Technical University of Crete (TUC)

Department of Electronics and Computer Engineering

Dissertation thesis

Προσαρμογή Αλγορίθμων
"FINGERPRINT RECOGNITION (NBIS)" ΚΑΙ "MULTIPLE SEQUENCE ALLIGNMENT PROGRAM(MAFFT)"
για εκτέλεση στον πολυεπεξεργαστή CELL BROADBAND ENGINE

Τζανουδακης Θεοδωρος
tzanouch@yahoo.gr

Committee
Παπαευσταθιου Ιωαννης, Associate Professor (Supervisor)
Δολλας Αποστολος, Professor
Ζερβακης Μιχαλης, Professor

Chania 2009

ABSTRACT

by Tzanoudakis Theodoros

Decades of prophesies have finally come true: programming with multiple processors has now entered the mainstream and in order to increase performance and save energy, most future processor chips will contain many processors, so called cores, that work in parallel. Is it due to Moore's Law that "the number of transistors on a reasonably priced integrated circuit tends to double every approximately 20 months" and now that we are closing to the size limits the need for multiple-cores has become inevitable? Could it just be the need for creating a power-efficient and cost-effective high performance system? The fact is that great hopes are being pinned on multiprocessors and parallel architecture as the answers for the continuing development of electronics and computing.

The purpose of this thesis is to port the fingerprint recognition algorithm used by FBI ("**NBIS**") and a multiple sequence alignment program used for amino acids or nucleotide sequences ("**MAFFT**") to the CELL BROADBAND ENGINE ("**CELL**") architecture and try to improve their performance. Cell is a multi-core processor developed by the alliance "STI" – Sony Computer Entertainment , Toshiba , IBM. After a 4-year period and a 400 million US $ budget the Cell was finally released on 2005 but the first major commercial application was in Sony's Playstation 3 game console ("PS3"). Since then many applications have used the Cell multiprocessor but the cheapest way for a developer to get his hands on the powerfull Cell still remains the PS3 game console. As DR. Peter Hofstee –Cell's Designer– said when asked what he felt while developing game technologies: *"I think games are an interesting area but quite clearly, Cell is **not just for Games**. There are many other areas it can be used. Games are the thing that inspired us to do it".*

After almost a year of studies, tries, experiments, tricks and unexpected problems this project was completed. Both algorithms were ported to the PS3 and with the use of threads on the most demanding processes we managed to achieve a significant improvement on the performance of the **MAFFT** program.

Contents

List of Figures

List of Tables

We all fall ! The point is how gracefully we stand back to our feet ...

Acknowledgements

This project was the result of a team effort and lasted over a year.

Firstly, I would like to express my appreciation to my project supervisor professor I.Papaefstathiou for his understanding apart from his expertise and cooperation.

Besides my supervisor, I would like to thank the rest of my thesis committee, professors A.Dollas and M.Zervakis for their assistance at all levels of my studies.

Very special thanks go to the postgraduate students P.Christou and G.Chrysos for their self-forgetful and constant guidance during this research.

Many thanks to the Microprocessor & Hardware Laboratory supervisor Mr.M.Kimionis for his help and encouragement as well as his cooperation.

Many thanks to my postgraduate and undergraduate student friends for all the great memories we shared.

And finally, I am grateful to my family and Norou for their constant psychological but for financial support.

This thesis as well as my degree are dedicated to my father E.Tzanoudakis for his presence and support during my studies.

Chapter 1

INTRODUCTION

by Tzanoudakis Theodoros

In the last decade there was an enormous improvement in processors' speeds without a corresponding improvement in bus or interconnection network speeds. This led the relative costs of communication and computation in shared-memory multiprocessors to change dramatically. Moreover, many parallel applications which depend on a delicate balance between the cost of communication and computation, do not execute efficiently on today's shared- memory multiprocessors. Those needs to break the physical limits of uniprocessing (by branch prediction or RAW dependencies etc.) while being cost and power effective at the same time were the motivation for the scientific and industrial communities to look for alternative architectures. Therefore, microprocessor engineers agree that multi-core designs will be the wave of the future, but they differ widely on how to implement them and surmount the many challenges they pose.

Today's major trend in computer architecture is the design of multi-core-systems-on-a-chip processors. The cores are typically integrated onto a single integrated circuit die (CMP – chip multiprocessor) or they may be integrated onto multiple dies in a single chip package. As a result, huge computational power is hold on a single chip and the re-design becomes an easier task as bigger systems can be built from commodity parts (ex. ordinary uniprocessors). Thus, the cost of redesigning has dropped while the fault tolerance increased (on N-processor system if one processor fails there are still (N-1) processors available). This fundamental change in our core computing architecture will require a fundamental change in how we

program. The art of multiprocessor programming is more complex than programming uniprocessor machines and requires an understanding of new computational principles, algorithms and programming tools. All these led to making the architecture simpler and more streamlined to the software. This is exactly the purpose of this thesis, initially to port the algorithms to a multicore processor and furthermore to extract as much computational power we can.

The processor we used is the multicore-processor Cell Broadband Engine (`Cell BE`). The Cell BE processor is the first implementation of the Cell Broadband Engine Architecture (CBEA), developed jointly by Sony, Toshiba, and IBM. In addition to Cell BE's use in the upcoming Sony PlayStation® 3 console, there is a great deal of interest in using it in Cell BE-based workstations, media-rich electronics devices, video and image processing systems, as well as several other emerging applications. The Cell BE includes one POWER^{TM} Processing Element (`PPE`) and eight Synergistic Processing Elements (`SPEs`). The Cell BE architecture is designed to be well-suited for a wide variety of programming models, and allows for partitioning of work between the PPE and the eight SPEs. This thesis shows that Cell BE can outperform other state-of-the-art processors, and in some cases by approximately an order of magnitude or more. The easiest way to explore its features its computational power is the Sony Playstation 3 Game console (PS3) which provides the developer with 6 128-bit SPEs (one is locked for testing and one from OS) in addition to a multithreaded 64-bit PPE.

The first application we had to port to the PS3, was the fingerprint recognition application (**NBIS**) developed by the *National Institute of Standards and Technology (**NIST**) for the Federal Bureau of Investigation (**FBI**) and the US Department of Homeland Security (**DHS**)*. A collection of application programs, utilities and source code libraries are freely provided in the non-export (meaning not export controlled) package NBIS and after a request to the NIST, the full export-controlled package with a large dataset for testing was sent to our lab. The software technology contained in this distribution is a culmination of more than a decade's worth of work for the FBI and DHS at NIST. In the late 60's researchers at NIST began work on the first FBI's AFIS system and during this period developed methods and produced databases for the DHS as well. Since the terrorist hit on the 11th of September 2001 the phrase, "Everything has changed," has been frequently stated. This is no less true for the Image Group at NIST. Within a couple of months, new initiatives were started that redirected work focused on law enforcement to new work focused on border control.Thus, the necessity of developing a fast and as accurate as possible fingerprint recognition system was maximized. After a couple of months experimenting and testing, we

managed to port the whole application to the PS3. Unfortunately, the lack between the software and the Cell architecture's characteristics caused a drop on the performance we could not't overcome.

The second application we had to port was a multiple sequence alignment program used for amino acids or nucleotide sequences (**"MAFFT"**). Mafft offers various multiple alignment methods such as L-INS-I (accurate for alignment of $<$ 200 sequences), FFT-NS-2 (fast for alignment of $<$ 10.000 sequences), etc. in order to improve the speed and the effectiveness of the requested sequence. Mafft is useful for optimizing protein alignments based on physical properties of the amino acids. In our thesis , not only did we port the application to the PS3 but we achieved a significant increase on the performance without changing the application's logic, only by making use of the Cell's computational power and its unique (so far) technical characteristics.

The original purpose was not to compare our benchmark results and their time of execution with a conventional system (General Purpose PC ex.Pentium IV) . The original purpose was to port the applications to the PS3 and try to reduce its execution time. Several problems (4.1.5) were faced at the first application and so we focused on the second application which looked very promising for a performance boost. Our suspicions become true and the performance results, after the project was completed, were the execution time of mafft to become comparable with that of a conventional Quad-core PC (developed 4 years after the Cell was released !!).

The rest of the thesis is organized as follows:

- In **Chapter 2**, **Section 2.1** introduces the `CELL` and Section 2.2 the `PS3` console we used for the implementation,
- **Chapter 3** outlines the `NBIS` (Section 3.1) and the `MAFFT` (Section 3.2) applications as well as their main algorithms,
- **Chapter** 4 is a detailed description of the development process we followed for the implementation and is consisted of 2 sections.
  Section 4.1 describes the implementation of `NBIS` while `MAFFT's` is explained on **Section** 4.2,
- **Chapter** 5 presents the evaluation results of the two implementations and finally
- **Chapter** 6 contains the conclusion and the related work suggestions for both applications of this thesis.

Chapter 2

CELL & Sony PS3

This chapter describes the hardware we used for the implementation of this project. The purpose of this chapter is to introduce to the reader the architecture of Cell and the PS3 game console. The concepts described in this chapter are the theoretical setting for multicore-programming on the PS3's Cell.

2.1 CELL BROADBAND ENGINE

2.1.1 Introduction

The CELL Broadband Engine (CBEA) defines an architecture well suited for a wide variety of compute and communication intensive applications. The first implementation of the CBEA is the nine-core version of CELL BROADBAND ENGINE and appears to be a good fit for a variety of signal processing applications. The architecture was developed by the "S.T.I - alliance" - Sony, Toshiba, IBM - after a 4-year scientific research and a budget over 400 million US$. In the first months of 2005 it was finally released and the first version was applied to the game console that was initially designed for, known as PLAYSTATION 3 GAME CONSOLE - Toshiba plans to use the Cell in digital home appliances and IBM in high-performance computers. It's computing capabilities though, were rapidly spread to the scientific community as the beginning of a new trend in the multicore processing architecture.[1] [11]

The CBEA chip consists of a traditional 64-bit General-Purpose Power-PC called **PPE** (2.1.2) and eight 128-bit Synergistic Power-PCs called **SPE's** (2.1.3). Onto the chip there are also integrated a high speed memory

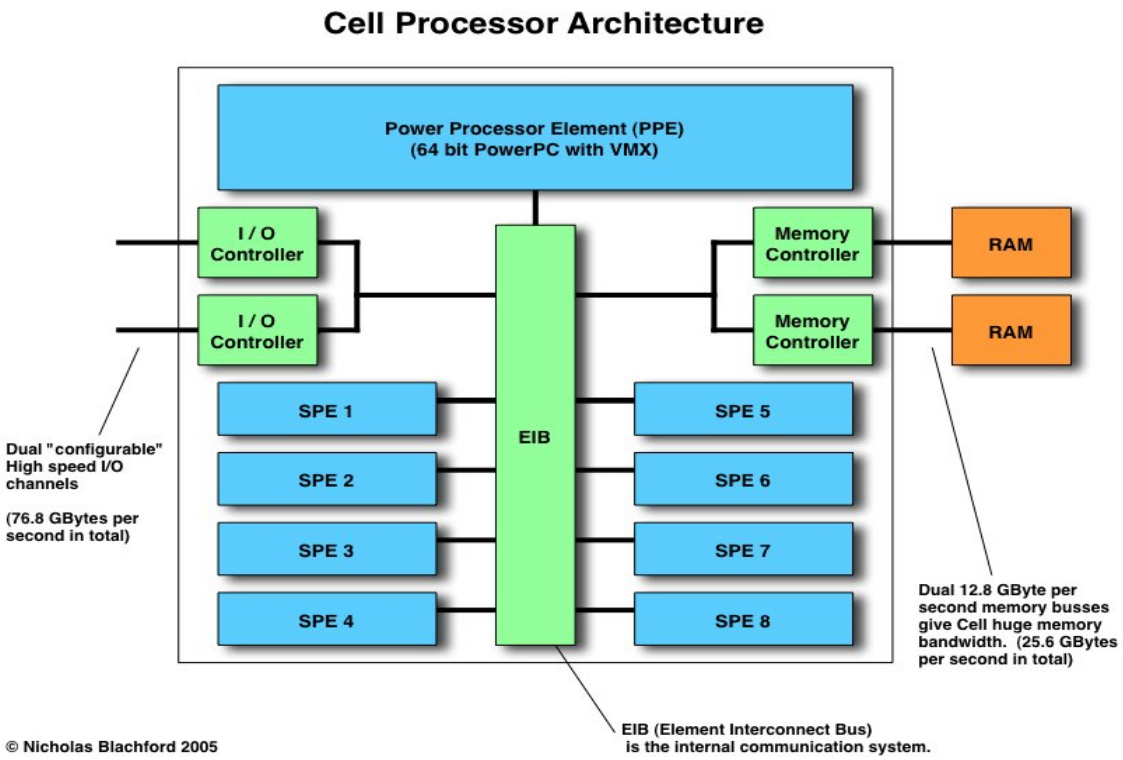

Figure 2.1: CBEA architecture's block diagram

controller, an I/O controller, an interrupt controller and caches. Every element of the chip is connected with a high bandwidth interconnection bus element, called **EIB** (2.1.4). **Figures** 2.1 , 2.2 give an architectural overview of the CBEA chip.[4] [13]

2.1.2 Power Processor Element - "PPE"

The *Power Processor Element* (PPE) (**Figure** 2.3) is a 64-bit Power-Architecture-compliant core optimized for design frequency and power efficiency. In comparison to more recent four-issue out-of-order processors the design of the PPE is simplified. The PPE is a dual-issue design that does not dynamically reorder instructions at issue time (e.g., "in-order issue"). The core interleaves instructions from two computational threads at the same time to optimize the use of issue slots, maintain maximum efficiency, and reduce pipeline depth. Simple arithmetic functions execute and forward their results in two cycles. Owing to the delayed-execution fixed-point

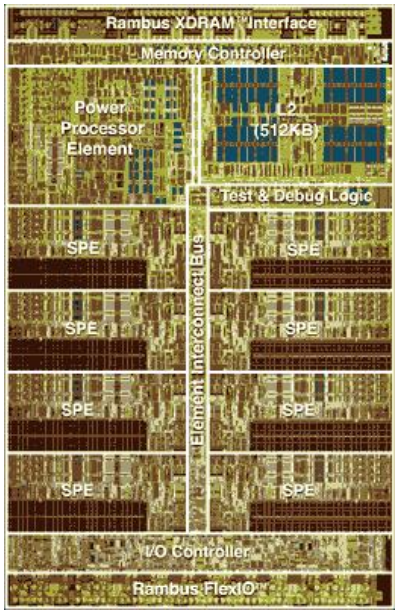

Figure 2.2: CBEA architecture die photo

pipeline, load instructions also complete and forward their results in two cycles while a double-precision floating-point instruction executes in ten cycles.

The PPE supports a conventional cache hierarchy with 32-KB first-level instruction and data caches and a 512-KB second-level cache. The second-level cache and the address-translation caches use replacement management tables to allow the software to direct entries with specific address ranges at a particular subset of the cache. This mechanism allows for locking data in the cache (when the size of the address range is equal to the size of the set). It can also be used to prevent overwriting data in the cache by directing data that is known to be used only once at a particular set. Providing these functions enables increased efficiency and increased real-time control of the processor.

