Automatic conversion of source code for multi core architecture

By

Malay A. Andhariya

Roll No: 09MCE001

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

May, 2011

Automatic conversion of source code for multi core architecture

Major Project

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Computer Science and Engineering

By

Malay A. Andhariya

Roll No: 09MCE001

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

May, 2011

Declaration

This is to certify that

i) The thesis comprises my original work towards the degree of Master of Technology in Computer Science and Engineering at Nirma University and has not been submitted elsewhere for a degree.

ii) Due acknowledgement has been made in the text to all other material used.

Malay A. Andhariya

Certificate

This is to certify that the Major Project entitled "Automatic conversion of source code for multi core architecture" submitted by Malay A. Andhariya (09MCE001), towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science and Engineering of Nirma University, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Dr. S.N. Pradhan
Guide, Professor and PG-Coordinator,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Prof. D. J. Patel
Professor and Head,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Dr K Kotecha
Director,
Institute of Technology,
Nirma University, Ahmedabad

Abstract

As personal computers have become more prevalent and more applications have been designed for them, the end-user has seen the need for a faster, more capable system to keep up with new applications. Speedup has been achieved by increasing clock speeds and, more recently, adding multiple processing cores to the same chip. The clock speeds of processors have reached physical limit, so multi cores are seen as viable way to increase the performance. IBM, in collaboration with Sony Computer Entertainment and Toshiba Corporation, established a relationship with the primary objective of developing a new processor with dramatically increased performance, responsiveness and security. The result of the relationship is Cell Broadband Engine (Cell/B.E.) technology, an advanced IBM Power Architecture technology-based microprocessor.

The Cell/B.E. processor is an asymmetric multi-core processor that is optimized for parallel processing and streaming applications. Though originally developed for gaming application, it is found to be useful for general purpose computing. The problem is, there is no application in market readily available with these library functions and can use all the processor of the architecture. The main objective of this thesis is to develop software, which is able to identify independent execution tasks and converts automatically in such a way that the modified code executes parallel on different cores and give the better performance.

This thesis work is mainly concentrated on the computational part of the source code. Report contains general study of the Cell BE architecture, SDK tool kit, different methods to get performance benefit, implementation of intermediate tool to find out functions and their dependencies from the source code and the implementation of the complete algorithm, testing and analysis of the developed software.

Acknowledgements

It gives me great pleasure in expressing thanks and profound gratitude to my guide **Dr. S.N. Pradhan,** Professor & M.Tech Coordinator, Department of Computer Science and Engineering, Institute of Technology, Nirma University, Ahmedabad for his valuable guidance and continual encouragement throughout part one of the Major project. I heartily thankful to him for his time to time suggestions and the clarity of the concepts of the topic that helped me a lot during this study.

I would like to extend my gratitude to **Prof. D.J.Patel,** H.O.D., Department of Computer Science and Engineering, Institute of Technology, Nirma University, Ahmedabad for fruitful discussions and valuable suggestions during meetings and for their encouragement.

I would like to thanks **Dr. Ketan Kotecha,** Honorable Director,Institute of Technology, Nirma University, Ahmedabad for providing basic infrastructure and healthy research environment.

I would also thank my Institution, all my faculty members in Department of Computer Science and my colleagues without whom this project would have been a distant reality. Last, but not the least, no words are enough to acknowledge constant support and sacrifices of my family members because of whom I am able to complete my dissertation work successfully.

- Malay A. Andhariya

09MCE001

Contents

List of Tables

List of Figures

Abbreviations

ALF	Accelerated Library Framework
BLAS	Basic Linear Algebra Subprograms
CellBE	Cell Broadband Engine
DaCS	Data Communication and Synchronization
DMA	Dynamic Memory Allocation
EDP	Enhanced Double Precision
EIB	Element Interconnect Bus
GPU	graphics processing units
MFC	Memory Flow Controller
MPMD	multiple-program-multiple-data
PPE	Power Processor Element
SDK	Software Development Toolkit
SIMD	Single Instruction Multiple Data
SPE	Synergistic Processor Elements

Chapter 1

Introduction

IBM, in collaboration with Sony Computer Entertainment and Toshiba Corporation, established a relationship with the primary objective of developing a new processor with dramatically increased performance, responsiveness and security. The result of the relationship is technology, an advanced IBM Power Architecture technology-based microprocessor. This report contains general study CellBE architecture, IBM toolkit to program for this architecture, libraries and system call, different methods for performance optimization on Cell BE, explanation of the definition, Intermediate tool to identify functions and their dependency in a source code, implementation of the complete software.

This report also contains testing and performance evaluation of the tool which has been used in the software to identify function and their dependency from any C source code. It also includes testing of the complete software and its performance analysis.

1.1 Objective of the Work

The objective of this research is to provide a better utilization of the resources of Cell BE to obtain better performance and to save manual efforts and time to learn the SDK and to prepare a program for Cell BE.

1.2 Scope of the Work

The scope of this work is to optimize Processor elements and to reduce the computation time with the play station 3 and other IBM servers which are using this architecture. Same method of developing the software can be applied for different multi core architecture also.

1.3 Motivation of the Work

- To identify the independent computational tasks from the given C/C++ source code.
- Increase the computation power by distributing the independent tasks to all the available processing elements.

1.4 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2, *Detail study of the CellBE architecture*, design goals, performance, advantages and application of it like PS3.

Chapter 3, *Programming concepts with CellBE*, SDK 3.0, components of SDK, CellBE libraries its installation.

chapter 4, *Explanation of the Definition*, methods for performance optimization, steps to be performed with the selected method, benefit of the software.

chapter 5, *Implementation of Dependency analysis tool*, Types of dependency, pseudo code, functions developed, output, dependency graph.

chapter 6, *Implementation of Automatic code converting software*, Algorithm, outputs, performance evaluation and limitations.

chapter 7, *Conclusion and Future work*, concluding remarks and future work is presented.

