalculateArray` function which is now coming from the stub library. This stub library will call the server side. The system on which we ran the application had the Zone Alarms Checkpoint Integrity Flex™ firewall installed. This software traps all socket connections for security purposes. This is a simple way to see that our application is now going to the network for its computational functions. This is shown in (Figure 7-20).

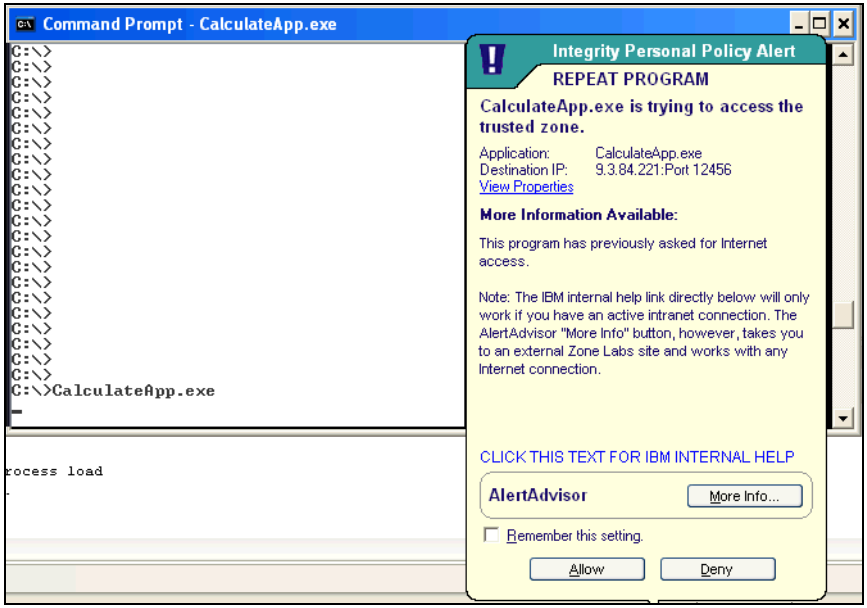


Figure 7-20 Firewall captured the acceleration request

Of course you don't want this to happen all the time, but this shows that we are going to the network. Once we tell the firewall to allow the connection to 9.3.84.221:12456, our Cell BE listening on port 12456, the server side of DAV will load the library containing our computation functions, get them to do their work and ship back the output data. Upon returning from the computational routines the application will continue its work as shown in (Figure 7-21 on page 487).

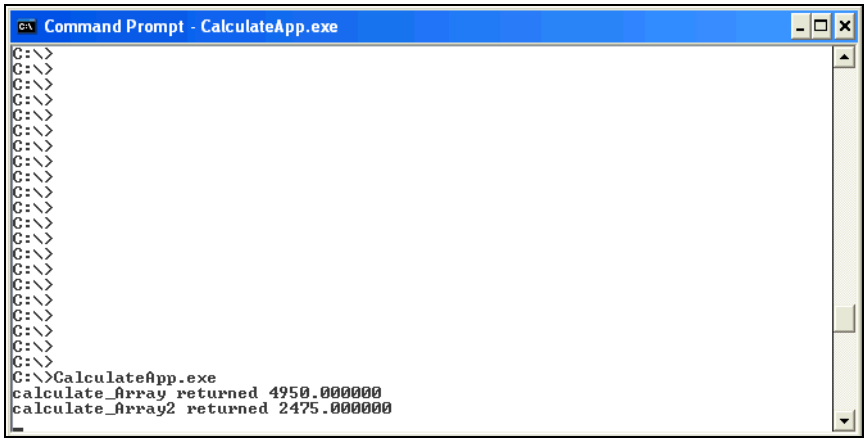


Figure 7-21 The results from the DAV server have come back.

If we look at the server side, we see that transactions get logged to the DAV log file. See (Figure 7-22). We are calling from 9.41.223.243.

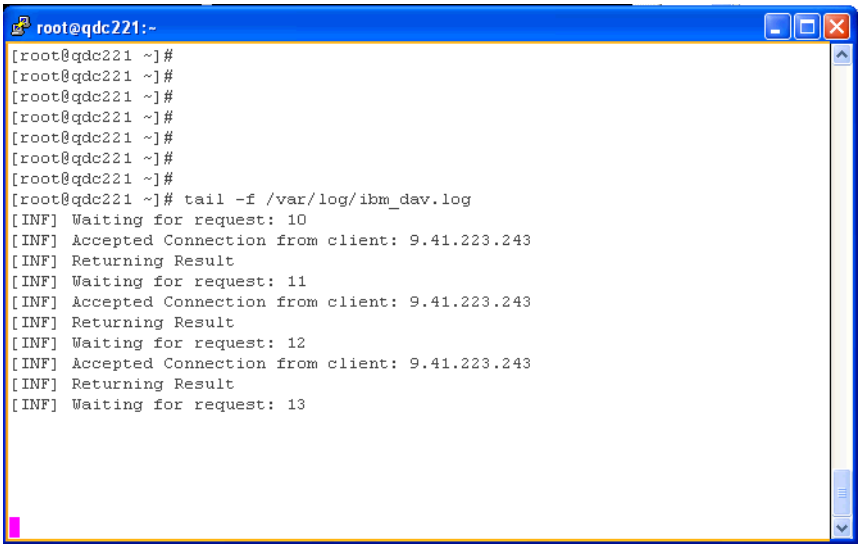


Figure 7-22 All transactions are logged

Looking at the processes runnning on the server, we see that we have two processes running on the host, the davStart daemon and the davService process forked to handle our request. See (Figure 7-23 on page 488).

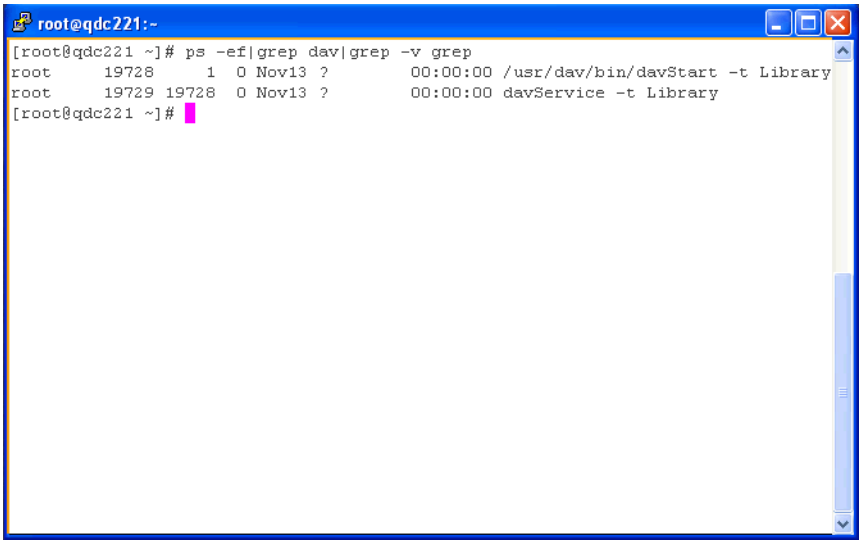


Figure 7-23 The DAV processes running on the server

We also see from the maps file that our shared libraries have been loaded into the DAV server process. Take a look at the seventh to tenth lines of output on (Figure 7-24 on page 489).

```
root@qdc221:~
[root@qdc221 ~]# cat /proc/19729/maps
00100000-00120000 r-xp 00100000 00:00 0                                  [vdso]
00270000-00290000 r-xp 00000000 00:14 3148462                            /lib/libgcc_s-4.1.2-20070503.so.1
00290000-002a0000 rw-p 00010000 00:14 3148462                            /lib/libgcc_s-4.1.2-20070503.so.1
005c0000-006e0000 r-xp 00000000 00:14 3281680                            /usr/lib/libstdc++.so.6.0.8
006e0000-006f0000 r--p 00110000 00:14 3281680                            /usr/lib/libstdc++.so.6.0.8
006f0000-00700000 rw-p 00120000 00:14 3281680                            /usr/lib/libstdc++.so.6.0.8
0fcb0000-0fcc0000 r-xp 00000000 00:14 3673416                            /usr/dav/services/Library/libLibrary.s
0fcc0000-0fcd0000 rw-p 00000000 00:14 3673416                            /usr/dav/services/Library/libLibrary.s
0fce0000-0fcf0000 r-xp 00000000 00:14 3673417                            /usr/dav/services/Library/libLibrary_d
0fcf0000-0fd00000 rw-p 00000000 00:14 3673417                            /usr/dav/services/Library/libLibrary_d
0fd10000-0fd20000 r-xp 00000000 00:14 3148440                            /lib/libdl-2.6.so
0fd20000-0fd30000 r--p 00000000 00:14 3148440                            /lib/libdl-2.6.so
0fd30000-0fd40000 rw-p 00010000 00:14 3148440                            /lib/libdl-2.6.so
0fd40000-0fe00000 r-xp 00000000 00:14 3148444                            /lib/libm-2.6.so
0fe00000-0fe10000 r--p 000b0000 00:14 3148444                            /lib/libm-2.6.so
0fe10000-0fe20000 rw-p 000c0000 00:14 3148444                            /lib/libm-2.6.so
0fee0000-0fef0000 r-xp 00000000 00:14 3673051                            /usr/dav/bin/libbridge.so
0fef0000-0ff00000 rw-p 00000000 00:14 3673051                            /usr/dav/bin/libbridge.so
0ff10000-0ffa0000 r-xp 00000000 00:14 3673050                            /usr/dav/bin/libaw.so
0ffa0000-0ffb0000 rw-p 00090000 00:14 3673050                            /usr/dav/bin/libaw.so
0ffc0000-0ffe0000 r-xp 00000000 00:14 3147812                            /lib/ld-2.6.so
0ffe0000-0fff0000 r--p 00010000 00:14 3147812                            /lib/ld-2.6.so
0fff0000-10000000 rw-p 00020000 00:14 3147812                            /lib/ld-2.6.so
10000000-10030000 r-xp 00000000 00:14 3673046                            /usr/dav/bin/davService
10030000-10040000 rw-p 00020000 00:14 3673046                            /usr/dav/bin/davService
10040000-10070000 rwxp 10040000 00:00 0                                  [heap]
f7e30000-f7fb0000 r-xp 00000000 00:14 3147815                            /lib/libc-2.6.so
f7fb0000-f7fc0000 r--p 00170000 00:14 3147815                            /lib/libc-2.6.so
f7fc0000-f7fd0000 rw-p 00180000 00:14 3147815                            /lib/libc-2.6.so
f7fe0000-f7ff0000 rw-p f7fe0000 00:00 0
ff7e0000-ff930000 rw-p ff7e0000 00:00 0                                  [stack]
[root@qdc221 ~]#
```

Figure 7-24 The maps file for the davService process

In essence, this is how IBM DAV works. Every application, provided it gets its computational functions from a DLL, can be accelerated on a Cell BE using DAV.

Visual Basic example: an Excel 2007 spreadsheet.

IBM DAV is a great tool for bringing Cell BE acceleration to Microsoft Windows applications running on Intel processors. It requires no source code changes to the application and provides the quickest path to Cell BE as it limits the porting effort to the number crunching functions only. Still, some usage considerations are important.

IBM DAV is best used for accelerating highly computational functions requiring litlle input and output. This is a general statement but it is exacerbated here by the fact that the data transfer is currently performed with TCP/IP which has high latency. It should be kept in mind that no state data can be kept on the

accelerator side between two invocations. Also, all the data needed by the accelerated functions need to be passed as arguments for the DAV runtime to be able to ship all the required information to the accelerator. The functions should be self-contained.

The davServer process running on the Cell BE is a 32 bit Linux program and we are therefore limited to 32 bit shared libraries on the server side.

Part 3

Application Re-engineering

In this part of the book we focus on specific application re-engineering topics: Monte Carlo Simulation, and FFT Algorithms.

8

Case study: Monte Carlo Simulation

Monte Carlo simulation is a popular computational technique used in Financial Services Sector, for example, to calculate stock prices, interest rates, exchange rates, commodity prices, and risk management that requires estimating losses with certain probability over a time period. Besides Financial Services Sector, Monte Carlo simulation method is widely used in other engineering and scientific areas, for example, Computational Physics. Calculating option pricing (option value) is an important area in Financial Services Sector. An option is an agreement between a buyer and a seller; the buyer of a European call option buys the right to buy a financial instrument for a preset price(strike price) at expiration (maturity). Similarly, the buyer of a European put option buys the right to sell a financial instrument for a preset price at expiration. The buyer of an option is not obligated to exercise the option; for example, if the market price of the underlying asset is below the strike price on the expiration date, then the buyer of a call option can decide not to exercise that option. In this chapter, we show how to implement Monte Carlo simulation on Cell BE to calculate option pricing based on Black-Scholes model by providing sample codes. We include techniques to improve the performance and provide performance data. Also, since mathematical functions such as log, exp, sine and cosine are used extensively in option pricing, we discuss the use of the following SDK 3.0 libraries by providing examples and Makefile:

- SIMD Math; this consists of SIMD versions(short vector versions) of the traditional libm math functions on Cell BE.
- MASS (Mathematical Acceleration Subsystem); this provides both SIMD and vector versions of mathematical intrinsic functions, which are tuned for optimum performance on Cell BE; MASS treats exceptional values differently and may produce slightly different results, compared to SIMD math.

;

8.1 Monte Carlo simulation for option pricing

Option pricing involves the calculation of option payoff values that depend on the price of the underlying asset. The Block-Scholes model is based on the assumption that the price of an asset follows geometric Brownian Motion, which is described by a SDE (Stochastic Differential Equation). An Euler scheme is used to discretize the SDE, and the resulting equation can be solve using Monte Carlo simulation[1]. In Example 8-1, we show the basic steps for Monte Carlo simulation to calculate the price S of an asset (stock) at different time steps $0 = t_0 < t_1 < t_2 < \ldots t_M = T$, where T is time to expiration(maturity). The current stock price at time $t_0, S(t_0) = S_0$, the interest rate, r, and the volatility, v, are known.

Example 8-1 Pseudo code for Monte Carlo cycles; N is the number of cycles and M is the number of time points in each cycle. r is the interest rate, v is the volatility

```
for i=0, 1, 2, .., N-1
{
    for j=0, 1, 2, .. M-1
    {
      get a standard normal random number X^i_j
      dt_j = t_j - t_j-1
   S^i(t_j) = S^i(t_j-1)*exp((r-0.5v^2)*dt_j+v*sqrt(dt_j)*X^i_j))

   }
}
```

In Example 8-2, we show a pseudo code example for calculating European option call and put values.

Example 8-2 European option pricing; S is the spot price and K is the strike price

```
    C^i = MAX(0,S[M-1] - K); i=0,1, ..., N-1

    P^i = MAX(K-S[M-1],0); i=0,1, ..., N-1

Average current call value = exp(-rT)*(C^0+C^1+....+C^N-1)/N

Average current put value = exp(-rT)*(P^0+P^1+....+P^N-1)/N
```

[1] See references [12] or [24] in "Other publications" on page 619.

In Example 8-3, we give a pseudo code to calculate Asian option call and put values.

Example 8-3 Asian option pricing; S is the spot price and K is the strike price.

```
B^i = (S^i(t_0) + S^i(t_1) + ....+ S^i(t_M-1))/M; i=0,1,2, ..., N-1
C^i = MAX(0,B^i-K); i=0,1,2, ..., N-1
P^i = MAX(B^i-K, 0); i=0,1,2, ..., N-1
Average current call value = exp(-rT)*(C^0+C^1+....+C^N-1)/N
Average current put value = exp(-rT)*(P^0+P^1+....+P^N-1)/N
```

We note that there are different types of options traded in the market; for example, an American call/put option gives the buyer the right to sell/buy the underlying asset at strike price on or before the expiration date[2].

The main computational steps for option values are in Example 8-1; further, the most time consuming part is getting the standard Gaussian(normal) random variables; these are random numbers that have the following probability density function with mean 0 and standard deviation 1::

$$f(x) = \frac{1}{2\Pi}\exp\left(-\frac{1}{2}x^2\right), \quad -\infty < x < \infty$$

8.2 Methods to generate Gaussian(normal) random variables

In this section, we discuss some of the algorithms available to generate Gaussian random numbers. The general procedure is as follows:

1. generate a 32-bit random unsigned integer x
2. convert x to a float or double uniform random variable y in (0,1)
3. transform y to a standard normal random variable z

Mersenne Twister[3] is a popular method to generate random numbers that are 32-bit unsigned integers. This method has good properties such as long period,

[2] For more details on American style option calculation methods, see reference [12] in "Other publications" on page 619.

[3] See reference [11] in "Other publications" on page 619.

good distribution property, and efficient use of memory. Also, there are other methods that are equivalently good. SDK3.0 provides Mersenne Twister as one of the random number generators[4]. This method takes an unsigned integer as seed and generates a sequence of random numbers. Next step is to change these unsigned 32-bit integers to uniform random variables in (0, 1). The final step is to transform the uniform random numbers to standard normal random numbers for which Box-Muller method or its variant polar method can be used. Box-Muller method as well as the polar method requires two uniform random numbers and returns two normal random numbers[5]. In the following example, we give a sample code to generate two standard normal single precision random numbers; the code can be easily changed to generate two standard normal double precision random numbers.

Example 8-4 Sample code to generate standard normal random numbers

```
//generate uniform random numbers
float rand_unif()
{
   float c1= 0.5f, c2 = 0.2328306e-9f;
   return (c1 + (signed) rand_MT() * c2);
}


//Box-Muller method
void box_muller_normal(float *z)
{
   float  pi=3.14159f;
   float t1,t2;


    t1 = sqrtf(-2.0f *logf( rand_unif() );
    t2 = 2.0f*pi*rand_unif();
    z[0] = t1 * cos(t2);
    z[1] = t1 * sin(t2);

}
//polar method
void pollar_normal(float *z)
{
    float t1, t2, y;

    do {
          t1 = 2.0f * rand_unif() - 1.0f;
```

[4] See reference [22] in “Other publications” on page 619.
[5] See reference [23] in “Other publications” on page 619.

```
            t2 = 2.0f * rand_unif() - 1.0f;
            y  = t1 * t1 + t2 * t2;
        } while ( y >= 1.0f );

        y  = sqrtf( (-2.0f * logf( y ) ) / y );
      z[0] = t1 * y;
      z[1] = t2 * y;
}
```

8.3 Parallel and vector implementation of Monte Carlo algorithm on Cell

In this section, we show how to parallelize and vectorize the above Monte Carlo simulation algorithm for option pricing on Cell. Also, we discuss the use of SIMD Math, MASS SIMD and MASS Vector libraries.

8.3.1 Logical steps

In Figure 8-1 we show the logical steps for parallelizing the Monte Carlo simulation.

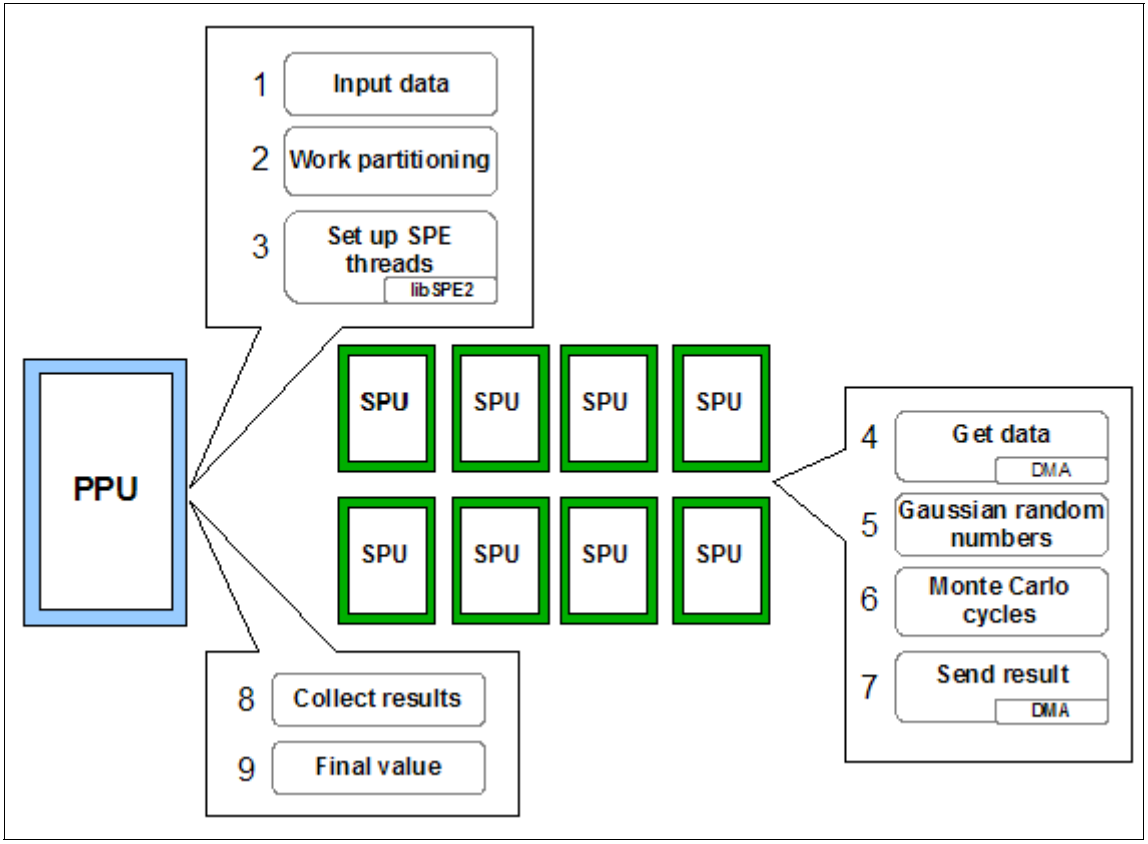


Figure 8-1 Logical steps: parallelizing the Monte Carlo simulation

The logical steps from Figure 8-1 are as follows:

1. **Input data:** Reads input values such as number of Monte Carlo simulations, spot_price, strike_price, interest_rate, volatility, time_to_maturity, num_of_ time steps.
2. **Work partitioning:** Decides how many SPEs to use based on the number of simulations required; partitions the work and prepares the data needed for each SPE.
3. **Set up SPE threads:** Creates SPE contexts and execution threads, loads the SPE program, starts the execution of SPE threads using libspe2 functions.
4. **Get data:** Each SPE gets initial seed values and data for Monte Carlo simulation using DMA.

5. **Gaussian random numbers**: Each SPE generates unsigned integer random numbers and converts them to standard Gaussian random numbers using Box-Muller transformation.
6. **Monte Carlo cycles**: Each SPE performs its portion of Monte carlo cycles and computes an average option value as result.
7. **Send the result**: Each SPE sends its result to PPE using DMA.
8. **Collect results**: Checks if work is done by SPEs and collects the results form SPEs.
9. **Final value**: Calculates the average of the results collected and computes the final option value.

In what follows we give more details for some of the above steps; also, because the computational steps for getting European option call and put values and Asian option values are similar, we restrict our discussion to European option call value.

In general, the number of Monte Carlo simulations, N, is very large(millions or hundreds of thousands) since the rate of convergence for Monte Carlo simulation method is 1/sqrt(N). Further, the Monte Carlo cycles (simulations) are independent; hence, we can divide the number of cycles, N in Example 8-1, by the number of available SPEs and distribute the work load among the SPEs in Step 2 in the graph. Further, in each SPE, four Monte Carlo cycles can be done simultaneously for single precision (two cycles for double precision) using Cell BE SIMD instructions. So, it is important that the number of Monte Carlo cycles allocated for each SPE is a multiple of four for single precision and two for double precision.

As we have noted before, in Example 8-1, the main computational part is generating standard normal random numbers, so it requires careful implementation to reduce the overall computing time. For computational efficiency and because of limited local stores available on SPEs, precomputing all random numbers and storing them should be avoided. Instead, generating the random numbers during the Monte Carlo cycles in each SPE is recommended. But, this poses a major challenge in the sense that we cannot simply implement the serial random number generators such as Mersenne Twister that generates a sequence of random numbers based on a single seed value as input. At a minimum, generating random numbers on SPEs in parallel requires different seed values as input[6]. However, using different seeds on different SPEs is not enough since the generated random numbers may be correlated; that is, the random numbers are not independent; hence, the quality of Monte Carlo simulations will not be good and that leads to inaccurate results. Note that these

[6] Note: the Mersenne Twister random number generator with different seeds on SPEs is used in reference [13], “Other publications” on page 619

remarks apply to all parallel machines, not specific to Cell BE. One way to avoid these problems is to use parallel random number generators such as Dynamic Creator[7] on Cell.

Dynamic Creator is based on Mersenne Twister algorithm, which depends on a set of parameters, called Mersenne Twister parameters, to generate a sequence of random numbers for a given seed. The main difference in Dynamic Creator algorithm is that we can make certain Mersenne Twister parameters different on different SPEs so that the generated sequences are independent to each other, resulting in high quality random numbers overall. Also, Dynamic Creator provides the capability to precompute these parameters, which can be done on PPE and saved in an array; note that this has to be done only once for a given period.

Following the logical steps given in step 2 of Figure 8-1, on PPE, we store the values such as number of simulations, J=N/number_of_SPUs, interest rate, r, volatility, v, number of time steps, Mersenne Twister parameters, and the initial seeds in the control structure, defined in Example 8-5, to transfer the data to SPEs. Based on these input values, in step 6 Figure 8-1, the average call value:

$C_k = (C^0 + C^1 + \ldots + C^{J-1})/J;\ k = 1, 2, \ldots$ number_of_SPUs,

where C^i is defined in Example 8-2 on page 495. As indicated in the final step in Figure 8-1, these results are combined to compute the present call value:

$\exp(-rT)*((C_1+C_2+\ldots+C_k)/$number_of_SPUs$)$

In Example 8-6, we provide a sample code for the main program on SPUs to calculate European option call value. First, in Example 8-5 we define the control structure that will be used to share data between PPU and SPUs.