The processor provides two simultaneous threads of execution within the processor and can be viewed as a two-way multiprocessor with shared dataflow. This gives software the effective appearance of two independent processing units. All architected states are duplicated, including all architected registers and special-purpose registers, with the exception of registers that

deal with system-level resources, such as logical partitions,memory, and thread control. Non-architected resources such as caches and queues, are generally shared for both threads except in cases where the resource is small or offers a critical performance improvement to multithreaded applications.

Having a 3.2 GHz clock ,the PPE looks like a very promising processor. The PPE's architecture though, is a lot simpler than a conventional's General-Purpose CPU (for example doesn't support branch prediction) causing applications' execution time to raise. The PPE was developed for being a controller that supervises the other cores (SPEs) and running the Operation System (OS) but for the compute intensive parts which are offloaded to the SPE (The PS3 reserves one SPE for OS). All other power-demanding applications need to be offloaded to the SPE's and thanks to the compliance with PowerPc architecture, existing applications can run on the Cell with a few -in some cases even out of the box - changes (4.2.3) and then optimized for performance using the SPE's.[5][6][7]

The PPU deals with instruction control and execution. It includes:

- the full set of 64-bit PowerPC registers
- 32 128-bit vector registers
- a 32-KB level 1 (L1) instruction cache
- a 32-KB level 1 (L1) data cache
- an instruction-control unit
- a load and store unit
- a fixed-point integer unit
- a floating-point unit
- a vector unit
- a branch unit and
- a virtual-memory management unit.

The PPSS handles memory requests from the PPE and external requests to the PPE from other processors or I/O devices. It includes:

- a unified 512-KB level 2 (L2) instruction and data cache,
- various queues,
- a bus interface unit that handles bus arbitration and pacing on the EIB.

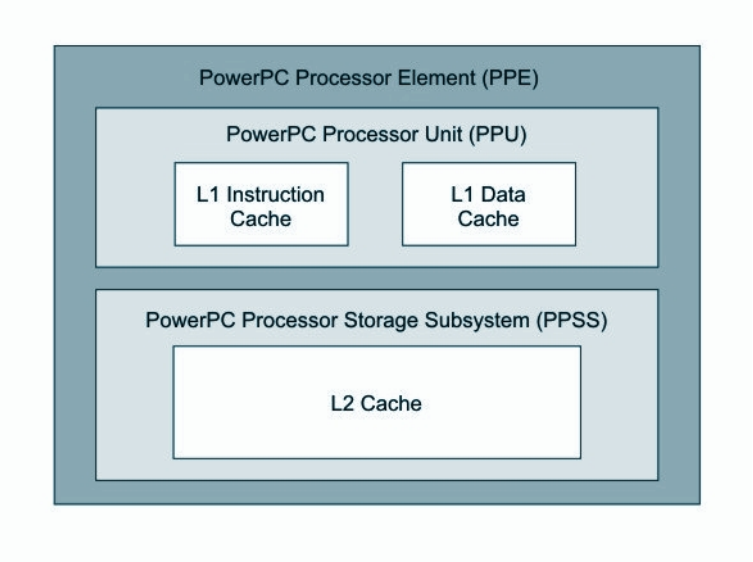

Figure 2.3: PPE block diagram

2.1.3 Synergistic Processor Elements - "SPE's"

The Cell has eight *Synergistic Processor Elements* (SPEs) and each of them consists of a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC) as shown to the block diagram (**Figure** 2.4). The architecture of the SPE is shown in **Figure** 2.5). The SPUs have less complex computational units than PPEs because they do not perform any system management functions. They have a single instruction, multiple data (SIMD) capability. They typically process data and initiate any required data transfers in order to perform their allocated tasks.([3])

An SPE is a RISC processor with 128-bit SIMD (Single Instruction, Multiple Data) organization for single and double precision instructions. With the current generation of the Cell, each SPE contains a 256 KB embedded SRAM for instruction and data, called "Local Storage" which is visible to the PPE and can be addressed directly by software. Each SPE can support up to 4 GB of local store memory. The local store does not operate like a conventional CPU cache since it is neither transparent to software nor does it contain hardware structures that predict which data to load.[4][9]

The SPEs contain a 128-bit, 128-entry register file and measures 14.5 mm2 on a 90 nm process. The SPU cannot directly access system memory.

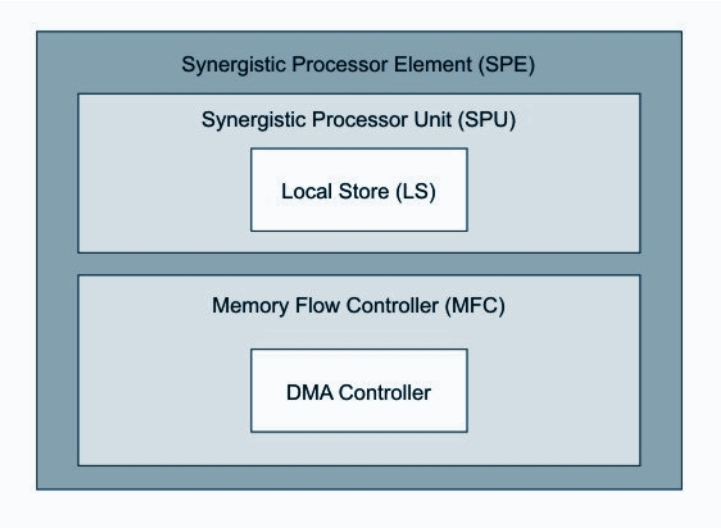

Figure 2.4: SPE block diagram

Instead, the 64-bit virtual memory addresses formed by the SPU must be passed from the SPU to the SPE memory flow controller (MFC) to set up a DMA operation within the system address space. The SPE implements a new instruction-set architecture optimized for power and performance on computing-intensive and media applications. The SPE operates on a local store memory (256 KB) that stores instructions and data. Data and instructions are transferred between this local memory and system memory by asynchronous coherent DMA commands, executed by the memory flow control unit included in each SPE. Each SPE supports up to 16 outstanding DMA commands. Because these coherent DMA commands use the same translation and protection governed by the pageand segment tables of the Power Architecture as the PPE, addresses can be passed between the PPE and SPEs, and the operating system can share memory and manage all of the processing resources in the system in a consistent manner. The DMA unit can be programmed in one of three ways:

1. with instructions on the SPE that insert DMA commands in the queues;
2. by preparing (scatter-gather) lists of commands in the local store and issuing a single "DMA list" of commands; or
3. by inserting commands in the DMA queue from another processor

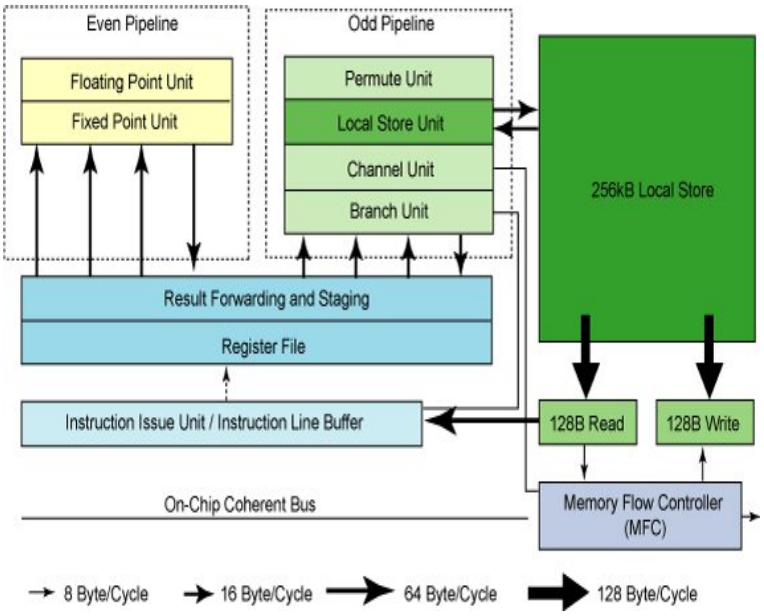

Figure 2.5: Synergistic Processor Element Architecture

Instructions are fetched 128 bytes at a time, and pressure on the local store is minimized. The highest priority is given to DMA commands, the next highest priority to loads and stores, and instruction (pre)fetch occurs whenever there is a cycle available. A special no-operation instruction exists to force the availability of a slot to instruction fetch when necessary. The execution units of the SPU are organized around a 128-bit dataflow. A large register file with 128 entries provides enough entries to allow a compiler to reorder large groups of instructions in order to cover instruction execution latencies. Simple fixed-point operations take two cycles, and single-precision floating-point and load instructions take six cycles. Two-way SIMD double-precision floating point is also supported, but the maximum issue rate is one SIMD instruction per seven cycles. All other instructions are fully pipelined.Up to two instructions are issued per cycle; one issue slot supports

fixed- and floating-point operations and the other provides loads/stores and a byte permutation operation as well as branches.[8]

2.1.4 Element Interconnection Bus - "EIB"

The PPE and SPEs communicate coherently with each other and with main storage and I/O through the *Element Interconnection Bus* ("**EIB**"). The EIB is a 4-ring structure (two clockwise and two counterclockwise) for data, and a tree structure for commands. Apart from connecting all the various on-chip system elements, EIB also includes an arbitration unit used for handling the requests of the elements on the ring.[12]

An element that needs to start data transfer, sends a data bus access request. The arbitration unit then, selects the ring that travels in the shortest transfer as long as it doesn't interfere with other in-flight transfers and gives priority to the memory controller requests. Finally, each ring may allow up to three concurrent data transfers whose paths do not overlap. The EIB's internal bandwidth is 96 bytes per cycle, and it can support more than 100 outstanding DMA memory requests between main storage and the SPEs.[14][5]

The memory-coherent EIB has two external interfaces, as shown (**Figure** 2.6):

1. The *Memory Interface Controller* (**MIC**) provides the interface between the EIB and main storage. It supports two Rambus Extreme Data Rate (XDR) I/O (XIO) memory channels and memory accesses on each channel of 1-8, 16, 32, 64, or 128 bytes.
2. The *Cell Broadband Engine Interface* (**BEI**) manages data transfers between the EIB and I/O devices. It provides address translation, command processing, an internal interrupt controller, and bus interfacing. It supports two Rambus FlexIO external I/O channels. One channel supports only non-coherent I/O devices. The other channel can be configured to support either non-coherent transfers or coherent transfers that extend the logical EIB to another compatible external device, such as another Cell Broadband Engine.

Each participant on the EIB has one 16B read port and one 16B write port. The limit for a single participant is to read and write at a rate of 16B per EIB clock (8B per system clock). Each SPU processor contains a dedicated DMA management queue capable of scheduling long sequences of transactions to various endpoints without interfering with the SPU's ongoing computations; these DMA queues can be managed locally or remotely as well, providing additional flexibility in the control model.

A Cell's processor clock speed is most often cited running at 3.2 GHz resulting the clock frequency at each channel to flow at a rate of 25.6 GB/s. Viewing the EIB in isolation from the system elements it connects, achieving twelve concurrent transactions at this flow rate works out to an abstract EIB bandwidth of 307.2 GB/s. Based on this view many IBM publications depict available EIB bandwidth as "greater than 300 GB/s". This number reflects the peak instantaneous EIB bandwidth scaled by processor frequency. However, other technical restrictions are involved in the arbitration mechanism for packets accepted onto the bus. The IBM Systems Performance group explains: *Each unit on the EIB can simultaneously send and receive 16B of data every bus cycle. The maximum data bandwidth of the entire EIB is limited by the maximum rate at which addresses are snooped across all units in the system, which is one per bus cycle. Since each snooped address request can potentially transfer up to 128B, the theoretical peak data bandwidth on the EIB at 3.2 GHz is 128B x1.6 GHz = 204.8 GB/s. All things considered the theoretic 204.8 GB/s number most often cited is the best one to bear in mind.* The IBM Systems Performance group has demonstrated SPU-centric data flows achieving 197 GB/s on a Cell processor running at 3.2 GHz so this number is a fair reflection on practice as well.

On-chip network design has become an increasingly important component of computer architecture. The Cell Broadband Engine's Element Interconnect Bus, with its four data rings and common command bus for end-to-end transaction control, interconnects more nodes than most commercial on-chip networks.[32]

2.1.5 Programming Models

On any processor, coding optimizations are achieved by exploiting the unique features of the hardware. In the case of the Cell Broadband Engine, the large number of SPEs, their large register file, and their ability to hide main-storage latency with concurrent computation and DMA transfers support many interesting programming models.

With the computational efficiency of the SPEs, software developers can create programs that manage dataflow as opposed to leaving dataflow to a compiler or to later optimizations.Many of the unique features of the SPE are handled by the compiler, although programmers looking for the best performance can take advantage of the features independently of the compiler. It is almost never necessary to program the SPE in assembly language. C intrinsics provide a convenient way to program the efficient movement and buffering of data. There are seven types of programming models:

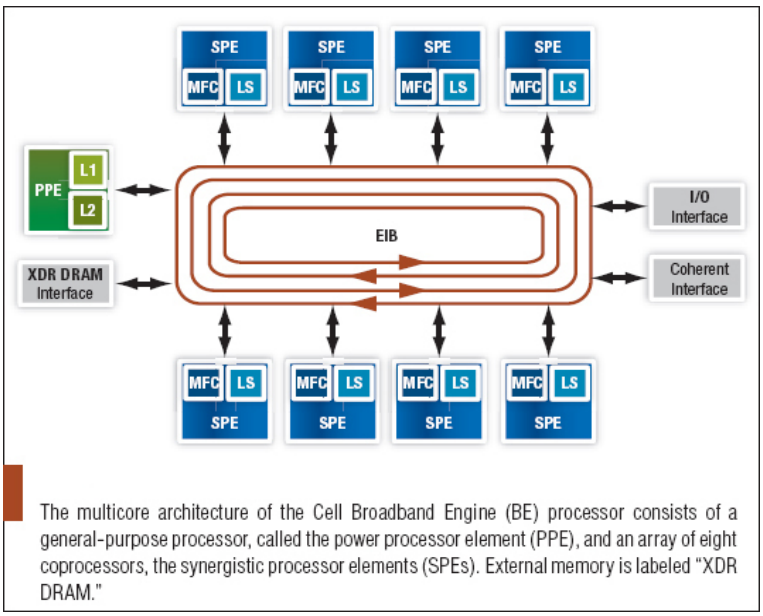

Figure 2.6: Element Interconnection Bus - **EIB**

1. *Function-Offload Model:*

   In the Function-Offload Model, the SPEs are used as accelerators for performance-critical procedures. This model is the quickest way to effectively use the Cell Broadband Engine with an existing application. In this model, the main application runs on the PPE and calls selected procedures to run on one or more SPEs. The Function-Offload Model is sometimes called the Remote Procedure Call (RPC) Model. The model allows a PPE program to call a procedure located on an SPE as if it were calling a local procedure on the PPE. This provides an easy way for programmers to use the asynchronous parallelism of the SPEs without having to understand the low-level workings of the MFC DMA layer. In this model, you identify which procedures should execute on the PPE and which should execute on the SPEs. The PPE and SPE source modules must be compiled separately, by different compilers.