Chapter 2

Cell BE Architecture

IBM, in collaboration with Sony Computer Entertainment and Toshiba Corporation, established a relationship with the primary objective of developing a new processor with dramatically increased performance, responsiveness and security. The results of the relationship is Cell Broadband Engine (Cell/B.E.) technology, an advanced IBM Power Architecture technology-based microprocessor.

2.1 Design goals

The group understood that conventional microprocessors have performance limitations and traditional improvements were not going to meet current and future demands for greater processor performance. Another equally important goal was to overcome the increasing power consumption, cooling resources and floor space needed by systems based on conventional microprocessors that were trying to meet the ever-increasing processing demands of organizations.

The Cell/B.E. processor is an asymmetric multi-core processor that is optimized for parallel processing and streaming applications. Unlike symmetric multi-core, cache-based architectures which may not be able to efficiently handle streaming applications, Cell/B.E. technology is designed to offer very high performance and fast response. The Cell/B.E. processor includes a and eight highly optimized engines called .

Cell/B.E. processor performance is about an order of magnitude better than traditional processors for applications that can take advantage of its SIMD capability. The PPE is intended to run the operating system and coordinate computation. Each SPE is able to perform mostly the same as, or better than, a General Purpose Processor with SIMD running at the same frequency. A key performance advantage comes from its eight decoupled eDP SPE SIMD engines with dedicated resources including large register files and channels.

Another important design difference is the memory architecture. Cell/B.E processor memory architecture uses dedicated, on-die local memory storage for each SPE with a separate unit for managing data transfers, plus the use of XDR RAMBUS memory that can operate at CPU clock speeds (>3GHz compared to <1GHz of DDR RAM used in PCs), eliminating the need for large L2 cache memory and data latency.

2.2 Architecture

The following figure shows the architecture block diagram of the Cell/B.E. processor. The PPE accesses the main memory via the L1 and L2 cache in the traditional way, thus accessing the whole main memory. There is a latency penalty caused by two layer caching. However, the SPEs can only operate directly on their own 256 KB local memory. Each SPE has a programmable MFC that allows fast data transfer between the main memory and the local store. The SPEs can also communicate with each other and with the PPE via the very high speed EIB. Most of the compute power of the Cell/B.E. processor comes from the SPEs. Each SPE is capable of executing up to 8 floating point or 32-bit integer operations per cycle, a total of 64 operations in a single cycle. Alternatively, they can execute 128 16-bit operations or 256 8-bit operations in parallel. A good application design for Cell/B.E. technology uses the PPE for main control and disk I/O tasks, while all computation intensive tasks are distributed to the SPEs.

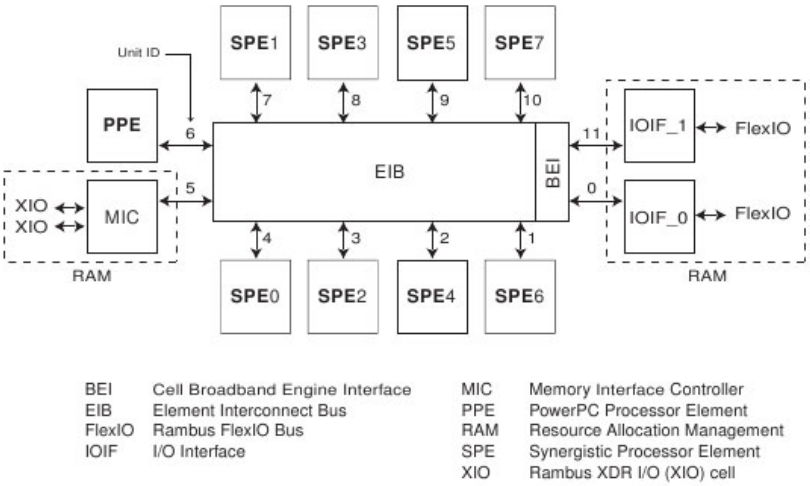

Figure 2.1: A block diagram of the Cell/B.E. processor[8]

2.3 Performance

Performance specification highlights of the Cell/B.E. processor [1]:

- 241 million transistors
- 9 cores, 10 parallel execution threads
- > 230 GFlops single precision
- Up to 25 GB/s memory bandwidth
- Up to 75 GB/s I/O bandwidth
- > 230 GB/s bandwidth on the EIB
- Frequency > 4GHz

2.4 Advantages

It helps companies to overcome the limitations of today's microprocessors and targets current and future compute-, data- and memory-intensive application workloads.

Currently, Cell/B.E. technology is available from IBM on select IBM BladeCenter products and from IBM Business Partners in industry-standard form factors. The Cell/B.E. is a microprocessor architecture jointly developed by Sony, Toshiba, and IBM. Originally used in cutting edge gaming applications to speed up physics simulations and catch up with the 3D graphics rendering speeds provided by advanced , Cell/B.E. architecture has now been integrated into enterprise-class technology systems.

The Cell/B.E. processor's breakthrough multi-core architecture and ultra high-speed communications capabilities deliver vastly improved, real-time response. Effectively delivering "supercomputer-like performance" by incorporating advanced multiprocessing technologies used in sophisticated IBM servers, Cell/B.E. is especially suitable for high-performance workloads. Matching the superior performance of the eHiTS algorithms with the acceleration derived from using the Cell/B.E processor allows the eHiTS Lightning application to provide unprecedented accuracy and throughput for flexible ligand docking to the life sciences Industry.