Example 8-5 Sample control structure to share data between PPU and SPUs

```
typedef struct _control_st {
  unsigned int  seedvs[4]; /* array of seeds  */
  unsigned int  dcvala[4]; /* MT parameters   */
  unsigned int  dcvalb[4]; /* MT parameters  */
  unsigned int  dcvalc[4]; /* MT parameters  */
           int  num_simulations; /* number of MC simulations  */
         float  spot_price;
         float  strike_price;
         float  interest_rate;
         float  volatility;
           int  time_to_maturity;
           int  num_time_steps;;
         float  *valp;
```

[7] See reference [10] in “Other publications” on page 619.

```
        char   pad[28]; /* padding */
} control_st;
```

Example 8-6 Sample SPU main program for Monte Carlo simulation

```
#include <spu_mfcio.h>

/* control structure */
control_st cb __attribute__ ((aligned (128)));
//


int main(unsigned long long speid, unsigned long long parm)
{
/*
  DMA control structure cb into local store.
*/
  spu_writech(MFC_WrTagMask, 1 << 0);
  spu_mfcdma32((void *)(&cb), (unsigned int)parm,
                                    sizeof(cb),0,MFC_GET_CMD);
  (void)spu_mfcstat(2);

//Get input values for Monte Carlo simulation.

  my_num_simulations = cb.num_simulations;
                   s = cb.spot_price;
                   x = cb.strike_price;
                   r = cb.interest_rate;
               sigma = cb.volatility;
                   T = cb.time_to_maturity;
                  nt = cb.num_time_steps;


//get seed
  seed = ((vector unsigned int){cb.seedvs[0], cb.seedvs[1],
                                cb.seedvs[2], cb.seedvs[3]});



//
//Get Mersenne Twister parameters that are different on SPUs

    A     = ((vector unsigned int){cb.dcvala[0], cb.dcvala[1],
                                   cb.dcvala[2], cb.dcvala[3]});
```

```
    maskB = ((vector unsigned int){cb.dcvalb[0], cb.dcvalb[1],
                                   cb.dcvalb[2], cb.dcvalb[3]});
    maskC = ((vector unsigned int){cb.dcvalc[0], cb.dcvalc[1],
                                   cb.dcvalc[2], cb.dcvalc[3]});

/Intialize the random number generator
    rand_dc_set(seed, A, maskB, maskC);

// compute European option average call value -- Monte Carlo simulation
    monte_carlo_eur_option_call(s, x , r, sigma, T, nt,
                                     ,my_sim_size, &value);

// send the value to PPU
spu_writech(MFC_WrTagMask, 1 << 0);
   spu_mfcdma32((void *)(&value),
        (unsigned int)(cb.valp), sizeof(float),0,MFC_PUT_CMD);

// wait for the DMA to complete
   (void)spu_mfcstat(2);

   return 0;
}
```

Attention: In control_st, padding, pad[28], is done so that its size is 128 bytes. otherwise, the DMA commands in the above program will give bus error because the minimum length for a DMA is 128 bytes.

8.3.2 Sample code for European option on SPU

Next, we provide a sample code to compute European option call value on SPUs. Since the Monte Carlo cycles are independent, four cycles can be done simultaneously by vectorizing the outer loop in Example 8-1 on page 495. The following sample code uses SPU vector intrinsics and SIMD Math functions.

Example 8-7 Sample SPU vector code for European option call value

```
#include <spu_intrinsics.h>
#include <simdmath/expf4.h>
#include <simdmath/sqrtf4.h>
//
#include <sum_across_float4.h>
```

```
//
/*
 s    spot price
 x    strike (exercise) price,
 r    interest rate
sigma volatility
t_m   time to maturity
nt    number of time steps
*/
void monte_carlo_eur_option_call(float s, float x, float r,
                 float sigma, float t_m,int nt,int nsim,float *avg)
{
      vector float c,v0,q,zv;
      vector float dtv,rv,vs,p,y,rvt,sigmav,xv;
      vector float sum,u,sv,sinitv,sqdtv;
      int i,j,tot_sim;
//
      v0      = spu_splats(0.0f );
      c       = spu_splats(-0.5f);
      dtv     = spu_splats( (t_m/(float)nt) );
      sqdtv   = _sqrtf4(dtv);
      sigmav  = spu_splats(sigma);
      sinitv  = spu_splats(s);
      xv      = spu_splats(x);
      rv      = spu_splats(r);
      vs      = spu_mul(sigmav, sigmav);
      p       = spu_mul(sigmav, sqdtv);
      y       = spu_madd(vs, c, rv);
      rvt     = spu_mul(y,dtv);

      tot_sim = ( (nsim+3)&~3 ) >> 2;
      sum     = spu_splats(0.0f );

      for (i=0; i < tot_sim; i++)
      {
        sv  = sinitv;
        for (j=0; j < nt; j++)
        {
          rand_normal(zv);
          u   = spu_madd(p , zv, rvt);
          sv  = spu_mul(sv,_expf4(u));
        }
           q  = spu_sub(sv,xv);

          sv  = _fmaxf4(q ,v0);
```

```
        sum = spu_add(sum,sv);
    }
    *avg = _sum_across_float4(sum)/( (float)tot_sim*4.0f);
```

Note that in the above example, in the step that computes `tot_sim`, we first make the number of simulations a multiple of four before dividing it by four. In addition to using simdmath functions expf4 and sqrtf4, we have also used the function _fmaxf4 to get the component-wise maximum of two vectors and SDK library function _sum_across_float4 to compute the sum of the vector components.

8.4 Generating Gaussian random numbers on SPUs

In this section, we discuss some techniques to generate Gaussian(normal) random numbers on SPUs. Note that the main parts in Example 8-7 are the routine `rand_normal` for generating standard Gaussian random numbers and the rest of the instructions for option value calculations. As we noted in section 8.2, “Methods to generate Gaussian(normal) random variables” on page 496, obtaining standard Gaussian random numbers requires two main computational steps;

1. Generating random unsigned integer numbers.
2. Computing standard normal random numbers, for example, using Box-Muller method or Polar method.

For step 1, we recommend a parallel random number generator such as Dynamic Creator[8]. We note that the library[9] consists of functions to generate different random numbers, but it doesn’t include Dynamic Creator. The code in Dynamic Creator for generating random numbers requires only integer computations, which can be easily changed to vector instructions using SPU intrinsics. For the vector version of Dynamic Creator, the components of a seed vector should be independent; thus, on 16 SPUs, 64 independent seed values, unsigned integers, are needed. As an example, one can use thread ids as seed values. We note that for Dynamic Creator. as indicated in Example 8-7, only

[8] See reference [10] in “Other publications” on page 619.
[9] See reference [23] in “Other publications” on page 619.

three Mersenne Twister parameters *A*, *maskB* and *maskC* that are different on SPUs need to be set when the random number generator is initialized; the rest of the Mersenne Twister parameters do not change and they can be inlined in the random number generator code.

For step2, we give a sample SPU code in Example 8-8, which is a vector version of Box-Muller method in Example 8-4; here, we convert the vectors of random unsigned integers to vectors of floats, generate uniform random numbers, and use Box-Muller transformation to obtain vectors of standard normal random numbers.

Example 8-8 Sample single precision SPU code to generate standard Gaussian random numbers.

```
#include <spu_mfcio.h>
#include <simdmath/sqrtf4.h>
#include <simdmath/cosf4.h>
#include <simdmath/sinf4.h>
#include <simdmath/logf4.h>

//

void rand_normal_sp(vector float *z)
{
    vector float u1,u2,v1,v2,w1,p1;
    vector float c1 = ((vector float) { 0.5f,0.5f,0.5f,0.5f});
    vector float c2 = ((vector float) { 0.2328306e-9f, 0.2328306e-9f,
                                        0.2328306e-9f, 0.2328306e-9f});
    vector float c3 = ((vector float) { 6.28318530f, 6.28318530f,
                                        6.28318530f, 6.28318530f});
    vector float c4 = ((vector float) { -2.0f,-2.0f,-2.0f,-2.0f});
    vector unsigned int y1, y2;


// get y1, y2 from random number generator.

//convert to uniform random numbers
       v1 = spu_convtf( (vector signed int) y1,  0) ;
       v2 = spu_convtf( (vector signed int) y2,  0) ;
       u1 = spu_madd( v1,  c2, c1);
       u2 = spu_madd( v2,  c2, c1);

// Box-Muller transformation
       w1 = _sqrtf4( spu_mul(c4,_logf4(u1)) );
       p1 = spu_mul(c3, u2);
     z[0] = spu_mul(w1, _cosf4(p1) );
```

```
    z[1] = spu_mul(w1, _sinf4(p1) );
}
```

In the above example, the C statement `(vector signed int) y1` converts the components of y1 that are unsigned integers to signed integers; the instruction `spu_convtf` converts each component of the resulting vector signed int to a floating-point value and divides it by 2^{scale}; scale=0. For double precision, the generated 32-bit unsigned integer random numbers are converted to floating-point values and extended to double precision values; note that a double precision vector will have only two elements because each element is 64-bit long. In the above example, the vector `y1` has four elements that are 32-bit unsigned integer random numbers. So, we can compute the required two double precision vectors of uniform random numbers for Box-Muller transformation by shuffling the components of y1; this avoids the computation of `y2` in the above example. To explain this idea, we give a sample code in Example 8-9.

Example 8-9 Sample double precision SPU code to generate standard Gaussian random numbers.

```
#include <spu_mfcio.h>
#include <simdmath/sqrtd2.h>
#include <simdmath/logd2.h>
#include <simdmath/cosd2.h>
#include <simdmath/sind2.h>

//

void rand_normal_dp(vector double *z)
{
    vector double u1,u2,v1,v2,w1,p1;
    vector double c1 = ((vector double) { 0.5, 0.5});
    vector double c2 = ((vector double) { 0.2328306e-9, 0.2328306e-9});
    vector double c3 = ((vector double) { 6.28318530, 6.28318530});
    vector double c4 = ((vector double) { -2.0, -2.0});

vector unsigned char pattern={4, 5, 6, 7, 0, 1, 2, 3, 12, 13, 14, 15,
8, 9, 10, 11};
    vector unsigned int y1, y2;

// get y1 from random number generator.

       y2 = spu_shuffle(y1, y1, pattern);
```

```
//convert to uniform random numbers
       v1 = spu_extend( spu_convtf( (vector signed int) y1,  0) ) ;
       v2 = spu_extend( spu_convtf( (vector signed int) y2,  0) ) ;
       u1 = spu_madd( v1,  c2, c1);
       u2 = spu_madd( v2,  c2, c1);

// Box-Muller transformation
       w1 = _sqrtd2( spu_mul(c4,_logd2(u1)) );
       p1 = spu_mul(c3, u2);
     z[0] = spu_mul(w1, _cosd2(p1) );
     z[1] = spu_mul(w1, _sind2(p1) );
}
```

Next, we discuss some ideas to tune the code on SPUs. In order to improve register utilization and instruction scheduling, the outer loop in Example 8-7 can be unrolled. Further, since a vector version of Box-Muller method as well as Polar method computes two vectors of standard normal random numbers out of two vectors of unsigned integer random numbers, we can use all generated vectors of normal random numbers by unrolling the outer loop, for example, to a depth of two or four.

Note that the in the above examples, we have used mathematical intrinsic functions such as exp, sqrt, cos and *sin* that are from *S*IMD math library, which takes advantage of SPU SIMD (vector) instructions and provides significant performance gain over the standard libm math library[10]. The Mathematical Acceleration Subsystem (MASS), available in SDK3.0, provides mathematical intrinsic functions that are tuned for optimum performance on PPU and SPU. MASS libraries provide better performance than SIMD math libraries for most of the intrinsic functions. In some cases the results for MASS functions may not be as accurate as the corresponding functions in SIMD math libraries, and MASS may handle the edges differently[11]. We found with our implementation of European option pricing that performance can be improved using MASS and the accuracy of the computed result using MASS is about the same as that using SIMD math. We note that the current version of MASS in SDK 3.0 provides only single precision math intrinsic functions.

Use of MASS inline functions instead of SIMD math functions is easy; for example, it requires only changing the SIMD math include statements in

[10] See reference [21] in “Other publications” on page 619.
[11] Note, a comparison of the accuracy of results between MASS functions and SIMD math functions can be found in reference [22] in “Other publications” on page 619.

Example 8-7 on page 503 and in Example 8-8 on page 506 by the corresponding include statements given in Example 8-10 and in Example 8-11, respectively.

Example 8-10 Include statements to include MASS intrinsic functions on SPUs

```
#include <mass/expf4.h>
#include <mass/sqrtf4.h>
```

Example 8-11 Sample include statements to include MASS intrinsic functions on SPUs.

```
#include <mass/sqrtf4.h>
#include <mass/cosf4.h>
#include <mass/sinf4.h>
#include <mass/logf4.h>
```

Alternatively, in the above examples, one can use MASS SIMD library functions instead of inlining them; for this, one needs to change the include statements and the function names. Example 8-12, which is a modified version of Example 8-8 on page 506, shows this. Further, the MASS SIMD library libmass_simd.a should be added at the link step; see the Makefile, Example 8-13 on page 510.

Example 8-12 Sample SPU code to generate standard Gaussian random numbers using MASS SIMD library

```
#include <spu_mfcio.h>
#include <mass_simd.h>
//

void rand_normal_sp(vector float *z)
{
    vector float u1,u2,v1,v2,w1,p1;
    vector float c1 = ((vector float) { 0.5f,0.5f,0.5f,0.5f});
    vector float c2 = ((vector float) { 0.2328306e-9f, 0.2328306e-9f,
                                        0.2328306e-9f, 0.2328306e-9f});
    vector float c3 = ((vector float) { 6.28318530f, 6.28318530f,
                                          6.28318530f, 6.28318530f});
    vector float c4 = ((vector float) { -2.0f,-2.0f,-2.0f,-2.0f});
    vector unsigned int y1, y2;


// get y1, y2 from random number generator.

//convert to uniform random numbers
```

```
        v1 = spu_convtf( (vector signed int) y1,  0) ;
        v2 = spu_convtf( (vector signed int) y2,  0) ;
        u1 = spu_madd( v1,  c2, c1);
        u2 = spu_madd( v2,  c2, c1);

// Box-Muller transformation
        w1 = sqrtf4( spu_mul(c4,logf4(u1)) );
        p1 = spu_mul(c3, u2);
      z[0] = spu_mul(w1, cosf4(p1) );
      z[1] = spu_mul(w1, sinf4(p1) );
}
```

Example 8-13 Sample Make file to use MASS SIMD library

```
CELL_TOP = /opt/cell/sdk/
SDKLIB = /opt/cell/sdk/prototype/sysroot/usr/lib

# Choose xlc over gcc because it gives slightly better performance
SPU_COMPILER = xlc
#

PROGRAMS_spu    := mceuro_spu

INCLUDE = -I/usr/spu/include

OBJS  = mceuro_spu.o

LIBRARY_embed   := mceuro_spu.a

# use default optimization because higher levels do not improve
CC_OPT_LEVEL          := -O3

IMPORTS = /usr/spu/lib/libmass_simd.a

##########################################################################
#
#                       make.footer
##########################################################################
#

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

```
        include ../../../../../make.footer
endif
```

It is straightforward to get a double precision version of the above example from Example 8-9 on page 507.

Recall that the number of Monte Carlo cycles for European Option pricing is typically very large -- millions or hundreds of thousands. In such cases, the call overhead for math functions can degrade the performance. In order to avoid call overhead and improve the performance, one can use the MASS Vector library[12] to provide the math functions to calculate the results for an array of input values with a single call. To link to MASS vector library, we need to replace *libmass_simd.a* by *libmassv.a* in *IMPORTS* in the Makefile, Example 8-13 on page 510. The following example shows how to restructure the code, Example 8-12 on page 509, to use MASS vector functions.

Example 8-14 Sample code to use MASS vector functions

```
#include <spu_mfcio.h>
#include <massv.h>

//define buffer length
#define BL  32
#define BL2  64

void rand_normal_sp(vector float *z)
{
    vector float c1 = ((vector float) { 0.5f,0.5f,0.5f,0.5f});
    vector float c2 = ((vector float) { 0.2328306e-9f, 0.2328306e-9f,
                                        0.2328306e-9f, 0.2328306e-9f});
    vector float c3 = ((vector float) { 6.28318530f, 6.28318530f,
                                        6.28318530f, 6.28318530f});
    vector float c4 = ((vector float) { -2.0f,-2.0f,-2.0f,-2.0f});
    vector unsigned int y[BL2]  __attribute__ ((aligned (16)));
    vector float u1[BL]  __attribute__ ((aligned (16)));
    vector float u2[BL]  __attribute__ ((aligned (16)));
    vector float w[BL]  __attribute__ ((aligned (16)));
    vector float v1,v2;
    int i, size=4*BL;

// get random numbers in the array y
//convert to uniform random numbers
```

[12] See reference [22] in “Other publications” on page 619.

```
      for (i=0, j=0; i < BL; i++, j+=2)
      {
        v1 = spu_convtf( (vector signed int) y[j],  0) ;
        v2 = spu_convtf( (vector signed int) y[j+1],  0) ;
        u1[i] = spu_madd( v1,  c2, c1);
        u2[i] = spu_madd( v2,  c2, c1);

      }

//MASS
      vslog ((float *) u1, (float *) u1, &size);



      for (i=0; i < BL; i++)
      {
        u1[i] = spu_mul(c4,u1[i]);
        u2[i] = spu_mul(c3,u2[i]);
      }

       vssqrt ( (float *)u1,  (float *)u1, &size);
       vssincos ( (float *)w, (float *)u2, (float *)u2, &size);

      for (i=0, j=0; i < BL; i++, j+=2)
      {
        z[j]   = spu_mul(u1[i], u2[i] );
        z[j+1] = spu_mul(u1[i], w[i] );
      }
```

Note that in the above example, pointers to vector floats such as u1 and u2 are cast to pointers of floats in the MASS vector function calls since the array arguments in the MASS vector functions are defined as pointers to floats; for details, see the prototypes for the SPU MASS vector functions in `/usr/spu/include/massv.h`.

8.5 Improving the performance

In this section, we discuss ideas to improve the performance and provide performance data[13]. With our European option pricing code implementation, we

found that xlc gives better performance than gcc. Further, besides using MASS, we used the following techniques to improve the performance.

- Unrolled the outer loop in Example 8-7 to improve register utilization and instruction scheduling. Note that, since a vector version of Box-Muller method as well as Polar method computes two vectors of standard normal random numbers out of two vectors of unsigned integer random numbers, all generated vectors of normal random numbers can be used by unrolling the outer loop, for example, to a depth of eight or four.
- Inlined some of the routines to avoid call overhead.
- Avoided branching (*if* and *else* statements) as much as possible within a loop.
- Reduced the number of global variables to help compiler with register optimization.

Moreover, we used -O3 compiler optimization flag with xlc; also, we tried higher level optimization flags such as -O5, which didn't make a significant performance difference, compared to -O3.

In Figure 8-2 on page 514, we provide the performance results in terms of millions of simulations per second (M/sec) for the tuned single precision code for European option on QS21(clock speed 3.0 GHz); the input values are given in Example 8-15 on page 513.

Example 8-15 Input values for Monte Carlo simulation

```
num_simulations = 200000000
     spot_price = 100.0
   strike_price = 50.0
  interest_rate = 0.10
     volatility = 0.40;
time_to_maturity = 1
  num_time_steps = 1
```

[13] Note: some general guidelines for improving performance of an application on SPUs are provided buy reference [14] in "Other publications" on page 619.

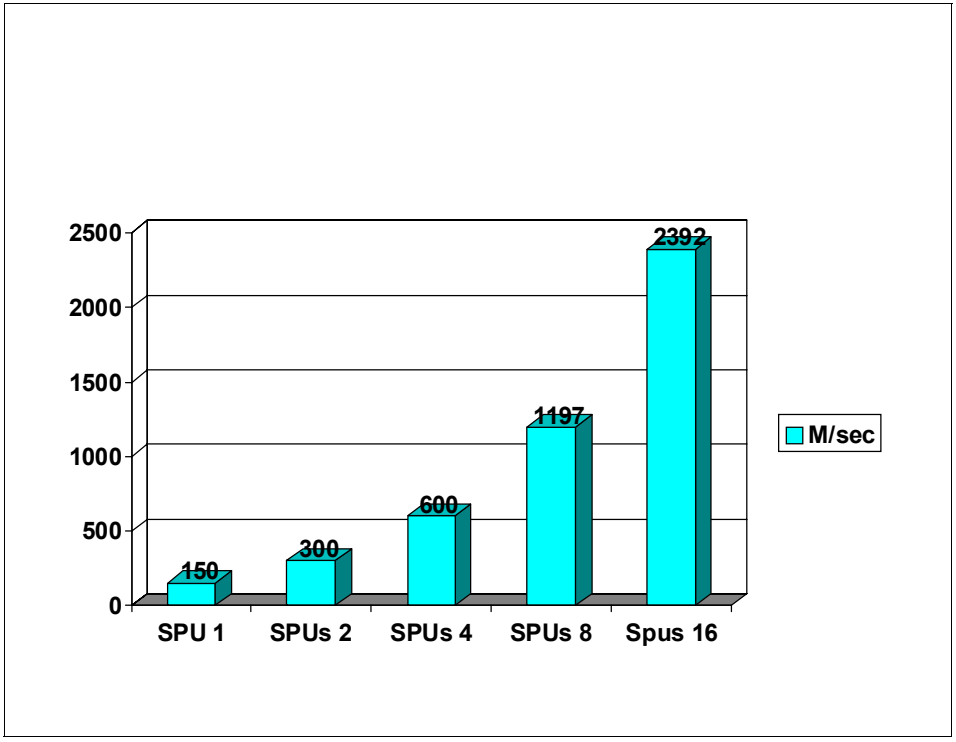


Figure 8-2 Performance of Monte Carlo simulation on QS21.

9

Case study: Implementing an FFT algorithm

This chapter describes the code development process and stages for an FFT library that is included as a prototype project in the Cell BE SDK 3. FFT algorithms are key in many problem domains including the siesmic industry.

The focus of this chapter is to provide a real world example of the code development process for the Cell BE processor running on an IBM QS21 Blade. The snippets of source code found in this chapter are included for illustration and do not consititue a complete FFT solution.

The following topics are presented in the following sections of this chapter:

- A description of the FFT algorithm implemented on Cell BE
- A description of the development process
- Evolution of select SPU code fragments during development
- Various performance improvement techniques

9.1 Motivation for an FFT algorithm

FFTs (Fast Fourier Transforms) are used in many applications that process raw data looking for a signal. There are many FFT algorithms ranging from relatively simple powers-of-two algorithms to powerful, but CPU intensive algorithms capable of working on arbitrary inputs. FFT algorithms are well suited to the Cell BE processor because they are floating point intensive and exhibit regular data access patterns.

The IBM Cell BE SDK 3.0 contains a prototype FFT library written to explicitly exploit the features of the Cell BE processor. This library uses several different implementations of an algorithm to solve a small class of FFT problems. The algorithm is based on a modified Cooley-Tukey type algorithm. All of the implementations use the same basic algorithm, but each implementation does something different to make the algorithm perform the best for a particular range of problem sizes.

The first step in any development process is to start with a good algorithm that maps well to the underlying architecture of the machine. The best compilers and hardware can not hide the deficiencies imposed by a poor algorithm - it is necessary to start with a good algorithm. The following are some of the considerations that went into selecting an algorithm:

- Requirement: support for problem sizes that can be factored into powers of 2, 3, and 5. This eliminated straight 'powers of two' or PFA (Prime Factor Algorithm) algorithms.
- A single problem should fit within the memory of an SPU. This kept the code simpler by eliminating the need for two or more SPUs to coordinate and work on a single problem.

9.2 Development Process

There is no formal set of rules or process for for developing or porting an existing application to the Cell BE processor. The program team that wrote this FFT library developed an interative development process mapped on top of a pre-defined set of logical stages. The interative development process was useful during some or all of the development statges.

Figure 9-1 on page 517 is a pictorial representation of the stages and process used during implementation of the FFT code for Cell BE processor. The following sections provide a description of this figure in more detail.

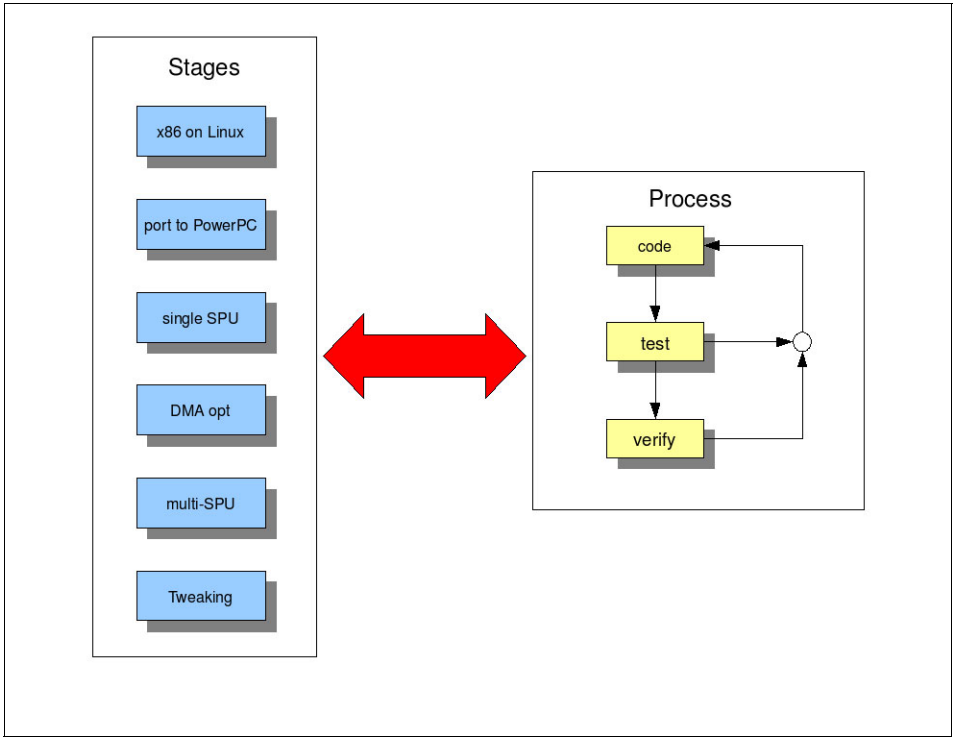


Figure 9-1 Development states and process

9.2.1 Code

The box labeled “code” represents the actual physical writing of code. This may include the implementation of a formal program specification or coding done without any documentation other than code directly from the programmers brain.

Experienced programers may require fewer code interations while less experienced programmers normally take more iterations. There is no concept of duration other than it is not unusal to break code sessions into logical functional elements.

Coding can be performed in whatever is convienent to the programmers. The IBM Eclipse IDE for Cell BE found in the SDK is a good choice for writing and debugging Cell BE code. Some of the team that implemented the FFT library used in IBM Eclipse IDE for Cell BE while others on the team make use of classic Linux based editors and debuggers or the C programmers friend, printf.

9.2.2 Test

The “test” box inside the Process box in Figure 9-1 on page 517 represents the testing of code that has been compiled. For Cell BE applications, testing is still a necessary and critical step in producing well performing applications.

The test process for Cell BE application will undoubtably require testing for code performance. Testing is normally focused on code that runs on the Cell BE SPU and the focus is more on making sure all code paths are processed and the accuracy of the output.

9.2.3 Verify

The “verify” box in the process box in Figure 9-1 on page 517 represents an important aspect of many Cell BE applications. The need for verification of code that runs on the SPU is of special importance. The Cell BE SPE single-precision floating point is not the same implementation as found in the PPU and other processors.

Code being ported from other processor platforms present a unique oppurtunity. The verification of Cell BE programs by comparing output from the original application for accuracy. A binary comparison of single-precision floating point be performed to eliminate conversions to and from textual format.

The FFT coding team chose to write a separate verification program to verify output. The verification code made use of a well known open source FFT library which is supported on PowerPC processors. The results from the Cell BE FFT code was compared at a binary level with the results from the open source FFT results. The example code in Example 9-2 on page 519 shows how single-precision floating point was converted to displayable hexidecimal and then converted back to single-precsion floating point. These functions are one way to compare SPE and PPE floats.

Example 9-1 on page 518 shows a simple C function from the test program which illustrates how two single-precsion floating point values representing an complex number is output in displayable hexidecimal form.

Example 9-1 Single-precision floats as displayable hexidecimal