   Remote procedure call The Function Offload or Remote Procedure Call (RPC) Model is implemented using stubs as proxies. A method stub , or simply stub , is a small piece of code used to stand in for some other code. The stub or proxy acts as a local surrogate for the remote procedure, hiding the details of server communication. The main code on the PPE contains a stub for each remote procedure on the SPEs.

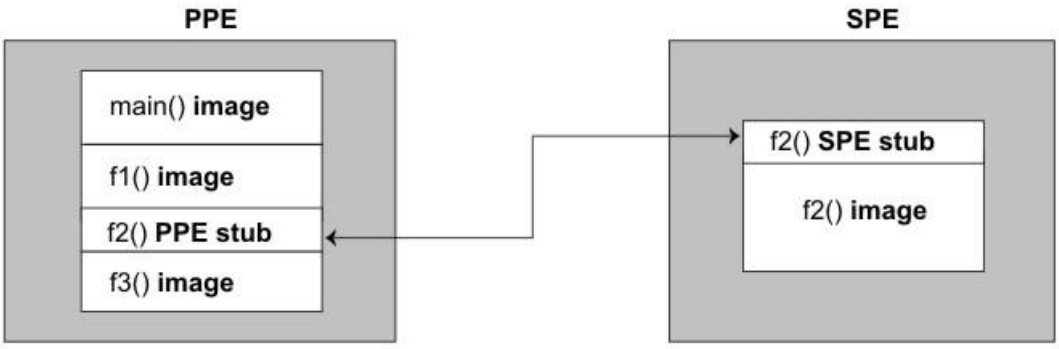

Figure 2.7: Function-Offload (or RPC) Model example

Each procedure on an SPE has a stub that takes care of running the procedure and communicating with the PPE. When the program on the PPE calls a remote procedure, it actually calls that procedure's stub located on the PPE. The stub code initializes the SPE with the necessary data and code, packs the procedure's parameters, and sends a mailbox message to the SPE to start its stub procedure. The SPE stub retrieves the parameters and executes the procedure locally on the SPE. The PPE program then retrieves the output parameters.

An example of the Function-Offload (or RPC) Model is shown in **Figure** 2.7

2. *Device-Extension Model:* The Device Extension Model is a special case of the Function-Offload Model in which the SPEs act like I/O devices or can also act as intelligent front ends to an I/O device while mailboxes can be used as command and response FIFOs between the PPE and SPEs.

3. *Streaming Model:* In the Streaming Model, each SPE, in either a serial or parallel pipeline, computes data that streams through. The PPE acts as a stream controller, and the SPEs act as stream-data processors. Although the SDK does not include a formal streaming language, most of the programs written for the Cell Broadband Engine are likely to use the streaming model to some extent. An algorithm that contains a computational kernel that streams packets of data through the kernel for each step in time is an example of the streaming model.

4. *Computation-Acceleration Model:* The Computation-Acceleration Model is an SPE-centric model that provides a smaller-grained and more integrated use of SPEs. The model speeds up applications that use computation-intensive mathematical functions without requiring significant rewrite of the applications. Most computation-intensive sections of the application run in parallel on SPEs. The PPE acts as a control and system-service facility. The work is partitioned manually by the programmer, or automatically by the compilers.

5. *Shared-Memory Multiprocessor Model:* The Cell Broadband Engine can be programmed as a shared-memory multiprocessor, using two different instruction sets. The SPEs and the PPE fully interoperate in a cache-coherent Shared-Memory Multiprocessor Model. All DMA operations in the SPEs are cache-coherent. Shared-memory load instructions are replaced by DMA operations from shared memory to local store (LS), followed by a load from LS to the register file. The DMA operations use an effective address that is common to the PPE and all the SPEs. Shared-memory store instructions are replaced by a store from the register file to the LS, followed by a DMA operation from LS to shared memory. The SPE's DMA lock-line commands provide the equivalent of the PowerPC Architecture atomic-update primitives (load with reservation and store conditional).

6. *Asymmetric-Thread Runtime Model:* Threads can be scheduled to run on either the PPE or on the SPEs, and threads interact with one another in the same way they do in a conventional symmetric multiprocessor. Scheduling policies are applied to the PPE and SPE threads to optimize performance. Although preemptive task-switching is supported on SPEs for debugging purposes, there is a runtime performance and resource-allocation cost. FIFO run-to-completion models, or lightweight cooperatively-yielding models, can be used for efficient task-scheduling. A single SPE can run only one thread at a time; it cannot support multiple simultaneous threads.

   The Asymmetric-Thread Runtime Model is flexible and supports all of the other programming models described in this chapter.

   Any program that explicitly calls *spe_context_create* and *spe_context_run* is an example of the Asymmetric-Thread Runtime Model. This is the fundamental model provided by the SDK's SPU Runtime Management Library, and it is identified by user threads (both PPE and SPE) running on the Cell Broadband Engine's heterogeneous processing complex.

7. *User-Mode Thread Model:* The User-Mode Thread Model refers to

one SPE thread managing a set of user-level functions running in parallel. The user-level functions are called microthreads which are created and supported by user software(also called user threads and user-level tasks) and in contrary to the SPE thread which is supported by the operating system, the OS is not involved. One advantage of this programming model is that the microthreads, running on a set of SPUs under the control of an SPE thread, have predictable overhead. A single SPE cannot save and restore the MFC commands queues without assistance from the PPE.[10][3]

Our implementations were based on a combination of the Function-Offload and the Asymmetric-Thread Runtime Models and are explained with details in **Chapter 4**.

Figure 2.8: PLAYSTATION 3 Game Console black edition - **PS3**

2.2 PLAYSTATION 3 game console

2.2.1 Introduction

"Though sold as a game console, what will in fact enter the home is a Cell-based computer" as Ken Kutaragi[1] stated when the *Playstation 3 game console*-("PS3") was released. The version we used in this thesis was the 60 GB version 1 black-edition PS3 which was released in Europe at November,2007. It includes an internal IEEE 802.11 b/g Wi-Fi, multiple flash card readers (SD/MultiMedia Card, CompactFlash Type I/Type II, Microdrive,Memory Stick/PRO/Duo), and a chrome colored trim . It also supports hardware-based PS2 emulation and SACD(Super Audio Cd) playback and was the first Blu-ray 2.0-compliant Blu-ray player on the market.[17] [33] [34] **Figure 7** (2.8) is an image of our Console.

[1]He is known as "The Father of the PlayStation", and its successors and spinoffs, including the PlayStation 2, PlayStation Portable, and the PlayStation 3

2.2.2 Operating Systems

The Cell project was driven by the need to develop a processor for next-generation entertainment systems. A powerful next-generation architecture that is designed to interface optimally with a user and broadband network in real time and could, if architected and designed properly, be effective in a wide range of applications in the digital home and beyond. The Broadband Processor Architecture is intended to have a life well beyond its first incarnation in the first-generation Cell processor.

In order to extend the reach of this architecture, and to foster a software development community in which applications are optimized to this architecture, an open (Linux**-based) software development environment was developed along with the first-generation processor - The *YELLOW DOG version 6* developed from *Terra Soft Solutions*. Nowadays ,there are various distributions that offer official and unofficial support such as "*Fedore Core 7*" and "*Debian*" operation systems and software has been developed for a wide range of applications far more than gaming.

One SPE is reserved for running the operation system while one more is used for yield reasons. Consequently, the developer has one PPE and six SPEs at his full command for programming any application as access is not allowed to the 2 reserved SPE's.

2.2.3 Graphic Card

For graphics the PS3 has a 256 MB GDDR3 *Nvidia RSX* graphic card with a clock frequency at 550MHz. RSX stands for *Reality Synthesizer* and is the graphics processing unit (GPU) co-developed by NVIDIA and Sony for the PlayStation 3 game console. Sony also claims a 1.8 TFLOPS floating point performance with full HD (up to 1080p) x 2 channels Multi-way programmable parallel floating point shader pipelines.(**??**)

2.2.4 Memory

The PS3 has a dual-channel 256 MB XDR (*Rambus Extreme Data Rate*) main memory providing nearly 200MB the operating system and its applications. As for storage there is a detachable 2.5" HDD of 60GB -in our version. The most recent versions reach up to 160GB capacity.

2.2.5 Networking

Unlike the standard PC's Ethernet controllers, PS3 has a built-in network card directly attached to the companion chip. Therefore, no PPE's intervention is required for data transfers. The PS3 has an internal IEEE 802.11 b/g Wi-Fi which means a total Net Bit Rate of 54Mbps and a maximum throughput of 22 Mbps. Unfortunately, the 2.4 GHz band frequency where the IEEE protocol operates is often used by other devices - microwaves or bluetooth devices and may result in interference issues within a 150feet range indoor.

2.2.6 I/O interface

Our version of PS3 has six ports (Front x 4, Rear x 2 (USB2.0)for USB connection and also supports :

- Memory Stick: standard/Duo, PRO x 1
- SD: standard/mini x 1
- CompactFlash: (Type I, II) x 1

Chapter 3

Implemented Algorithms

3.1 NBIS

3.1.1 Introduction

The first application we had to port to the PS3, was the fingerprint recognition application (**NBIS**) developed by the *National Institute of Standards and Technology (**NIST**) for the Federal Bureau of Investigation (**FBI**) and the US Department of Homeland Security (**DHS**)*. In the late 60's researchers at NIST began work on the first FBI's AFIS system and during this period developed methods and produced databases for the DHS as well. Since the terrorist hit on the 11th of September 2001 the phrase, "Everything has changed," has been frequently stated. This is no less true for the Image Group at NIST. Within a couple of months, new initiatives were started that redirected work focused on law enforcement to new work focused on border control. Thus, the necessity of developing a fast and as accurate as possible fingerprint recognition system was maximized and new methods are applied in order to cover those needs.[21][22][23]

3.1.2 NBIS packages

The NBIS software is organized in two categories: non-export controlled and export controlled. The *non-export controlled* NBIS software is organized into five major packages:

1. *PCASYS* is a neural network based fingerprint pattern classification system; Pcasys is a pattern classification system designed to automatically categorize a fingerprint image as an arch, left or right loop, scar, tented

arch, or whorl. Identifying a fingerprint's class effectively reduces the number of candidate searches required to determine if a fingerprint matches a print on file. For example, if the unknown fingerprint is an arch, it only needs to be compared against all arches on file. These types of "binning" strategies are critical for the FBI to manage the searching of its fingerprint repository.

2. *MINDTCT* is a fingerprint minutiae detector; It takes a fingerprint image and locates features in the ridges and furrows of the friction skin, called minutiae. Points are detected where ridges end or split, and their location, type, orientation, and quality are stored and used for search. There are 100 minutiae on a typical tenprint, and matching takes place on these points rather than the 250,000 pixels in the fingerprint image.

These 3 methods all conduct image binarization of the fingerprint. *It should be noted* that these systems were developed independently of each other, so although these processing steps are in common, different algorithms are applied in each. Further study is required to determine if one system's algorithmic approach is better than the other.

3. *NFIQ* is a neural network based fingerprint image quality algorithm. It takes a fingerprint image and analyzes the overall quality of the image returning an image quality number ranging from 1 for highest quality to 5 for lowest. The quality of the image can be extremely useful in knowing the likely performance of a fingerprint matcher on that image.

4. *AN2K7* is a reference implementation of the ANSI/NIST-ITL 1-2000 "Data Format for the Interchange of Fingerprint, Facial, Scar Mark & Tattoo (SMT) Information" standard; It contains a suite of utilities which facilitate reading, writing, manipulating, editing and displaying the contents of ANSI/NIST files.

5. *IMGTOOLS* is a collection of image utilities, including encoders and decoders for Baseline and Lossless JPEG and the FBI's WSQ specification. It is a large collection of general-purpose image utilities such as image encoders and decoders supporting Baseline JPEG, Lossless JPEG, and the FBI's specification of Wavelet Scalar Quantization (WSQ). There are utilities supporting the conversion between images with interleaved and non-interleaved color components; colorspace conversion between RGB and YCbCr; and the format conversion of legacy files in NIST databases.

The *export controlled* NBIS software is organized into two major packages:

1. *NFSEG* is a fingerprint segmentation system useful for segmenting four-finger plain impressions, It takes a four-finger plain impression fingerprint

image (called a slap) and segments it into four separate fingerprint images. These single finger plain impression images can then be used for single finger matching versus either rolled images or other plain impression fingerprint images. NFSEG will also take a single finger rolled or plain impression image and isolate the fingerprint area of the image by removing the white space.

2. *BOZORTH3* is a minutiae based fingerprint matching system. It uses the minutiae detected by MINDTCT to determine if two fingerprints are from the same person, same finger. It can analyze fingers two at a time or run in a batch mode comparing a single finger (probe) against a large database of fingerprints (gallery).

The NFSEG and BOZORTH3 software are subject to U.S. export control laws. It is our understanding that NFSEG and BOZORTH3 software fall within ECCN 3D980, which covers software associated with the development, production or use of certain equipment controlled in accordance with U.S concerns about crime control practices in specific countries. This source code is written in ANSI "C", and has been developed to compile and execute under the Linux operating system and MAC OS-X operating system using the GNU gcc compiler and gmake utility.

The reasons of our process selection are described in detail in **Chapter** 4.1.2.

3.1.3 PCASYS package

Automatic fingerprint classification is a subject of interest to developers of an Automated Fingerprint Identification System (AFIS). In an AFIS system, there is a database of file fingerprint cards, against which incoming search cards must be efficiently matched. Automatic matchers now exist that compare fingerprints based on their patterns of ridge endings and bifurcations (the minutiae). However, if the file is very large, then exhaustive matching of search fingerprints against file fingerprints may require so much computation as to be impractical. In such a case, the efficiency of the matching process may be greatly increased by partitioning the file fingerprints based on classification.

The basic method used by the PCASYS fingerprint classifier consists of, first, extracting from the fingerprint to be classified an array (a two-dimensional grid in this case) of the local orientations of the fingerprint's ridges and valleys. Second, comparing that orientation array with similar arrays made from prototype fingerprints ahead of time. The comparisons are actually performed between low-dimensional feature vectors made from

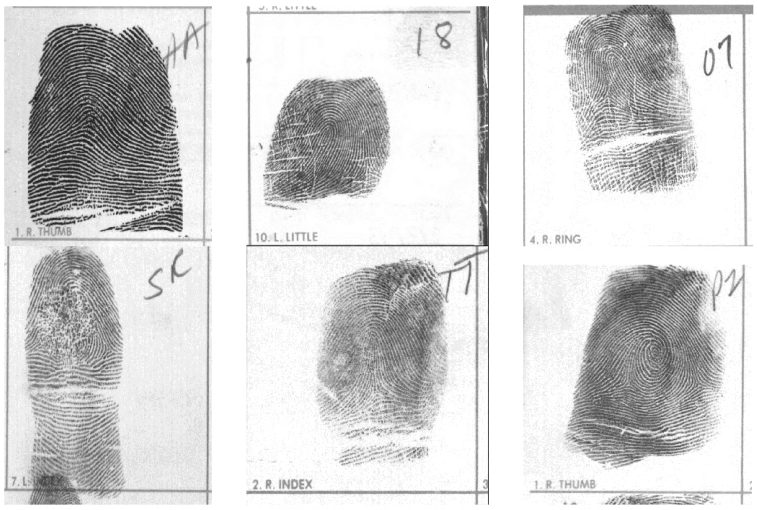

Figure 3.1: Six pattern-level classes example

the orientation arrays, rather than using the arrays directly, but that can be thought of as an implementation detail.