The following table shows execution times and performance improvement reached for one example program, the Lennard-Jones 6-12 Potential calculation, part of the eHiTS program. The next table illustrates the performance gains achieved in various

System	**PPU only - Power Mac**	**Dual core Intel Xeon processor 2.4 GHz**	**IBM Bland center**
Time	3m34s	1m14s	1.0s
Performance improvement	1x	3x	214x

Table 2.1: execution times and performance improvement[8]

calculations on the IBM BladeCenter QS21 with two 3.2GHz Cell/B.E. processors compared to traditional CPUs at equivalent clock speeds.

eHiTS Scoring	112x
Rigid fragment Docking	58x
Pose matching	37x
Flexible Chain Fitting	92x
Conformation Minimization	84x
Final Optimization	55x
Total (Complete flexible Docking)	62x

Table 2.2: performance gains achieved in various calculations[8]

2.5 Application - PLAYSTATION 3

The PLAYSTATION 3 is unusual for a gaming console for two reasons. First, it is incredibly more open than any previous console. While most consoles do everything possible to prevent unauthorized games from being playable on their system, the PS3 goes in the other direction, even providing direct support for installing and booting foreign operating systems. Of course, many of the game-related features such as video acceleration are locked out for the third-party operating systems, but this series focuses on more general-purpose and scientific applications anyway.

The real centerpiece for the PS3, however, is its processor – the Cell Broadband Engine chip (often called the Cell BE chip). The Cell BE architecture is a radical departure from traditional processor designs. The Cell BE processor is a chip consisting of nine processing elements (note the PS3 has one of them disabled, and one of them reserved for system use, leaving seven processing units at your disposal). The main processing element is a fairly standard general-purpose processor. It is a dual-thread Power Architecture element, called the Power Processing Element, or PPE for short. The other eight processing elements, however, are a different story.

The other processing elements within the Cell BE are known as Synergistic Processing Elements, or SPEs. Each SPE consists of:

- A vector processor, called a Synergistic Processing Unit, or SPU
- A private memory area within the SPU called the local store (the size of this

area on the PS3 is 256K)

- A set of communication channels for dealing with the outside world
- A set of 128 registers, each 128 bits wide (each register is normally treated as holding four 32-bit values simultaneously)
- A Memory Flow Controller (MFC) which manages DMA transfers between the SPU's local store and main memory

While the Cell BE processor has been out for a while in specialized hardware, the PS3 is the first Cell BE-based device that has been affordable and readily available. And, with Linux, anyone who wants to can program it.

Chapter 3

Programming Concepts

The performance exhibited by the Cell/B.E. is not delivered simply by recompiling existing C/C++ code for the Cell/B.E. processor. There are several key differences in coding for the Cell/B.E. processor compared to traditional CPUs:

- The PPE and the SPE are not binary compatible. The code needs to be compiled with a different compiler to generate code fragments that run on the SPE. Simple POSIX threads cannot be scheduled by the OS to run on the SPE.
- A thread running on an SPE can only directly access the 256 KB local storage of that SPE. This small amount of memory must contain the code, the data and the execution stack at any one time. Of course, data or code can be shuffled back and forth between the main memory and the local store via DMA calls, but those have to be explicitly programmed and managed by the code. Furthermore, the DMA calls in the code have to be designed to use double buffering or similar tricks to streamline data and avoid stalling due to latency.
- The power of the SPEs comes from SIMD vector operations, where the same instruction is executed for multiple data entries. An SPE using dual pipe can execute 8 instructions per cycle. These are two different instructions each executed on 4 parallel data units and those data units must reside in a consecutive block of memory or in a single 128-bit register. High performance can therefore

only be reached if the data is organized in a very specific way that is suitable for SIMD calculations.

- SPEs do not have branch prediction hardware. This was one of the simplifications compared to traditional CPUs that allowed the engineers to pack more computation cores into the chip. The downside of this is that branches in the code are more costly and lead to a loss in pipeline efficiency. A single branch miss costs an 18-cycle delay and 144 (from 18x8) potential instruction executions are lost! Therefore, code must be written by minimizing branches. If a choice needs to be made, it is often worth computing both alternatives and selecting the most appropriate one from the results rather than inserting a branch to compute only what is necessary.

3.1 SDK 3.0[12]

SDK for Multicore Acceleration V3.0 contains the essential tools required for developing programs for the Cell/B.E. based server the IBM BladeCenter QS21 with libraries, frameworks, and application development tools. The Synergistic Processing Elements (SPE) will be used for offloading numerically intensive computing functions from more general-purpose x86 and IBM POWER-based processors.

The SDK is designed to allow customers, research institutions, and universities to port and optimize applications and algorithms quickly with existing software development staff.

Focusing on programmer productivity, the SDK includes tools for improving application development and performance optimization. Market segment enablement is offered through libraries such as BLAS.

The SDK is offered through IBM Passport Advantage with support through IBM support channels. Along with the SDK, a number of beta and prototype components are available.

The beta and prototype components found on developerWorks are available under

the International License for Early Release of Programs.
Four development platforms can be used to facilitate application development: x86, x86fi64, PPC64, and Cell/B.E. Architecture.
The SDK V3 is closely aligned with Red Hat Enterprise Linux 5 Update 1.

3.2 The SDK contains the following components

- An Eclipse-based IDE for building, compiling, and debugging applications leveraging the compilers, programming model frameworks, and analysis tools of the SDK.

- Development libraries and frameworks:

  - The ALF provides a programming environment for data and task parallel applications and libraries. The ALF API is designed to provide library developers a set of interfaces to simplify library development on heterogeneous multicore systems. Library developers can use the framework to offload the computationally intensive work to the accelerators, facilitating the development of more complex applications by combining several function offload libraries.

  - The DaCS library provides a set of services designed to ease the development of applications and application frameworks in a heterogeneous multitiered (for example, memory hierarchy) system. The DaCS services are implemented as a set of APIs that provide a layer of architectural neutrality for application developers on a variety of multicore memory hierarchy systems.

  - BLAS is a widely used API for commonly used linear algebra operations in HPC and other scientific domains. BLAS is widely used as the basis for other high-quality linear algebra software.