```
typedef union {
  unsigned int i;
  float        f;
} Conv_t;

void printOutput( float f1, float f2 ) {
```

```
  if ( Output ) {
    Conv_t t1, t2;
    t1.f = f1;
    t2.f = f2;
    printf( "%08x %08x\n", t1.i, t2.i );
  }
  else {
    printf( "%f %f\n", f1, f2 );
  }
}
```

Figure 9-2 is a code snippet from the verification program which reads displayable hexadecimal from stdin and converts it into a complex datatype. The verify program reads multiple FFT results from stdin which explains why variable p is a two dimensional matrix.

Example 9-2 Reading Complex numbers from displayable hexidecimal format

```
typedef struct {
   foat real;
   floatimag;
} Complex;

Complex *t = team[i].srcAddr;
MT_FFTW_Complex *p = fftw[i].srcAddrFC;
unsigned int j;
for ( j=0; j < n; j++ ) {
      volatile union {
        float f;
        unsigned int i;
      } t1, t2;
      scanf( "%x %x\n", &t1.i, &t2.i );
#define INPUT_FMT "i=%d j=%d r=%+13.10f/%8.8X i=%+13.10f/%8.8X\n"
         (verbose ? fprintf(stdout, INPUT_FMT, i, j, t1.f, t1.i, t2.f,
t2.i) : 0);
      t[j].real = p[j][0] = (double)t1.f;
      t[j].imag = p[j][1] = (double)t2.f;
    } // for j  (number of elements in each fft)
```

The code shown in Example 9-2 on page 519 will accurately reconstitute a single-precision floating point value this is only true for data generated by the same mathematical, floating-point representations (i.e. PowerPC to PowerPC). The FFT library verification program compares single-precision floating point

values between PowerPC and SPE single-precision format. The format of the two floating point representations are similar and can normally be ignored for verificaiton purposes.

9.3 Development Stages

The development stages list in Figure 9-2 were arrived at prior to the beginning of the code development and later revised. The purpose of introducing these stages was to provide a way to monitor progress of the project and seemed like a practical way to develop the solution.

The following sections describe in more detail the different stages and include sample code to illustrate the evolution of some aspects of the code that are unique to the Cell BE processor.

9.3.1 x86 implementation

The goal or deliverable from this stage was a functional FFT implementation that ran on x86 Linux hardware. Very little effort was invested in producing code that performed optimally on x86; the goal was to prove that the basic algorithm produced correct results and would meet our requirements for running in an SPU.

This version of the FFT code included an initial functional interface from a test program to the actual FFT code and a primative method for displaying the results. The verification of FFT output was essential to ensure accuracy.

9.3.2 Port to PowerPC

The x86 FFT implementation was ported to PowerPC hardware by recompiling the x86 source code on an IBM QS20 Blade. The effort to do this was almost trivial as can be expected.

The PowerPC version of the code performed much slower than the x86 version of the code. This is due to the difference in the relative power of the PPU portion of the Cell BE compared to a fairly high end dual core x86 CPU on which the x86 code was developed. Even though this code was simple and single threaded, the x86 processor core used for development is a much more power processor core than the PPU. The good news for Cell BE is that the power of the chip lies in the eight SPUs, which we would be taking advantage of shortly. (The PPU would eventually be left to just managing the flow of work to the SPUs.)

9.3.3 Single SPU

The first real specific Cell BE coding task was to take the PowerPC (PPU) code and run the compute kernel on a single SPU. This involved restructuring the code by adding calls to the SDK to load an SPU module. More code was added to implement a simple DMA model for streaming data into the SPU and streaming it back to main store memory.

Example 9-3 shows a code snippet demonstrating how a single SPU thread is started from the main FFT PPU code. The use of the spe_context_run() is a synchronous API and will block until the SPU program finishes execution. The multiple-SPU version of this code using pThread support is the prefered solution and is discussed in 9.3.5, “Using multiple SPUs” on page 523.

Example 9-3 Runing a single SPU

```
#include <stdio.h>
#include <stdlib.h>
#include <libspe2.h>

   spe_context_ptr_t ctx;
   unsigned int entry = SPE_DEFAULT_ENTRY;

   /* Create context */
   if ((ctx = spe_context_create (0, NULL)) == NULL) {
      perror ("Failed creating context");
      exit (1);
   }

   /* Load program into context */
   if (spe_program_load (ctx, &dma_spu)) {
      perror ("Failed loading program");
      exit (1);
   }

   /* Run context */
   if (spe_context_run(ctx,&entry,0,buffer,(void *)128,NULL)< 0) {
      perror ("Failed running context");
      exit (1);
   }
   /* Destroy context */
   if (spe_context_destroy (ctx) != 0) {
      perror("Failed destroying context");
```

```
    exit (1);
}
```

The program above could as easily be compiled and run as a spulet.

9.3.4 DMA Optimization

The initial versions of the code were CPU intensive and slow because they did not use the SIMD features of the SPUs. The code was not well structured or optimized. As a result, the time spent doing DMA transfers was very small relative to the time spent computing results. In this stage of the project DMA optimization was not considered to be important. Early measurements showed that without double buffering, DMA transfer time was only about 2 percent of the total time it took to solve a problem.

The time spent in the computation phase of the problem shrank as the code became more optimized using various techniques described later in this section. As a result, DMA transfer time was growing larger relative to the time spent performing the computation phase. Eventually the team turned their attention from focusing on the computation phase to the growing DMA transfer time.

The programming team knew double buffering could be used to hide the time it takes to do DMA transfers if done correctly. At this phase of the project, it was determined that DMA transfers were about 10 percent of the total time it took to solve a problem. Correctly implemented DMA transfers were added to the code at this phase to overlap with current on-going computations. Given the optimization effort had been put into the base algorithm, being able to reduce our run time per problem by 10 percent was worth the effort in the end.

However, double buffering does not come for free. In this case, the double buffering algorithm required three buffer areas, one buffer for incoming data and outgoing DMAs, and two for the current computation. The use of a single buffer for both input and output DMAs was possible by observing that the data transfer time was significantly smaller than compute time. It would be possible to allocate the third buffer area when working on a small problem, but for the largest problems memory was already constrained and another buffer area would not be possible.

The solution to the problem was to have several implementations of the algorithm, where each specific implementation would use a different data transfer strategy:

- For large problems where an extra data buffer was not possible, double buffering would not be attempted.
- For smaller problems where an extra data buffer was possible, one extra data buffer would be allocated and used for background data transfers, both to and from SPU memory.

The library on the PPU that dispatched the problems to the SPUs now would have to look at the problem size provided by the user and choose between two different implementations. The appropriate implementation would then be loaded on an SPU, and problems suitable for that implemention would be sent to that SPU.

With this solution, the code was flexible enought to have it both ways - small problems were able to take advantage of double buffering, while large problems were still possible to execute.

9.3.5 Using multiple SPUs

The original project requirements specified that no single FFT problem would be larger than 10,000 points. This allowed for implementing a serial FFT algorithm where a single SPU could solve the largest sized FFT problem instead of a more complicated parallel algorithm using multiple SPUs in concert. (The SDK provides an example of a large parallel FFT algorithm in the sample codes.)

As a result, extending the code to run on multiple SPUs allowed for increasing the throughput, not the maximum problem size. All of the SPUs that get used will work independently, getting their work from a master work queue maintained in PPU memory.

9.4 Strategies for using SIMD

The SPUs are designed from the ground up as vector units. If a coding style does not make use of these vector registers, 75 percent of the potential performance is lost.

FFT implementations based on 'powers-of-two' algorithms are relatively easy to implement on a vector processor because of the regular data access patterns and boundary alignments of the data. A non 'powers-of-two' algorithm is significantly more difficult to map to a vector architecture because of the data access patterns and boundary issues.

The FFT prototype library chose two different vectorization strategies:

- Striping multiple problems across a vector
- Synthesizing vectors by loop unrolling

9.4.1 Striping multiple problems across a vector

A relatively easy way to use the vector registers of the SPU is to treat each portion of the register independently from other portions of the register. For example, given four different floating point values from four different problems, all of the floating point values can be resident in a single register at a time. This allows for computation on four different problems in a register at the same time. The code to do this a simple extension of the original scalar code, except now instead of loading one value from one problem into a register, the four values are loaded from four problems into a single vector register. One way to visualize this technique is to consider it as ‘striping’ multiple problems across a single register.

If there are four FFT problems in memory and those problems are of the same length then this is trivial to implement. All four problems will be done in lockstep, and the code which performs the indexing, the twiddle calculations, and other code can be reused for all four problems.

The problem with this approach is that it requires multiple FFT problems be resident in SPU memory. For the very smallest problem sizes (up to around 2500 points) this is feasible, but for larger problem sizes this technique could not be leveraged because the memory requirements were too great for a single SPU.

9.4.2 Synthesizing vectors by loop unrolling

For the larger problems in the FFT prototype, ‘striping’ multiple problems across a register was not possible because there was not enough space in SPU

memory to have multiple problems resident. But it was still necessary to make use of the vector features of the SPU for performance reasons.

The solution to this problem was to:

- Unroll the inner computation loop by a factor of four to give four sets of scalar values to work with.
- Use shuffle operations to synthesize vector registers from the scalar values.
- Perform the math using the vector registers.
- Split the results in the vector registers apart and put them back into their proper places in memory.

The advantage of this solution is that it allowed the use of the vector capabilities of the SPU with only one FFT problem resident in memory. Parallelism was achieved by performing loop unrolling on portions of the code.

A disadvantage of this solution was the need to constantly assemble and disassemble vector registers. Although there was a large net gain, a 4x speedup was not realized by using the vector registers. This is because the time spent assembling and disassembling the vector registers was fairly large compared to the time spent computing with those vector registers. (On a more intensive calculation this technique would have shown a greater improvement.)

9.4.3 Measuring and tweaking performance

The primary tools for measuring performance of the FFT code were the simulator, the spu-timing tool, and a stopwatch. (An actual stopwatch was not used, but a simple stopwatch was implemented in the code). Since the time the FFT was developed additional performance tools have become available, some of which are explained in this book.

The spu-timing tool allowed for analyzing the code generated by the compiler to look for inefficient sections of code. Understanding why the code was inefficient provided insight into where to focus efforts on generating better perfoming code. The simulator gave insight as to how the code would behave on real hardware. The dynamic profiling capability of the simulator let us verify our assumptions about branching behavior, DMA transfer time, and overall SPU utilization. And finally the stopwatch technique provided verification that the changes to the code were actually faster than the code that was replaced.

Below are some of the performance lessons that were learned during this project:

Use the SIMD Math Library

An FFT algorithm is a very intensive user of the sinf() and cosf() function in the math library. These are expensive functions to call so minimizing their use is critical.

An important feature of the SIMD Math library are special vector versions of sinf() and cosf() that will take four angles in a vector as input and return four sine or cosine values in a vector as output. These functions were used to dramatically reduce the amount of time spent computing sines and cosines. The source code for the functions is available in the SDK which allows you do adapt them for your specific needs. A merged sinf() and cosf() function was coded that generates eight outputs (four sines and four cosines) from a single vector of input angles. (This merged function was even faster than calling the two vector functions separately.) There are also inline versions of the functions available, which are used to improve branching performance.

Use code inlining and branch hints to improve performance

In this code two methods were used to improve performance:

Code inlining

Code inlining is a great way to improve branching performance, because the fastest branch is the one that you didn't take. During the development cycle code was often written in functions or procedures to keep the code modular. During the performance tuning stage these functions and procedure calls were reviewed for potential candidates that could be inlined. Besides reducing branching, inlining has a side benefit of giving the compiler more code to work with, allowing it to possibly make better use of the SPU pipelines.

A certain amount of judgement has to be used when inlining code - if the code is called from many locations the compiled size of the code may grow to an unacceptable level. It is also possible to have so much code in a basic block that the compiler starts to spill registers to the stack, which is leads to degraded performance.

Branch hint directives

Most of the branching in the FFT code was caused by loop structures. The compilers will generate the correct branch hints for loop structures, so we did not worry about them. In the rare place where ‘if-then-else' logic was needed an analysis of the code was performed to determine which path was the most used, and a branch hint was inserted by hand. This improved the performance of the runtime by about one percent which was well worth the effort. Code with lots of 'if-then-else' logic where the most taken path can be predicted would probably benefit more from branch hints than this code did.

Using the Shuffle intrinsic effectively

In section 9.4.1, “Striping multiple problems across a vector” on page 524 a technique was discussed where four FFT problems in memory were 'striped' across vector registers. This allowed all four problems to be computed in lockstep using code that was basically an extension of the original scalar code.

When the FFT problems first come into memory they are in an interleaved format where the real portion of a complex number is followed immediately by the imaginary portion of the complex number. Loading a vector from memory would result in two consecutive complex numbers from the same problem in a vector register. That would then require shuffle operations to split the reals from the imaginaries, and more shuffle operations to end up with four problems striped across the vector registers.

It is most efficient to do all of the data re-arrangement after the problems are in SPU memory, but before computation begins. The example code in Example 9-4 is a simple data structure used to represent the problems in memory and a loop that does the re-arrangement:

Example 9-4 Rearranging complex numbers

```
typedef union {
  Complex_t prob[4][MAX_PROB_SIZE_C2C_4];
  union {
    struct {
      float real[MAX_PROB_SIZE_C2C_4*4];
      float imag[MAX_PROB_SIZE_C2C_4*4];
    } sep;
    struct {
      vector float real[MAX_PROB_SIZE_C2C_4];
      vector float imag[MAX_PROB_SIZE_C2C_4];
    } vec;
  } u;
} Workarea_t __attribute__ ((aligned (128)));

Workarea_t wa[2];
// Separate into arrays of floats and arrays of reals
// Do it naively, one float at at time for now.
short int i;
for ( i=0; i < worklist.problemSize; i++ ) {
  wa[0].u.sep.real[i*4+0] = wa[1].prob[0][i].real;
  wa[0].u.sep.real[i*4+1] = wa[1].prob[1][i].real;
  wa[0].u.sep.real[i*4+2] = wa[1].prob[2][i].real;
  wa[0].u.sep.real[i*4+3] = wa[1].prob[3][i].real;
  wa[0].u.sep.imag[i*4+0] = wa[1].prob[0][i].imag;
  wa[0].u.sep.imag[i*4+1] = wa[1].prob[1][i].imag;
```

```
  wa[0].u.sep.imag[i*4+2] = wa[1].prob[2][i].imag;
  wa[0].u.sep.imag[i*4+3] = wa[1].prob[3][i].imag;
}
```

The data structure provides access to memory in three ways:

- Four problems where the reals and imaginaries are interleaved
- One array of floats representing all of the reals, and one array of floats representing all of the imaginaries
- One vector array of floats representing all of the reals, and one vector array of floats representing all of the imaginaries. (Note that these arrays have one fourth as many elements as the previous arrays because they are vector arrays.)

On entry to the code wa[1] contains four problems in interleaved format. This code copies all of the interleaved real and imaginary values from wa[1] to wa[0], where they appear as separate arrays of reals and imaginaries. At the end of the code the view in wa[0] is of four problems stripped across vectors in memory.

While this code works, it is hardly optimal:

- Only four bytes is moved at a time.
- It is scalar code so the compiler must shuffle the floats into the preferred slots of the vector registers, even though the values are going to be written back to memory immediately.
- When writing each value to memory, the compiler must generate code to load a vector, merge the new data into the vector, and store the vector back into memory.

A review of the assembly code generated by the compiler and annotated by the spu_timing tool (see Example 9-5) shows that the code is very inefficient for this architecture:

Example 9-5 spu_timing output for suboptimal code

```
                                                    .L211:
002992 1                                         2          hbrp    # 2
002993 0                                          3456      shli    $17,$25,3
002994 0                                           4567     shli    $79,$25,4
002995 0                                            5678    shli    $35,$25,2
002996 0D                                            67     il      $61,4
002996 1D                                            6      lnop
002997 0D                                             78    a       $10,$17,$82
002997 1D 012                                         789   lqx     $39,$17,$82
002998 0D                                              89   a       $78,$17,$30
002998 1D 0123                                         89   lqx     $36,$79,$80
002999 0D 0                                             9   ai      $6,$35,1
```

```
002999 1D 012                                                  9
003000 0D 01
003000 1D 0123
003001 0   1234
003002 0    2345
003003 0D    34
003003 1D    3
003004 1      4567
003005 0       56
003006 0D       67
003006 1D       6789
003007 0D        7890
003007 1D        7890
003008 0D         89
003008 1D         8901
003009 0D          90
003009 1D          9012
003010 0D           01
003010 1D           0123
003011 0D            12
003011 1D            1234
003012 0D             23
003012 1D             234567
003013 0D              34
003013 1D              345678
003014 1                456789
003015 1                 5678
003019 1                  ---9012
003023 1                      ---3456
003024 0d                         45
003027 1d                         ---789012
003028 1                              890123
003029 1                               901234
003034 1                                ----4567
003038 1                                     ---8901
003042 1                                         ---234567
003043 1                                             345678
003044 1                                              456789
003049 1  012                                          ----9
003053 1  ---3456
003057 1     ---789012
003058 1         890123
003059 1          901234
003064 1           ----4567
003068 1                ---8901
003072 1                    ---234567
003073 1                        345678
003074 1                         456789
003079 1                          ----9012
003083 1                               ---3456
003087 1                                   ---789012
003088 1                                       890123
003089 1                                        901234
003094 1                                         ----4567
003098 1  01                                          ---89
003102 1  --234567                                       -
003103 1    345678
003104 1     456789
003109 1      ----9012
003113 1           ---3456
003117 1               ---789012

003118 1                   8901
```

```
        cwx     $38,$79,$80
        ai      $7,$35,2
        cwx     $8,$79,$24
        shli    $71,$6,2
        shli    $63,$7,2
        a       $70,$17,$29
        hbrp    # 2
        rotqby  $37,$39,$10
        ai      $5,$35,3
        a       $62,$17,$28
        cwx     $32,$71,$80
        shli    $54,$5,2
        cwx     $21,$63,$80
        ai      $11,$10,4
        shufb   $34,$37,$36,$38
        ai      $77,$78,4
        cwx     $75,$71,$24
        ai      $69,$70,4
        cwx     $67,$63,$24
        ai      $60,$62,4
        cwx     $14,$54,$80
        ai      $27,$27,1
        stqx    $34,$79,$80
        ai      $26,$26,-1
        lqx     $33,$17,$30
        lqx     $25,$71,$80
        cwx     $58,$54,$24
        rotqby  $31,$33,$78
        shufb   $23,$31,$25,$32
        ori     $25,$27,0
        stqx    $23,$71,$80
        lqx     $22,$17,$29
        lqx     $19,$63,$80
        rotqby  $20,$22,$70
        shufb   $18,$20,$19,$21
        stqx    $18,$63,$80
        lqx     $16,$17,$28
        lqx     $13,$54,$80
        rotqby  $15,$16,$62
        shufb   $12,$15,$13,$14
        stqx    $12,$54,$80
        lqx     $9,$10,$61
        lqx     $4,$79,$24
        rotqby  $2,$9,$11
        shufb   $3,$2,$4,$8
        stqx    $3,$79,$24
        lqx     $76,$78,$61
        lqx     $73,$71,$24
        rotqby  $74,$76,$77
        shufb   $72,$74,$73,$75
        stqx    $72,$71,$24
        lqx     $68,$70,$61
        lqx     $65,$63,$24
        rotqby  $66,$68,$69
        shufb   $64,$66,$65,$67
        stqx    $64,$63,$24
        lqx     $59,$62,$61
        lqx     $56,$54,$24
        rotqby  $57,$59,$60
        shufb   $55,$57,$56,$58
        stqx    $55,$54,$24
.L252:
        brnz    $26,.L211
```

Notice the large number of stall cycles. The loop body takes 126 cycles to execute.

The code in Example 9-6, while less intuitive, shows how the shuffle intrinsic can be used to dramatically speed up the rearranging of complex data:

Example 9-6 Better performing complex number rearrangement code

```
// Shuffle patterns
    vector unsigned char firstFloat  = (vector unsigned char){  0,  1,  2,  3, 16, 17, 18, 19,
0,  0,  0,  0,  0,  0,  0,  0 };
    vector unsigned char secondFloat = (vector unsigned char){  4,  5,  6,  7, 20, 21, 22, 23,
0,  0,  0,  0,  0,  0,  0,  0 };
    vector unsigned char thirdFloat  = (vector unsigned char){  8,  9, 10, 11, 24, 25, 26, 27,
0,  0,  0,  0,  0,  0,  0,  0 };
    vector unsigned char fourthFloat = (vector unsigned char){ 12, 13, 14, 15, 28, 29, 30, 31,
0,  0,  0,  0,  0,  0,  0,  0 };

    vector unsigned char firstDword  = (vector unsigned char){  0,  1,  2,  3,  4,  5,  6,  7,
16, 17, 18, 19, 20, 21, 22, 23 };

   vector float *base0 = (vector float *)(&wa[1].prob[0]);
   vector float *base1 = (vector float *)(&wa[1].prob[1]);
   vector float *base2 = (vector float *)(&wa[1].prob[2]);
   vector float *base3 = (vector float *)(&wa[1].prob[3]);
   short int i;
   for ( i=0; i < worklist.problemSize; i=i+2 ) {

      // First, read a quadword from each problem
      vector float q0 = *base0;
      vector float q1 = *base1;
      vector float q2 = *base2;
      vector float q3 = *base3;

      vector float r0 = spu_shuffle(
                        spu_shuffle( q0, q1, firstFloat ),
                        spu_shuffle( q2, q3, firstFloat ),
                        firstDword );

      vector float i0 = spu_shuffle(
                        spu_shuffle( q0, q1, secondFloat ),
                        spu_shuffle( q2, q3, secondFloat ),
                        firstDword );

      vector float r1 = spu_shuffle(
                        spu_shuffle( q0, q1, thirdFloat ),
```