The algorithm may be outlined in 10 steps:

1. `Segmentor:` The segmentor produces, as its output, an image that is 512x480 pixels in size by cutting a rectangular region of these dimensions out of the input image

2. `Image Enhancement:` The enhancement of an input square is done by first performing the forward two-dimensional fast Fourier transform (FFT) to convert the data from its original (spatial) representation to a frequency representation. Next, a nonlinear function is applied that tends to increase the power of useful information (the overall pattern, and in particular the orientation, of the ridges and valleys) relative to noise. Finally, the backward 2-d FFT is done to return the enhanced data to a spatial representation before snipping out the middle 16×16 pixels and installing them into the output image.

3. `Ridge-Valley Orientation Detector:` This step detects, at each pixel location of the fingerprint image, the local orientation of the ridges and valleys of the finger surface, and produces an array of regional averages of these orientations. This is the basic feature extractor of the classification.

4. `Registration:` Registration is a process that the classifier uses in order to reduce the amount of translation variation between similar orientation arrays. When finding a consistent feature it essentially translates the array, bringing that feature to standard location.

5. `Feature Set Transformation:` This step applies a linear transform to the registered orientation array. Transformation accomplishes the reduction of the dimensionality of the feature vector from its original 1680 dimensions to 64 dimensions (PNN) and 128 dimensions (MLP).

6. `Karhunen-Loève Transform:` In order to transform the high-dimensional feature vectors representing each fingerprint which is 1680 elements(28*30 orientation vectors * two components per orientation vector) into much lower-dimensional ones in such a way that would not be detrimental to the classifiers , the K-L transform algorithm was applied.

7. `Probabilistic Neural Network Classifier - "PNN":` This step takes as its input the low-dimensional feature vector that is the output of the transform and it determines the class of the fingerprint.The algorithm classifies an incoming feature vector by computing the value, at its point in feature space, of spherical Gaussian kernel functions centered at each of a large number of stored prototype feature vectors. The largest normalized activation, which is the estimated posterior probability of the hypothesized class, is a measure of the confidence that may be assigned to the classifier's decision.

8. `Multi-Layer Perceptron Neural Network Classifier - "MLP":` This alternative classifier takes as input the low-dimensional feature vector, non-optimized, and a set of MLP weights. The weights are the result of several training runs of MLP in which the weights are optimized to produce the best results with the given training data.

9. `Auxiliary Classifier: Pseudo-ridge Tracer:` This step takes a grid of ridge orientations of the incoming fingerprint and traces pseudo-ridges, which are trajectories that approximately follow the flow of the ridges. It is a whorl detector for the missclassified by the classifiers whorl fingerprints.

10. `Combining the Classifier and Pseudo-ridge Outputs:` This final processing module takes the outputs of the main Neural Network (NN) classifier and the auxiliary pseudo-ridge tracer, and makes the decision as to what class, and confidence, to assign to the fingerprint.

Orientation arrays or matrices like the ones used in PCASYS were produced in early fingerprint work at Rockwell, CALSPAN, and Printrak. The detection of local ridge slopes came about naturally as a side effect of binarization

algorithms that were used to pre-process scanned fingerprint images in preparation for minutiae detection. Early experiments in automatic fingerprint classification using these orientation matrices were done by Rockwell, improved upon by Printrak, and work was done at NIST (formerly NBS). Wegstein, of NBS, produced the R92 registration algorithm that is used by PCASYS and did important early automatic classification experiments.

Figure 3.1 is an example of a six pattern-level classes.

3.1.4 Pcasys input - output

Input The algorithm reads a sequence of image files, each depicting one box as scanned from a fingerprint card, and classifies each fingerprint, using a Multi-Layer Perceptron (MLP) or Probabilistic (PNN) Neural Network, to one of six pattern−level classes: Arch, Left loop, Right loop, Scar, Tented arch, and Whorl. The type of classifier MLP or PNN is chosen in the parameters file pcasys/parms/pcasys.prs. Pcasys may optionally make an output file, containing a results line for each fingerprint and a summary at the end showing the error rate and the "confusion matrix", and it optionally writes progress messages to the standard output. After a request to the *National Institute of Standards and Technology* (**NIST**) the **NBIS EXPORT CONTROL SOURCE CODE CD-ROM** with a set of 2700 **WSQ** compressed grayscale fingerprint images are included to support the testing of PCASYS.

Output The output file has a line for each of the fingerprints that were classified. Each line shows: the fingerprint filename; the actual class (A, L, R, S, T, and W stand for the pattern-level classes arch, left loop, right loop, sear, tented arch, and whorl); the output of the classifier (a hypothesized class and a confidence); the output of the auxiliary pseudo-ridge tracing whorl detector (whether or not a concave-upward shape, a "conup," was found); the final output of the hybrid classifier, which is a hypothesized class and a confidence; and whether this hypothesized class was right or wrong. Figure 3.2 is the output showing the first and last 10 sample images using the PNN classifier:

3.1.5 The Join_Lets process

The most consuming process - in terms of CPU execution time of the pcasys package (Section 4.1.2) was the join_lets process. This process was part of general routines responsible for supporting WSQ image compression contained in the util.c source code. The image is firstly converted into unsigned character pixels which are converted to floating points in the range of +-128.0. The variances between the image's wavelet subbands are then calculated and the image's wavelet subbands as well as every image's

```
s0024301.wsq: is W; nn: hyp W, conf 0.59; conup y; hyp W, conf 1.00; right
s0024302.wsq: is R; nn: hyp R, conf 0.88; conup n; hyp R, conf 0.88; right
s0024303.wsq: is R; nn: hyp R, conf 1.00; conup n; hyp R, conf 1.00; right
s0024304.wsq: is R; nn: hyp R, conf 1.00; conup n; hyp R, conf 1.00; right
s0024305.wsq: is R; nn: hyp R, conf 0.99; conup n; hyp R, conf 0.99; right
s0024306.wsq: is L; nn: hyp L, conf 0.99; conup n; hyp L, conf 0.99; right
s0024307.wsq: is L; nn: hyp L, conf 0.94; conup n; hyp L, conf 0.94; right
s0024308.wsq: is L; nn: hyp L, conf 0.99; conup n; hyp L, conf 0.99; right
s0024309.wsq: is L; nn: hyp L, conf 1.00; conup n; hyp L, conf 1.00; right
s0024310.wsq: is L; nn: hyp L, conf 1.00; conup n; hyp L, conf 1.00; right
…
s0026991.wsq: is W; nn: hyp W, conf 1.00; conup y; hyp W, conf 1.00; right
s0026992.wsq: is W; nn: hyp W, conf 1.00; conup y; hyp W, conf 1.00; right
s0026993.wsq: is T; nn: hyp A, conf 0.79; conup n; hyp A, conf 0.79; wrong
s0026994.wsq: is W; nn: hyp W, conf 1.00; conup y; hyp W, conf 1.00; right
s0026995.wsq: is W; nn: hyp W, conf 1.00; conup y; hyp W, conf 1.00; right
s0026996.wsq: is W; nn: hyp W, conf 0.84; conup y; hyp W, conf 1.00; right
s0026997.wsq: is W; nn: hyp W, conf 0.75; conup y; hyp W, conf 1.00; right
s0026998.wsq: is L; nn: hyp L, conf 0.84; conup n; hyp L, conf 0.84; right
s0026999.wsq: is W; nn: hyp W, conf 1.00; conup y; hyp W, conf 1.00; right
s0027000.wsq: is W; nn: hyp W, conf 0.96; conup y; hyp W, conf 1.00; right

pct error: 7.07

A           L           R           S           T           W
A    41( 83.7)    3(  6.1)    0(  0.0)    0(  0.0)    4(  8.2)    1(  2.0)
L     3(  0.4)  784( 97.5)    3(  0.4)    0(  0.0)    5(  0.6)    9(  1.1)
R     7(  1.0)    6(  0.8)  699( 95.1)    0(  0.0)    5(  0.7)   18(  2.4)
S     0(  0.0)    4( 80.0)    0(  0.0)    0(  0.0)    1( 20.0)    0(  0.0)
T    19( 22.6)   26( 31.0)   14( 16.7)    0(  0.0)   25( 29.8)    0(  0.0)
W     1(  0.1)   35(  3.4)   27(  2.6)    0(  0.0)    0(  0.0)  960( 93.8)
```

The last part of the output file is a brief summary of the results. First, there is the percent error, i.e. the percentage of the fingerprints that were classified incorrectly. Following this is a confusion matrix. It has the same format as Table 2 and Table 3, described in the next section.

Figure 3.2: Pcasys output example

subband block are quantized. The wavelet decomposition of an image and the wavelet subband decomposition are computed and a lossy floating point pixmap is reconstructed from a WSQ compressed datastream. Join_lets is then called for the reconstruction of the image from the wavelet subbands.

3.2 MAFFT

3.2.1 Introduction

An important step in various types of comparative studies of biological sequences is the Multiple sequence alignment (`SMSA`). SMSA is used in phylogenetic inference, conserved region detection, structure prediction of ncRNAs and proteins and many other situations. For an easy MSA problem, such as an alignment consisting of a small number ($<\sim 100$) of short ($<\sim 5,000$) sequences with global and high similarity (percent identity of $>\sim 40\%$ for protein cases and $>\sim 70\%$ for nucleotide cases), most of the current programs return a correct MSA, and no special consideration is needed. However, if all three of these conditions are not met, then the construction of an MSA can be a difficult task from both computational and biological viewpoints.[18]

There is an established method based on the Dynamic Programming (DP) algorithm for calculating- a pairwise alignment (an alignment between two sequences) with a time complexity of O(L2), where L is the sequence length. Theoretically, the DP algorithm can be extended for cases of more than two sequences, but the time and space complexities of the naively extended algorithm, O(LN), are impossibly large, where N is the number of sequences. Finding the exactly optimum MSA quickly becomes computationally intractable when the number of sequences in- creases. Considerable efforts have been made to obtain the optimum MSA of ∼10 sequences, which is still substantially smaller than the alignment size biologists now need. Therefore,some sort of heuristics are inevitable.[19][20]

Even if the optimal MSA is successfully obtained, it is not always the correct solution from a biological viewpoint. This suggests that we should pay attention to a biologically relevant objective function, as well as to algorithmic techniques for obtaining the optimum solution. This is one of the reasons why various multiple sequence alignment schemes have been extensively studied to date, but there is no definitive one. Moreover, the accuracy of multiple alignment is improved by adding homologs or profiles. This is because homologs make family-specific information available and enrich the profiles used in the multiple alignment processes. Recent protein MSA studies indeed tended to use external sequence information. Therefore, for an alignment program, the ability to handle many sequences is an important factor for yielding accurate results, as well as for large-scale analyzing.

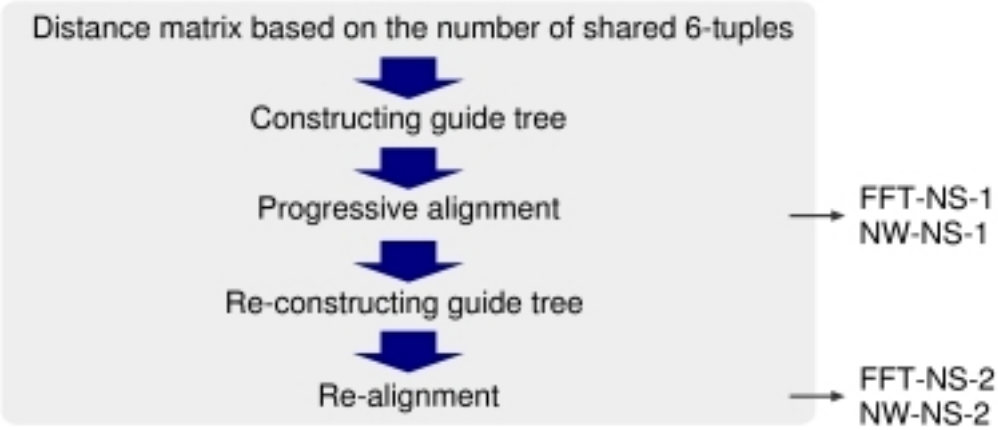

Figure 3.3: MAFFT - Progressive method - algorithm dataflow

3.2.2 MAFFT algorithm dataflow

MAFFT offers various multiple alignment strategies. They are classified into three types:

1. the progressive method,
2. the iterative refinement method with the WSP score and the
3. the iterative refinement method using both the WSP and consistency scores.

In general, there is a trade off between speed and accuracy. The order of speed is $a > b > c$, whereas the order of accuracy is $a < b < c$. Apart from being the faster , the progressive method has been more friendly to this thesis' target as explained on Section 4.2.2. Therefore our implementation was focused on the progressive methods (1^{st} method)

These are simple progressive methods like ClustalW. By using the several new techniques described below, these options can align a large number of sequences (up to ∼5,000) on a standard desktop computer. The most used progressive methods are:

- `FFT-NS-1` which is the simplest progressive option in MAFFT and one of the fastest methods currently available. Firstly it makes a rough distance matrix by counting the number of shared 6-tuples (see below) between every sequence pair. A guide tree is then being built and finally the sequences are aligned according to the branching order.
- `FFT-NS-2` where the distance matrix already used in FFT-NS-1 is recomputed and a second progressive alignment is carried out which basically optimise's the 1st method.

Figure 3.3 shows the algorithm's dataflow

Argument	**Differences**	**Default**
–retree 1	Approximately two times faster but more rough than default	–retree 2
–maxiterate 2	Enhances the accuracy but not applicable to many sequences	–maxiterate 0
–memsave	Memory saving but approximately two times slower	auto
–fft	For long (∼1,000,000 nt) conserved sequences	auto
–nofft	For many (∼5,000) sequences	auto
–parttree	For extremely many ($> 10,000$) sequences	disabled
–dpparttree	For extremely many ($> 10,000$) sequences	disabled
–fastaparttree	For extremely many ($> 10,000$) sequences	disabled
–partsize 1000	More accurate than default	–partsize 50
–groupsize 1	Does not align. Recommended to be used with –reorder. The sequences will be sorted according to similarity.	–groupsize (large)
–treeout	Outputs the guide tree	disabled

Table 3.1: Mafft optional input arguments

Mafft (versions > 5.850) has as a default setting to automatically select a moderately fast method that can process a large dataset the number of sequences ($< 10,000$). It may also take many optional arguments as inputs (shown on table 3.1 which are for manual selection of the appropriate combination in cases of abnormal termination or if there are extremely many ($> 10,000$) sequences to be aligned.

TYPE	INPUTS - OUTPUT
char	**seq1
char	**seq2
double	*eff1
double	*eff2
int	icyc
int	jcyc
int	alloclen
LocalHom (structure)	localhom
float	*impmatch
char	*sgap1
char	*sgap2
char	*egap1
char	*egap2
float	wm

Table 3.2: A__Align optional input arguments

3.2.3 A__Align - The most demanding process

The most time-consuming process of the MAFFT application was the A__Align process. This process is responsible for reconstructing the guide tree after the progressive alignment of the original tree. It calls other routines in order to create matrices with matching scores. When the new matrices are created, in order to make the alignment of the sequences, A__Align produces a score value between the new and the old score tables.