  - SIMD math libraries are available for the PowerPC Processor Element (PPE) Vector/SIMD Multimedia Extension and the Synergistic Processing Unit (SPU).

- Performance tools:

  - The FDPR-Pro tool gathers information for feedback-directed optimization through static code analysis.
  - Performance Debugging Tool provides tools to analyze the execution of Cell/B.E. applications and track problems to optimize performance.

- Example source code:

  - Examples, libraries, demos, and code to demonstrate the use of tools, libraries, and hardware features are available. A tutorial to guide the user through the creation of an example application is also included.

- Mathematical Acceleration Subsystem (MASS) consists of libraries of mathematical intrinsic functions tuned for optimum performance on SPE and PPE. These libraries offer improved performance over the standard mathematical library routines, are thread-safe, and can be used by C, C++, and Fortran applications. The PPE libraries support both 32-bit and 64-bit compilations.

- Compiler:

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

- IBM XL C/C++ compiler 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 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 PowerPC Processor Element (PPE) and Synergistic Processor Element (SPE).
- GNU ADA compiler comes in an existing Cell/B.E. processor and an x86 cross-compiler. The initial version of this compiler supports code generation for the PPU processor.
- IBM XL Fortran for Multicore Acceleration for Linux, the latest addition to the IBM XL family of compilers, is a cross compiler. It adopts proven high-performance compiler technologies that are used in its compiler family predecessors. It also adds new features that are tailored to exploit the unique performance capabilities of processors that are compliant with the new CBEA.

3.3 CellBE Libraries[12]

- The SPE Runtime Management Library (libspe) constitutes the standardized low-level application programming interface (API) for application access to the Cell/B.E. SPEs. This library provides an API to manage SPEs that are 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. Their use may lead to non-portable code and dependencies on certain implemented versions of the library.

- 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 that are available in both the PPU and SPU in the CBEA. 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.

- The Mathematical Acceleration Subsystem (MASS) consists of libraries of mathematical intrinsic functions, which are tuned specifically for optimum performance on the Cell/B.E. processor. Currently the 32-bit, 64-bit PPU, and SPU libraries are supported. These libraries offer the following advantages:

  - Include both scalar and vector functions, are thread-safe, and support both 32-bit 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

- The library:

  - The Basic Linear Algebra Subprograms (BLAS) 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 IBM SDK for Multicore Acceleration supports only real single-precision (SP) and real double-precision (DP) versions. All SP and DP routines in the three levels of standard BLAS are supported on the PPE. These routines are available as PPE APIs and conform to the standard BLAS interface.
- Some of these routines are optimized by using the SPEs and show a marked increase in performance compared to the corresponding versions that are implemented solely on the PPE. These optimized routines have an SPE interface in addition to the PPE interface. The SPE interface does not conform to, yet provides a restricted version of, the standard BLAS interface. Moreover, the single precision versions of these routines have been further optimized for maximum performance by using various features of the SPE.

- ALF Library:
  - The 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 heterogeneous multicore 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.
  - The ALF supports the programming module where multiple programs can be scheduled to run on multiple accelerator elements at the same time. The ALF library includes the following functionality:
    * Data transfer management

    * Parallel task management
    * Double buffering and dynamic load balancing

- The library:
  - The Data Communication and Synchronization (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, the library includes the following services:
    * 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 that provide 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). HEs usually run a full operating system and submit work to the specialized processes that run in the AEs.

3.4 Installation of SDK[1]

3.4.1 Getting source:

Grab the following from IBM DeveloperWorks, as directed in a directory e.g. /tmp/sdkfiles/

- cell-install-3.1.0-0.0.noarch.rpm

- oCellSDK-Devel-RHELfi3.1.0.0.0.iso
- CellSDK-Extras-RHELfi3.1.0.0.0.iso

3.4.2 Prepare to install:

Apply the commands

- rpm -ivh /tmp/sdkfiles/cell-install-3.1.0-0.0.noarch.rpm
- /opt/cell/cellsdk uninstall

That did some clean-ups/housekeeping. Redid the install binaries as instructed:

- yum install *ppu* *spu* numactl numactl-devel

3.4.3 Mounting:

Apply the commands

- mkdir /mnt/sdk
- mkdir /mnt/sdkextras

That creates mount point and to mount the installation files.

- mount -o loop /tmp/sdkfiles/CellSDK-Devel-RHELfi3.1.0.0.0.iso /mnt/sdk
- mount -o loop /tmp/sdkfiles/CellSDK-Extras-RHELfi3.1.0.0.0.iso /mnt/sdkextras

3.4.4 Installing the base SDK stuff:

Apply the commands

- cd /mnt/sdk/ppc64/
- yum –nogpgcheck localinstall *

That installs SDK and now to install extra packages.

- cd /mnt/sdkextras/ppc64/
- yum –nogpgcheck localinstall *

and finally unmount the image.

Chapter 4

Problem Definition

4.1 Methods for performance optimization[8]

There are numbers of way by which a programme can be written to get performance benefit of CellBE. They are explained as follows.

4.1.1 Vector data types intrinsics

The vector data types intrinsics is a set of intrinsics that is provided to support the VMX instructions, which follow the AltiVec standard.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.
VMX data types and Vector/SIMD Multimedia Extension intrinsics can be used in a seamless way throughout a C-language program. The programmer does not need to set up to enter a special mode. The intrinsics may be either defined as macros within the system header file or implemented internally within the compiler.

4.1.2 Task parallelism and managing SPE threads

Programs that run on the Cell/B.E. processor typically partition the work among the eight available SPEs since each SPE is assigned with a different task and data to work on. Regardless of the programming model, the main thread of the program is executed on the PPE, which creates sub-threads that run on the SPEs and off-load some function of the main program.
There are two main methods to load SPE programs:

- **Static loading of SPE object** Statically compile the SPE object within the PPE program. At run time, the object is accessed as an external pointer that can be used by the programmer to load the program into local storage. The loading itself is implemented internally by the library API by using DMA.