```
                                spu_shuffle( q2, q3, thirdFloat ),
                              firstDword );

    vector float i1 = spu_shuffle(
                        spu_shuffle( q0, q1, fourthFloat ),
                        spu_shuffle( q2, q3, fourthFloat ),
                        firstDword );
     wa[0].u.vec.real[i]   = r0;
     a[0].u.vec.real[i+1] = r1;
     wa[0].u.vec.imag[i]   = i0;
     wa[0].u.vec.imag[i+1] = i1;
     base0++;
     base1++;
    base2++;
    base3++;
  }
```

This code executes much better on the SPU for the following reasons:

- All loads and stores to main memory are done using full quadwords. This makes the best use of memory bandwidth.
- The compiler is working exclusivley with vectors, avoiding the need to move values into 'preferred slots'

The code annotated with the spu_timing tool is shown in Example 9-7.

Example 9-7 spu_timing output for more optimal code

```
                                                                .L211:
002992 0D                                  23                   ai      $74,$16,1
002992 1D                                  234567               lqd     $72,0($15)
002993 0D                                   3456                shli    $59,$16,4
002993 1D                                   345678              lqx     $73,$15,$27
002994 0D                                    4567               shli    $56,$74,4
002994 1D                                    456789             lqx     $70,$15,$26
002995 0D                                     56                cgt     $55,$20,$22
002995 1D 0                                   56789             lqx     $71,$15,$25
002996 0                                       67               ori     $16,$22,0
002997 0                                        78              ai      $15,$15,16
002998 0d                                       89              ai      $22,$22,2
002999 1d 012                                   -9              shufb   $68,$72,$73,$23
003000 1  0123                                                  shufb   $66,$72,$73,$21
003001 1   1234                                                 shufb   $69,$70,$71,$23
003002 1    2345                                                shufb   $67,$70,$71,$21
003003 1     3456                                               shufb   $64,$72,$73,$18
003004 1      4567                                              shufb   $65,$70,$71,$18
003005 1       5678                                             shufb   $62,$72,$73,$19
003006 1        6789                                            shufb   $63,$70,$71,$19
003007 1         7890                                           shufb   $61,$68,$69,$17
003008 1          8901                                          shufb   $60,$66,$67,$17
003009 1           9012                                         shufb   $58,$64,$65,$17
003010 1            0123                                        shufb   $57,$62,$63,$17
003011 1             123456                                     stqx    $61,$59,$80
```

```
003012 1              234567                                  stqx    $60,$59,$24
003013 1               345678                                 stqx    $58,$56,$24
003014 1                456789                                stqx    $57,$56,$80
                                                            .L252:
003015 1                 5678                                 brnz    $55,.L211
```

This loop body is much shorter at only 23 cycles. It also has virtually no stall cycles. The body of the second loop is approximately five times faster than the body of the first loop. However, examination of the code shows that the second loop is executed one half as many times as the first loop. With the savings between the reduced cycles for the loop body and the reduced number of iterations the second loop works out to be 10 times faster than the first loop. This result was verified with the simulator as well.

Part 4

Systems

In this part of the book we provide a chapter that covers detailed system installation, configuration, and management topics.

10

SDK 3.0 and Bladecenter QS21 System Configuration

In this chapter, we will discuss installing the IBM BladeCenter QS21, Installing SDK 3.0 on a QS21, Firmware considerations, Blade management considerations, and finally suggesting a method for installing a distribution so that it utilizes a minimal amount of the QS21 Bladecenter's resources.

The sections covered in this book are as follows:

- IBM BladeCenter QS21 characteristics
- Installing the operating system
- Installing SDK 3.0 on a QS21 Bladecenter
- Firmware considerations
- Options for managing multiple blades
- Method for installing a minimized distribution

The SDK3.0 Installation Guide, available through IBM's Alphaworks website, covers in great detail the necessary steps to install the operating system on a QS21 blade as well as the additional required steps for setting up a diskless system. This chapter places such detail into consideration, and addresses topics complementary to that guide.

10.1 BladeCenter QS21 Characteristics

The Bladecenter QS21 has two 64-bit Cell Broadband Engine (Cell BE) processors directly mounted onto the blade planar board in order to provide multiprocessing capability. The memory on a BladeCenter QS21 consists of 18 XDR memory modules per Cell BE chip, which creates 1GB of memory per Cell BE chip.

The Bladecenter QS21 is a single-wide blade which uses one Bladecenter H slot and can coexist with any other Blade in the same chassis. To ensure compatibility with existing Blades, the BladeCenter QS21 provides two midplane connectors that contain Gigabit Ethernet links, USB ports, power and a unit management bus. The local service processor supports environmental monitoring, front panel, chip initialization, and the BladeCenter unit Advanced Management Module Interface.

The blade includes support for an optional Infiniband expansion card and an optional Serial Attached SCSI (SAS) card.

Additionally, the BladeCenter QS21 has the following major components:

- 2 Cell BE processor chips (Cell BE-0 and Cell BE-1) operating at 3.2 GHz
- 2 GB XDR system memory with ECC, 1 GB per Cell BE chip v 2 Cell BE companion chips, one per Cell BE chip
- 2x8 PCIe as High Speed Daughter Cards (HSDC)
- 1*PCI-X as Daughter Card
- Interface to optional DDR2 memory, for use as the I/O Buffer
- Onboard Dual Channel Gb-Ethernet controller BCM5704S
- Onboard USB controller NEC uPD720101
- 1 BladeCenter PCI-X expansion card connector
- 1 BladeCenter High-Speed connector for 2 times x 8 PCIe buses
- 1 Special additional I/O expansion connector for 2 times x 16 PCIe buses
- 4 DIMM slots (2 slots per Cell BE companion chip) for optional I/O Buffer DDR2 VLP DIMMs
- Integrated Renesas 2166 Service processor (BMC supporting IPMI and SOL)

An important characteristic to point out about the BladeCenter QS21 is that it does not contain onboard hard disk or other storage. Storage on the BladeCenter QS21 can be allocated through network or a SAS attached device. The following section will cover installing an operating system, through network storage, on a QS21 Blade.

Note The Bladecenter QS21 support is only available with the Bladecenter H Type 8852 Unit.

10.2 Installing the Operating System

The BladeCenter QS21 does not contain local storage, storage can be provided through a SAS device or through another server in the same network. We will briefly cover the steps for accomplishing the installation of an operating system through network storage.

Section10.24, “Example for installing through network storage” on page 544, will show how to apply these steps through the use of bash scripts.

10.2.1 Important Considerations

Before proceeding further with the installation process, there are considerations to be made first.

- The BladeCenter QS21 is accessible only through SOL or serial interface. For serial interface, a specific UART breakout cable will be needed.
- A POWER-based system with storage is needed for initial installation of Linux, which is needed for a root filesystem.

  Note: If you do not have a POWER based system, you can execute your initial installation on a BladeCenter QS21 with USB storage attached.

- A DHCP server is needed to upload the kernel zImage to the BladeCenter QS21, this can only be done through ethernet modules on the BladeCenter H chassis.
- While the media tray on the BC-H chassis does work on the BladeCenter QS21, it is not a supported feature.
- For RHEL5.1, an individual kernel zImage will need to be created for each BladeCenter QS21.

Note: Through the use of cluster management tools such as DIM or xCAT, the process of creating inividual zImages can become automated. For DIM, it has the capabality of applying one RHEL5.1 kernel zimage to multiple BladeCenter QS21s.For more details, refer to “DIM implementation on BladeCenter QS21s” on page 565.

- For Fedora 7, the same kernel zImage can be applied across multiple blades and is accessible through the Barcelona supercomputing site.
- Each Blade must have it’s own separate root filesystem over the NFS server. This restriction applies even if the root filesystem is read only.

Note: While each BladeCenter QS21 must have it’s own root filesystem, some directories can be shared as read only amongst multiple BladeCenter QS21s.

- SWAP is not supported over NFS. Any root filesystem that will be NFS mounted cannot have SWAP space.
- SELinux cannot be enabled on nfsroot clients.
- SDK3.0 supports both Fedora7 and RHEL5.1, but it is officially supported only for RHEL5.1
- External internet access is needed for installing SDK3.0 open source packages.

Note: If your BladeCenter QS21 does not have external internet access, you can download the SDK3.0 open source components from the Barcelona Supercomputing website from another machine that does have externel internet access and apply them to the BladeCenter QS21.

10.2.2 Managing and accessing the Blade server

There are currently six options for managing and configuring the blade server, this includes being able to access the blade’s console.

Advanced Management Module through the web interface

The Advanced Management Module (AMM) is a management and configuration program for the BladeCenter system. Through it’s web interface, the AMM allows for configuring the Bladecenter unit, including components like the Bladecenter

QS21. Systems status and an event log is also accessible to monitor errors related to the chassis or it's connected blades.

For further information on the extent of the offerings provided by the Advanced Management Module, please refer to the *IBM BladeCenter Advanced Management Module User's Guide.*

Advanced Management Module through Command-line

In addition to the AMM being accessible through a web browser, it can also be directly accessed through command-line interface. Through this interface, you can issue commands to control the power and configuration of the blade server along with other components of the BladeCenter unit.

For further information and instructions on using the command line interface, please refer to the *IBM BladeCenter Management Module Command-Line Interface Reference Guide*.

Serial over LAN

The Serial over LAN (SOL) connection is one option for accessing the blade's console, accessing the blade's console allows for viewing firmware progress and accessing the Linux terminal. By default, the blade server sends output and receives over the SOL connection.

In order to establish an SOL connection, you must ensure to configure the SOL feature and start an SOL session. The Bladecenter and Advanced Management Module must also be configured properly in order to enable SOL connection.

For further information and details on establishing SOL connection, please refer to the *IBM BladeCenter Serial over LAN Setup Guide.*

Serial Interface

Another method aside from SOL for accessing the BladeCenter QS21 server's console is through a serial interface. This can be accomplished through the use of a specific UART cable connected to the BladeCenter H chassis. This particular cable is not included with included with the BladeCenter H, so it must be accessed separately.

Ensure that the following parameters for serial connection are set on the terminal client:

- 115200 baud
- 8 data bits
- No parity
- One Stop bit

- No flow control

Note that by default, input is provided to the blade server through SOL connection, thus, if you prefer input to be provided through a device connected to the serial port, ensure you press any key on that device while the server boots.

For further information and details on establishing serial interface connection, please refer to the *BladeCenter H Type 8852 Installation and User's Guide.*

SMS utility program

The System Management Services (SMS) utility program is another utility which can provide, in some cases, more information than what is accessible through the Advanced Management Module.

To access the SMS utility program, you must have input to the Blade's console (accessible either through serial interface or SOL) as it's booting up. Early into it's boot process, as shown below, make sure, to type "F1" as it boots up.

Example 10-1 Initial BladeCenter QS21 boot up

```
QS21 Firmware Starting
 Check ROM  = OK
 Build Date = Aug 15 2007 18:53:50
 FW Version = "QB-1.9.1-0"

Press "F1" to enter Boot Configuration (SMS)

Initializing memory configuration...
MEMORY
 Modules    = Elpida 512Mb, 3200 MHz
 XDRlibrary = v0.32, Bin A/C, RevB, DualDD
 Calibrate  = Done
 Test       = Done

SYSTEM INFORMATION
 Processor  = Cell/B.E.(TM) DD3.2 @ 3200 MHz
 I/O Bridge = Cell Companion chip DD2.x
 Timebase   = 26666 kHz (internal)
 SMP Size   = 2 (4 threads)
 Boot-Date  = 2007-10-26 23:52
 Memory     = 2048MB (CPU0: 1024MB, CPU1: 1024MB)
```

Additional configurations on the SMS utility that can't be implemented on the Advanced Management Module are the following:

– SAS configurations

- Choosing which firmware image to boot from, whether TEMP or PERM
- Choosing Static IP for network boot up, otherwise, the default method is strictly DHCP.

For more information on firmware, please refer to “Firmware considerations” on page 560.

10.2.3 Installing through Network Storage

Because the Bladecenter QS21 does not contain storage, and you cannot directly install Linux on a network device attached to the BladeCenter QS21, you must create an initial installation on a disk. From this initial installation, you can establish a network boot up that can be used by the BladeCenter QS21.

Due to Cell’s Power based architecture, the system to create the initial installation for storage must be on a 64-bit POWER based system. After this installation, you then copy the resulting root file system to a Network File System (NFS) server, make it network bootable so that it can be mounted via NFS, and adapt it to the specifics of an individual blade server.

Setting up the root filesystem

First we want to obtain a root filesystem that can be NFS mounted to the BladeCenter QS21.

Below are the steps for this NFS setup:

1. Install RHEL5.1 or Fedora 7 on a 64-bit POWER based system
2. Copy the root filesystem from the POWER-based installation to an NFS Server

> **Note:** You can create multiple copies of this filesystem if you want to apply the same distribution on multiple BladeCenter QSS1 systems. For more information on this, please refer to section 10.2.4 “Example for installing through network storage” on page 544.

3. Edit the copied root filesystem to reflect the specific blade you want to mount the filesystem to.

> **Note:** In addition to changing basic network configuration files on the root filesystem, you will also need to change a few specific files to enable NFS root, these files wil be covered in the example shown in section “Example for installing through network storage” on page 544.

Obtaining a zImage with NFS root enabled

The RHEL5.1 and Fedora 7 default kernels do not have options that would allow NFS root enabled. Due to this reason, you won't be able to boot an nfsroot system on these default kernels.

Because of these reasons, you will need to have a zImage with an initial RAM disk (initrd) that supports booting from NFS and apply it to your BladeCenter QS21 through a Trivial File Transfer Protocol (TFTP) server.

Below are the steps for obtaining a zImage and applying it to the tftp server:

1. Obtain or create your zImage file:
   a. For Fedora 7, you can download the zImage file from: http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/sdk3.0/zImage.initrd-2.6.22-5.20070920bsc.
   b. For RHEL5.1, you will need to create the zImage file. All steps provided below should be applied on the POWER based system that contains the filesystem and kernel that you want to mount onto the BladeCenter QS21. You can use the following steps for this:
      i. Make sure the correct boot configuration is stored in the zImage. To do this, ensure that BOOTPROTO=dhcp on /etc/sysconfig/network-scripts/ifcfg-eth0
      ii. Create the initrd image by using the following command:

```
# mkinitrd --with=tg3 --rootfs=nfs --net-dev=eth0 \
--rootdev=<nfs server>:/<path to nfsroot> \
~/initrd-<kernel-verson>.img <kernel-version>
```

      iii. Create the zImage by using the following command:

```
# mkzimage /boot/vmlinuz-<kernel-version>
/boot/config-<kernel-version> \
/boot/System.map-<kernel-version> <initrd> \
/usr/share/ppc64-utils/zImage.stub <zImage>
```

2. Apply your created or downloaded zImage to the exported directory of your TFTP server

Applying the zImage and root filesystem

Now that you've obtained the most important components needed to boot up a BladeCenter QS21, these being the root filesystem to be mounted and the zImage file, you'll need to establish how to pass these onto the BladeCenter QS21.

When a BladeCenter QS21 system boots up, the first component it will need to access is the kernel, which is achieved through loading the zImage file. Once the kernel is successfully loaded and booted up, the root filesystem is mounted via NFS.

The zImage file can be provided to the BladeCenter QS21 communicating with a Dynamic IP Configuration (DHCP) server which can provide the zImage file via TFTP.

Here are the steps you will need to follow in ensuring this

1. Ensure DHCP and TFTP packages are installed and the corresponding services are enabled on the server.
2. Place your zImage file in a directory that will be exported. Ensure this directory is exported and TFTP server is enabled by editing the /etc/xinet.d/tftp file:

```
disable = no
server_args             = -s <directory to export> -vvvvv
```

3. Edit /etc/dhcpd.conf file to reflect your settings for the zImage by editing the “filename” argument.

> **Note:** If you are booting up a Fedora 7 filesystem, you will need to add the following option entry to your /etc/dhpd.conf file:
>
> ```
> option root-path “<NFS server>:<path to nfsroot>”;
> ```

Booting up the BladeCenter QS21

Now that the root filesystem has been created, modified and configured for being exported, and the zImage file has been placed on a TFTP server, you can proceed to boot your QS21.

The BladeCenter QS21 can boot from:

- The optical drive of the BladeCenter unit media tray
- A SAS storage device, typically one or more hardisks attached to the BladeCenter unit.
- A storage device attached to the network.

To boot up through a device attached to the network, ensure that the boot sequence for the BladeCenter QS21 is set to “Network”. This configuration can be established through the AMM web-browser by going to **Blade Tasks** → **Configuration** → **Boot Sequence** → . You should now be able to boot up your Bladecenter QS21 system.

10.2.4 Example for installing through network storage

We will now show an example on applying the steps mentioned in Section 13.2.3. In this example, we will cover some specific modifications to the root filesystem in order for it to successfully boot up on the BladeCenter QS21. We will implement these steps mainly thru the use of bash scripts, to show how these steps can be applied in an aautomated fashion.

On section 10.5, “Example DIM implementation on BladeCenter QS21 cluster” on page 572, we cover an example of implementing a cluster of QS21s through the use of the Distributed Image Management tool.

This example will assume that we’ve already installed the Linux distribution of choice on a POWER-based system.

First, let’s establish the parameters that will be used for this example:

Example 10-2 Network settings specific to this example

```
Linux Distribution: RHEL5.1
Kernel Version: 2.6.18-53.el5
POWER based system with initial install IP & hostname:
192.168.170.30/POWERbox
NFS, PTFTP & DHCP Server: 192.168.170.50
QS21 Hostnames: qs21cell51-62
QS21 Private Network IP Address (eth0) : 192.168.170.51-62
NFS root path: /srv/netboot/QS21/RHEL5.1/boot/192.168.170.51
```

Note: Notice that we will be using the NFS, TFTP and DHCP server as the same system. Ideally, you would want them to be the same, but you can have the NFS server be a different machine if preferred for storage purposes.

First, we utilize the “cell_build_master.sh” script, this script will copy the root tree directory from the RHEL5.1 installation on a POWER based system to the NFS server. It will also modify some files in this master root filesystem so as to prepare it for nfsroot.

```
root@dhcp-server# ./cell_build_master.sh 192.168.170.50 \
/srv/netboot/QS21/RHEL5.1/master
```

Example 10-3 cell_build_master.sh script

```
#!/bin/bash
################################################################
# QS21 Cell Build Master Root Tree                             #
#                                                              #
```

```
#                                                                    #
######################################################################

### Show help information ######################################
usage()
{
  cat << EOF
  ${0##*/} - Creates master root tree for Cell QS21.

  usage: $0 [POWER_MACHINE] [DESTINATION_PATH]

  Arguments:
    POWER_MACHINE       Server where '/' will be copied from
    DESTINATION_PATH    Path where the master directory wil be
                        stored

  Example:

        ./cell_build_master 10.10.10.1 /srv/netboot/qs21/RHEL5.1/master

        This will copy the root directory from machine
        '10.10.10.1' and store it in /srv/netboot/qs21/RHEL5.1/master

EOF
  exit 1
}

if [ $# != 2 ]; then
  usage
fi

POWER_MACHINE=$1
DESTINATION_DIR=$2
RSYNC_OPTS="-avp -e ssh -x"

set -u
set -e
### Check if master tree already exists ##########################
test -d $DESTINATION_DIR || mkdir -p $DESTINATION_DIR

### Copy root filesystem from POWER-based machine to NFS server ###
rsync $RSYNC_OPTS $POWER_MACHINE:/ $DESTINATION_DIR

### Remove 'swap', '/' and '/boot' entries from /etc/fstab ########
grep -v "swap" $DESTINATION_DIR/etc/fstab | grep -v " / " \
```

```
| grep -v " /boot" > $DESTINATION_DIR/etc/fstab.bak
### Ensure SELinux is disabled ###################################
sed -i "s%^\(SELINUX\=\).*%\\1disabled%" \
$DESTINATION_DIR/etc/selinux/config
```

Note that /etc/fstab has some changes that are placed on /etc/fstab.bak, this backup file will eventually overwrite the /etc/fstab file. For now we have a master copy for this distribution, we can now apply this to multiple BladeCenter QS21s.

Next, we'll grab a copy of this master root filesystem and edit some files to make it more specific to the individual BladeCenter QS21s. We will use the "cell_copy_rootfs.sh" script.

```
root@dhcp-server# ./cell_copy_rootfs.sh \
/srv/netboot/qs21/RHEL5.1/master /srv/netboot/qs21/RHEL5.1/boot/ \
192.168.70.30 192.168.70.51-62 -i qs21cell 51 - 62
```

Notice that in this case, we're copying the master root filesystem to 12 individual BladeCenter QS21s, additionally, we are configuring some files in each of the copied root filesystems so as to accurately reflect the network identity of each corresponding BladeCenter QS21.

Example 10-4 cell_copy_rootfs.sh script

```
#!/bin/bash
#####################################################################
# QS21 Cell Copy Master Root Filesystem                             #
#                                                                   #
#                                                                   #
#####################################################################

### Show help information ########################################
usage()
{
  cat << EOF
  ${0##*/} - Copy Master Root Tree for individual BladeCenter QS21.

  usage: $0 [MASTER] [TARGET] [NFS_IP] [QS21_IP] -i [QS21_HOSTNAME]

  Arguments:
    MASTER              Full path of master root filesystem
    TARGET              Path where the root filesystems of the blades
                        will be stored.
    NFS_IP              IP Address of NFS Server
    QS21_IP             IP Address of QS21 Blade(s).  If creating
                        for multiple blades, put in range form:
```

```
                                10.10.2-12.10
 -i [QS21_HOSTNAME]     Hostname of Bladecenter QS21. If creating
                        root filesytems for multiple blades, use:

                                -i <hostname> <first> - <last>
 -h                     Show this message

  Example:

    ./cell_copy_master.sh /srv/netboot/qs21/RHEL5.1/master \
/srv/netboot/qs21/RHEL5.1 10.10.10.50 10.10.10.25-30 \
-i cell 25 - 30

    This will create root paths for QS21 blades cell25 to cell30,
    with IP addresses ranging from 10.10.10.25 to 10.10.10.25.
    These paths will be copied from
    /srv/netboot/qs21/RHEL5.1/master into
    /srv/netboot/qs21/RHEL5.1/cell<25-30> on NFS server 10.10.10.50

EOF
 exit 1
}

### Process QS21 IP Address passed ##############################
proc_ip (){
        PSD_QS21_IP=( `echo "$1"` )
        QS21_IP_ARY=( `echo $PSD_QS21_IP | sed "s#\.# #g"` )
        QS21_IP_LENGTH=${#QS21_IP_ARY[*]}
        for i in $(seq 0 $((${#QS21_IP_ARY[*]} - 1)))
        do
                PSD_RANGE=`echo ${QS21_IP_ARY[i]} | grep "-"`
                if [ "$PSD_RANGE" != "" ]; then
                        RANGE=`echo ${QS21_IP_ARY[i]} | \
                        sed "s#-# #"`
                        QS21_IP_ARY[i]=new
                        QS21_TEMP=`echo ${QS21_IP_ARY[*]}`
                        for a in `seq $RANGE`; do
                                NEW_IP=`echo ${QS21_TEMP[*]} | \
                                sed "s#new#$a#" | sed "s# #.#g"`
                                QS21_IP[b]=$NEW_IP
                                ((b++))
                        done
                        echo ${QS21_IP[*]}
                        break
                fi
```

```
        done
        if [ -z "$QS21_TEMP" ]; then
                echo $PSD_QS21_IP
        fi
}

### Show usage if no arguments passed ############################
if [ $# = 0 ]; then
  usage
fi

MASTER=$1
TARGET=$2
NFS_IP=$3
QS21_IP=$4
shift 4

QS21_IP=( `proc_ip "$QS21_IP"` )

### Capture QS21 Hostname(s) #####################################
while getopts hi: OPTION; do
case $OPTION in
  i)
    shift
    BLADES=( $* )
    ;;
  h|?)
    usage
    ;;
esac
done

### If a range of blades is provided, process them here ##########
if [ "${BLADES[2]}" = "-" ]; then
        BASE=${BLADES[0]}
        FIRST=${BLADES[1]}
        LAST=${BLADES[3]}
        BLADES=()
        for i in `seq $FIRST $LAST`;
        do
                BLADES[a]=${BASE}${i}
                ((a++))
        done
fi
```

```
### Ensure same number of IP and Hostnames have been provided ####
if [ "${#BLADES[*]}" != "${#QS21_IP[*]}" ] ; then
        echo "Error: Mismatch in number of IP Addresses & Hostnames"
        exit 1
fi

### Creation & configuration of individual Blade paths ###########
for i in $(seq 0 $((${#BLADES[*]} - 1)))
do
        ### Check if master root filesystem already exists #######
        test -d  "${TARGET}${BLADES[i]}" && \
        { echo "target \"$TARGET/${BLADES[i]}\" exists"; exit 1; }

        ### Copy master root filesystem for  a specific blade ####
        /usr/bin/rsync -aP --delete ${MASTER}/* \
        ${TARGET}${BLADES[i]}

        #### Edit /etc/fstab for specific machine ################
        echo "${NFS_IP}:${TARGET}${BLADES[i]}                  \
           nfs      tcp,nolock      1 1" >>
$TARGET/${BLADES[i]}/etc/fstab.bak
        echo "spufs                    /spu                     \
        spufs   defaults        0 0" >>
$TARGET/${BLADES[i]}/etc/fstab.bak
        cp -f $TARGET/${BLADES[i]}/etc/fstab.bak
$TARGET/${BLADES[i]}/etc/fstab
        mkdir $TARGET/${BLADES[i]}/spu

        #### Setup Network ######################################
        echo "Now configuring network for target machine...."

        SUBNET=`echo $NFS_IP | cut -f1-2 -d.`
        sed -i "s%^${SUBNET}.*%${QS21_IP[i]} ${BLADES[i]}   \
        ${BLADES[i]}%" $TARGET/${BLADES[i]}/etc/hosts
        sed -i "s%^\(HOSTNAME\=\).*%\\1${BLADES[i]}%" \
        $TARGET/${BLADES[i]}/etc/sysconfig/network
done
echo "Tree build and configuration completed."
```

Note: There may be situations where you want to have two ethernet connections for the BladeCenter QS21.

For example, if on eth0 you have a private network and you want to establish public access to the BladeCenter QS21 via eth1. In this case, make sure you edit the ifcnfg-eth1 file located on '/etc/sysconfig/network-scripts/'. So in our example, we would edit '/srv/netboot/QS21boot/192.168.170.51/etc/sysconfig/network-scripts/ifcfg -eth1' so that it reflects the network settings for BladeCenter QS21.

Now we have a root filesystem modified and ready to be NFS mounted to our BladeCenter QS21. But before this, we need to create a corresponding zImage. For this particular purpose, we will use the "BLUEcell_zImage.sh" script, this script will be ran on the POWER based system where the RHEL5.1 installation was done.