A__Align is called at least once per input sequence. For every single sequence it creates two pairs of sequences -called "clusters"- based on the weight tree already created. Every pair consists of sequence parts, the number of whom depends on the nature of the input. For most of our datasets the maximum length of a part was less than 500 characters. In more details, the input sequence is separated into `seq1` and `seq2` and their elements number is set by the values of `cluster1` and `cluster2`. Furthermore, it takes as input the table `dis_consweight` which contains the weights of each part as calculated in the previous step of mafft. The table also contains a default `fpenalty` value for the cost of empty sequence parts and two vectors `sgap` and `egap` used for defining any blank spaces of the sequence.

Input: . At the stage of alignment MAFFT calls A__align. A__Align takes as input arguments few pointers to the original score table as well as some static integer and float values used for data transfer between the score matrices.

Output: . After the alignment A__Align returns one float value, called `wm` which is the totalscore of the matching between the matrices and is also the benchmark of the method's accuracy and the two aligned parts of the sequence.

Table 3.2 shows the input and output arguments of A__Align

Chapter 4

Implementation

The main purpose of this chapter is to explain the overall development flow of this thesis' implementation. The processes of enabling the **NBIS** (Section 4.1) and the **MAFFT** (Section 4.2) applications on the PS3's Cell processor are presented in this chapter. Details are also given for the programming models, the level of parallelism and the data partitioning procedure that were used in each application. Finally, some of the problems we encountered as well as their solutions are described at the end of each section.

4.1 NBIS Implementation

4.1.1 Introduction

This section gives an overall development flow of our implementation of the NBIS application on the CELL processor. This implementation was not successful in terms of boosting the performance of an existing application. The problems we faced, instead, obliged us to stop prematurely. They provided us, on the other hand, with a theoretical base not only for porting an application to the CELL but also for the necessity to have the source code written for a multithreaded execution. After porting the Fingerprint recognition algorithm to the Cell's PPE, our implementation was completed by the disappointing execution times (we used *Intel Vtune Performance Analyzer* along with the *Linux* `time` function).

Our implementation is separated into 4 steps each described in details in the following sections:

- Section 4.1.2 describes the *performance analysis* of the given application and shows the execution profile for each of the packages of NBIS (described in Section 3.1.2),
- Section 4.1.3 describes the process of porting the *PCASYS* package to the PPE and refers to the problems we faced at this stage,
- Section 4.1.4 describes the data analysis we did to the most time-consuming function called `Join_Lets`
- Section 4.1.5 gives an overall of the problems that ended this implementation without porting to the SPE's.

In the final subsection 4.1.6, solutions and a brief conclusion for the requirements when building applications like NBIS, are provided.

4.1.2 Performance analysis

The first step of our implementation was to make the profile of the original source code and discover the hotspots that would be offloaded to the Synergistic Processing Units. NBIS came along with a 300 page manual describing all the packages' and their main processes' algorithms and provided us as well with inputs and outputs for testing. For measuring the performance of the packages we used INTEL's Vtune Analyzer distribution 9_1 for Linux. The machine we used for the profiling was a Pentium IV at 2.66 GHz, with Ubuntu 8.04 OS.

Bozorth3 package

After reading the manual and having realized the need of every package (as described in Section 3.1.2) we decided to port the `bozorth3` package to CELL. Bozorth3 is responsible for determining if two fingerprints are from the same person, same finger. It uses the minutiae detected by MINDTCT to analyze fingers two at a time or run in a batch mode comparing a single finger (probe) against a large database of fingerprints (gallery) and come with a matching result.

Unfortunately vtune's results (showed on Table 4.1) were discouraging as there was no demanding (in terms of computing) process with a significant percentage of CPU usage. Instead, depending on the dataset different processes would prove to be the most time-consuming. For a thread designer this means that he\she should follow one of the following designing paths:

- The first way would be to create threads for more than one processes that have a sum of CPU time at least over 75% in order to have a considerable influence on the total execution time

- Another way would be to try to fit as much code to the SPEs which is regarded to be faster than the PPE. The problem with this method, except for the limited sizes of the SPE's Local Storage and the considerable enough cost of DMA transfers, is the architecture of the SPE's itself. As explained on Section 2.1.3, the SPE's were developed for calculations and vector multiplication applications. When it comes to architecture though, the SPEs do not include "Out-of-order execution", "branch prediction" or caches (they have their own Local Store but does not support the cache's characteristics of a Pentium). This is a trade off which gives you more computing power along with low power consumption, but the simplicity of the architecture results in an even worse execution time than the PPE (already proved to be slower than a Pentium IV).[30][31]

We chose the first path for this project's implementation. The next step was to find the most time-consuming functions of the first three processes. The results on Table 4.1 show that the most compute-intensive functions were once again not consuming a considerable percentage of the CPU time. Furthermore, there were different functions for each of the processes and with varying percentages as well. For our implementation this meant that had to be limited in terms of the arguments and the datasets tested. Thus, to find a formula where we could apply the *Threading Processing* that was the initial target of this thesis. The cost of limiting the applicability of our

project was seriously taken under consideration and we decided to profile the other packages of NBIS in order to find a package with more "appealing" results.

BOZORTH3 on Pentium IV				
Dataset	**Argument**	**1st process (%CPU)**	**2nd process (%CPU)**	**3rd process (%CPU)**
mates	-M	bz_match_score (53%)	bz_comp (17,1%)	bz_match (9,4%)
gallery	-M	bz_match_score (49,6%)	bz_match (21,3%)	sort_order_decreasing (7,3%)
probes	-P	bz_match_score (43,3%)	bz_comp (28,2%)	bz_match (11,7%)
gallery	-v -M	bz_match_score (39%)	bz_comp (26,4%)	sort_order_decreasing (6,7%)

Table 4.1: Bozorth3 process analysis : datasets - Processes (their CPU_time % percentage)

Pcasys Package

`Pcasys` was the second package we did the profiling to and its vtune's results had been more positive than Bozorth3's. Pcasys is a fingerprint classifier that reduces the number of candidate searches required to determine if a fingerprint matches a print on file. For more details for the package Pcasys refer to Sections 3.1.2 3.1.3 .
Table 4.2 shows the three most time-consuming processes and their % percentage of CPU time for a few different datasets we tested.

Pcasys on Pentium IV			
Dataset	**1st process (% CPU)**	**2nd process (% CPU)**	**3rd process (% CPU)**
1 image	join_Lets (55,2%)	rors(6,2%)	passb4 (5,6%)
10 images	join_Lets (49,8%)	fft2dr (6,8%)	passbf4 (5,8%)
100 images	join_Lets (51,4%)	rors (4,9%)	enhnc (3,9%)
270 images	join_Lets (53,7%)	passb4 (5,9%)	rors (5,6%)
1000 images	join_Lets (53,6%)	rors (5,8%)	passbf4 (5,7%)
2700 images	join_Lets (54.6%)	rors (6,2%)	enhnc (5.6%)

Table 4.2: Pcasys process analysis : datasets - Processes (their CPU_time% percentage)

4.1.3 Porting to PPE

When we started this project over a year ago, our knowledge for Thread Processing on a multicore processor was limited to what the few manuals for Cell Programming and PS3 programming said about porting an application. Unfortunately, all the examples and the methos produced in the manuals had in common the way to treat the Makefiles. To be more precise, the use of two Makefiles (one in the SPU folder and one in the PPU's) was implemented. On the other side, when trying to port a huge application , there might be many different Makefiles and executables located into different directories (in our case we had 140 different Makefiles and over 500 executables placed in different folders)

The next step seemed to be the creation of a new Makefile for the PPE. A new Makefile that would do exactly what the other 140 Makefiles did but would call the ppu-gcc compiler instead of the original gcc. This process of creating the Makefile finally lead us to a dead end. After a lot of tries and changes to all the Makefiles the build errors continued to appear to the console. The complexity of the Makefiles of the NBIS distribution we had then was above our limits. The Makefiles were written in the "old style" ! The new-style Makefiles are more compact and easier to get correct for certain features (such as CONFIG_ options that enable more than one file). The "new-style variables" are simpler and more powerful than the "old-style variables". Moreover, the "new-style" Makefiles support associative indexing (a single line simple assignment equals to eight lines of code in an "old-style" Makefile). As a result, many subdirectory Makefiles shrank more than 60 %. Our project was divagating from its original purpose to Software's Engineering fields and ways to manipulate Makefiles.

Writer's Note : In time, all arch Makefiles and subdirectory Makefiles will convert to the new style.

Fortunately a new distribution of NBIS was released during our experiments and although the Makefiles were once again in different folders they had been written this time in "new-style". This distribution was developed and released for more recent machines including Linux distributions using the `GNU gcc` compiler and the `gmake` utility. To us, this was an unexpected gift apart from a sign to stop and erase a couple months of work that seemed to lead our research to a dead end in the first place.

With the new distribution nothing has changed on the original source code but for the Makefile's structure. There were also no new flag optimizations compared to the previous Makefiles. Therefore, and due to the new process

analysis we did with Vtune, packages and their processes' demands had been stable and so we continued with the `Pcasys` package.

The makefiles of NBIS looked for the gcc compiler by default (it was declared in the parent Makefile located in the top folder of NBIS). We changed that value to ppu-gcc and added few lines of source to raise some flags needed for the PPE. Additionally, we had to include to the Makefile a footer file (containing makefile definitions supplied by the SDK for producing programs) and a couple of header files. A few changes to the binaries folders and their permissions and our Makefile was finally ready for compiling.

The application was built without errors or warnings on PS3's Yellow Dog (version 6_0) and the step *porting to the PPE* was successful. We used the `time` process of Linux apart from Intel's Vtune and both showed that the PPE with its 3.2 GHz frequency was more than x2 slower than the Pentium IV running at 2.66 GHz. The scientific community backed us up with many applications that proved the PPE's execution time to be significantly slower than a Pentium's. Table 4.3 shows the execution times of NBIS for the Pentium IV and the PPE for the same datasets.

Pcasys : Pentium IV vs PPE		
Dataset	**Pentium IV**	**PPE**
1 image	0.398	0.940
10 images	4.024	8.765
100 images	39.450	99.389
270 images	106.119	237.709
2700 images	1024.703	2334.699

Table 4.3: Pcasys execution times : PPE vs Pentium IV

4.1.4 Data analysis of Join_lets

Join_lets proved to be the most demanding function of the Pcasys package as it consumed around 45 % of the total execution time of pcasys executable. With the use of system calls we discovered that join_lets would run 40 times per input file we want to compare. The datasets we tested were consisted of 270 sample files each. As a result, join_lets was called $10800(= 2700 * 40)$ times per dataset. With a closer look to join_lets source code we discovered that there where dependencies between the variables but more importantly it consisted of over ten "branch-loops" also depended with each other. For our implementation this meant that neither the method of data partitioning nor the "data-column" could be applied. The next step to solve this problem would be a data analysis of the Join_Lets function, hoping it would fit to the limited size of the SPEs.

More system calls were used to the original source code printing to the console the sizes of the input arguments. Fortunately, the total size of data input had been around 4 KB ($< 256KB$ available in the Local Stores). Two ideas for implementation could be applied on this case. As already mentioned join_lets runs 40 times per full compare between the dataset and the 270 samples we used. The first idea was to try and fit all 40 runs in every SPU, thus have 6 parallel Synergistic processors with different input files doing 40 compares each with the Database. This method would have a maximum theoretical speed-up of $1/((1-p)+n/p)$, where p=fraction of the code and n=number of cores. The other idea was to have all the SPEs working on the same input file doing a different compare each until the end of the file. The structure of the source though (had dependencies) in addition to the cost of the many DMA transfers required for sending and receiving data lead to the selection of the first method for our implementation.

4.1.5 Overall of NBIS problems

As mentioned in previous section our implementation would be to try and fit the join_lets process to the SPEs in order to deploy Cell's performance abilities. On the other side, we had to limit our thesis' target in terms of applicability as well as in achieving a speed-up in the performance.

Due to the reasons described in Sections 4.1.2 and 4.1.4 our range of Pcasys' potential methods along with our dataset were seriously smaller than the target of this thesis. This thesis' target was to port an existing application to the Cell microprocessor. Porting an existing application means that we should include into our implementation most -if not all- potential arguments, or datasets it originally supported. Moreover, depending on the input data of an algorithm, an application may call different routines or functions. Porting should also support most of these execution methods of the original application, but in our case this could not be applied. Therefore, we rested our hopes to achieving a performance boost in the execution on the CELL, even for limited data and methods.

When we ported the Pcasys to the Cell's PPE we measured it's execution time with the `mfttb` command supported by `spu_intrinsics.h` library. With the use of `time` and the `Intel's Vtune Analyzer` we also measured the execution time of Join_lets process on a Pentium IV running at 2.66 GHz. The PPE proved to be too slow even for a Pentium running at lower frequencies. Furthermore, join_lets would consume a maximum of only 55% of the total execution time. This means that no matter what approach we may have followed for implementation (any method applied only to the Join_Lets function) it would have an effect only on the 55% of the total time. In the best case scenario (SPE's execution time limiting to zero) we would still be considerably slower than the Pentium IV.[2]

4.1.6 Conclusion

Our first try to the CELL could not be regarded as a success. We faced difficulties in every step of our implementation and no matter what method we may have used for porting, we would still be slower than a Pentium IV. The simplicity of the CBEA architecture proved to be the most important factor that led to our premature ending of the NBIS implementation.

In addition to the slow architecture, NBIS source structure was not friendly to multi-processors. The algorithms had not been created for multithreaded execution and due to the dependencies existed in them, optimization by parallel execution could not be applied. In order to test the performance abilities of the CELL, the need for rewriting source for this application seems to be the only way. New algorithms that support multithreading should be implemented, along with changes to the sequential structure of this application.

Unless these are applied, there will be no significant speed-up in the performance and even an ancient Pentium would look like a speed-horse compared to the Cell Broadband Engine.

Writer's Note : Although this design came to a dead end it proved to be -apart from interesting- extremely helpful as provided us with essential knowledge on multicoring, later applied on the `MAFFT` application.

4.2 MAFFT Implementation

4.2.1 Introduction

This section gives an overall development flow of our implementation of the MAFFT application on the CELL processor. In next section (4.2.2), the programming model we chose for our implementation and the reasons for our choice are provided along with the **Application Enablement Process** (`AEP`) of the implementation. Our implementation consists of five main steps:

1. Make the profile of the original code (Section 4.2.3) of MAFFT ,
2. Port the application only to the PPE processor (Section 4.2.4,)
3. Data flow analysis and setting the PPE to act as a control (Section 4.2.5),
4. The DMA (Direct memory access) data transfers to the SPE's Local Storage (Section 4.2.6) and
5. The overall execution implementation with the 6 SPE's available on the PS3 (Section 4.2.7).

4.2.2 Programming Model & AEP

The programming model we chose for our implementation was the function-offload model with the use of Asymmetric-Threads (Asymmetric-Thread Runtime Model).The models are explained in **section** 2.1.5. We chose these models as they are suggested for optimizing an existing application. After we statically identified the critical on performance functions , we decided which functions should be *offloaded* to the SPEs. Using these methods was the quickest way to speed-up the computation-intensive functions of an existing application (In Chapter 5 the need for re-writing sources is explained).