- **Dynamic loading of SPE object** 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 local storage of the SPE. This method is more flexible because you can decide, at run time, which program to load, for example, depending on the run time parameters. By using this method, you save linking the SPE program with the PPE program at the cost of lost encapsulation, so that the program is now a set of files, and not just a single executable.

4.2 Dynamic loading of SPE for task parallelism

This method best suites to this these as it loads SPE threads dynamically so whenever any individual function call identifies, we can call an SPE thread to compute that task independently. It is also very simple to call an SPE thread from PPE execution. It requires to create an image of the SPE function independently, Create a linux thread from PPE program and load that image in to the thread along with some required parameters.
Following are the steps to do the same:

- Find out the independent functions.
- Copy each individual function code into a separate file for SPEs.
- Copy the main code in to the main file for PPE.
- Create the image of the SPE files.
- Create the linux thread from the PPE program at whenever the function calls and copy the image of that SPE file.

4.3 Problem with the definition

As explained in the architecture, in CellBE each SPE and PPE has its own separate local storage and a common global storage memory. Following is the detailed storage of CellBe.

As shown in figure, all the storage spaces are connected to each other through a high

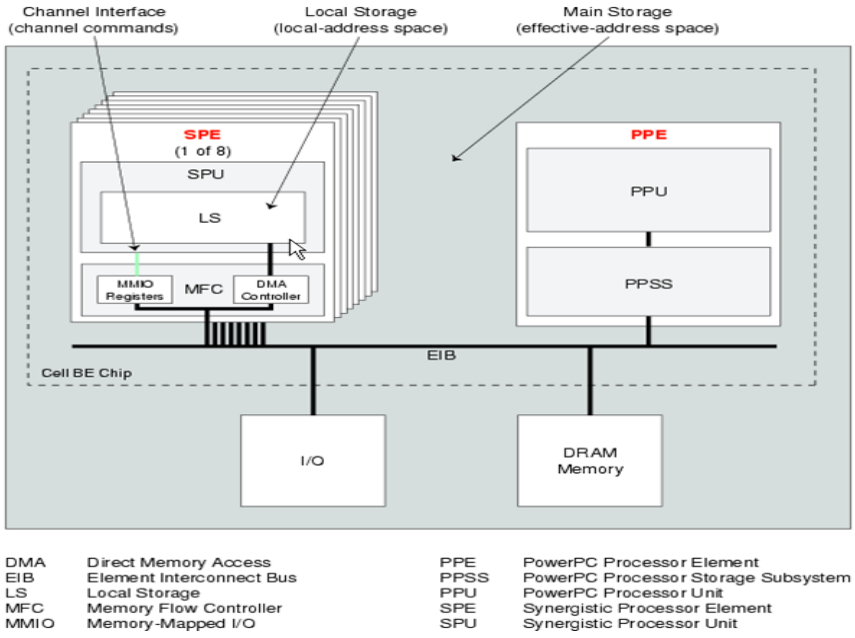

Figure 4.1: Storage of a CellBE architecture[12]

speed bus so that data transfer takes less amount of time. The main challenge is to copy required variables in to respective local storage of SPE and PPE.
To solve this problem this algorithm puts the global variables into a common header file and includes that file in each SPE and PPE file. It also finds out the local variables of each function and puts them into the the local storage space at run time.

4.4 Benefits of this approach

Cell BE architecture is one of the multi core architectures available to get the higher performance. This kind of architectures required a specific system calls and programming structure to get its performance benefit. For a developer, it is very difficult and a time consuming task to learn such new programming method. This software is developed to automatic convert the normal code written in C programming language into a code, which runs nicely on the Cell BE architecture to utilize its hardware features and get optimal performance. Same algorithm can be used to develop such softwares for different multi core architectures like CUDA to save the man power and time.

Chapter 5

Dependency analysis tool

This tool is developed to find out a functional dependency graph for a C program. Here for this thesis i have considered programs written in C programming language only.

5.1 Dependencies

There may many kind of dependencies in a program like, Functional dependency, Variable dependency, Module dependency, etc. This tool generates a functional dependency graph and doesn't consider any other kind of dependency.
There are mainly three kind of functional dependency.

a. **Direct Dependency** is a kind of dependency in which a function is called within another function e.g.

```
void fun1{
    ...
    fun2()
    ...
}
```

fun1 is directly dependent on fun2.

b. **Indirect Dependency** is a kind of dependency in which return value of a function is passed as an argument of another function e.g.

```
void fun1{
    ...
    a=fun2()
    fun3(a)
    ...
}
```

fun3 is indirectly dependent on fun2.

c. **Transitive Dependency** is a kind of dependency in which a function called in a function is already dependent on another function e.g.

```
void fun1{
    ...
    fun2()
    ...
}
void fun2{
    ...
    fun3()
    ...
}
```

fun1 is Transitively dependent on fun3.

5.2 Pseudo code

This tool is made to run on Yellow Dog Linux, so most of the coding has been done using C++ programming and Linux Shell scripts. This tool internally uses three kind of storage such as stack, 1D link list and 2D link list at different level, for different purposes. Following is the Pseudo code for the tool Which provides enough knowledge, how it works.

```
Read (main file of the software)
Get (all the included c files)
Write (into a stack)
Get (global variables)
Write (into a common header file)
WHILE stack is not empty
    Read (a file on the top of the stack)
    Get (find out the declared functions)
        IF it is a main function THAN
            Write (into main_ppu.c file)
        ELSE
            Write (into FUN_spu.c file)
        END IF
    Read (these files line by line)
    Get (variable declared in that line and find out dependant functions)
    Write (variables into a link list and functions in to a 2D link list)
END WHILE
```

5.3 Functions Developed

Following are the functions, developed within the tool.

- **RmvCmt** removes comments from a file, to consider only the part of program

which is going to be compiled.