```
root@POWERbox# ./cell_zImage 192.168.70.50 2.6.18-53.el5
/srv/netboot/qs21/RHEL5.1/boot/ -i cell 51 - 62
```

Here, we've created 12 zImage files, one for each BladeCenter QS21.

Example 10-5 cell_zImage.sh script

```
#/bin/bash
#####################################################################
# QS21 zImage Creation Script                                       #
#                                                                   #
#                                                                   #
#                                                                   #
#####################################################################

set -e

### Show help information ###########################################
usage()
{
  cat << EOF
  ${0##*/} - creates zImage for Cell QS21.

  usage: $0 [NFS_SERVER] [KERN_VERSION] [NFS_PATH] -i [IDENTIFIER]

  Arguments:
    NFS_SERVER          Server that will mount the root filesystem
                        to a BladeCenter QS21.
```

```
    KERN_VERSION        Kernel version zImage wil be based on
    NFS_PATH            Path on NFS_SERVER where the root
                        filesystems for blades will be stored

 -i [identifier]        Directory name that will identify specific
                        blade on NFS_SERVER.  If creating zImage
                        for multiple blades, use:

                                -i <hostname> <first> - <last>
 -h                     Show this message

  Example:

    ./cell_zImage.sh 10.10.10.1 2.6.18-53.el5 \
/srv/netboot/qs21/RHEL5.1 -i cell 25 - 30

    This will create zImages for QS21 blades cell21 to cell34, based
    on kernel 2.5.18-53.el5, and whose NFS_PATH will be
    10.10.10.1:/srv/netboot/qs21/RHEL5.1/cell<25-30>

EOF
  exit 1
}

if [ $# = 0 ]; then
  usage
fi

NFS_SERVER=$1
KERN_VERSION=$2
NFS_PATH=$3

shift 3
while getopts hi: OPTION; do
case $OPTION in
  i)
    shift
    BLADES=( $* )
    ;;
  h|?)
    usage
    ;;
esac
done
```

```
### Ensure to set BOOTPROTO=dhchp on eth0 config files ###########
sed -i "s%^\(BOOTPROTO\=\).*%\\1dhcp%" \
/etc/sysconfig/network-scripts/ifcfg-eth0

### If a range of blades is provided, process them here ##########
if [ "${BLADES[2]}" = "-" ]; then
        BASE=${BLADES[0]}
        FIRST=${BLADES[1]}
        LAST=${BLADES[3]}
        BLADES=()
        for i in `seq $FIRST $LAST`;
        do
                BLADES[a]=${BASE}${i}
                ((a++))
        done
fi

### Create the initrd and zImages ################################
for QS21 in ${BLADES[*]}; do

        ### Create initrd image ##################################
        mkinitrd --with "tg3" --net-dev "eth0" \
        --rootdev=${NFS_SERVER}:${NFS_PATH}/${QS21} --rootfs=nfs \
        initrd-nfs-${QS21}-${KERN_VERSION}.img ${KERN_VERSION}

        ### Create kernel zImage #################################
        mkzimage vmlinuz-${KERN_VERSION} config-${KERN_VERSION}  \
        System.map-${KERN_VERSION} \
        initrd-nfs-${QS21}-${KERN_VERSION}.img \
        /usr/share/ppc64-utils/zImage.stub \
        zImage-nfs-${QS21}-${KERN_VERSION}.img
        rm -rf initrd-nfs-${QS21}-${KERN_VERSION}.img > \
        /dev/null 2>&1
        echo "zImage has been built as zImage-nfs-${QS21}-\
${KERN_VERSION}.img"
done
```

Finally, we copy the zImage file to our DHCP server:

```
root@POWERbox# scp /boot/zImage.POWERbox-2.6.18-53.el5
root@192.168.170.50:/srv/netboot/QS21/RHEL5.1/images/
```

Now that we've placed the two most important components, the root filesystem in our NFS server and the zImage file in our DHCP server, we can more forward to ensure these are accessible by our BladeCenter QS21

First, we edit our /etc/exports file in our NFS server so as to give the proper access permissions, our file will look like this:

```
/srv/netboot/QS21/RHEL5.1/boot/192.168.70.51
192.168.170.0(rw,sync,no_root_squash)
```

We now edit our /etc/xinet.d/tftp file, we're mostly interested in the "server_args" and "disable" parameters, our file now looks as such:

Example 10-6 Sample /etc/xinet.d/tftp configuration file

```
#default: off
# description: The tftp server serves files using the trivial file
#transfer protocol.  The tftp protocol is often used to boot diskless
#workstations, download configuration files to network-aware printers,
#and to start the installation process for some operating systems.
service tftp
{
   socket_type             = dgram
        protocol                = udp
        wait                    = yes
        user                    = root
        server                  = /usr/sbin/in.tftpd
        server_args             = -vvv -s /srv/netboot/QS21/images
        disable                 = no
        per_source              = 11
        cps                     = 100 2
        flags                   = IPv4
}
```

Now we pay attention to our /etc/dhcpd.conf file in our DHCP server. In order for changes made to the dhcpd.conf file to take into affect, you have to restart the dhcpd service each time you make a change on the dhcp.conf file.

To minimize this need, we'll create soft link that points to the zImage file. In the future, if we want to change the zImage for our BladeCenter QS21, we change where the soft link points to instead of having to edit the dhcpd.conf file and restart the dhcpd service.

```
root@192.168.170.50# ln -snf \
/srv/netboot/QS21/RHEL5.1/images/POWERbox/zImage-POWERbox-2.6.18-53\
.el5 /srv/netboot/QS21/images/QS21BLADE
```

Next, we ensure our /etc/dhcpd.conf file is properly set, our file now looks like this:

Example 10-7 Sample entry on /etc/dhcpd.conf file

```
subnet 192.168.170.0 netmask 255.255.255.0 {
next-server 192.168.170.50;
}
host QS21Blade{
    filename “QS21Blade";
    fixed-address 192.168.170.51;
    option host-name "POWERbox";
    hardware ethernet 00:a1:b2:c3:d4:e5;
}
```

Notice for the “filename” variable, we have called it “QS21Blade”, this file is named in reference to the directory specified on the variable “server_args”, defined under /etc/xinet.d/tftp’ file. Recall that “POWERbox” is a soft link we assigned earlier that points to the appropriate zImage file of interest.

Let’s assume that we did edit our /etc/dhcpd.conf file, in this case, we’ll have to restart the dhcpd service as such:

```
root@192.168.170.50# service dhcpd restart
```

We are now completed with this setup, assuming our BladeCenter QS21 is properly configured for booting up via Network, we can move forward and restart our BladeCenter QS21.

10.3 Installing SDK3.0 on BladeCenter QS21

The BladeCenter QS21 and SDK 3.0 supports the following operating systems:

- Red Hat Enterprise Linux 5.1
- Fedora 7

Furthermore, while SDK 3.0 is available for both of these Linux distributions, SDK3.0 support is available only for Red Hat Enterprise Linux 5.1.

Additionally, note that the following SDK3.0 components are not available for RHEL5.1

- Crash SPU Commands
- Cell Performance Counter
- OProfile
- SPU-Isolation

- Full-System Simulator and Sysroot Image

The following table provides the SDK 3.0 SO images provided for each distribution, how they're obtainable, and their corresponding contents.

Table 10-1 Red Hat Enterprise Linux (RHEL5).1 ISO Images

Product Set	ISO Name	Location
The Product package is intended for production purposes. This package contains access to IBM support and all of the mature technologies in SDK3.0.	CellSDK-Product-RHEL_.3.0.0.1.0.iso	http://www.ibm.com/software/howtobuy/passportadvantage
The Developer package is intended for evaluation of the SDK in a development environment and contains all of the mature technologies in SDK 3.0	CellSDK-Devel-RHEL_3.0.0.1.0.iso	http://www.ibm.com/developerworks/power/cell/downloads.html
The Extras package contains the latest technologies in the SDK. These packages are usually less mature or are technology preview code that may or may not become part of the generally available product in the future.	CellSDK-Extra-RHEL_3.0.0.1.0.iso	http://www.ibm.com/developerworks/power/cell/downloads.html

Table 10-2 Fedora 7 ISO Images

Product Set	ISO Name	Locations
The Developer package is intended for evaluation of the SDK in a development environment and contains all of the mature technologies in SDK 3.0	CellSDK-Devel-Fedora_3.0.0.1.0.iso	http://www.ibm.com/developerworks/power/cell/downloads.html

Product Set	ISO Name	Locations
The Extras package contains the latest technologies in the SDK. These packages are usually less mature or are technology preview code that may or may not become part of the generally available product in the future.	CellSDK-Extra-Fedora_3.0.0.0.1.0.iso	http://www.ibm.com/developerworks/power/cell/downloads.html

In addition to these packages, there are a set of open source SDK components which are usually accessed directly through YUM from a direcotory on the Barcelona Supercomputing Web site.

If your BladeCenter QS21 does not have outside internat access, you can download these components into one of your local servers that does have external access and install the RPMs manually on the BladeCenter QS21 of interest. The table below outlines these open source components.

Table 10-3 SDK 3.0 Open Source Components

Component	RHEL5.1	Fedora 7
Crash SPU Commands	Not Available	http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/sdk3.0/CellSDK-Open-Fedora/cbea/
GCC Toolchain	http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/sdk3.0/CellSDK-Open-RHEL/cbea/	http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/sdk3.0/CellSDK-Open-Fedora/cbea/
LIBSPE/LIBSPE2	Included with distribution	http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/sdk3.0/CellSDK-Open-Fedora/cbea/
netpbm	Included with distribution	http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/sdk3.0/CellSDK-Open-Fedora/cbea/

Component	RHEL5.1	Fedora 7
numactl	http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/sdk3.0/CellSDK-Open-RHEL/cbea/	http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/sdk3.0/CellSDK-Open-Fedora/cbea/
Oprofile	Not Available	http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/sdk3.0/CellSDK-Open-Fedora/cbea/
Sysroot Image	Not Available	http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/sdk3.0/CellSDK-Open-Fedora/cbea/

10.3.1 Pre-installation steps

Before proceeding forward to the SDK 3.0 installation on a BladeCenter QS21, ensure the following preparatory steps are established.

1. Ensure the BladeCenter QS21 has the appropriate firmware, level QB-01.08.0-00 or higher. For further information on firmware, please refer to the “Firmware Considerations” section of this chapter.
2. Ensure YUM updater daemon is disabled as it cannot be running when installing SDK. To turn YUM updater daemon off, type:

   ```
   # /etc/init.d/yum-updatesd stop
   ```

3. For RHEL5.1 only:
   a. If you plan on installing FDPRO-Pro, you must ensure compat-libstdc++ RPM is installed first, otherwise, the FDPR-Pro installation will fail.
   b. Ensure that the LIBSPE2 libraries provided in the RHEL5.1 Supplementary CD are installed, these rpms are:
      - libspe2-2.2.0.85-1.el5.ppc.rpm
      - libspe2-2.2.0.85-1.el5.ppc64.rpm
   c. Ensure that elfspe utility is provided by installing elfspe2 rpms as provided in the RHEL5.1 Supplementary CD, these rpms are:
      - elfspe2-2.2.0.85-1.el5.rpm

- libspe2-devel-2.2.0.85-1.el5.ppc.rpm (if application development applies)
- libspe2-devel-2.2.0.85-1.el5.ppc64.rpm (if application development applies)

After installing this rpm, ensure spufs is loaded correctly by doing the following steps:

i. `# mkdir -p /spu`

ii. If if you want to mount spufs immediately:

```
# mount /spu
```

To ensure it automatically mounts on boot, place the following line under /etc/fstab:

```
spufs /spu spufs defaults 0 0
```

4. Ensure that rsync, sed, tcl, and wget packages are installed in your BladeCenter QS21, the SDK Installer requires these packages.
5. If you will be installing SDK3.0 through ISO images, ensure to create the /tmp/sdkiso directory and place the SDK3.0 ISO images in this directory.

Now that you've established the prerequisites for installation, you can proceed to installation.

10.3.2 Installation Steps

The installation for SDK3.0 is mostly covered by the SDK Installer, which is obtained after installing the cell-install-<rel>-<ver>.noarch rpm.

After you install the cell-install RPM, you can install SDK3.0 through the use of the cellsdk script, using the iso option:

```
# cd /opt/cell
# ./cellsdk --iso /tmp/cellsdkiso install
```

Note: If you prefer to install through a GUI, you can add the --gui flag when executing the cellsdk script.

Make sure to read the corresponding license agreements, these licenses being GPL and LGPL. Additionally, either the International Programming License Agreement (IPLA) or the International License Agreement for Non-Warranted Programs (ILAN). If you install the "Extras" ISO, you will also be presented with the International License Agreement for Early Release of Programs (ILAER).

Once the license agreements have been read and established, confirm to YUM install the specified SDK.

10.3.3 Post-Installation Steps

Now you have installed the default components for SDK3.0. The only necessary steps at this point are related to configuration updates for YUM.

First, if you do not intend on installing additional packages, you'll need to unmount the isos that were automatically mounted when you ran the install. To do this, run the following command.

```
# ./cellsdk --iso /tmp/cellsdkiso unmount
```

Note: If you find the need to install additional packages that are included in the SDK3.0 but are not part of the default install, make sure to mount these isos once again. You can accomplish this using the following command:

```
# ./cellsdk --iso /tmp/cellsdkiso mount
```

Afterwards, run '**yum install <package_name>**'

Next, you'll have to edit the /etc/yum.conf file from preventing automatic updates from overwriting certain SDK components. Add the following clause to the [Main] section of this file to prevent YUM update from overwriting SDK versions of the following runtime RPMs:

```
exclude=blas kernel numactl oprofile
```

The next and final step is to re-enable the yum updater daemon that was initially disabled before SDK3.0 installation:

```
# /etc/init.d/yumupdater start
```

You've now established all of the necessary post-installation steps. If you are interested in installing additional SDK3.0 components for development purposes, please refer to the IBM Software Development Kit for Multicore Acceleration Version 3.0.0 Installation Guide.

For additional details on SDK3.0 installation and installing a supported distribution on a BladeCenter QS21 system, please refer to the IBM Software Development Kit for Multicore Acceleration Version 3.0.0 Installation Guide, located at http://www.ibm.com/alphaworks/tech/cellsw/download.

10.4 Firmware considerations

Firmware for the BladeCenter QS21 comes primarily through two packages, one is through the baseboard management controller (BMC) firmware, and the other through the basic input/output system (BIOS) firmware. In a more detailed manner, here is what each firmware packages covers;

- BMC firmware
  - Communicates with advanced management module
  - Controls power on
  - Initializes board, including the Cell BE processors and clock chips
  - Monitors the physical board environment
- System firmware
  - Takes control once BMC has successfully initialized the board
  - Acts as basic input/output system (BIOS)
  - Includes boot-time diagnostics and power-on self test
  - Prepares the system for operating system boot

It is important that both of these packages match at any given time in order to avoid problems and system performance issues.

10.4.1 Updating firmware for the BladeCenter QS21

When updating the firmware for the BladeCenter QS21, it is highly recommended that both the BMC and system firmware are updated, with the system firmware being updated first. Both of these packages can be downloaded from `http://www.ibm.com/support/us/en`.

Checking current firmware version

There are two ways to check the firmware level on a BladeCenter QS21. The first way is through the Advanced Management Module, while the other is through the command line.

The Advanced Management Module can give you information on not only the system firmware, but also the BMC firmware. Through this interface, you can view the build identifier, release, and revision level of both firmware types. These are viewable under **Monitors** → **Firmware VPD**.

You can also view the system firmware level through the command line by typing the following command:

```
# xxd /proc/device-tree/openprom/ibm,fw-vernum_encoded
```

On the output, the value of interest is the last field, which starts with “QB”, this will be your system firmware level.

Updating system firmware

To update the system firmware, you can download the firmware update script fro IBM’s online support site, once downloaded into your BladeCenter QS21, run the following comand on your running BladeCenter QS21:

```
# ./<firmware_script> -s
```

This will automatically update the firmware silently and reboot the machine, you can also extract the .bin system firmware file into a directory:

```
# ./<firmware_script> -x <directory to extract to>
```

Once you have obtained the .bin file into a chosen directory, you can update the firmware using the following command under Linux:

```
# update_flash -f <rom.bin file obtained from IBM support site>
```

Note: Both of the previous commands will cause the BladeCenter QS21 machine to reboot, so make sure you run this while having access to the machine’s console, either via serial connection or through SOL connection.

Now that you’ve updated and successfully booted up to the new system firmware, you must ensure that you have a backup copy of this firmware image on your server. There are always two copies of the system firmware image on the blade server, these being TEMP and PERM.

TEMP	Firmware image normally used in the boot process. When you update the firmware, it is the TEMP image that is updated.
PERM	This is a backup copy of the system firmware boot image. Should your TEMP image be corrupt, you can recover to a working firmware from this copy. More info is provided on this process later in this section.

You can check from which image the Blade server has booted up from by running the following command:

```
# cat /proc/device-tree/openprom/ibm,fw-bank
```

If the output returns a “P”, this means you have booted on the PERM side, you will usually boot on the TEMP side unless that particular image is corrupt.

After you have successfully booted up your blade server from the TEMP image with the new firmware, you can copy this image to the backup PERM side by typing the following command:

```
# update_flash -c
```

Additionally, you can accomplish this task of copying from TEMP to PERM by typing the following command:

```
# echo 0 > /proc/rtas/manage_flash
```

Please refer to the “Recovering a working system firmware ” section if you encounter problems with your TEMP image file.

Updating BMC firmware

Once you’ve obtained and uncompressed .tgz file, you will have the BMC firmware image whose filename will be BNBT<version number>.pkt.

To update, first, log into the corresponding BC-H Advanced Management Module through a browser. Once you’ve logged in, make sure to turn off the BladeCenter QS21 whose firmware you’re going to update. Next, go to **Blade Tasks** → **Firmware Update**, select the blade server of interest, then select “Update” and “Continue” on the following screen.

You should now be set with your updated BMC firmware, you can now boot up your BladeCenter QS21.

Updating firmware for Infiniband Daughter Card

If you have the optional, Infiniband daughter card on your BladeCenter QS21, you may have to occasionally update the firmware. For this task to be completed, the “openib-tvflash” package will need to be installed.

First, obtain the .tgz packaged file from the IBM support site, uncompress it on your BladeCenter QS21.

```
# tar xvzf cisco-qs21-tvflash.tgz
```

Next, run the following command:

```
# ./tvflash -p <firmware .bin file>
```

At this point you should have your Infiniband daughter card firmware updated. For further information on features and supplementary information, please refer to the “Topspin_LinuxHost_ReleaseNotes_3.2.0” under http://www.ibm.com/support/en/us.

Recovering to a working system firmware

The system firmware is contained in two separate images of the flash memory, these being the temporary (TEMP) and the permanent (PERM) image. Usually, when a BladeCenter QS21 boots up, it will boot from the TEMP image. However, in instances where he TEMP image is corrupt or damaged, the system will then boot up from the PERM image.

To choose which image to boot from, you can do this by accessing the SMS utility and on the main menu and selecting “Firmware Boot Side Options”. For more on accessing the SMS utility, please refer back to the THIS SECTION.

To check which image your machine is currently booted on, type the following command:

```
# cat /proc/device-tree/openprom/ibm,fw-bank
```

A returned value of “P” means you’ve booted from the PERM side.

If you have booted from the PERM side and would like to boot from the TEMP side instead, First copy the PERM image to the TEMP image by running the following command:

```
# update_flash -r
```

Shut down the blade server, restart the blade system management processor through the Advanced Management Module and turn the BladeCenter QS21 back on.

10.5 Options for managing multiple blades

As previously covered, the BladeCenter QS21's characteristic of being a diskless system implies the need for additional configurations to ensure initial functionality. Some of these steps, more specifically for RHEL5.1, require individual BladeCenter QS21 configurations, which can become mundane when amplified to a cluster network of BladeCenter QS21s.

This section will introduce two tools in particular which can be implemented to not only set an ease to such configuration steps for scalabe purposes, but also establish such cluster environment in an organized fashion. These two tools are Extreme Cluster Administration Toolkit (xCAT) and Distributed Image Management for Linux Clusters (DIM).

10.5.1 Distributed Image Management

Distributed Image Management for Linux Clusters (DIM) is a tool that was developed for scalable image management purposes. This tool allows diskless blades in particular, to run a Linux distribution over the network. Addtionally, traditional maintenance tasks can be easily and quickly applied to multitude of blades at the same time. Distributed Image Management for Linux Clusters was first implemented for use in IBM's MareNostrum supercomputer, which consists of over 2,500 IBM JS21 Blades.

Distributed Image Management for Linux Clusters is a cluster image management utility, it does not contain tools for cluster monotoring, event management, or remote console management. DIM's primary primary focus is to adddress the difficult task of managing Linux distribution images for all nodes of a fairly sized cluster.

Some additional characterstics of DIM are the following:

- Automated IP and DHCP configuration through an XML file that describes the cluster network and naming taxonomy.
- Allows set up of multiple images(i.e. can have Fedora and RHEL5.1 images setup for one blade) for every node in parallel
- Allows for fast incremental maintenance of filesystem images, changes such as user ID's, passwords, and RPM installations.
- Manages multiple configurations to be implemented accross a spectrum of individual blades, these being:
  - IP addresses
  - DHCP

- NFS
- File system images
- network boot images (BOOTP/PXE)
- Node remote control

While DIM is not a comprehensive cluster management tool, if so needed, it can by complemented by other cluster management tools such as xCAT.

DIM has been tested on BladeCenter QS21s and the latest release supports DIM implementation on Cell blades.

DIM implementation on BladeCenter QS21s

The most recent release of DIM supports and documents how to implement it for BladeCenter QS21 node clusters. DIM has the ability to provide configuration, set-up, and provisioning for all these nodes through one or more image servers. We will show generalized steps for implementing DIM on a cluster of QS21s, and will follow forward with this in providing an example implementation.

The following prerequsities are needed before proceeding to installing and setting up DIM:

- A single, regular disk-based installation on a POWER-based machine.
- A POWER-based or x86-64 machine to be the DIM server
- An image server where the master and node trees will be stored(This can be the same as the DIM-Server). It is recommended to have at least 20GB of storage space + .3 GB per node.
- The the following software:
  - BusyBox
  - Perl Net-SNMP
  - PERL XML-Simple
  - PERL XML-Parser

We will assume that the POWER-based installation has been already established, ensure that when you do the initial install, the “Development Tools” package is included. We will also assume that the dim server in this case also contain the distribution that will be deployed to the BladeCenter QS21s.

We recommend establishing a local YUM repository to ease the process of package installation and for instances where your BladeCenter QS21 does not have external internet access. Assuming you will be applying the rpms from a distribution install DVD, You can accomplish this through the following steps:

```
# mount /dev/cdrom /mnt
  rsync -a /mnt /repository
  umount /dev/cdrom
  rpm -i /repository/Server/createrepo-*.noarch.rpm
   createrepos /repository
```

Next, create the /etc/yum.repos.d/local.repo configuration file to reflect this newly created repository:

```
[local]
name=Local repository
baseurl=file:///repository
enabled=1
```

DIM server setup steps

On your DIM server, ensure the following packages are installed and enabled:

- dhcp
- xinetd
- tftp-server

Because DIM will loop mount the BladeCenter QS21 images, you'll need to increase the number of allowed loop devices. Additionally, you'll want to automatically start DIM_NFS upon bootup of the dim-server. You can accomplish both both of these tasks by editing the /etc/rc.d/rc.local file so that it's configured as shown below:

Example 10-8 /etc/rc.d/rc.local file for DIM server.