The process of enabling an application (`Application Enablement Process`) on Cell processor (Figure 4.1 shows our Application Enablement Process) can be iterative on every hotspot of the application. Each one can be dealt with the steps mentioned above until the performance is good enough. Therefore, this AEP may be applied in different applications and should give depending on the algorithm an increase in the performance.

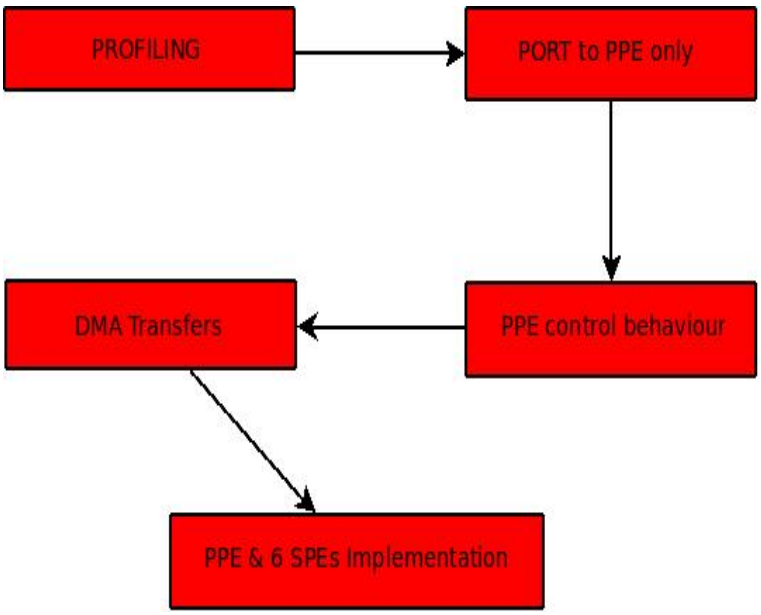

Figure 4.1: Application Enablement Process

4.2.3 Profiling the original code

The first step of our implementation was to make the profile of the application. In order to find the most demanding functions of the **MAFFT** application we used *INTEL's Vtune Analyzer 9.1 for Linux* on a Pentium IV 2.66 GHz with Ubuntu v.8.04 .(Details of measuring the performance are given on **Chapter** 5)

We measured all the methods described in Chapter 3.2.2 with vtune. The results for the progressive methods ,in contrast with **NBIS**'s performance analysis (Section 4.1.2), have been an auspicious start for us. Table 4.4 shows that the most-demanding process measured in execution time was by far the disstbfast (when running with option –retree 1 - 1^{st} *progressive method*) or the tbfast (with option –retree2 - 2^{nd} *progressive method*). For an implementation on the Cell processor this is a case where the *function-offload* model is suggested as the way to start. The measurement of those two processes showed that the function A__align was the most computation-intensive function , as shown on Table 4.5.

The results of the profiling showed us the way for the implementation. Based on these , we decided to choose the *Function-Offload Model* and port to the SPE's only the **A__Align** function.

Writer's Note : There is no need offloading the other functions as the cost of DMA transfers - needed for the data transfers to the SPE's LS. - is significant and any potential speed-up from porting to the SPEs would be lost.[24][27][28][29][26]

DISSTBFAST		
Dataset	**CPU (%)**	**Execution time (Sec)**
ex10x766	40	0.336
ex10x1460	57	1.320
ex100x766	45	1.768
ex59x5271	41	53.254
flyDNA10x766	43	0.180
flyDNA100x766	48	0.812
flyDNA100x1403	49	4.649

Table 4.4: Vtune system analysis for *MAFFT*

A__Align		
Dataset	**CPU (%)**	**Execution time (Sec)**
ex10x766	85	0.285
ex10x1460	87	1.148
ex100x766	84	1.485
ex59x5271	89	47.910
flyDNA10x766	77	0.138
flyDNA100x766	81	0.657
flyDNA100x1403	79	3.673

Table 4.5: Vtune functions analysis for *Disstbfast*

4.2.4 Port to PPE only

Porting to the PPE proved to be an easy case for us. The Makefile was written for any Linux distribution (with kernel 2.6) and so there were very few changes we had to do in order to built the application on the Yellow Dog. The main parameter we had to alter was the compiler the makefiles looked for , from gcc to ppc-gcc. This automatically sets the Power-Pc compiler environment needed for the PPE of the PS3 to run the application. The

source files were all located in the same directory and so no other changes were needed except for changing the binaries' installation directory and their permissions to root's. The default flag for optimizations was initially at -O3 and we tested the performance with the optimizations and without them (-O0) as we had already done on the Pentium IV.

With or without the O3 flag , the PPE (running at 3.2GHz) was 4x times slower than the Pentium IV (running at 2.66 GHz). This can be explained as the role of the PPE is to behave as a controller of the SPE's and so it was made with a lot simpler architecture than the other general purpose processors. Most modern microprocessors devote a large amount of silicon to executing as many instructions as possible at once by executing them "out-of-order" (OOO). This type of design is widely used but it requires hefty amounts of additional circuitry and consumes large amounts of power. With the PPE, IBM have not done this and have instead gone with a much simpler design which uses considerably less power than other PowerPC devices - even at higher clock rates. This design has however the downside of potentially having rather erratic performance on branch laden applications. Such a simple CPU needs the compiler to do a lot of the scheduling work that hardware usually does; so a good compiler will be essential. In conclusion , there is nothing impressive about the PPE's architecture other than it is a small, fast, efficient core. Against a Pentium 4 or an Athlon 64, the PPE would lose undoubtedly, but the PPE's architecture is one answer to a shift in the performance.

Table 4.6 shows the execution time for the PPE and the Pentium IV with the use of -O3 optimization flag.

MAFFT		
Dataset	**Pentium IV**	**PPE**
ex10x766	0.336	0.705
ex10x1460	1.320	2.414
ex100x766	1.768	4.292
ex59x5271	68.322	122.234
flyDNA10x766	0.180	0.336
flyDNA100x766	0.812	2.080
flyDNA100x1403	1.912	4.649

Table 4.6: Pentium IV vs PPE execution times

Writer's Note : Should Cell ever make its way into a PC, the PPE would definitely have to be "beefed" up, or at least paired with multiple other PPEs.

4.2.5 PPE control behavior

As already mentioned in section 2.1.3 the SPEs operate on a 256KB Local Store, each of the SPUs possess , for data transfers and instruction fetching. After having decided which method to use for the implementation, our next step was to locate the calls of our function (A__Align) during the `mafft's` execution. The use of "printf" showed that our function was called once per sequence of any input file and there were two different ways of calling A__Align. It was called either directly from the `treebase` function or the treebase would call the `F_Align` function and it would then call A__Align. Since the methods aren't conflicting, threads should be created just before the call of A__Align on both cases.

The PPE will start running the Mafft and it will pause it's execution just before the call of A__Align from processes `treebase` or `F_align` - depending on the input sequence of the dataset. Afterwards, it creates a thread and starts sending data to the SPEs' Local Storages. At this point we had to determine the amount of data needed to be transfered to the SPEs' LS. The data size of the inputs was different not only between the datasets but also from sequence to sequence. In addition to this, the limited size of the LS was taken under consideration. Therefore, the method of `data partitioning` was implemented to partition the data and to solve any bus errors (as the ppu-gcc compiler identifies them). Details for the method are given on Section 4.2.6.

The declaration of our function was initially located at the SAlignmm.c file and was removed as it would be offloaded to the SPEs by any calls to our function. Therefore , in order to solve the "duplicate or unused declaration" warnings, the A__Align source was saved to a new C source file (`spe_A__Align.c`). The PPE's role has turned from computing to being just a controller of the SPEs. It stops it's execution , creates the context for the SPE's and loads the program to the SPEs. From the current thread the execution starts and it is and the PPE waits for the return of the SPEs. After fetching the data from the SPE's it destroys the context and continues its execution. The scheduling process had some overhead and this proved to be crucial for the design as it takes a significant amount of time compared to the execution time. In order to reduce the overhead, context was created just before and destroyed immediately after one SPE_program load. Figure 4.2 shows the PPE's control behavior.

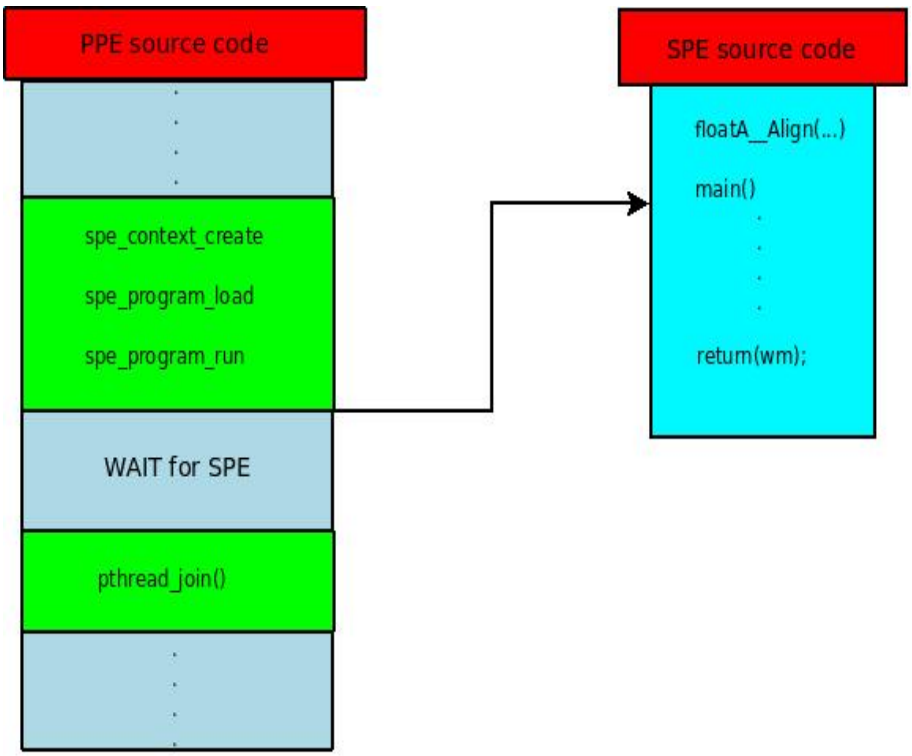

Figure 4.2: PPE control

4.2.6 DMA transfers

The architecture of Cell defines that the SPEs use data only from their LS. The need for data transfer was fulfilled by the commands supported by the CELL for DMA transfers. In order to initiate the request for data, we used the `SPE initiated transfers` as the `PPE initiated` transfers are slower than the SPEs. Moreover, there are size limitations to a single transfer (maximum supported is 16KB per transfer) and the data should be a multiple of 16. These problems were solved with the use of the `mfcget` and `mfcputs` commands used for moving data from LS to main storage and vice versa in addition to a 16-byte data alignment implemented on the control block of the data.

As mentioned in previous section our input arguments are not stable in terms of size and a special treatment was needed in order to avoid the bus errors or any mistakes in the output. Thankfully, instead of using the `libspe` header we used the newest version (`libspe2`) and the posix threads supported. The deprecated libspe version is asynchronous. That means, if you try to create more threads than SPEs available, the program tries to

create threads but sometimes fails because the SPEs are busy. On the other side , the libspe2 header with posix threads is synchronous. They will block and queue the incoming thread until the requested resources are free again without the overhead we would get from trying this with the old libspe. [16]. The use of posix threads is essential for creating, manipulating and managing threads, as well as synchronize between threads using mutexes and signals. Their ability to be build on different machines (Solaris, PPE(Cell's), PowerPcs etc.) is very useful for future work implementations.

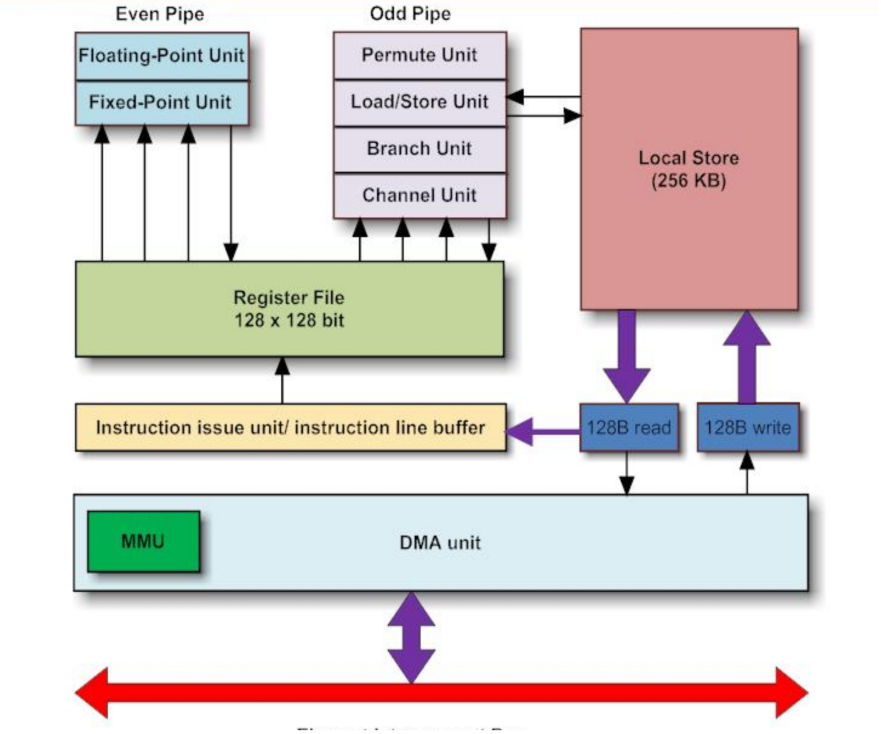

Figure 4.3: DMA transfers from and to LS

After fetching the input arguments from the main memory to the Local Storage of each SPE, the SPEs are ready for execution. While processing the input they stall the transfers for as long it takes to the SPE to complete the execution and return the result value (in our case a float `wm`) to the main memory. Then, they are free again for receiving the data from the main store. We categorized our datasets into two categories in terms of data size. Those whose data were less from the size limits of LS (256KB) and those whose data had an overhead. Fortunately, most of our datasets could fit on the Local Storage except for very long sequences for which we had to

fetch data more than once. The main categorization of the dataset, though, was made depending on the input sequence each dataset used. Depending on the sequence the *FFT-Transform* our method uses, becomes slower and has to create tables with a double size than the original (cases where the sequences did not have similarities) while on "long-distanced" sequences the *FFT* becomes more applicable thus faster, as it creates tables half of the original size. Therefore, our inputs where separated into two groups:

- `examples` :We tested four different in terms of size and sequence number datasets and they are the cases in which we saw the best performance boost.
- `flyDNA` :We tested three different sequences in which mafft's execution time was a lot faster on the Pentium than the PPE and even faster than our final implementation with the six Synergistic processing units available.

DATASETS		
No.	**Examples**	**flyDnas**
1	ex10x766	flyDNA10x766
2	ex100x766	flyDNA100x766
3	ex10x1460	flyDNA100x1403
4	ex59x5271	-

Table 4.7: Two categories of dataset used for the implementation

4.2.7 Implementation with 6 SPEs

This stage of our implementation started with creating an SPE source file. We copied the original A__Align source and added the headers needed for the SPEs. Those headers were also placed to the SPE_Makefile in addition to the headers necessary for the threading procedure and the communication with the PPE. In more details, we created a folder called mafft as a parent directory and created 2 subfolders called SPU and PPU.