- **FindFile** finds included .c files from a file so that later on it can scan all the .c files and find out functions and their dependency graph.

- **FindVar** finds variables declared in a file to put those variables in to the local memory of SPEs accordingly.

- **FindFcn** finds functions defined in a file and stores them in to a 1D link list.

- **FindDpn** finds functional dependency for each function and generate a graph.

5.4 Output

This tool gives two kind of output.

a. **Separate .c files** :- This tool will give a independent .c file for each function definition.

b. **Functional dependency graph** :- This tool also provides a functional dependency graph for each function declared in the program.

Following is the complete explanation of the output formate of this tool with the help of an example.

5.4.1 Input file

Here is the tst.c file which is given as the input to this tool.

```
#include<stdio.h>
int x;
void tst1(){
    int i,j;
```

```
    tst2();
    printf("Hi. this is test 1.\n");
}
void tst2(){
    int i,j;
    printf("Hi. this is test 2.\n");
}
int main(){
    int i,j;
    printf("Hi. this is main.\n");
    tst1();
    return 0;
}
```

This file has three member functions tst1, tst2 and main defined. tst1 is dependent on tst2, main is dependent on tst1 and tst2 is an independent function in this file.

5.4.2 Output files

Here are the separate .c files which this tool gives as output for the tst.c file described above.

- **tst2.c**

```
#include<stdio.h>
int x;
void tst2(){
    int i,j;
    printf("Hi. this is test 2.\n");
}
```

- **tst1.c**

```
#include<stdio.h>
int x;
void tst1(){
    int i,j;
    tst2();
    printf("Hi. this is test 1.\n");
}
```

- **main.c**

```
#include<stdio.h>
int x;
int main(){
    int i,j;
    printf("Hi. this is main.\n");
    tst1();
    return 0;
}
```

5.4.3 Dependency graph

Following is the dependency graph, in the form of 2D link list, generated for the file tst.c described above.

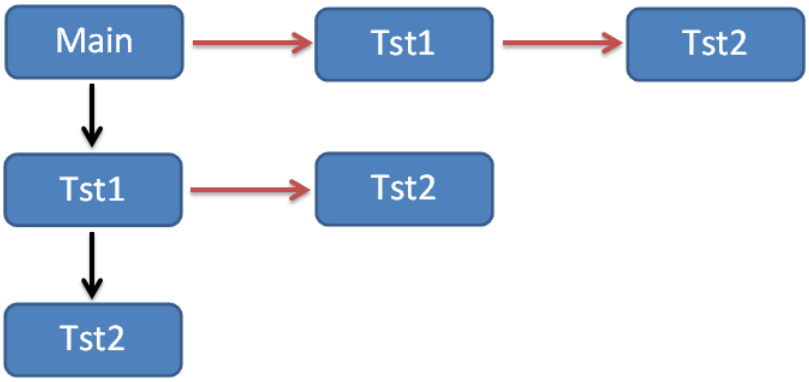

Figure 5.1: Dependency graph

5.5 Performance evaluation

This tool has been applied no of different c programs to evaluate its performance and the Following is the performance evaluation graph generated from the results.

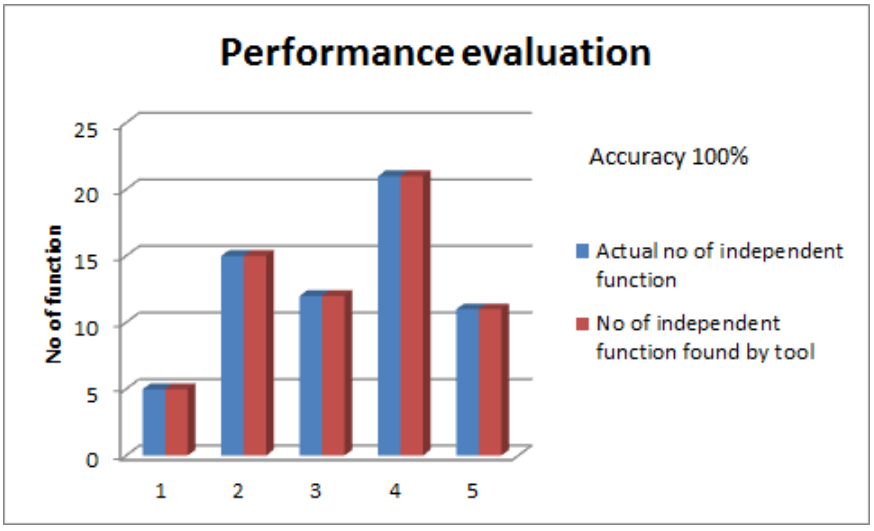

Figure 5.2: Performance evaluation

Chapter 6

Automatic code conversion

This algorithm is developed to convert a normal program written in c programming language which runs on a single CPU into a complete package including number of files which can run properly on a CellBE architecture and can get performance benefits of the architecture.

6.1 Algorithm

```
Read main file of the software
Find out all the included c files
Read all the c files one by one
    Find out the global variables and included files
    Write them into a common header file
    Find out declared functions
    If it is a main function Than
        Write it into main_ppu.c file
    Else
        Write it into FUN_spu.c file

Read all the FUN_spe.c files
```

```
Find out functional dependence
If there is a inter functional dependence Than
    Write the function definition into that FUN_spe.c file
else
    Leave the FUN_spe.c file as it is

Read main_ppu.c file line by line
Write appropriate SDK code at respective place
Find out the function calls in the definition
Call appropriate SPE threads instead of function calls

Include the header file to all the SPU and PPU files.

Compile all the FUN_spe.c files using SPU compiler
Compile the main_ppu.c file using PPU compiler
Run the main file
```

6.2 Output

This algorithm takes a .c file as an input and gives a complete package with number of files. Following are the input and output

6.2.1 Input file

Consider a program, with two files test1.c and test2.c is given as an input to this algorithm. Among these two files test1.c is the main file which includes test2.c file to use its functionality. These files are as follows.