```
#!/bin/sh
#
# This script will be executed *after* all the other init scripts.
# You can put your own initialization stuff in here if you don't
# want to do the full Sys V style init stuff.
DIM_SERVER=<DIM_IP_ADDRESS>
touch /var/lock/subsys/local
if grep -q "^/dev/root / ext" /proc/mounts; then
        #commands for the DIM Server
        modprobe -r loop
        modprobe loop max_loop=64
        if [ -x /opt/dim/bin/dim_nfs ]; then
                /opt/dim/bin/dim_nfs fsck -f
                /opt/dim/bin/dim_nfs start
```

```
        fi
else
        # commands for the DIM Nodes
        ln -sf /proc/mounts /etc/mtab
        test -d /home || mkdir /home
        mount -t nfs $DIM_SERVER:/home /home
        mount -t spufs spufs /spu
fi
if [ -x /etc/rc.d/rc.local.real ]; then
. /etc/rc.d/rc.local.real
fi
```

Note that you'll have to replace the variable DIM_SERVER with your particular DIM server IP address. Next, run the following commands to edit /etc/init.d/halt:

```
root@dim-server# perl -pi -e \
's#loopfs\|autofs#\\/readonly\|loopfs\|autofs#' /etc/init.d/halt

root@dim-server# perl -pi -e \
's#/\^\\/dev\\/ram/#/(^\\/dev\\/ram|\\/readonly)/#' /etc/init.d/halt
```

This edit will prevent the reboot command from failing on BladeCenter QS21 nodes as without this fix, the QS21 will try to unmount its own root file system when switching into runlevel 6.

Now you can reboot your DIM server and we'll proceed to installing DIM.

DIM Software Install

Now, we'll show the steps for installing DIM on your designated DIM server.

First, obtain the latest DIM rpm from the IBM alphaworks website, this rpm can be downloaded from http://www.alphaworks.com/tech/dim.

Next, download the additional needed software:

- For Busybox:

  http://www.busybox.net/downloads/busybox-1.1.3.targ.gz

- For Perl Net-SNMP:

  http://www.cpan.org/modules/by-module/Net/Net-SNMP-5.1.0.tar.gz

- For Perl XML-Parser:

  http://www.cpan.org/modules/by-module/XML/XML-Parser-2.34.tar.gz

- For Perl XML-Simple

http://www.cpan.org/modules/by-module/XML/XML-Simple-2.18.tar.gz

Place all of these downloaded packages and the DIM rpm into a created /tmp/dim_install directory. Install the DIM rpm from this directory.

Next, add /opt/dim/bin to the PATH enviroment variable:

```
root@dim-server# echo 'export PATH=$PATH:/opt/dim/bin' >> \
$HOME/.bashrc
root@dim-server# . ~/.bashrc
```

Install the additional PERL modules required for DIM (Net-SNMP, XML-Parser, XML-Simple), you can ignore the warnings that may be displayed.

```
root@dim-server# cd /tmp/dim_install
                make -f /opt/dim/doc/Makefile.perl
                make -f /opt/dim/doc/Makefile.perl install
```

Next, build Busybox and copy the BusyBox binary to the DIM directory:

```
root@dim-server# cd /tmp/dim_install
                tar xzf busybox-1.1.3.tar.gz
                cp /opt/dim/doc/busybox.config \
                busybox-1.1.3/.config
                patch -p0 < /opt/dim/doc/busybox.patch
                cd busybox-1.1.3
                make
                cp busybox /opt/dim/dist/busybox.ppc
                cd /
```

Note: The Busybox binary must be built on a POWER based system, otherwise the kernel zimage will not boot up on a BladeCenter QS21.

At this point DIM and it's dependent packages has been installed on your DIM server, we can move forward to setup steps.

DIM setup

This section will outline some of the basic setup steps, provide descriptions and options to these steps, and additionally point out some specific configuration files where modifications can be made to meet your specific network needs.

First, we'll point three DIM configuration files in particular that will need to be modificed to meet your specific network setup.

– /opt/dim/config/dim_ip.xml

  This file you will need to initially create, you can copy one of the template examples named "dim_ip.xml.<example>" and modify it as you see fit. The main purpose of this file is to set a DIM IP configuration for your Cell clusters. It will define your Cell DIM nodes, their corresponding hostnames, ip addresses and it will also define the DIM server.

– /opt/dim/config/dim.cfg

  This file defines the larger scale locations of the NFS, DHCP and DIM configuration files. For the most part, the default values will apply in most cases, however these variables may frequently change:

  • DIM_DATA

    This is the directory where the all of the data relevant to the distro you're deplying accross your network will be stored. This data includes the zImages, the master root filesystem, and the blade filesystem images.

  • NFS_NETWORK

    This will define your specific IP address deployment network.

  • PLUGIN_MAC_ADDR

    This will define the IP address of your BladeCenter H, it is through this variable that DIM will access the BladeCenter H to obtain MAC address info on the QS21 nodes along with executing basic, operational, blade specific commands.

– /opt/dim/config/<DISTRO>/dist.cfg

  This file will define some variables that are specific to your distribution deployment. It will also define the name of the DIM_MASTER machine, should it be different. Some variables of interest that you may need to change:

  • KERNEL_VERSION

    The kernel version you'll want for creating bootable zImages on your particular distribution.

  • KERNEL_MODULES

    The modules that you'll want to and aren't enabled by default on your kernel. Note that the only module you'll need for BladeCenter QS21 to function is "tg3".

  • DIM_MASTER

This variable defines the machine that contains the master root filesystem. In cases where the distribution you want to deploy in your cluster is in a different machine, you'll specify which machine to grab the root filesystem from. Otherwise, if it is located in the same box as your DIM server, you can leave the default value.

– /opt/dim/<DIST>/master.exclude

  This file contains a list of directories that wil be excluded from being copied to the master root filesystem.

– /opt/dim/<DIST>/image.exclude

  This file contains a list of directories that wil be excluded from being copied to the image root filesystem of each BladeCenter QS21.

Before you proceed further to the DIM commands which will begin to create your images, ensure that you adapt the files and variables defined above to meet your particular network needs and preferences.

Now, you can execute the DIM commands which will create you distribution and node specific images. You can find the extent of the DIM commands offered under the /opt/dim/lib directory. Each one of these commands should have an accessible manual page on your system, run "**man <dim_command>**".

First, create the DIM master directory for your distribution of interest:

```
root@dim-server# dim_sync_master -d <distribution>
```

Now, build the DIM network boot image:

```
root@dim-server# dim_zImage -d <distribution>
```

Next, create the the read-only and "x" read-write DIM images representing the number of nodes in your network.

```
root@dim-server# dim_image -d <distribution> readonly
dim-server[y]-blade[{1..x}]
```

Where "y" is the corresponding DIM server number(if there is more than one) and "x" represents the number of BladeCenter QS21 nodes. Add all of these DIM images to /etc/exports:

```
root@dim-server# dim_nfs add -d <distribution> all
```

Now mount and confirm all the DIM images for NFS exporting:

```
root@dim-server# dim_nfs start
                dim_nfs status
```

Note: Ensure that you have enabled the max number of loop devices on your dim-server, otherwise, you may see an error related to this when running the command above. To increase the number of loop devices, run:

```
root@dim-server# modrpobe -r loop
                 modprobe loop max_loop=64
```

Next, set the base configuration of DHCPD with your own IP subnet address:

```
root@dim-server# dim_dhcp add option -O
UseHostDeclNames=on:DdnsUpdateStyle=none
                dim_dhcp add subnet -O Routers=<IP_subnet> \
dim-server[1]-bootnet1
                dim_dhcp add option -O DomainName=dim
```

The following steps will require that the BladeCenter QS21 be connected in the BladeCenter H chassis, the steps mentioned previously did not require this. In this instance, you now add each QS21 blade to the dhcp.conf file through the following command:

```
root@dim-server# dim_dhcp add node -d <distribution> \
dim-server[y]-blade[x]
```

Once again, where “y” represents the dim-server number(if there are is more than one dim server) and “x” represents the QS21 blade. Once you’re done adding all of the nodes of interest, ensure to restart the dhcp service:

```
root@dim-server# dim_dhcp restart dhcpd
```

You should now be ready to boot up your QS21 to your distribution as configured under DIM.

Once your setup is complete, you may want to eventually add aditionally software to your nodes, for these purposes, it is recommended you apply software maintenance on the original machine where the root filesystem was copied from, then, you can sync the master root filesystem and DIM images.

```
root@powerbox# dim_sync_master -d <distribution> <directory
updated>..<directory updated>..
root@powerbox# dim_sync_image -d <distribution> <directory
updated>..<directory updated>..
```

Notice that you’ll list out the directories to sync on the master and DIM images, depending on which directories are affected by your software installation.

Example DIM implementation on BladeCenter QS21 cluster

We will now show an example of applying DIM on a small cluster of BladeCenter QS21s.

This example will address the issue where POWER-based servers are in limited availability. We have two initial installs on a POWER based machine, while the dim-server will be an X-series machine.

Here are the variables for our example:

Example 10-9 Settings for DIM implementatoin example

```
Distributions: Fedora 7 and RHEL5.1
Fedora 7 Server:f7-powerbox
RHEL5.1 Server: rhel51-powerbox
DIM Boot Image Creation Server: POWER-based
qs21-imagecreate(192.168.20.70)
Dim Server: xSeries qs21-dim-server(192.168.20.30)
Kernel Version: 2.6.22-5.20070920bsc and 2.6.18-53.el5
Dim Nodes: qs21cell{1..12}(192.168.20.71..82)
BladeCenter H IP address: 192.168.20.50
```

We've chosen to use both Fedora 7 and RHEL5.1 distributions to split half of our cluster between RHEL5.1 and Fedora. Also, note that we have have our DIM server and our image server be two different machines.

Notice that our dim-server will be an System x server while our DIM Boot Image server will be a POWER based system. We have chosen our boot image server to be a POWER based system because a POWER based system is needed for creating the kernel zImage file. Due to the fact we're working with two different distributions, we will need to copy the root filesystem from two different power-based installations into our System x server.

We will install DIM only on the dim server and when needing to create a zImage, mount the DIM directory to our boot image server. We will specify the machine if a procedure is to applied to only one of these two servers.

Note: If you do not have a POWER based system, you can install your distribution on a BladeCenter QS21 by using USB storage, copy this installed filesystem to your non-POWER based dim-server, and use the USB storage strictly for creating needed boot zImage files.

We won't show the steps for the initial hard-disk installation on a POWER based machine, we'll only mention that we did ensure to include the "Development Tools' package in our installation.

We'll want to give our dim-server the ability to partially manage the BladeCenter's QS21 from the dim-server's command prompt. In order to achieve this, we'll need to access the Advanced Management Module (AMM) through a web browser. Once we access the AMM, we'll go to **MM Control** → **Network Protocols** → **Simple Network Management Protocol (SNMP)** and set the following values:

```
SNMPv1 Agent    : enable
Community Name  : public
Access Type:    : set
Hostname or IP  : 192.168.20.50
```

Since one of our distributions is Fedora7, we need to install the kernel rpm that is provided on the Barcelona Supercomputing web site on our Fedora 7 machine:

```
root@f7-powerbox# wget \
http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/s
dk3.0/kernel-2.6.22-5.20070920bsc.ppc64.rpm

                  rpm -ivh --force --noscripts \
kernel-2.6.22-5.20070920bsc.ppc64.rpm

                  depmod -a 2.6.22-5.20070920bsc
```

First we'll want to ensure we have increased the numer of max loop devices and start the nfs service by default. This is accomplished by editing the file as shown in Example 10-8 on page 566, the file will look the same, except we'll assigning the variable DIM-Server to the IP address in our example, 192.168.20.70.

Next, we'll make the two changes needed on the /etc/init.d/halt file on both the f7-powerbox and rhel51-powerbox:

```
root@f7-poerbox# perl -pi -e \
                 's#loopfs\|autofs#\\/readonly\|loopfs\|autofs#'\
/etc/init.d/halt

                 perl -pi -e \
                's#/\^\\/dev\\/ram/#/(^\\/dev\\/ram|\\/readonly)/#'\
/etc/init.d/halt
```

Finally, we'll add all of the QS21 blade hostnames under /etc/hosts, this will only apply to the dim-server:

```
root@dim-server# echo "192.168.20.50 mm mm1" >> /etc/hosts

                 echo "192.168.20.70 dim-server" >> /etc/hosts

                 for i in `seq 71 82`; do echo "192.168.20.$i cell$i\
b$i"; done >> /etc/hosts
```

Now we've established all the steps before proceeding to the DIM software install.

For DIM software install, first we'll download the DIM RPM from the IBM Alphaworks website (http://www.alphaworks.com/tech/dim) and the required dependent software for DIM.

We will create a /tmp/dim_install directory and place our downloaded DIM rpm on there, this will apply to both the dim-server and the dim-storage machine.

```
root@dim-server # mkdir /tmp/dim_install
```

The script "dim_install.sh" will setup the procedures for installing our DIM software, this script will be ran on both servers as well:

```
root@dim-server # ./dim_install.sh
```

Example 10-10 dim_install.sh script

```
#/bin/bash
#####################################################################
# DIM Installation Script                                           #
#                                                                   #
#                                                                   #
#                                                                   #
#####################################################################

set -e

### Download the additional sofware needed by DIM ###############
cd /tmp/dim_install
wget http://www.busybox.net/downloads/busybox-1.1.3.tar.gz
wget http://www.cpan.org/modules/by-module/Net/Net-SNMP-5.1.0.tar.gz
wget http://www.cpan.org/modules/by-module/XML/XML-Parser-2.34.tar.gz
wget http://www.cpan.org/modules/by-module/XML/XML-Simple-2.18.tar.gz

### Add /opt/dim/bin to PATH environment variable ###############
echo "export PATH=$PATH:/opt/dim/bin" >> $HOME/.bashrc
chmod u+x ~/.bashrc
~/.bashrc

### Install perl modules required for DIM ########################
cd /tmp/dim_install
make -f /opt/dim/doc/Makefile.perl
make -f /opt/dim/doc/Makefile.perl install
```

```
### Add /opt/dim/bin to PATH environment variable ###############
echo "export PATH=$PATH:/opt/dim/bin" >> $HOME/.bashrc
. ~/.bashrc
echo "Completed"
```

Note: Ensure that the following packages are installed before executing the script above:

- gcc
- expat-devel

The needed busybox binary must be built on a POWER based system. Because of this, we'll build this binary on our boot image creation POWER machine.

```
root@qs21-imagecreate# cd /tmp/dim_install
                       tar xzf busybox-1.1.3.tar.gz
                       cp /opt/dim/doc/busybox.config \
busybox-1.1.3/.config
                       patch -p0 < /opt/dim/doc/busybox.patch
                       cd busybox-1.1.3
                       make
                       scp busybox \
root@qs21-dim-server:/opt/dim/dist/busybox.ppc
```

Now that 'we've installed DIM on our server, we'll have to make some modifications to setup DIM.

The first file of interest is /opt/dim/config/dim_ip.xml, we'll use one of the example templates (dim_ip.xml.example4) and modify it to our needs. This will need to be modified on both servers. The xml file is provided below.

```
root@dim-server# cp /opt/dim/config/dim_ip.xml.example4 \
/opt/dim/config/dim_ip.xml
```

Example 10-11 /opt/dim/config/dim_ip.xml file

```
<?xml version="1.0" ?>
<!-- $Id: dim_ip.xml.example4 1654 2007-09-02 09:03:44Z morjan $ -->
<!--

    Simple DIM IP configuration file for CELL
        - boot network 192.168.70.0/24 (eth0)
        - user network 10.0.0.0/8      (eth1, optional)
        - 13 DIM Nodes  (blade 1-13)
```

```
        - 1  DIM Server (blade 14)

    Examples:
    DIM Component                          IP-Address        Hostname    Comment

=============================================================================================
dim-server[1]-bootnet1               192.168.70.14     s1         DIM Server JS21 Slot 14 eth0
dim-server[1]-blade[1]-bootnet1      192.168.70.1      cell1      Cell Blade 1    Slot 1 eth0
dim-server[1]-blade[13]-bootnet1     192.168.70.13     cell13     Cell Blade 13   Slot 13 eth0
dim-server[1]-blade[1]-usernet1      10.0.0.1          cell1u     Cell Blade 1    Slot 1 eth1
-->
<dim_ip xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="dim_ip.xsd">
    <configuration name="dim">
        <component name="server" min="1" max="1">
            <network name="bootnet1">
                <mask>255.255.255.0</mask>
                <addr>192.168.20.0</addr>
                <ip>192, 168, 20, 30</ip>
                <hostname>dim-server%d, server</hostname>
            </network>
            <component name="blade" min="1" max="13">
               <network name="bootnet1" use="1">
                   <mask>255.255.255.0</mask>
                   <addr>192.168.20.0</addr>
                   <ip>192, 168, 20, ( blade + 70 )</ip>
                   <hostname>qs21cell%d, ( blade + 70 )</hostname>
               </network>
               <network name="usernet1">
                   <mask>255.0.0.0</mask>
                   <addr>10.10.10.0</addr>
                   <ip>10, 10, 10, ( blade + 70 )</ip>
                   <hostname>qs21cell%du, blade</hostname>
               </network>
            </component>
        </component>
    </configuration>
</dim_ip>
```

We'll note the main changes done form the example template revolve around the IP addresses, tand hostnames for both the DIM server and the individual BladeCenter QS21s. Note that this file is open to be modified to meet your specific hardware considerations.

Next we modify /opt/dim/config/dim.cfg to fit our needs:

Example 10-12 /opt/dim/config/dim.cfg file

```
#-------------------------------------------------------------------------
# (C) Copyright IBM Corporation 2004
# All rights reserved.
#
#
#-------------------------------------------------------------------------
# $Id: dim.cfg 1563 2007-07-11 10:12:07Z morjan $
#-------------------------------------------------------------------------

# DIM data directory (no symbolic links !)
DIM_DATA                /var/dim

# DHCPD config file
DHCPD_CONF              /etc/dhcpd.conf

# dhcpd restart command
DHCPD_RESTART_CMD       service dhcpd restart

# NFS config file
NFS_CONF                /etc/exports

# NFS server restart command
NFS_RESTART_CMD         /etc/init.d/nfsserver restart

# NFS export options read-only image
NFS_EXPORT_OPTIONS_RO   rw,no_root_squash,async,mp,no_subtree_check

# NFS export options read-write image
NFS_EXPORT_OPTIONS_RW   rw,no_root_squash,async,mp,no_subtree_check

# NFS export option network
NFS_NETWORK             192.168.20.0/255.255.255.0

# TFTP boot dir
TFTP_ROOT_DIR           /srv/tftpboot

# plugin for BladeCenter MAC addresses
PLUGIN_MAC_ADDR         dim_mac_addr_bc -H 192.168.20.50

# name of dim_ip configuration file
DIM_IP_CONFIG           dim_ip.xml

# name of zimage linuxrc file
```

```
LINUX_RC                linuxrc

# network interface
NETWORK_INTERFACE       eth0

# boot type (NBD|NFS)
BOOT_TYPE               NFS
```

In the file provided above, we mainly changed the NFS_NETWORK, TFTP_ROOT_DIR, and PLUGIN_MAC_ADDR variables so as to reflect our settings, the file is shown to provide a glimpse on what can additionally be configured.

Because we are working with two distributions, we'll need to create entries for both distributions.

```
root@qs21-storage# mkdir /opt/dim/config/Fedora7
                   mkdir /opt/dim/dist/Fedora7
                  mkdir /opt/dim/config/RHEL51
                   mdkir /opt/dim/dist/RHEL51
                   cp /opt/dim/CELL/* /opt/dim/config/Fedora7
                   cp /opt/dim/dist/CELL/* /opt/dim/dist/Fedora7/
                   cp /opt/dim/CELL/* /opt/dim/config/RHEL51
                   cp /opt/dim/dist/CELL/* /opt/dim/dist/RHEL51
```

As can be seen, we copied all the files under /opt/dim/CELL and /opt/dim/dist/CELL into our distribution config directories. The files copied from /opt/dim/CELL we're "dist.cfg", "image.exclude", and "master.exclude". We'll configure these files to meet our needs. For the files under /opt/dim/dist/CELL, we shouldn't need to change modify them.

Below are the "dist.cfg" files for Fedora7 and RHEL5.1 respectively.

Example 10-13 /opt/dim/config/Fedora7/dist.cfg file

```
#-------------------------------------------------------------------
# (C) Copyright IBM Corporation 2006
# All rights reserved.
#
#-------------------------------------------------------------------
# $Id: dist.cfg 1760 2007-10-22 16:30:11Z morjan $
#-------------------------------------------------------------------
```

```
# Image file size for readwrite part in Megabytes
IMAGE_SIZE_RW           200

# Image file size for readonly part in Megabytes
IMAGE_SIZE_RO           5000

# Read only directories
DIR_RO                      bin boot lib opt sbin usr lib64

# Read write directories
DIR_RW                      root dev etc var srv selinux

# Create additional directories
DIR_ADD                      media mnt proc readonly sys spu huge tmp
home

# Ethernet DHCP trials
ETHERNET_DHCP_TRIALS    3

# Reboot delay in seconds on failure
REBOOT_DELAY            60

# Zimage command line options
ZIMAGE_CMDLINE         not used

# Mount options readwrite tree
MOUNT_OPTIONS_RW        rw,tcp,nolock,async,rsize=4096,wsize=4096

# Mount options readonly tree
MOUNT_OPTIONS_RO        ro,tcp,nolock,async,rsize=4096,wsize=4096

# Boot method
BOOT_METHOD             bootp

# Name of zimage linuxrc file
LINUX_RC                linuxrc

# Kernel release (uname -r)
KERNEL_VERSION          2.6.22-5.20070920bsc

# Kernel modules
KERNEL_MODULES          tg3.ko sunrpc.ko nfs_acl.ko lockd.ko nfs.ko

# DIM master node       (hostname | [user@]hostname[:port]/module)
DIM_MASTER              f7-powerbox
```

```
# DIM server            (hostname | [user@]hostname[:port]/module
[,...])
DIM_SERVER              localhost
```

The only variables we have changed here are the KERNEL_VERSION, KERNEL_MODULES, and DIM_MASTER. We additionally modified MOUNT_OPTIONS_RW and MOUNT_OPTIONS_RO variables to include “tcp”.

The rest of the variables we have left with their default values. In this example, the only difference between the Fedora7 and RHEL5.1 configuration file are the KERNEL_VERSION, KERNEL_MODULES, and DIM_MASTER variables.

Example 10-14 /opt/dim/config/RHEL51/dist.cfg file

```
#-------------------------------------------------------------------------
# (C) Copyright IBM Corporation 2006
# All rights reserved.
#
#-------------------------------------------------------------------------
# $Id: dist.cfg 1760 2007-10-22 16:30:11Z morjan $
#-------------------------------------------------------------------------

# Image file size for readwrite part in Megabytes
IMAGE_SIZE_RW           200

# Image file size for readonly part in Megabytes
IMAGE_SIZE_RO           5000

# Read only directories
DIR_RO                      bin boot lib opt sbin usr lib64

# Read write directories
DIR_RW                      root dev etc var srv selinux

# Create additional directories
DIR_ADD                      media mnt proc readonly sys spu huge tmp
home

# Ethernet DHCP trials
ETHERNET_DHCP_TRIALS    3

# Reboot delay in seconds on failure
REBOOT_DELAY            60
```

```
# Zimage command line options
ZIMAGE_CMDLINE         not used

# Mount options readwrite tree
MOUNT_OPTIONS_RW        rw,nolock,async,rsize=4096,wsize=4096

# Mount options readonly tree
MOUNT_OPTIONS_RO        ro,nolock,async,rsize=4096,wsize=4096

# Boot method
BOOT_METHOD             bootp

# Name of zimage linuxrc file
LINUX_RC                linuxrc

# Kernel release (uname -r)
KERNEL_VERSION          2.6.18-53.el5

# Kernel modules
KERNEL_MODULES          tg3.ko fscache.ko sunrpc.ko nfs_acl.ko lockd.ko
nfs.ko

# DIM master node       (hostname | [user@]hostname[:port]/module)
DIM_MASTER              rhel51-powerbox

# DIM server            (hostname | [user@]hostname[:port]/module
[,...])
DIM_SERVER              localhost
```

The “image.exclude” and “master.exclude” are text files which contain a list of directories to be excluded from image root filesystems and the master root filesystem. We will leave the default directories on these files.

Next, we’ll create the master image. First, we’ll need to copy the root filesystems from the initial Fedora 7 and RHEL5.1 machines into our qs21-storage server.

We’ve already specified these initial machines in our /opt/dim/config/<DIST>/dist.cfg file, now we want to create public ssh public key authentication for each machine.

```
root@dim-server# ssh-keygen -t dsa

root@dim-server# cat ~/.ssh/id_dsa.pub | ssh root@f7-powerbox "cat \
- >> ~/.ssh/authorized_keys"
```

```
root@dim-server# cat ~/.ssh/id_dsa.pub | ssh root@rhel51-powerbox \
"cat - >> ~/.ssh/authorized_keys"
```

Now, we can move forward to creating the master images:

```
root@dim-server# dim_sync_master -d Fedora7
                dim_sync_master -d RHEL5.1
```

We now create the zimages, in this case, since our dim-server is an Xseries machine, we'll cannot run the command on the dim-server machine. Instead, we will mount the /opt/dim, /var/dim and /srv/tftpboot direcotories from the Xseries dim server onto a power-based machine, in this case, our boot image server.

```
root@dim-server# echo "/opt/dim" >> /etc/exports
root@dim-server# echo "/var/dim" >> /etc/exports
root@dim-server# echo "/srv/tftpboot" >> /etc/exports
```

Now that we've made these directories exportable from the dim-server, we'll mount them on the power based boot image server:

```
root@qs21-imagecreate# mkdir /opt/dim
                       mkdir /var/dim
                       mkdir /srv/tftpboot
                       mount dim-server:/opt/dim /opt/dim
                       mount dim-server:/var/dim /var/dim
                       mount dim-server:/srv/tftpboot /srv/tftpboot
                       echo "export PATH=$PATH:/opt/dim/bin" >> \
$HOME/.bashrc
                       . ~/.bashrc
```

Next, we'll want to create the zImage files on the dim boot image server and afterwards, umount the directories:

```
root@qs21-imagecreate# dim_zimage -d Fedora7
                       dim_zimage -d RHEL5.1
                       umount /opt/dim
                       umount /var/dim
                       umount /srv/tftpboot
```

We return to the dim-server and create the images

```
root@dim-server# dim_image -d Fedora7 readonly \
dim-server[1]-blade[{1..5}]

                dim_image -d RHEL5.1 readonly \
dim-server[1]-blade[{6..12}]

                dim_nfs add -d Fedora7 all

                dim_nfs add -d RHEL5.1 all

                dim_nfs start
```

Next we'll add the base configuration to our dhcpd.conf file, this will apply to our dim-server only:

```
root@dim-server# dim_dhcp add option -O \
UseHostDeclNames=on:DdnsUpdateStyle=none

                dim_dhcp add subnet -O Routers=192.168.20.100 \
dim-server[1]-bootnet1

                dim_dhcp add option -O DomainName=dim
```

Now, we'll add the entries into our /etc/dhcp.conf file:

```
root@dim-server# for i in 'seq 1 5'; do dim_dhcp add node -d Fedora7\
dim-server[1]-blade[i]; done

                for i in 'seq 6 12'; do dim_dhcp add node -d \
RHEL5.1 dim-server[1]-blade[i]; done

               dim_dhcp restart
```

Finally, we can use DIM to ensure each blade is configured to boot up via network and also boot up each QS21 blade node.