PPU_folder The PPU folder contained all the original folders with the changes described on Section *Porting to PPE only* (4.2.3). When building an application on an SPE, an embedded library is created (*mylib.a*). This embedded library had also to be imported to the PPE. In order to allow consolidation of the spu program into the ppe binary we added two more flags to the original Makefile (*-lspe2 -lpthread*). Those flags were used only for the object files that needed the A__Align function and had to look for the SPE executable. Furthermore, we had to create the threads, send data to the Local Stores and receive the result back to the main memory, on every process that called A__Align. All the other folders and their contents didn't need any changes. An example script of our thread process is shown on figure 4.4.

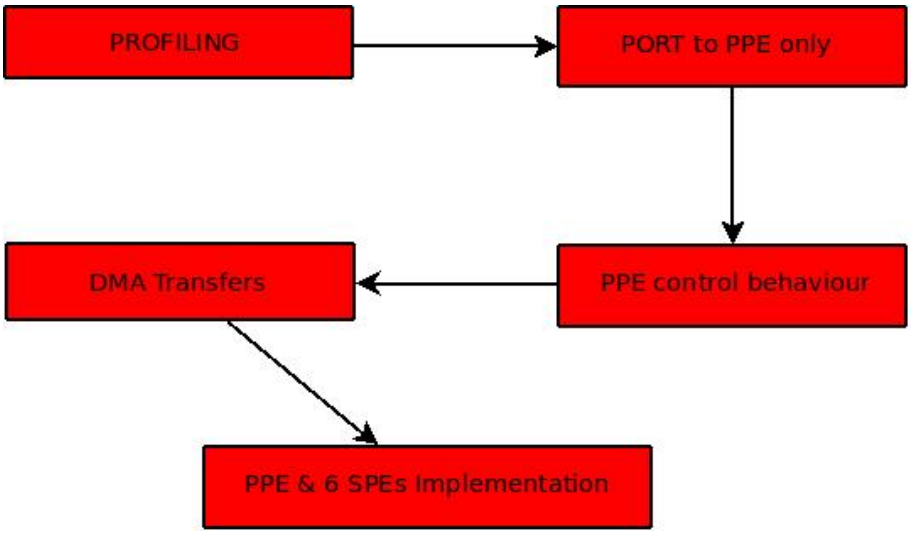

Figure 4.4: PPU source

SPU_folder: This folder contained the SPE_A__Align source file written in C with the posix threads and their headers included. In this folder we placed a new Makefile (`SPE_Makefile`) where we set the name of our `SPU_program` and embedded the library *mylib.a*. Furthermore, we had to manually include the files containing any headers or functions' declarations that A__Align called. When building an SPU_application a new executable (with the name set as `SPU_program`) is created apart from the libraries already discussed. It goes without saying that this executable has to be created before the PPU's in order to be available for the PPE. Finally, when setting an SPE_application's name we automatically set the *spu_gcc* compiler along with some flags (`-W -Wall -Winline -Wno-main -I. -I`) needed for the compiling and the join of the SPU with the PPU. When the SPE_source is built, the SPE_executable along with the object file *spe_A__align-embed64.o* and the embedded library mylib.a are created in the folder spu.

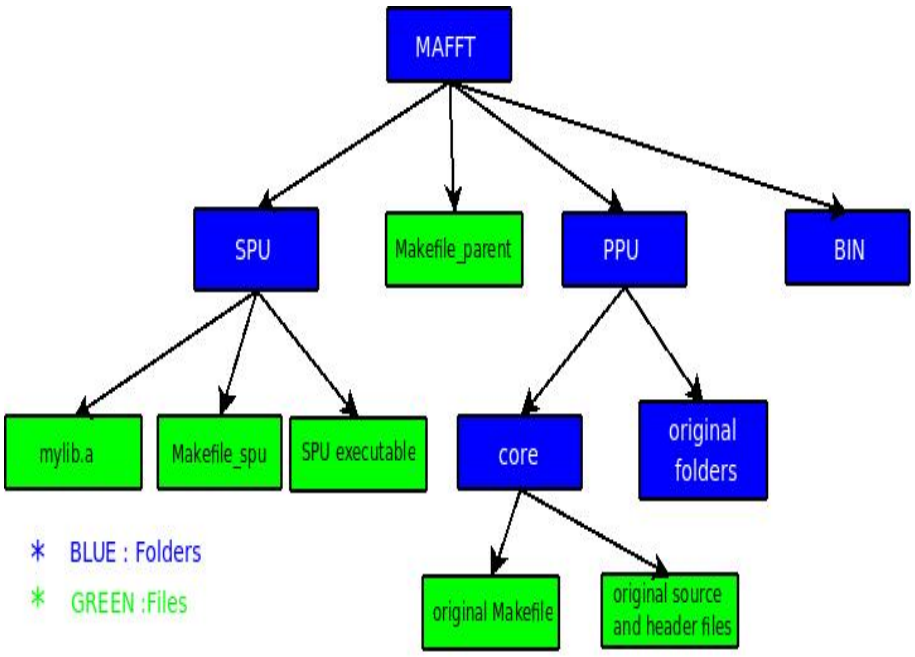

Figure 4.5: Folders organized in a tree

MAFFT -1st progressive method			
Dataset	**Pentium IV**	**PPE**	**PPE & SPEs**
ex10x766	0.336	0.705	0.198
ex100x766	1.768	4.292	1.263
ex10x1460	1.320	2.414	0.778
ex59x5271	62.322	122.321	40.503
flyDNA10x766	0.180	0.336	0.225
flyDNA100x766	0.812	2.080	1.898
flyDNA100x1403	1.912	4.649	3.815

Table 4.8: Pentium IV vs PPE vs PPE & SPEs execution times

Parent_makefile: In the parent directory (`mafft`) we created another Makefile (Makefile_parent) in order to connect the two Makefiles placed in the spu and the ppu folders, with each other. In more details, we first build the makefile of the spu folder and then the ppu's. The original Makefile needed two instructions for build (`make` and `make install`). This conflicted with the build on the SPU which had no tags at all. After a few more changes to the ppe's Makefile we managed for our implementation to need only one instruction (a single `make`) on the top folder where the Makefile_parent is located. A couple bus errors were then faced but were all solved by the initialization of our control block of data.

With the use of the system call *fprintf* in our spe_source we were certain that the SPE_code we implemented was executed and even before timing the results it was obvious that there was a significant improvement on the performance compared with the original PPE's execution time. The time simulations proved our suspicions and the *CELL* **JUST** become a powerfull processor. Especially on the slower datasets("examples") the execution time was enough faster than the Pentium. Unfortunately, for the fast cases where the FFT is applicable most, the architecture of the Pentium was too fast for us. An example of those time measurements is given on Table 4.8 while Tables 4.9 4.10 show the speed-up between the Pentium and our implementation and the PPE versus our implementation.

MAFFT -1st progressive method			
Dataset	**Pentium IV**	**PPE & SPEs**	**Speed-up**
ex10x766	0.336	0.198	1,8x
ex100x766	1.768	1.263	1,4x
ex10x1460	1.320	0.778	1,7x
ex59x5271	62.322	40.503	1,5x
flyDNA10x766	0.180	0.225	-0.8x
flyDNA100x766	0.812	1.898	-0.4x
flyDNA100x1403	1.912	3.815	-0.5x

Table 4.9: Pentium IV vs PPE & SPEs execution times - SPEEDUP

MAFFT -1st progressive method			
Dataset	**PPE**	**PPE & SPEs**	**Speed-up**
ex10x766	0.705	0.198	3.5x
ex100x766	4.292	1.263	3.4x
ex10x1460	2.414	0.778	3.1x
ex59x5271	125.321	40.503	3.1x
flyDNA10x766	0.336	0.225	1.5x
flyDNA100x766	2.080	1.898	1.1x
flyDNA100x1403	4.649	3.815	1.3x

Table 4.10: PPE vs PPE & SPEs execution times - SPEEDUP

Chapter 5

Evaluation

Introduction

This chapter's purpose is to compare the performance of our designs for the `NBIS` and the `MAFFT` applications with the performance of other processors. Apart from the explanation of the results, the measuring procedure as well as the tools we used are also presented in the chapter. This chapter is organized in two sections. Section 5.1 shows the evaluation results of `NBIS` while the results of `MAFFT` are in Section 5.2.

5.1 NBIS evaluation

Introduction

In sections 4.1.5 4.1.6 we showed that our implementation on the NBIS application ended earlier than we expected and consequently this section is also shorter than expected. This Section is organized into two subsections. The time analysis and the measuring methods used for the Pentium IV are provided in the first section (5.1.1), while the results after porting to the PPE are shown in Section 5.1.2.

5.1.1 Evaluation on Pentium

Our reference machine was a Pentium IV running at 2.66 GHz with an 1GB DDR RAM running at 333 MHz under the Linux operating system *UBUNTU version 8.04 Hardy*. The execution time of the total application was measured with the `time` function supported by ???. We also used *INTEL's VTUNE PERFORMANCE ANALYZER* version 9_1 for Linux to discover the most consuming processes and their percentage of the total execution time.

`TIME` function The time function supported by our O/S showed us the execution time of Pcasys. It provided us with three different time information:

1. `real:` The total real time it took to the PC to return the exit value along with other processes running on the PC.
2. `user:` The total time of our application only- both the computational and the application's system calls times are calculated,
3. `system:` The total time of the system calls of our application.

Different O/S treat system calls differently and so the printfs-system calls were removed. In all datasets tested the system calls execution time was less than 0.01 % of the user time and therefore we used the `user` time as our reference. The time function proved to be extremely accurate and most runs (on the same dataset each time) had almost the same `user` value.

Pcasys - dataset:mates			
RUN	**real**	**user**	**system**
1^{st}	0.203	0.104	0.002
2^{nd}	0.212	0.106	0.002
3^{rd}	0.198	0.101	0.002
4^{th}	0.204	0.103	0.001

Table 5.1: Pcasys dataset mates time analysis

Tables 5.1 5.2 5.3 show our application execution times for different datasets, running on our reference machine, as they were printed to our console.

Vtune Analyzer Vtune is used for analyzing throughout the development process to produce faster, more efficient code. Source and disassembly views show the developer exactly the lines of code and clockticks they require. Therefore, for our reference computer (CPU at 2.66 GHz) we had to divide the number of clockticks calculated from Vtune with the CPU frequency. On a General-Purpose system like our Pentium many processes are running apart from our application on the same time (*Round-Robin system*). It is natural, therefore, to have different number of clockticks on every single run even with the same input dataset (due to the different `misses` or `stalls` that may take place while execution). In order to be accurate with our measurements we ran each dataset more than 10 times in order to get an

Pcasys - dataset:gallery			
RUN	**real**	**user**	**system**
1st	0.102	0.028	0.001
2nd	0.107	0.030	0.001
3rd	0.103	0.035	0.002
4th	0.114	0.029	0.002

Table 5.2: Pcasys dataset gallery time analysis

Pcasys - dataset:probes			
RUN	**real**	**user**	**system**
1st	0.071	0.008	< 0.001
2nd	0.079	0.009	< 0.001
3rd	0.064	0.008	< 0.001
4th	0.069	0.009	< 0.001

Table 5.3: Pcasys dataset probes time analysis

average performance analysis. The average performance demands are shown in Section 4.1.2 while the clockticks of different dataset runs are shown in table 5.4.

Pcasys		
Dataset	**total(%)**	**join_lets(%)**
1 images	54	47
10 images	62	50
100 images	47	49
270 images	51	45
2700 images	53	44

Table 5.4: Pcasys vtune analysis

5.1.2 Cell's evaluation

After porting successfully to the PPE, the PS3 was able to run the Pcasys application. We used the `time` function also supported by our O/S (Yellow Dog) and measured the performance of the PPE. The execution time results of pcasys for the same datasets tested on the Pentium are shown on Tables 5.5 to 5.8.

Pcasys - dataset:1 image			
RUN	**real**	**user**	**system**
1st	1.988	0.940	0.236
2nd	1.997	0.939	0.233
3rd	1.989	0.939	0.234
4th	2.003	0.941	0.233

Table 5.5: Pcasys (1 image) time analysis

Pcasys - dataset:100 images			
RUN	**real**	**user**	**system**
1st	90.755	87.385	2.605
2nd	89.864	87.315	2.598
3rd	89.997	87.934	2.599
4th	91.235	87.885	2.601

Table 5.6: Pcasys (100 images) time analysis

Pcasys - dataset:270 images			
RUN	**real**	**user**	**system**
1st	125.705	106.119	3.740
2nd	123.864	106.123	3.700
3rd	125.426	107.231	3.799
4th	129.235	106.385	3.744

Table 5.7: Pcasys (270 images) time analysis

Pcasys (2700 images)			
RUN	**real**	**user**	**system**
1st	2407.729	2330.699	66.137
2nd	2411.543	2332.001	67.132
3rd	2418.012	2330.301	66.989
4th	2415.232	2331.432	67.129

Table 5.8: Pcasys (2700 images) time analysis

5.2 MAFFT evaluation

Introduction

This section is organized in 3 subsections each describing and containing details of the execution time, the methods and tools applied for the analysis of the Mafft application. In Section 5.2.1 the results of our reference machine are demonstrated while in Section 5.2.2 we show the time measurements of the PS3's Cell. In final Section 5.2.3, the final results of our implementation are being shown on tables comparing the PS3's performance versus the (unexpectedly impressive for its age - and characteristics) Pentium IV.

Mafft - dataset:ex10x1460			
RUN	**real**	**user**	**system**
1st	1.051	0.764	0.008
2nd	1.076	0.812	0.080
3rd	1.075	0.836	0.080
4th	1.059	0.842	0.081

Table 5.9: Mafft dataset ex10x1460 time analysis

5.2.1 Pentium

For the time analysis of our application we used the same tools described in Section 5.1.1 for the NBIS application. Due to the measurements shown in tables 5.9 to 5.14 we calculated the execution time of `A__Align` for the same 3 datasets. The execution time of those datasets in addition to few more datasets tested as inputs in A__Align are shown in Tables 5.15 to 5.20.

Mafft - dataset:ex59x5271			
RUN	**real**	**user**	**system**
1st	68.088	61.924	0.992
2nd	67.438	60.962	0.982
3rd	71.824	61.596	0.964
4th	69.452	60.835	0.930

Table 5.10: Mafft dataset ex59x5271 time analysis

Mafft - dataset:flyDNA100x1403			
RUN	**real**	**user**	**system**
1st	2.859	1.824	0.088
2nd	2.617	1.796	0.098
3rd	2.946	1.844	0.088
4 4th	2.952	1.925	0.085

Table 5.11: Mafft dataset flyDNA100x1403 time analysis

Mafft - dataset:ex10x1460		
RUN	**total**	**A__Align**
1st	1.320	1.148
2nd	1.348	1.153
3rd	1.369	1.198
4th	1.405	1.288

Table 5.12: Mafft (ex10x1460) vtune analysis

5.2.2 Porting to PPE only

At this stage of our implementation we used the `time` function supported by the Yellow Dog in addition to the use of `dynamic counters` for dynamic profiling. We used time-base registers whom frequency is set by default at 79.8 MHz instead of the 3.2 GHz clock of the PPE. We included the library `spu_intrinsics.h` which supports the commands required for setting the time counters. At this point of our implementation the time values of `user time` function were almost identical with the dynamic counters measurements. Finally, we used a 5bit precision on the counters while the `time` function has a default value of 3-bit.
Tables 5.22 5.23 and 5.24 show the PPE's user execution time counted with the two methods described.