- **test1.c**

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

```
#include "test2.c"
int x;
void tst1(){
    int i,j;
    tst3();
    for(i=0;i<9999;i++)
        for(j=0;j<9999;j++);
    printf("Hi. this is test 1.\n");
}
void tst2(){
    int i,j;
    tst3();
    for(i=0;i<9999;i++)
        for(j=0;j<9999;j++);
    printf("Hi. this is test 2.\n");
}
int main(){
    int i,j;
    tst1();
    tst2();
    for(i=0;i<9999;i++)
        for(j=0;j<9999;j++);
    printf("Hi. this is main.\n");
    return 0;
}
```

- **test2.c**

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

```
void tst3(){
    int i,j;
    for(i=0;i<9999;i++)
        for(j=0;j<9999;j++);
    printf("Hi. this is tst 3.\n");
}
```

There are three sub function and a main functions are declared. main function is dependent on functions tst1 and tst3, which are dependent on function tst2.

6.2.2 Output files

This algorithm gives a separate .c file for each function and one .sh file to compile the program. following are the output files.

- **tst1_spu.c**

```
#include "hdr_ppu.h"
#include <stdint.h>
#include <spu_intrinsics.h>
#include <spu_mfcio.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)));

void tst3(){
   int i,j;
   for(i=0;i<9999;i++)
       for(j=0;j<9999;j++);
```

```
   printf("Hi. this is tst 3.\n");
}

void tst2() {
    int i;
    printf("Hi. this is test 2.\n");
}

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

    int i,j;

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

    mfc_get((void*) &ctx, argp, sizeof(ctx), tag_id, 0, 0);
    mfc_write_tag_mask(1<<tag_id);
    mfc_read_tag_status_all();

    tst3();
    for(i=0;i<9999;i++)
        for(j=0;j<9999;j++);
    printf("Hi. this is tst 1.\n");

    mfc_tag_release(tag_id);
    return 0;
```

```
}
```

- **tst2_spu.c**

```
#include "hdr_ppu.h"
#include <stdint.h>
#include <spu_intrinsics.h>
#include <spu_mfcio.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)));

void tst3(){
    int i,j;
    for(i=0;i<9999;i++)
        for(j=0;j<9999;j++);
    printf("Hi. this is tst 3.\n");
}

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

    int i,j;

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

```
    mfc_get((void*) &ctx, argp, sizeof(ctx), tag_id, 0, 0);
    mfc_write_tag_mask(1<<tag_id);
    mfc_read_tag_status_all();
    tst3();
    for(i=0;i<9999;i++)
        for(j=0;j<9999;j++);
    printf("Hi. this is tst 2.\n");

    mfc_tag_release(tag_id);
    return 0;
}
```

- **main_ppu.c**

```
#include "hdr_ppu.h"
#include <stdint.h>
#include <libspe2.h>
#include </opt/cell/sdk/usr/include/cbe_mfc.h>
#include <pthread.h>
volatile char in_data[BUFF_SIZE] __attribute__ ((aligned(128)));
volatile char out_data[BUFF_SIZE] __attribute__ ((aligned(128)));
volatile parm_context ctx[NUM_SPES] __attribute__ ((aligned(16)));
spe_program_handle_t *program[BUFF_SIZE];
typedef struct spu_data {
    spe_context_ptr_t spe_ctx;
    pthread_t pthread;
    void *argp;
} spu_data_t;
spu_data_t data[NUM_SPES];
```

```
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);
}
int main(){
int ppunum;

int i,j;

char spe_names[NUM_SPES][20] = {"tst2","tst1"};
for( ppunum=0; ppunum<NUM_SPES; ppunum++){
   ctx[ppunum].ea_in = (uint64_t)in_data + ppunum*(BUFF_SIZE/NUM_SPES);
   ctx[ppunum].ea_out= (uint64_t)out_data + ppunum*(BUFF_SIZE/NUM_SPES);
   data[ppunum].argp = &ctx;
}
for( ppunum=0; ppunum<NUM_SPES; ppunum++){
   if ((data[ppunum].spe_ctx = spe_context_create (0, NULL)) == NULL) {
       perror("Failed creating context"); exit(1);}
   if (!(program[ppunum] = spe_image_open(&spe_names[ppunum][0]))) {
       perror("Fail opening image"); exit(1);}
   if (spe_program_load ( data[ppunum].spe_ctx, program[ppunum])) {
       perror("Failed loading program"); exit(1);}
}
```

```
for( ppunum=0; ppunum<NUM_SPES; ppunum++){
   if(pthread_create(&data[ppunum].pthread,NULL,&spu_pthread,&data[ppunum]))
       perror("Failed creating thread"); exit(1);}
}
for( ppunum=0; ppunum<NUM_SPES; ppunum++){
   if (pthread_join (data[ppunum].pthread, NULL)) {
       perror("Failed joining thread"); exit (1);}
   if (spe_context_destroy( data[ppunum].spe_ctx    )) {
       perror("Failed spe_context_destroy"); exit(1);}
}
for(i=0;i<9999;i++)
   for(j=0;j<9999;j++);
printf("Hi. this is main.\n");

printf(")PPE:) Complete running all super-fast SPEs"); return (0); }
```

- **hdr_ppu.h**

```
#include<stdio.h>
#include<stdint.h>
#define NUM_SPES 2

#define BUFF_SIZE 256
typedef struct{
    uint64_t ea_in;
    uint64_t ea_out;
} parm_context;
```

- **run.sh**

```
g++ find_var.cpp -o fndvar -Wno-deprecated
g++ find_fnc.cpp -o fndfcn -Wno-deprecated
./fndfcn $1
./cmpl.sh
echo "Compiled successfully..."
echo "To run : ./main "
```

- **cmpl.sh**