```
root@dim-server# for i in 'seq 1 12'; do dim_bbs -H mm1 i network;\
done

                for i in 'seq 1 12'; do dim_bctool -H mm1 i on; done
```

We have now completed implementing DIM on 12 of our QS21 nodes, using both Fedora7 and RHEL5.1 as deployment distributions.

10.5.2 Extreme Cluster Administration Toolkit

The Extreme Cluster Administration Toolkit (xCAT) is a toolkit used for deployment and administration of Linux clusters, with many of it's features taking advantage of the IBM xSeries® hardware.

xCAT is written entirely using scripting languages such as korn, shell, perl, and Expect. Many of these scrips can be altered to reflect the needs of your particular

network. xCAT provides cluster management through four main branches, these being automated installation, hardware management and monitoring, software administration, and remote console support for text and graphics.

Here is a more detailed view of xCAT's offerings:

- Automated Installation
  - Network booting with PXE or Etherboot/GRUB
  - Red Hat installation with Kickstart
  - Automated Parallel installation for RedHat and SuSE
  - Automated Parallel installation via imaging with Windows, or other operating systems
  - Other OS installation using imaging or cloning
  - Automatic node configuration
  - Automatic errata installation
  - Hardware management and monitoring
- Hardware Management and monitoring
  - Supports the Advanced Systems Management features in IBM xSeries

    Remote Power control (on/off/state) via IBM Management Processor Network and/or APC Master Switch

    Remote Network BIOS/firmware update and configuration on extensive IBM hardware

    Remote vital statistics (fan speed, temperatures, voltages)

    Remote inventory (serial numbers, BIOS levels)

    Hardware event logs

    Hardware alerts via SNMP

    Create and manage diskless clusters.
  - Supports remote power control switches for control of other devices

    APC MasterSwitch

    BayTech Switches

    Intel EMP
  - Traps SNMP alerts and notify administrators via e-mail
- Software administration
  - Parallel shell to run commands simultaneously on all nodes or any subset of nodes

- Parallel copy and file synchronization
- Provides Installation and configuration assistance of the HPC software stack

  Message Passing Interface: Build scripts, documentation, automated set-up for MPICH, MPICH-GM, and LAM

  Maui and PBS for scheduling and queuing of jobs

  GM for fast and low latency inter-process communication using Myrinet

– Remote console support for text and graphics
  - Terminal servers for remote console

    Equinox ELS and ESP

    iTouch In-Reach and LX series

    Remote Console Redirect feature in IBM ^ xSeries BIOS
  - VNC for remote graphics

As can be seen, the offerings from xCAT are mostly geared towards automating many of the basic setup and management steps for small and larger, more complex clusters.

As previously mentioned, xCAT's offerings are broad to the extend that they can complement the cluster management offerings that DIM doesn't provide. Additionally, xCAT does provide it's own method of handling diskless clusters. While we wont' go into further detail on xCAT's full offerings and implementation, we briefly will cover xCAT's solution to diskless clusters.

Diskless systems on xCAT

Similar to DIM, there is a stateless cluster solution that can be implemented with xCAT provides an alternative method for installing and configuring diskless systems.

Warewulf[1] is a tool utilized in conjunction with xCAT in providing a stateless solution for High Performance Computing (HPC) clusters. It was originally designed and implemented by the Lawrence Berkley National Laboratory as part of the Scientific Cluster Support (SCS) program to meet the need of a tool that would allow deployment and management of a large number of clusters.

Warewulf's main purpose is to provide ease to maintaining and monitoring stateless images and nodes as well as allowing such tasks to be applied in a scalable fashion.

[1] See http://www.perceus.org/portal/project/warewulf

Offerings provided by Warewulf are the following:

- Master/Slave relationship
  - Master Nodes

    Supports interactive logins and job dispatching to slaves

    Gateway between outside world and cluster network

    Central Management for all nodes
  - Slave Nodes

    Slave nodes optimized primarily for computation

    Only available on private cluster network(s)
- Multiple physical cluster network support
  - Fast ethernet administration
  - High speed data networks(i.e. Myricom, Infiniband, GigE)
- Modular design which facilitates cusomization
  - Can change between kernel, linux distribution, cluster applications
- Network booting
  - Boot image is built from Virtual Node Filesystem(VNFS)

    A small chroot'able Linux distribution residing on the master node

    Network boot image created using VNFS as a template

    Destined to live in the RAM on the nodes
  - Nodes boot utilizing Etherboot

    Open source project that faciliatates network booting

    Uses DHCP and TFTP to obtain boot image
  - Implements RAM-disk filesystems

    All nodes capable of running diskless
  - Account user management

    Builds password file for all nodes

    Standard authentication schemes (files, NIS, LDAP)

    Rsync used to push files to nodes

As shown above, the extent of Warewulf's offerings rest on providing customization and scalability on a small or large cluster.

There are many similarities between Warewulf and DIM, among them, mostly revolving around easing the process of installation and management in an

efficient, scalable manner. Both of these tools automate many of the initial install steps required for a BladeCenter QS21 to boot up.

The primary difference between these two node install and management tools rests on machine state. Warewulf provides solutions for stateless machines through the use of RAM-disk filesystems that are shared and read only. DIM preserves the state of inidividual nodes by using NFS root methods to provide read/write images on each node along with making certain components of the file system read only.

Warewulf version 2.6.x in conjuction with xCAT version 1.3 has been tested on BladeCenter QS21s. DIM version 9.14 has been tested on BladeCenter QS21s.

These offerings are both good solutions whose benefits depend on the needs of a particular private cluster setup. In instances where storage is a limited resource and maintaining node state is not important, Warewulf may be better suited for you needs. If storage is not a limited resource and maintaining the state of individual nodes is important, then DIM may be the preferred option.

10.6 Method for installing a minimized distribution

The BladeCenter QS21's diskless characteristic coupled with the common need to have the operating system utilize a minimal amount of resources, brings forth the topic covered in this chapter. Achieving this process not only saves storage space but also minimizes memory usage.

This solution of minimal usage of resources by the operating system can be further extended by decreasing the size of the kernel zImage that is loaded to the BladeCenter QS21. Additionally, the storage footprint of the root filesystem utilized by each BladeCenter QS21 can be minimized through decreasing the directories to be mounted or making the some of the directories read only.

Those two topics will not be covered in this section as they are beyond the scop of this documentation. Additionally, DIM addresses both of these issues and provides a resource efficient root filesystem and kernel zImage.

We will cover further steps that can be taken during and briefly after installation in removing packages that won't be neccessary in most cases for a BladeCenter QS21. This example will be shown for RHEL5.1 only, but can closely be applied to Fedora 7 installs as well.

Note: These steps are all to be implemented on a POWER based system that contains actual disk storage. The product of this process will then be mounted on to a BladeCenter QS21.

10.6.1 During installation

Detailed steps of installing on RHEL5.1 will not be covered in this section. The main point of interest we want to establish during installation are the packages to be installed. When you are in the “Package Installation” step of the process, ensure to select “Customize Software Selection” as shown below:

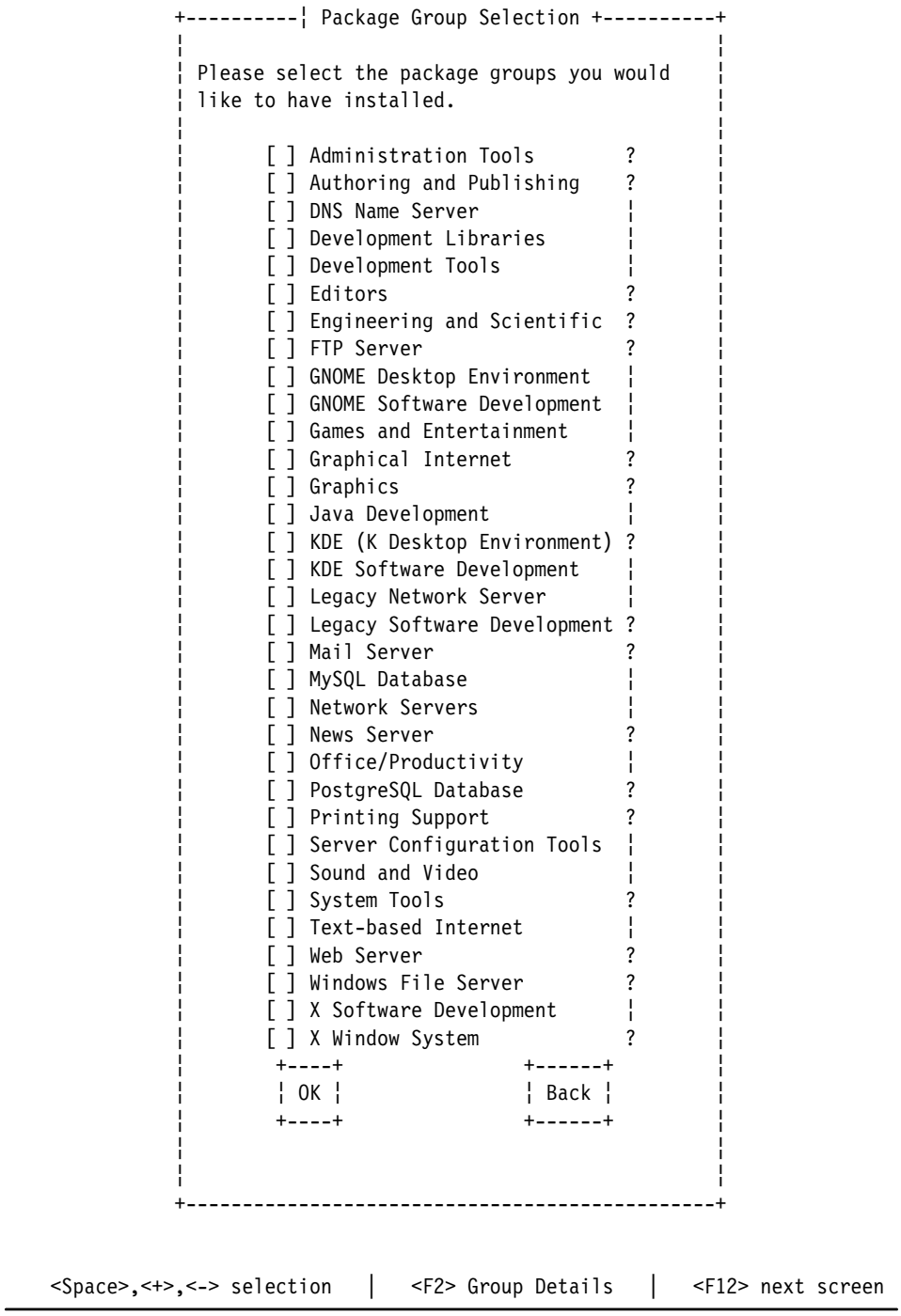
Example 10-15 Package Selection step of RHEL5.1 Installation Proccess

```
                +----------------¦ Package selection +-----------------+
                ¦                                                      ¦
                ¦ The default installation of Red Hat Enterprise       ¦
                ¦ Linux Server includes a set of software applicable   ¦
                ¦ for general internet usage. What additional tasks    ¦
                ¦ would you like your system to include support for?   ¦
                ¦                                                      ¦
                ¦               [ ] Software Development               ¦
                ¦               [ ] Web server                         ¦
                ¦                                                      ¦
                ¦                                                      ¦
                ¦                                                      ¦
                ¦                                                      ¦
                ¦           [*] Customize software selection           ¦
                ¦                                                      ¦
                ¦          +----+                  +------+           ¦
                ¦          ¦ OK ¦                  ¦ Back ¦           ¦
                ¦          +----+                  +------+           ¦
                ¦                                                      ¦
                ¦                                                      ¦
                ¦                                                      ¦
<               +------------------------------------------------------+
  <Tab>/<Alt-                                                      xt screen
```

In the following screen, you will be asked to specify which group of software packages you would like to install, some will have already been selected by default, ensure to de-select all of the packages as shown below:

Example 10-16

```
Welcome to Red Hat Enterprise Linux Server
```


Package Group Selection
Please select the package groups you would like to have installed.
[ ] Administration Tools
[ ] Authoring and Publishing
[ ] DNS Name Server
[ ] Development Libraries
[ ] Development Tools
[ ] Editors
[ ] Engineering and Scientific
[ ] FTP Server
[ ] GNOME Desktop Environment
[ ] GNOME Software Development
[ ] Games and Entertainment
[ ] Graphical Internet
[ ] Graphics
[ ] Java Development
[ ] KDE (K Desktop Environment)
[ ] KDE Software Development
[ ] Legacy Network Server
[ ] Legacy Software Development
[ ] Mail Server
[ ] MySQL Database
[ ] Network Servers
[ ] News Server
[ ] Office/Productivity
[ ] PostgreSQL Database
[ ] Printing Support
[ ] Server Configuration Tools
[ ] Sound and Video
[ ] System Tools
[ ] Text-based Internet
[ ] Web Server
[ ] Windows File Server
[ ] X Software Development
[ ] X Window System
OK
Back
<Space>,<+>,<-> selection | <F2> Group Details | <F12> next screen

Once you've ensured that none of the package groups have been selected, you can proceed forward with your installation.

10.6.2 Post-installation package removal

Now we'll focus on removing packages that were installed by default. In order to ease the process of package removal and dependency check, we strongly recommend you establish a local yum repository in your installed system.

Our main approach towards determining which packages to remove from our newly installed system is geared towards keeping the common purpose of a BladeCenter QS21 server into consideration. We will remove packages related to graphics, video, sound, documentation/word processing,network/email, security, and other categories which contain packages for the BladeCenter QS21 which may be unneccessary.

Graphics and audio

A great majority of the packages that are installed by default are related to Xorg and GNOME. Despite the fact we have selected to not install these particular package groups during our installation process, there are nevertheless packages that get installed related to these tools.

First, we'll remove the Advanced Linux Sound Architecture (ALSA) library, the removal of this particular package through YUM, will also remove other dependent packages:

```
# yum remove alsa-lib
```

Table 10-4 Packages removed with alsa-lib

Package	GNOME	Sound	Other Graphics	Doc	Other
alsa-utils		X			
antlr					X
esound		X			
firstboot					X
gjdoc				X	
gnome-mount	X				
gnome-python2	X				
gnome-python2-bonobo	X				

Package	GNOME	Sound	Other Graphics	Doc	Other
gnome-python2-canvas	X				
gnome-python2-extras	X				
gnome-python2-gconf	X				
gnome-python2-gnomevfs	X				
gnome-python2-gtkhtml2	X				
gnome-vfs2	X				
gtkhtml2			X		
gtkhtml2-ppc64			X		
java-1.4.2-gcj-compat					X
libbonoboui					X
libgcj	X				
libgcj-ppc64					X
libgnome					X
libgnomeui	X				
rhn-setup-gnome	X				
sox	X				

Next, remove the ATK library, which adds accessibility support to applications and graphical user interface toolkits. Removing this package will also remove the following packages through YUM:

```
# yum remove atk
```

Table 10-5 Packages removed with atk

Package	GNOME	Other Graphics	Other
GConf2			X
GConf2-ppc64			X
authconfig-gtk		X	
bluez-gnome			X
bluez-utils			X

Package	GNOME	Other Graphics	Other
gail		X	
gail-ppc64		X	
gnome-keyring	X		
gtk2		X	
gtk2-ppc64		X	
gtk2-engines		X	
libglade2		X	
libglade2-ppc64		X	
libgnomecanvas		X	
libgnomecanvas-ppc64		X	
libnotify			X
libwnck		X	
metacity		X	
notification-daemon			X
notify-python			X
pygtk2		X	
pygtk2-libglade		X	
redhat-artwork		X	
usermode-gtk		X	
xsri		X	

Next, remove the X.org X11 runtime library:

```
# yum remove libX11
```

Table 10-6 Packages removed with libX11

Package	X.org	Other Graphics	Other
libXcursor	X		
libXcursor-ppc64	X		
libXext	X		

Package	X.org	Other Graphics	Other
libXext-ppc64	X		
libXfixes	X		
libXfixes-ppc64	X		
libXfontcache	X		
libXft	X		
libXft-ppc64	X		
libXi	X		
libXi-ppc64	X		
libXinerama	X		
libXinerama-ppc64	X		
libXpm	X		
libXrandr	X		
libXrandr-ppc64	X		
libXrender	X		
libXrender-ppc64	X		
libXres	X		
libXtst	X		
libXtst-ppc64	X		
libXv	X		
libXxf86dga	X		
libXxf86misc	X		
libXxf86vm	X		
libXxf86vm-ppc64	X		
libxkbfile	X		
mesa-libGL		X	
mesa-libGL-ppc64		X	
tclx			

Package	X.org	Other Graphics	Other
tk		X	X

Next, remove X.org X11 libICE runtime library:

```
# yum remove libICE
```

Table 10-7 Packages removed with libICE

Package	X.org	Other
libSM	X	
libSM-ppc64	X	
libXTrap	X	
libXaw	X	
libXmu	X	
libXt	X	
libXt-ppc64	X	
startup-notification		X
startup-notification-ppc64		X
xorg-x11-server-utils	X	
xorg-x11-xkb-utils	X	

Remove the vector graphics library next, cairo:

```
# yum remove cairo
```

Table 10-8 Packages removed with cairo

Package	Doc	Printing	Other Graphics
cups		X	
pango	X		
pango-ppc64	X		
paps	X		
pycairo			X

Continue trying to remove remaining X.org packages:

```
# yum remove libfontenc
```

This will also remove “libXfont” and “xorg-x11-font-utils” packages

```
# yum remove x11-utils
```

This will also remove “xorg-x11-utils” package

```
# yum remove libFS libXau libXdmcp xorg-x11-filesystem
```

Remove the following remaining graphics packages:

```
# yum remove libjpeg
```

This will also remove “libtiff” package

```
# yum remove libart_lgpl fbset libpng gnome-mime
```

Now, wrap up the audio packages removal:

```
# yum remove audiofile libvorbis talk
```

There are other graphics related packages which will be removed by default due to dependencies on other non-graphics packages.

Documentation, word processing, and file manipulation packages

Next, remove packages that are related with documentation, word processing and file manipulation.

First, remove the file comparison tool, diffutils:

```
# yum remove diffutils
```

Table 10-9 Packages removed with diffutils

Packages	X.org	SELinux	GNOME	Other
chkfontpath	X			
policycoreutils		X		
rhpxl	X			
sabayon-apply			X	
selinux-policy		X		
selinux-policy-targeted		X		
urw-fonts				X
xorg-x11-drv-evdev	X			
xorg-x11-drv-keyboard	X			

Packages	X.org	SELinux	GNOME	Other
xorg-x11-drv-mouse	X			
xorg-x11-drv-vesa	X			
xorg-x11-drv-void	X			
xorg-x11-fonts-base	X			
xorg-x11-server-Xnest	X			
xorg-x11-server-Xorg	X			
xorg-x11-xfs	X			

Remove the relevant packages that are left in this category:

```
# yum remove aspell aspell-en ed words man man-pages groff bzip2 zip\
unzip
```

Network, email and printing packages

Now you can move forward and remove any network, printing and email related packages from your default distribution insall. First, remove the network packages.

```
# yum remove ppp
```

This will remove packages “rp-pppoe” and “wvdial”.

```
# yum remove avahi avahi-glib yp-tools ypbind mtr lftp NetworkManager
```

Now remove printing related packages:

```
# yum remove cups-libs mgetty
```

And finally, the email related packages

```
# yum remove mailx coolkey
```

Security, management and DOS-related packages

Because we cannot utilize SELinux for the BladeCenterQS21, we can look to remove these packages from our default install.

```
# yum remove checkpolixy setools libsemanage
```

Now remove the machine management related packages.

```
# yum remove conman anacron dump vixie-cron
```

And now you remove the DOS-related packages

```
# yum remove mtools unix2dos dos2unix dosfstools
```

USB and others

Now you can remove USB packages and other packages that will most likely not be needed for the typical BladeCenter QS21 implementation.

```
# yum remove cccid usbutils
```

We now cover the remaining packages to be removed from the RHEL5.1 default installation:

```
# yum remove bluez-libs irda-utils eject gpm hdparm hicolor \
ifd-egate iprutils parted pcsc-lite pcsc-lite-libs smartmontools \
wpa_supplicant minicom
```

At this point, we've removed packages mostly relevant to graphics, audio, word-processing, neworking and other tools. This process described above should cut your number of installed packages about one half.

10.6.3 Shutting off services

The final step we're providing in making your system faster and more efficient is a small list of remaining services that can be turned off. This will save run-time memory along with speed up your BladeCenter QS21 boot process. Use 'chkconfig' as shown below to turn off the services specified.