5.2.3 Porting to PPE and SPE's available

This section presents the time results of our final implementation and compares it with the previous results of the Pentium and the PS3. The tools we used were once again the `time` function and the `dynamic counters`.

Mafft - dataset:ex59x5271		
RUN	**total**	**A__Align**
1st	53.255	47.945
2nd	55.878	49.233
3rd	58.345	51.322
4th	55.656	49.350

Table 5.13: Mafft dataset ex59x5271 vtune analysis

Mafft - dataset:flyDNA100x1403		
RUN	**total**	**A__Align**
1st	1.912	1.432
2nd	1.879	1.359
3rd	1.946	1.473
4th	1.902	1.415

Table 5.14: Mafft dataset flyDNA100x1403 vtune analysis

In this stage, though, we added few new time counters to the PPE implementation. In more details, we used counters at the stage where A__Align was first called until it returned its output value (*pscore*) to the function that called it. By doing so, we managed to time the A__Align process running on the SPEs (SPE_A__Align).

Table 5.25 show the execution time of our implementation versus the PPE and the Pentium for different datasets tested.

The time measurements to the SPE's with the use of dynamic counters is shown on Table 5.26 and is compared with A__Align's execution time on the PPE and the Pentium.

A__Align - dataset:ex10x1460		
RUN1	**CPU (%)**	**Execution time (Sec)**
1st	85.32	1.149
2nd	86.12	1.138
3rd	84.77	1.022
4th	85.44	1.213

Table 5.15: A__Align (dataset ex10x1460)execution time Pentium

A__Align - dataset:ex59x5271		
RUN1	**CPU (%)**	**Execution time (Sec)**
1st	88.55	62.345
2nd	86.12	59.887
3rd	84.77	60.349
4th	85.44	61.234

Table 5.16: A__Align (dataset ex59x5271)execution time - Pentium

A__Align - dataset: flyDNA100x1403		
RUN1	**CPU (%)**	**Execution time (Sec)**
1st	77.32	1.431
2nd	74.46	1.392
3rd	77.32	1.485
4th	75.32	1.402

Table 5.17: A__Align (dataset flyDNA100x1403)execution time Pentium

A__Align - dataset: ex10x766		
RUN1	**CPU (%)**	**Execution time (Sec)**
1st	85.39	0.295
2nd	84.22	0.296
3rd	86.53	0.297
4th	86.28	0.298

Table 5.18: A__Align (dataset ex10x766)execution time Pentium

A__Align - dataset: ex100x766		
RUN1	**CPU (%)**	**Execution time (Sec)**
1st	85.39	1.459
2nd	84.22	1.479
3rd	86.53	1.472
4th	86.28	1.476

Table 5.19: A__Align (dataset ex100x766)execution time Pentium

A__Align - dataset: flyDNA10x766		
RUN1	**CPU (%)**	**Execution time (Sec)**
1st	75.92	0.142
2nd	74.54	0.137
3rd	76.35	0.144
4th	76.81	0.139

Table 5.20: A__Align (dataset flyDna10x766)execution time Pentium

A__Align - dataset: flyDNA100x766		
RUN1	**CPU (%)**	**Execution time (Sec)**
1st	75.92	0.652
2nd	74.54	0.647
3rd	76.35	0.644
4th	76.81	0.687

Table 5.21: A__Align (dataset flyDna100x766)execution time - Pentium

Mafft - dataset:ex10x1460		
RUN	**time function**	**Dynamic counters**
1st	2.615	2.59235
2nd	2.322	2.32152
3rd	2.642	2.64005
4th	2.263	2.26125

Table 5.22: Mafft dataset ex10x1460 on PPE time analysis

Mafft - dataset:ex59x5271		
RUN	**time function**	**Dynamic counters**
1st	108.321	108.20213
2nd	107.444	107.21235
3rd	109.655	109.42014
4th	106.525	106.23429

Table 5.23: Mafft dataset ex59x5271 on PPE time analysis

Mafft - dataset:flyDNA100x1403		
RUN	**time function**	**Dynamic counters**
1st	4.649	4.64912
2nd	4.648	4.64792
3rd	4.882	4.88300
4th	4.901	4.90098

Table 5.24: Mafft dataset flyDNA100x1403 on PPE time analysis

MAFFT -1st progressive method			
Dataset	**Pentium IV**	**PPE**	**PPE & SPEs**
ex10x766	0.336	0.705	0.198
ex100x766	1.768	4.292	1.263
ex10x1460	1.320	2.414	0.778
ex59x5271	62.322	122.321	40.503
flyDNA10x766	0.180	0.336	0.225
flyDNA100x766	0.812	2.080	1.898
flyDNA100x1403	1.912	4.649	3.815

Table 5.25: MAFFT: PPE & SPEs vs Pentium IV vs PPE

A__Align -1st progressive method			
Dataset	**Pentium IV**	**PPE**	**PPE & SPEs**
ex10x766	0.285	0.599	0.098
ex100x766	1.485	3.581	0.711
ex10x1460	1.148	2.021	0.393
ex59x5271	52.910	102.192	29.916
flyDNA10x766	0.138	0.231	0.119
flyDNA100x766	0.657	1.365	1.221
flyDNA100x1403	1.546	3.673	2.937

Table 5.26: A__ALIGN : PPE & 6SPEs vs Pentium IV vs PPE execution times

Chapter 6

Conclusions & Future Work

Introduction

This chapter's purpose is to summarize the results of this thesis, as well as this project's contribution. The chapter is organized in two sections- one for each application- and at the end of each section, some thoughts for future work are being presented.

6.1 NBIS

6.1.1 Conclusions

The lack of experience we had on multicore programming techniques and their development tools made this part of the thesis more than challenging. In addition, the NBIS application we used for this project has a huge source with high complexity; multiplying therefore our development problems. Furthermore, the disappointing [1] results of the processes analysis we did with `Vtune Analyzer` created another obstacle to our implementation.

Although many problems were faced and solved during this project, this work was to terminate before reaching it's final point. At the beginning of this project, after having studied the "parallel programming" techniques, we decided to follow the "function-offload" method for our implementation. Unfortunately, the applicability of the original application had to be seriously limited and the metrisis of the execution times on the PPE versus the Pentium IV had also been disappointing. Consequently, our implementation of the NBIS application ended prematurely and we decided to apply our knowledge on the MAFFT program.

[1] Due to the algorithm's structure

6.1.2 Future Work

Our implementation was limited to applying the "offload-function" model techniques to the NBIS application. Therefore, we covered almost every aspect of this particular model and came to the conclusion that this model would be no effective on this application. Some suggestions for future work are:

- A new approach with another programming model (Section 2.1.5). Our suggestion would be to try the "Streaming Model" or the "Shared-Memory Multiprocessor Model".
- The best choice would be to reconstruct the source code of the application. The need for limiting the dependencies of the existing algorithm is essential in order to achieve speed-up from parallel processing.
- For our model, our only suggestion would be to send more than ten processes of the application to the SPE's (they should cover at least a total of 75% of the total execution time). Not only this is painful path to follow, the speed-up should NOT be taken for granted as it would still depend on the parallelization of the processes and the cost of the data transfers.

6.2 MAFFT

6.2.1 Conclusions

Except for the porting to the PPE -thus enabling a game console to execute the MAFFT program- this project's main contribution was the achievement of a significant speed-up to the MAFFT's execution time on the Cell multiprocessor. At this stage of our implementation though, we already had the experience with the NBIS application and consequently most of the problems faced on the tools were rapidly solved. The new kind of problems we faced were about the porting to the SPE procedure. The fact that our function would be called from different processes depending on the dataset and the method of execution made the implementation a little trickier. Data partitioning matters occurred creating "bus errors" during the data transfers and therefore scheduling and synchronization issues came up in order to resolve these.

The Cell was initially designed for games and multimedia but could also be regarded as a very promising architecture for scientific computations. The absence of specialized software tools in addition to the low-level architecture of the PPE (and even more of the SPEs) force the developer to take into consideration most- if not all-of the following matters:

- Porting to the PPE may be trickier than expected if the version of the application is old
- The Local storages of the SPEs have a limited size leading to the need of data partitioning.
- DMA transfers also have limitations
- Scheduling issues depending on the algorithm's structure and it's processes nature.

6.2.2 Future Work

This project is a complete work which exhausted most of the possible improvements that could be done with the use of the "function-offload" model. Our suggestion for future work would be to focus on a different model. The "Streaming Model" is again our proposal for potential speed-up or any "SPE-centric" model. The developer should bear in mind that whatever design is used, most of the code emphshould be ported to the SPEs and the PPE should react like a controller to the Synergistic units avoiding as much computational work as possible.

A few suggestions for future work to our implementation are proposed below:

- SIMD vectorization of the SPE's (SIMD on the PPE would not be as effective) in order to allow a single instruction to be applied to multiple data elements.
- Send more processes (not only the most time-consuming) to the SPEs in order to effect the total execution time even more.
- Categorize the dataset input [2]. By doing so, we would able to predict the number of A__Align calls; thus the percentage of the total execution time and the speed-up.

[2] This requires cooperation with a more specialized to Biology issues scientist

Bibliography

[1] J.A.Kahle, et al., *"Introduction to the Cell Microprocessor,"* IBM Systems Journal, vol.49, no.4/5, 2005.
17

[2] S. Williams, et al., *"The potential of the Cell processor for scientific computing,"* in Third Conference on Computing Frontiers CF'06, New York, 2006.
55

[3] IBM, *"Cell Broadband Engine Programming Tutorial,"* Oct.2007, Version 3.0.
21, 29

[4] IBM, *"Cell Broadband Engine Programming Handbook,"* Apr.2007, Version 1.1.
18, 21

[5] IBM, *"Cell Broadband Engine Architecture Forum.[Online],"* http://www.ibm.com/developerworks/forums/forum.jspa?forumID=739\&cat=46.
20, 24

[6] IBM, *"Cell Broadband Engine Resource Center.[Online],"* http://www.ibm.com/developerworks/power/cell/documents.html.
20

[7] H. Peter Hofstee IBM Corporation *"Introduction to the Cell Broadband Engine"* Aug.2005.
20

[8] Daniel A. Brokenshire (brokensh@us.ibm.com), *"Maximizing the power of the Cell Broadband Engine processor: 25 tips to optimal application performance"* IBM STI Design Center,June 2006.
24

[9] IBM, *"SPU Instruction Set Architecture,"* Jan.2007, Version 1.2.
21

[10] IBM RedBooks, *"Programming the Cell Broadband Engine Examples and Best Practices,"* Dec.2007.
29

[11] Torsten Hoefler, *"The Cell Processor -A short Introduction-,"* Nov.2005.
17

[12] IBM, *"Cell Programming Tips & Techniques,"* Systems and Technology Group,Sept.2005.
24

[13] IBM, *"Software Development Kit for Multicore Acceleration,"* Programmer's Guide, version 3.0.
18

[14] Cédric Augonnet, *"An introduction to IBM Cell Processor,"* Amsterdam,2007.
24

[15] University of Georgia, *"One-Day IBM Cell Programming Workshop at Georgia Tech,"* IBM Systems and Technology Group,2007.

[16] IBM.[Online] *"Changes in libspe: How libspe2 affects Cell Broadband Engine programming,"* http://www.ibm.com/developerworks/library/pa-libspe2/
63

[17] Sony, *"Playstation.[Online],"* http://gr.playstation.com/ps3/index.html.
30

[18] MAFFT version 6 [Online] *"Multiple alignment program for amino acid or nucleotide sequences,"* http://align.bmr.kyushu-u.ac.jp/mafft/software/.
41

[19] Kazutaka Katoh,Hiroyuki Toh2Recent, *"Developments in the MAFFT multiple sequence alignment program,"* Aug.2008.
41

[20] Kazutaka Katoh, Kazuharu Misawa1, Kei-ichi Kuma, Takashi Miyata, *"MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform,"* Department of Biophysics, Institute of Molecular Evolutionary Genetics, Pennsylvania State University Apr.2002.
41

[21] NBIS[Online] *NIST BIOMETRIC IMAGE SOFTWARE (NBIS)* http://fingerprint.nist.gov/NFIS/, National Institute of Standards and Technology.
33

[22] National Institute of Standards and Technology, *"User's Guide to NIST Biometric Image Software (NBIS) -non-export-control,"* C. I. Watson, M. D. Garris, E. Tabassi, C. L. Wilson, R. M. McCabe, S. Janet and K. Ko, Oct.2004.
33

[23] C. I. Watson, M. D. Garris, E. Tabassi, C. L. Wilson, R. M. McCabe, S. Janet and K. Ko, *"User's Guide to Export Controlled Distribution of NIST Biometric Image Software (NBIS-EC),"* National Institute of Standards and Technology, Oct.2004.
33

[24] A.Chow, G.Fossum, and D. Brokenshire, *"A Programming Example: Large FFT on the Cell Broadband Engine,"*Parallel computing, vol33, 2007.
59

[25] Michael Gschwind, David Erb, Sid Manning, and Mark Nutter, *"An Open Source Environment for Cell Broadband Engine System Software,"* Jun.2007.

[26] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, Katherine Yelick, *"The Potential of the Cell Processor for Scientific Computing,"* May 2006.
59

[27] Chang-Burm Cho, Asmita V. Chande, Yue Li and Tao Li, *"Workload Characterization of Biometric Applications on Pentium 4 Microarchitecture,"* 2005.
59

[28] Daniel Jimenez-Gonzalez, Xavier Martorell, Alex Ramirez, *"Performance Analysis of Cell Broadband Engine for High Memory Bandwidth Applications,"* Barcelona Supercomputing Center (BSC), Sept.2007.
59

[29] Vipin Sachdeva, Michael Kistler, Evan Speight, and Tzy-Hwa Kathy Tzeng, *"Exploring the Viability of the Cell Broadband Engine for Bioinformatics Applications,"* IEEE, Jan.2007.
59

[30] Michael Gschwind, Peter Hofstee, Brian Flachs, Martin Hopkins, *"Synergistic Processing in Cell's Multicore Architecture,"* IEEE, Apr.2006.
49

[31] Sadaf R Alam, Jeremy S Meredith, Jeffrey S Vetter, *"Balancing Productivity and Performance on the Cell Broadband Engine,"* Oak Ridge National Laboratory,2007.
49

[32] Thomas William Ainsworth, Timothy Mark Pinkston, *"Characterizing the Cell EIB On-Chip Network,"* IEEE Micro, vol. 27, no. 5, pp. 6-14, Sep./Oct. 2007.
25

[33] Sony Computer Entertainment Inc. (SCEI), *"Outline of PLAYSTATION 3 (PS3) computer entertainment system,"* 1st release,2005.
30

[34] wikipedia.[Online] *"PLAYSTATION 3,"* http://en.wikipedia.org/wiki/PlayStation_3.
30