```
spu-gcc tst1_spu.c -o tst1
spu-gcc tst2_spu.c -o tst2
ppu-gcc main_ppu.c -o main -lspe2 -lpthread
```

6.3 Performance Evaluation

To compare the performance of the program, the same program with two files having three functions with few dependancies has been written and executed using different parallel programming methods. The result shows comparative performance benefit of the program converted using this algorithm. Following are the result with respect to Real, User and System time usage.

Here the data in the table shaws that the execution of a program distributade on different processors using this tool take comparatively less time as compare to sequential, multi-threading and multi-processing wxwcution of the same program on the same machine.

	real	**user**	**sys**
sequential program	0m9.225s	0m8.945s	0m0.082s
using multi-processing	0m8.268s	0m7.952s	0m0.051s
using multi-threading	0m8.787s	0m14.503s	0m0.116s
Distributed on different processors using this algorithm	0m4.757s	0m1.738s	0m0.026s

Table 6.1: Performance evaluation

6.4 Limitations

This algorithm has some limitations because of the limitations of the architecture and some other issue, which are mentioned below.

- **Less Independent functions, Less benefit :**

  This algorithm works better with the program having more number of independent functions. It will not affect to the performance of a sequential program with no independent function.

- **SPE limitation :**

  SPE has a limitation that it can not call another SPE. Only PPE can assign works to SPE and can call its thread so this algorithm will consider inter functional dependence as a sequential program.

- **IO interface :**

  Because of limited number of resources, Performance of the program degrades as the number of IO interface increases. Each PPU and SPU have to wait for the resource to be free if it is using by someone else.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Cell BE is a very powerful architecture for computational tasks and can perform better as compare to other architecture if it is used with proper programming. A simple program written for a single CPU also runs properly on this architecture but it uses power of PPU only, though it is developed using multi-threading or multi-processing programming techniques. To get the benefit of the architecture some of the system calls and some intrinsics should be used in the program, and to learn programming with CellBE SDK is a very time consuming and difficult task. A good programmer will take more than a month to learn architecture of CellBE and to be aware of the system calls of SDK.

This algorithm is useful to convert a sequential program into a program which runs properly on a multi-core architecture to get its architectural benefit. I have applied this algorithm for CellBE architecture. I have developed a tool which convert a sequential C language program into a list of programs which runs nicely and optimally on CellBE architecture. This tool saves time and affords of a developer to learn a complete SDK functionality.

There are some limitations with the architecture and the SDK, And so this tool also has several limitations which can be further resolved to make this tool more better.

7.2 Future Work

After working on this topic for almost a year, i am able to complete several tasks and to prove some of the things. I have found out some of the tasks and problems which can be worked on and can be resolved.

- IBM SDK provides supporting system calls and intrinsic for C, C++ and machine level programming language only. I have applied this algorithm and develop a tool which converts a program written in C. Same algorithm can be applied to make a tool for programs written in other programming languages like C++.

- This architecture supports parallel computing in two ways. One is independent computing using execution on SPUs and second is vectorization in which same operation like addition, multiplication etc... can be applied on multiple data at a time using SIMD. This algorithm only takes care of first method so second method of optimization (Vectorization) can also be applied to optimize the performance. Mix approach can also be tried to get the best performance.

- This SDK has some limitations like an SPE can not call another SPE. Because of such limitations only PPE can assigned a task to SPEs and can call them for computation, inter function dependence is not taken care in this algorithm. This problem can be solved by different approach.

Web References

[1] http://www.ibm.com

[2] http://www.cs.berkeley.edu/∼dsw/

[3] http://cs.ecs.baylor.edu/∼donahoo/tools/valgrind

[4] http://publications.csail.mit.edu/abstracts/abstracts06/pgbovine/pgbovine.html

[5] http://cs.swan.ac.uk/∼csoliver/oksatlibrary/internetfihtml/doc/doc/Valgrind/3.5.0/html/n
writing-tools.html

References

[1] Inbal Ronen, Nurit Dor, Sara Porat and Yael Dubinsky,*"Combined Static and Dynamic Analysis for Inferring Program Dependencies Using a Pattern Language"* ,IBM Haifa Research Lab Mount Carmel, Haifa 31905, Israel,–2006.

[2] Jeremy W. Nimmer and Michael D. Ernst*"Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java"*, MIT Lab for Computer Science 200 Technology Square Cambridge, MA 02139 USA.

[3] Miryung Kim and Andrew Petersen, *An Evaluation of Daikon: A Dynamic Invariant Detector, miryung, petersen*,at cs.washington.edu.

[4] Kenichi Nakao,*Lettix LDM:Analysis of Software Artifacts*, –Mar 25, 2008.

[5] Wolfram Amme, Peter Braun, Eberhard Zehendner, *Data Dependence Analysis of Assembly Code*, Computer Science Department Friedrich Schiller University Jena, –Septembre 1999.

[6] Francoise Balmas, *DDFgraph: a Tool for Dynamic Data Flow Graphs Visualization*, Universite Paris 8 (France).

[7] Stephen McCamant, *Dynamic Variable Comparability Analysis for C and C++ Programs"*,MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge MA, 02139 USA.

[8] *"programing with CellBE"*,IBM.

[9] philip J. Guo, *A Scalable Mixed-Level Approach to Dynamic Analysis of C and C++ Programs*,May 16, 2005.

[10] Seunghwa Kang and David A. Bader,*"Optimizing JPEG2000 Still Image Encoding on the Cell Broad- band Engine"*,Georgia Institute of Technology, Atlanta, GA 30332, 2008.

[11] Krste Asanovic and Ras Bodik,*"The Landscape of Parallel Computing Research: A View from Berke- ley"*,UC Berkeley EECS, Aug 18, 2009.

[12] *"C/C++ Language Extensions for Cell Broadband Engine Architecture"*, IBM, 2008.

Index