```
# chkconfig --level 12345 atd off
  chkconfig --level 12345 auditd off
   chkconfig --level 12345 autofs off
   chkconfig --level 12345 cpuspeed off
   chkconfig --level 12345 iptables off
   chkconfig --level 12345 ip6tables off
   chkconfig --level 12345 irqbalance atd off
   chkconfig --level 12345 isdn off
   chkconfig --level 12345 mcstrans off
   chkconfig --level 12345 rpcgssd off
   chkconfig --level 12345 rhnsd off
```

With the distribution stripped down to a lower amount of installed packages and a minimum amount of services running, you can copy this root filesystem to a master directory and on forward to individual BladeCenter QS21s for deployment. As stated prior, DIM takes steps at implement further resource

efficiency, such implementation can be complemented with the process shown in this section.

Part 5

Appendixes

In this part of the book we provide two Appendixes:

- Appendix A, “SDK 3.0 Topic Index” on page 601
- Appendix B, “Additional material” on page 609

A

SDK 3.0 Topic Index

This appendix contains a cross reference of programming topics relating to Cell BE application development to the Cell BE SDK 3.0 document containing that topic. This information in the following tables are subject to change when a new Cell BE SDK is released by IBM.

Table A-1 Programming topics cross reference

Topic	Cell BE SDK 3.0 Documentation
C and C++ Standard Libraries	C/C++ Language Extensions for Cell BE Architecture
Access Ordering	Cell Broadband Engine Architecture
Aliases, Assemlber	SPU Assembly Language Specification
Audio Resample Library	Audio Resample Library DELETED: See change log of Example Library API Reference, Version 3.0
Cache Management (software-managed cache)	Programming Guide V3.0 <-- missing VARIABLE
CBEA-Specific PPE Special Purpose Registers	Cell Broadband Engine Architecture

Topic	Cell BE SDK 3.0 Documentation
Completion Variables (Sync library)	Example Library API Reference
Composite Intrinsics	C/C++ Language Extensions for Cell BE Architecture
Conditional Variables (Sync library)	Example Library API Reference
Data Types and Programming Directives	C/C++ Language Extensions for Cell BE Architecture
Debug Format (DWARF)	SPU Application Binary Interface Specification, Version 1.7 <-- missing VARIABLE
DMA Transfers and Inter-Processor Communication	Cell Broadband Engine Programming Handbook
Evaluation Criteria for Performance Simulations	Performance Analysis with Mambo
Extensions to the PowerPC Architecture	Cell Broadband Engine Architecture
FFT Library	Example Library API Reference
Floating-Point Arithmetic on the SPU	C/C++ Language Extensions for Cell BE Architecture
Game Math Library	Example Library API Reference
Histograms	Example Library API Reference
I/O Architecture	Example Library API Reference
Image Library	Example Library API Reference
Instruction Set and Instruction Syntax	SPU Assembly Language Specification
Large Matrix Library	Example Library API Reference
Logical Partitions and a Hypervisor	Cell Broadband Engine Programming Handbook
Low-Level Specific and Generic Intrinsics	C/C++ Language Extensions for Cell BE Architecture
Low-Level System Information	SPU Application Binary Interface Specification

Topic	Cell BE SDK 3.0 Documentation
Mailboxes	Cell Broadband Engine Programming Handbook Cell Broadband Engine Programming Tutorial
Math Library	Example Library API Reference
Memory Flow Controller	Cell Broadband Engine Architecture Cell Broadband Engine Programming Handbook Cell Broadband Engine Programming Tutorial
Memory Map	Cell Broadband Engine Registers Cell Broadband Engine Architecture Cell Broadband Engine Programming Handbook
Memory Maps	Cell Broadband Engine Architecture
MFC Commands	Cell Broadband Engine Architecture Cell Broadband Engine Registers
Multi-Precision Math Library	Example Library API Reference
Mutexes	Example Library API Reference
Noise LibraryPPE	DELETED: See change log of Example Library API Reference, Version 3.0 Cell Broadband Engine SDK Libraries
Object Files	SPU Application Binary Interface Specification
Objects, Executables, and SPE Loading	Cell Broadband Engine Programming Handbook
Oscillator Libraries	DELETED: See change log of Example Library API Reference, Version 3.0 Cell Broadband Engine SDK Libraries
Overview of the Cell Broadband Engine Processor	Cell Broadband Engine Programming Handbook
Parallel Programming	Cell Broadband Engine Programming Handbookk
Performance Data Collection and Analysis with Emitters	IBM Full-System Simulator Performance Analysis
Performance Instrumentation with Profile Checkpoints and Triggers	IBM Full-System Simulator Performance Analysis
Performance Monitoring	Cell Broadband Engine Programming Handbook
Performance Simulation and Analysis with Mambo	IBM Full-System Simulator Performance Analysis

Topic	Cell BE SDK 3.0 Documentation
PowerPC Processor Element	Cell Broadband Engine Programming Handbook Cell Broadband Engine Architecture
PPE Interrupts	Cell Broadband Engine Programming Handbook
PPE Multithreading	Cell Broadband Engine Programming Handbook
PPE Oscillator Subroutines	DELETED: See change log of Example Library API Reference, Version 3.0 Cell Broadband Engine SDK Libraries
PPE Serviced SPE C Library Functions PPE-Assisted Funcitons	Security SDK Installation and User's Guide
Privileged Mode Environment	Cell Broadband Engine Architecture
Problem State Memory-Mapped Registers	Cell Broadband Engine Architecture
Program Loading and Dynamic Linking	SPU Application Binary Interface Specification
Shared-Storage Synchronization	Cell Broadband Engine Programming Handbook
Signal Notification	Cell Broadband Engine Programming Handbook SPE Runtime Management library C/C++ Language Extensions for Cell BE Architecture Cell Broadband Engine Programming Tutorial
IMD Programming	Cell Broadband Engine Programming Handbook
SPE Channel and Related MMIO Interface	Cell Broadband Engine Programming Handbook
SPE Context Switching	Cell Broadband Engine Programming Handbook
SPE Events	Cell Broadband Engine Programming Handbook SPE Runtime Management library
SPE Local Storage Memory Allocation	SEARCH-MORE Cell Broadband Engine SDK Libraries
SPE Oscillator Subroutines	DELETED: See change log of Example Library API Reference, Version 3.0 Cell Broadband Engine SDK Libraries
SPE Programming Tips	Cell Broadband Engine Programming Handbook Cell Broadband Engine Programming Tutorial

Topic	Cell BE SDK 3.0 Documentation
SPE Serviced C Library Functions	Security SDK Installation and User's Guide
SPU and Vector Multimedia Extension Intrinsics	C/C++ Language Extensions for Cell BE Architecture
SPU Application Binary Interface	SPU Application Binary Interface Specification Cell Broadband Engine Programming Handbook
SPU Architectural Overview	SPU Instruction Set Archtecture
SPU Channel Instructions	SPU Instruction Set Archtecture Cell Broadband Engine Architecture Cell Broadband Engine Programming Handbook
SPU Channel Map	Cell Broadband Engine Architecture
SPU Compare, Branch, and Halt Instructions	SPU Instruction Set Archtecture
SPU:Constant-Formation Instructions	SPU Instruction Set Archtecture
SPU Control Instructions	SPU Instruction Set Archtecture
SPU Floating-Point Instructions	SPU Instruction Set Archtecture
SPU Hint-for-Branch Instructions	SPU Instruction Set Archtecture
SPU Integer and Logical Instructions	SPU Instruction Set Archtecture
SPU Interrupt Facility	SPU Instruction Set Archtecture Cell Broadband Engine Programming Handbook
SPU Isolation Facility	Cell Broadband Engine Architecture
SPU Load/Store Instructions	SPU Instruction Set Archtecture
SPU Performance Evaluation	Performance Analysis with Mambo
SPU Performance Evaluation Criteria and Statistics	Performance Analysis with Mambo
SPU Rotate and Mask	Cell Broadband Engine Programming Handbook
SPU Shift and Rotate Instructions	SPU Instruction Set Archtecture
SPU Synchronization and Ordering	Cell Broadband Engine Programming Handbook

Topic	Cell BE SDK 3.0 Documentation
Storage Access Ordering	Cell Broadband Engine Architecture Cell Broadband Engine Programming Handbook PowerPC Virtual Environment Architecture - Book II
Storage Models	Cell Broadband Engine Architecture
Sync Library	Example Library API Reference
Synergistic Processor Elements	Cell Broadband Engine Programming Handbook Cell Broadband Engine Architecture
Synergistic Processor Unit	SPU Instruction Set Archtecture Cell Broadband Engine Architecture
Synergistic Processor Unit Channels	Cell Broadband Engine Architecture
Time Base and Decrementers	Cell Broadband Engine Programming Handbook
User Mode Environment	Cell Broadband Engine Architecture
Vector Library	Example Library API Reference
Vector/SIMD Multimedia Extension and SPU Programming	Cell Broadband Engine Programming Handbook
Virtual Storage Environment	Cell Broadband Engine Programming Handbook

The Cell BE SDK 3.0 documentation is installed as part of the install package regardless of the product selected. The following is a list of the online documentation.

Software Development Kit (SDK) 3.0 for Multicore Acceleration

- IBM SDK for Multicore Acceleration Installation Guide
- Cell Broadband Engine Programming Tutorial
- Cell Broadband Engine Programming Handbook
- Security SDK Installation and User's Guide

Programming Tools and Standards

- C/C++ Language Extensions for Cell BE Architecture
- IBM Full-System Simulator Users Guide and Performance Analysis
- IBM XL C/C++ single-source compiler
- SPU Application Binary Interface Specification
- SIMD Math Library Specification
- Cell BE Linux Reference Implementation Application Binary Interface Specification

- SPU Assembly Language Specification

Programming Library Documentation

- ALF Programmer's Guide and API Reference For Cell
- ALF Programmer's Guide and API Reference For Cell For Hybrid-x86 (P)
- BLAS Programmer's Guide and API Reference (P)
- DACS Programmer's Guide and API Reference For Cell
- DACS Programmer's Guide and API Reference- For Hybrid-x86 (prototype)
- Example Library API Reference
- Monte Carlo Library API Reference Manual (prototype)
- SPE Runtime Management library
- SPE Runtime Management Library Version 1.2 to 2.2 Migration Guide
- SPU Timer Library (prototype)

Hardware Documentation

- PowerPC User Instruction Set Architecture - Book I
- PowerPC Virtual Environment Architecture - Book II
- PowerPC Operating Environment Architecture - Book III
- Vector/SIMD Multimedia Extension Technology Programming Environments Manual
- Cell Broadband Engine Architecture
- Cell Broadband Engine Registers
- Synergistic Processor Unit (SPU) InSample caption

B

Additional material

This book refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser at:

ftp://www.redbooks.ibm.com/redbooks/SG247575

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG247575.

Using the Web material

The additional Web material that accompanies this book includes the following files:

- **SG247575_addmat.zip**

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.

Additional material content

The additional materials file for this book is structured as shown in below.

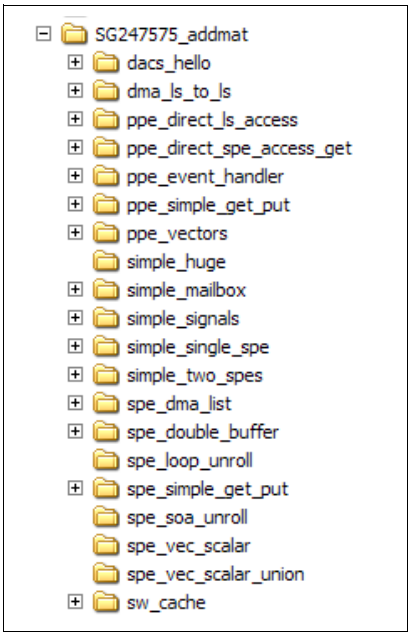


Figure B-1 Contents of additional materials file SG247575_addmat.zip

In the rest of this appendix we describe each of the contents of the additional material examples in more detail.

DaCS programming example

This section describe a DaCS code example which is related to the Chapter 4.7.1, “DaCS - Data Communication and Synchronization” on page 284.

DaCS synthetic example

File name: `dacs_hello_ide.tar`

Description: The tar file contains an IDE project in a format suitable for import using the `File->Import` menu in Eclipse. This code calls almost all the DaCS functions and is both an introduction to the API and a good way to check that DaCS is functioning correctly.

Although this is an IDE project, it can also be compiled outside Eclipse using the make command.

The contents of the dacs_hello.tar is listed below.

Example: B-1 dacs_hello.tar contents

```
dacs_hello/.cdtproject
dacs_hello/.project
dacs_hello/.settings/org.eclipse.cdt.core.prefs
dacs_hello/Makefile
dacs_hello/Makefile.example
dacs_hello/README.txt
dacs_hello/dacs_hello.c
dacs_hello/dacs_hello.h
dacs_hello/spu/Makefile
dacs_hello/spu/Makefile.example
dacs_hello/spu/dacs_hello_spu.c
```

Task parallelism and PPE programming examples

This section describe code example which are related to the 4.1, “Task parallelism and PPE programming” on page 78.

Simple PPU vector/SIMD code

- Directory name: `ppe_vectors`
- Description: A code that demonstrate some simple vector/SIMD insructions on a PPU program.
- Related book example: Example 4-1 on page 81

Running a single SPE

Directory name: `simple_single_spe`

Description: A code that demonstrate how a PPU program can insitate a single SPE thread.

Related book examples: Example 4-2 on page 86, Example 4-3 on page 86, and Example 4-4 on page 88.

Running multiple SPEs concurrently

Directory name: *simple_two_spes*

Description: A code that demonstrate how a PPU program can insitate a multiple SPE threads concurrently. The program execute two threads but can easily extended to use more threads.

Related book examples: Example 4-5 on page 90 and Example 4-6 on page 92.

Data transfer examples

This section describes code examples which are related to 4.3, “Data transfer” on page 109.

Direct SPE access ‘get’ example

Directory name: `ppe_direct_spe_access_get`

Description: A PPU program that uses ordinary load and store instructions to directly access the problem state of some SPE and intiate DMA ‘get’ command.

Related book example: Example 4-14 on page 106

SPU initiated basic DMA between LS and main storage

Directory name: `spe_simple_get_put`

Description: Demonstrate how SPU program initiated DMA transfers between LS and main storage. This example also shows how to use the stall-and-notify mechanism of the DMA, and implementing event handler on the SPU program to handle the stall-and-notify events.

Related book examples: Example 4-16 on page 122 and Example 4-17 on page 123

SPU initiated DMA list transfers between LS and main storage

Directory name: `spe_dma_list`

Description: Demonstrate how SPU program initiated DMA list transfers between LS and main storage. This example also shows how to use the stall-and-notify

mechanism of the DMA, and implementing event handler on the SPU program to handle the stall-and-notify events.

Related book examples: Example 4-19 on page 128 and Example 4-20 on page 129

PPU initiated DMA transfers between LS and main storage

Directory name: `ppe_simple_get_put`

Description: A PPU program that intiate DMA transfers between LS and main storage.

Related book examples: Example 4-22 on page 141 and Example 4-23 on page 142

Direct PPE access to LS of some SPE

Directory name: `ppe_direct_ls_access`

Description: A PPU program that uses ordinary load and store instructions to directly access the local store of some SPE.

Related book example: Example 4-24 on page 144

Multistage pipeline using LS to LS DMA transfer

Directory name: `dma_ls_to_ls`

Description: Uses DMA transfer between LS to LS to implement a multistage pipeline programming mode. This example doesn't pretend to provide the most optimized multistage pipeline model but just to demonstrate potential usage of the LS to LS transfer and potenitally a starting point for developing a highly optimized multistage pipeline model.

Related book example: None.

SPU software managed cache

Directory name: `sw_cache`

Description: Demonstrate how SPU program to intiate the software managed cache and later use it either to perform synchronous data access using safe mode, or to perform asynchronous data access using unsafe mode.

Related book examples: Example 4-25 on page 150, Example 4-26 on page 151, and Example 4-27 on page 152.

Double buffering

Directory name: `spe_double_buffer`

Description: A SPU program that implemented double buffering mechanism.

Related book examples: Example 4-28 on page 158, Example 4-29 on page 159, and Example 4-30 on page 161

Huge pages

Directory name: `simple_huge`

Description: A PPU program that uses buffers that are allocated on huge pages.

Related book example: Example 4-33 on page 167.

Inter-processor communication examples

This section describe code example which are related to the Chapter 4.4, “Inter-processor communication” on page 174.

Simple mailbox

Directory name: `simple_mailbox`

Description: A simple PPU and SPU program that demonstrate how to use the SPE inbouind and outbound mailboxes.

Related book examples: Example 4-35 on page 183, Example 4-36 on page 185, and Example 4-39 on page 195.

Simple signals

Directory name: `simple_signals`

Description: A simple PPU and SPU program that demonstrate how to use the SPE singal notification mechanism.

Related book examples: Example 4-37 on page 191, Example 4-38 on page 194, Example 4-39 on page 195, Example 4-40 on page 198.

PPE event handler

Directory name: `ppe_event_handler`

Description: An example of PPU program that implement an event handler that handles several SPE events.

Related book example: Example 4-41 on page 204.

SPU programming examples

This section describe code example which are related to the Chapter 4.6, “SPU programming” on page 240

SPE loop unrolling

Directory name: `spe_loop_unroll`

Description: A SPU program that demonstrate how to do the loop unrolling technique to schieve better performance.

Related book example: Example 4-56 on page 260

SPE SOA loop unrolling

Directory name: `spe_soa_unroll`

Description: A SPU program that demonstrate how to do the loop unrolling using SOA data organization to schieve better performance.

Related book example: Example 4-57 on page 263

SPE scalar to vector conversion using `insert` and `extract` intrinsics

Directory name: `spe_vec_scalar`

Description: A SPU program that demonstrate how to cluster several scalars into vector using `spu_insert` intrinsics, perform some SIMD operations on them and extract them back to their scalar shape using `spu_extract` intrinsic.

Related book example: Example 4-63 on page 273

SPE scalar to vector conversion using unions

Directory name: `spe_vec_scalar_union`

Description: A SPU program that demonstrate how to cluster several scalars into vector using unions.

Related book examples: Example 4-64 on page 275 and Example 4-65 on page 276.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

This is the first IBM Redbook on the Cell Broadband Engine. There are no other related IBM Redbooks or Redpieces at this time.

Other publications

These publications are also relevant as further information sources:

1. K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, K. Yelik. "The Landscape of Parallel Computing Research: A View from Berkeley". Technical report, EECS Department, University of California at Berkeley, UCB/EECS-2006-183, December, 2006.
2. Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, Katherine Yelick. "Scientific Computing Kernels on the Cell Processor". International Journal of Parallel Computing, 2007.
3. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. “Design Patterns. Elements of Reusable Object-Oriented Software”. Addison Wesley, 1994.
4. Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill. “Patterns for Parallel Programming”. Addison Wesley, 2004.
5. Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, Toshio Nakatani. “AA-sort : A new parallel sorting algorithm for multi-core SIMD processors”. International Conference on Parallel Architecture and Compilation Techniques, 2007.
6. Marc Snir. “Programming design patterns, patterns for high performance computing”. http://www.cs.uiuc.edu/homes/snir/PDF/Dagstuhl.pdf, Feb 2006.
7. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra. "MPI: The Complete Reference". Massachusetts Institute of Technology, 1996.
8. Phillip Colella. “Defining software requirements for scientic computing”. 2004.

9. P. Dubey. “Recognition, Mining and Synthesis Moves Computers to he Era of Tera”, Technology@Intel Magazine. Feb, 2005.
10. Makoto Matsumoto and Takaji Nishimura, Dynamic Creation of Pseudorandom Number Generators.http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dgene.pdf
11. M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3-30 (1998)
12. Glasserman, Paul , “Monte Carlo Methods in Financial Engineering”, Springer Verlag, October 2003.
13. http://dl.alphaworks.ibm.com/technologies/cellsw/cellFMwhitepaper.pdf
14. Daniel A. Brokenshire, "Maximizing the power of the Cell Broadband Engine processor: 25 tips to optimal application performance developerWorks", http://www-128.ibm.com/developerworks/power/library/pa-celltips1
15. IDC, “Solutions for the data center’s thermal challenges”, http://www-03.ibm.com/systems/pdf/IDC_Cool_Blue_Whitepaper_1-24-2007.pdf,
16. Kursad Albayraktaroglu, Jizhu Lu, Michael Perrone, Manoj Franklin, “Biological sequence analysis on the Cell BE. HMMer-Cell”, 2007, http://sti.cc.gatech.edu/Slides/Lu-070619.pdf,
17. Christopher Mueller, “Synthetic programming on the Cell BE”, 2006, http://www.cs.utk.edu/~dongarra/cell2006/cell-slides/06-Chris-Mueller.pdf,
18. Digital Medics, “Multigrid Finite Element Solver.”, 2006, http://www.digitalmedics.de/projects/mfes,
19. Thomas Chen, Ram Raghavan, Jason Dale, Eiji Iwata, “Cell Broadband engine Architecture and its first implementation, a performance view”, november 2005, http://www.ibm.com/developerworks/power/library/pa-cellperf/
20. David Kunzmann, Gengbin Zhang, Eric Bohm, Laxmikant V. Kale, ”Charm++, offload api and the Cell processor”, 2006, http://charm.c.uiuc.edu/papers/CellPMUP06.pdf,
21. SIMD Math Function Library reference, http://www.ibm.com/chips/techlib/techlib.nsf/techdocs/6DFAEFEDE179041E872572420078236 7?S_TACT=105AGX16&S_CMP=LP
22. Mathematical Acceleration Subsystem, http://www.ibm.com/support/search.wss?rs=2021&tc=SSVKBV&q=mass_cbeppu_docs&rankfile=08
23. Monte Carlo Library API Reference, http://www.ibm.com/chips/techlib/techlib.nsf/techdocs/8D78C965B984D1DE00257353006590B7?S_TACT=105AGX16&S_CMP=LP
24. Domingo Tavella, Quantitative Methods in Derivatives Pricing, John Wiley & Sons, Inc., 2002
25. Mike Acton, Eric Christensen, “Developing technology for ratchet and clank future : tools of destruction”, http://sti.cc.gatech.edu/Slides/Acton-070619.pdf, june 2007.

26. Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, James Demmel, “Optimization of sparse matrix-vector multiplication on emerging multicore platforms”, Supercomputing 2007.
27. Eric Christensen, Mike Acton, “Dynamic Code Uploading on the SPU”, http://www.insomniacgames.com/tech/articles/0807/files/dynamic_spu_code.txt, may 2007.

Online resources

These Web sites are also relevant as further information sources:

- Cell BE site on IBM DeveloperWorks with complete documentation

  http://www-128.ibm.com/developerworks/power/cell/
- Distributed Image Management (DIM) on Alphaworks

  http://alphaworks.ibm.com/tech/dim
- Extreme Cluster Adminstration Toolcat (xcat)

  http://www.alphaworks.ibm.com/tech/xCAT/
- oprofile on sourceforge.net
  http://oprofile.sourceforge.net
- IBM Dynamic Application Virtualization
  http://www.alphaworks.ibm.com/tech/dav
- MASS web site
  http://www.ibm.com/software/awdtools/mass

How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

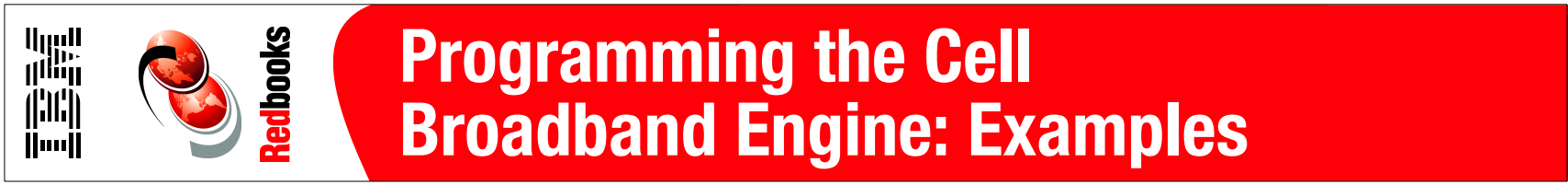
W

X

Y

Z

(1.5" spine)
1.5"<-> 1.998"
789 <->1051 pages

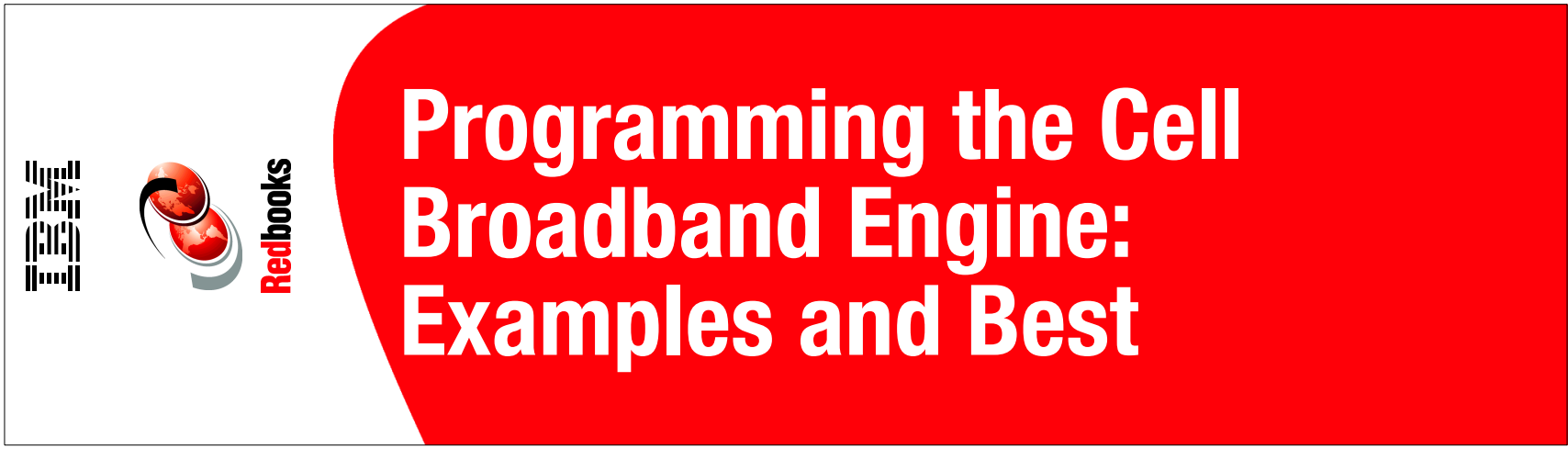


(1.0" spine)
0.875"<->1.498"
460 <-> 788 pages

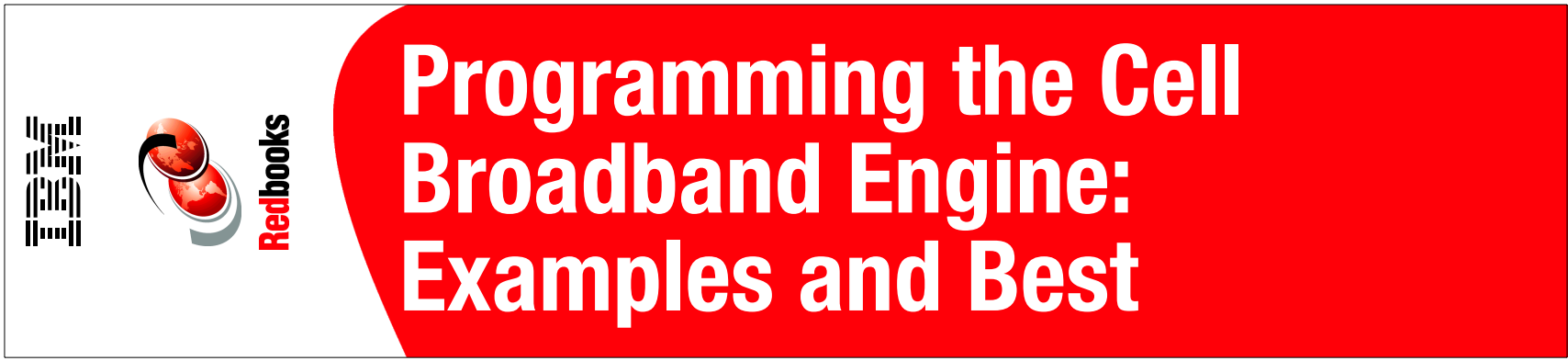
(0.5" spine)
0.475"<->0.875"
250 <-> 459 pages

(0.2"spine)
0.17"<->0.473"
90<->249 pages

(0.1"spine)
0.1"<->0.169"
53<->89 pages



(2.5” spine)
2.5”<->nnn.n”
1315<-> nnnn pages



(2.0” spine)
2.0” <-> 2.498”
1052 <-> 1314 pages

Draft Document for Review February 15, 2008 5:00 pm

Programming the Cell Broadband Engine Examples and Best

Practical code development and porting examples included

Make the most of SDK 3.0 debug and performance tools

Understand and apply different programming models and strategies

This Redbook will provide an examples driven programming manual that defines and illustrates various best practice development strategies, using the latest Cell BE SDK, the Full-System Simulator, and actual Cell BE systems, to illustrate how to develop both libraries and applications. This Redbook should compliment the library of existing Cell BE Programming Manuals, tutorials, and other resources with a more practical reference illustrating the most recent methods for leveraging the platform, with special emphasis on industry specific end-to-end examples, including debugging methods. This Redbook should lay the foundation for version 1 of a programmer's handbook. A handbook that could be updated as each version of the Cell BE SDK is released, along with major revisions to the hardware platform(s). SDK programs and sample code developed to demonstrate programming methods in this book are available for download.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7575-00 ISBN