國 立 交 通 大 學

電子工程學系 電子研究所碩士班

碩 士 論 文

在 PlayStation 3 上的即時多媒體處理

Real-time Multimedia Processing on PlayStation 3

研究生： 陳慶至
指導教授： 劉志尉 博士

中 華 民 國 九 十 六 年 十 月

在 PlayStation 3 上的即時多媒體處理

Real-time Multimedia Processing on PlayStation 3

研 究 生：陳慶至　　Student: Ching-Chih Chen

指導教授：劉志尉 博士　　Advisor: Dr. Chih-Wei Liu

國 立 交 通 大 學

電子工程學系 電子研究所碩士班

碩士論文

A Thesis
Submitted to Department of Electronics Engineering & Institute of Electronics
College of Electrical and Computer Engineering
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of Master
in
Electronics Engineering

October 2007

Hsinchu, Taiwan, Republic of China

中 華 民 國 九 十 六 年 十 月

在 PlayStation 3 上的即時多媒體處理

研究生：陳慶至　　　　　　　　　　指導教授：劉志尉 博士

國立交通大學
電子工程學系 電子研究所

摘要

現今的多媒體發展非常迅速，為了提升資料速率(data rate)和壓縮比率(compression ratio)，使得演算法的複雜度不斷增加。所以通常會使用可程式化處理器而非特定功能積體電路(ASIC)，以因應各種新穎的多媒體標準。在處理器為主的平台上，只需要作軟體的更新，就可以讓新的多媒體應用能夠順利運行，延長產品在市場的壽命。在多媒體應用中，除了傳統處理器所利用到的指令層級的平行度(ILP)和資料層級的平行度(DLP)之外，多執行緒(multi-thread)或多核心(multi-core)架構進一步利用執行緒(thread)層級的平行度(TLP)。多核心處理器能提供很高的運算能力去處理複雜的應用，但是多核心處理器的程式撰寫非常困難。多核心程式撰寫有兩個最主要的問題是，由工作分派所產生的同步花費和工作之間資料傳遞所產生的溝通花費。本論文中採用以幅為基礎(frame-based)的資料切割方式，去消除工作之間的資料依靠關係(data dependency)，減少同步的花費。溝通的問題則是利用資料流動最佳化來解決。在一個著名的多核心架構—PlayStation 3 上，可以使用這些技巧對一個 1080p 多媒體影片解碼器作最佳化。由實驗結果可知，這些最佳化的技巧能使解碼器加速 20 倍，最後超過即時(real-time)的限制以 60Hz 的頻率解碼出 1080p 的影片。

Real-time Multimedia Processing on PlayStation 3

Student: Ching-Chih Chen Advisor: Dr. Chih-Wei Liu

Department of Electronics Engineering
Institute of Electronics
National Chiao Tung University

ABSTRACT

Today's multimedia applications evolve very fast. In order to improve the data rate and compression ratio, the complexity of algorithm is enhanced. Instead of ASIC, programmable processors are usually used to deal with the variety of new multimedia standards. Processor-based architectures can use software patches to keep up with new multimedia applications, thus extend products' time-in-market. However, conventional processor architectures are unable to provide sufficient computing power for real-time constraints. Beyond instruction-level-parallelism (ILP) and data-level-parallelism (DLP) used in conventional processors, multithreaded/multi-core architecture further acquire thread-level-parallelism (TLP) to exploit parallelism in multimedia applications. Multi-core processor can provide high computing power to support complex applications, but multi-core programming is much more difficult. There are two main issues in multi-core programming, synchronization overhead introduced by task parathion and communication overhead arising from data communications between tasks. In this thesis, we adopt frame-based data partition to eliminate the data dependency between tasks, thus minimize synchronization overhead. Communication problem is solved using dataflow optimization. Using these techniques, a multimedia video decoder processing 1080p frames is then optimized on a famous multi-core architecture – PlayStation3. The experimental results show that the optimization techniques make the decoder run 20-time faster, which finally exceed real-time constraint to decode 1080p frame at 60 Hz.

誌 謝

研究生涯轉眼即逝，兩年來受到許多人幫助及鼓勵，才能如此順利完成碩士學業，在此致上最深的感激。

感謝劉志尉老師。老師的豐富學養及學者風範，使我在專業知識及研究態度上，更臻成熟，感謝老師兩年來的指導。同時，感謝口試委員：張錫嘉教授及闕河鳴教授。謝謝你們在百忙之中，撥冗參與論文口試，並對我的研究給予寶貴的意見，讓此篇論文更加完備充實。

感謝林泰吉學長。泰吉學長不厭其煩地對我的研究工作步步導引，有他的指導協助，才能有今日的成果。學長亦師亦友，研究所其間與學長相處的點點滴滴，都是求學階段最美好的回憶。

感謝實驗室學長學弟們。感謝陳信凱學長、歐士豪學長、林禮圳學長及郭羽庭學長，感謝學長們在研究生生涯中的各項協助及鼓勵。感謝洪正堉、顏于凱、呂進德、林彥呈、張巍瀚及李岳泰，謝謝學弟們在研究工作上的一切幫忙。

感謝一起打拼的好伙伴：王炳勛、卓毅、鄧翔升、劉士賢及卓志宏。這兩年，我們一同經歷了挑燈夜戰的努力，也共同分享研究成果的喜悅。

最後，感謝我最親愛的家人。爸、媽、哥，感謝你們一路上的支持及鼓勵，沒有你們就沒有今日的我，我愛你們。

謹將此篇論文獻給所有曾支持我、協助我的人，衷心的感謝並祝福你們。

慶至
謹誌於 新竹
2007 秋

CONTENTS

List of Tables

List of Figures

1 Introduction

The software-only solutions for media-rich consumer-electronics devices get more and more popular. This kind of solution decrease the development and manufacturing cost as long as the performance meets the real-time requirements of multimedia processing. The problem is how to reach the required level of performance on an advanced processor-based platform provided by the processor designers.

1.1 Multimedia Processing

The data rate and compression ratio of multimedia processing are improved as the complexity of algorithm grows. In multimedia decoding applications, the high-definition (HD) resolution is a basic requirement in many markets, such as DTV, multimedia games, and multimedia playing on monitors. The even higher performance pursued by consumers make engineers design more powerful devices while keeping the price low.

There are 3 major parallelisms in multimedia applications: instruction-level parallelism (ILP), task-level parallelism (TLP), and data-level parallelism (DLP). These parallelisms could be exploited by 3 techniques respectively: the multi-issue, the single instruction multiple data (SIMD), and the multi-thread or multi-core architecture.

The high-end consumer electronics need to run versatile multimedia applications. For examples, audio standards are AAC, MP3, Dolby Digital (AC3), etc. And multimedia standards are M-JPEG, MPEG-1, 2, and 4, H.263, H.264, etc. Thus the implementation of multimedia coding by software is a cost-effective solution. Processor-based architectures can use software patches to keep up with new multimedia applications. However, conventional single-core processor architectures are unable to provide sufficient computing power for advanced real-time multimedia processing. Thus the parallelisms in multimedia applications should be exploited by processor-based system with high performance to meet the real-time specifications.

1.2 Multi-core Processors

In the recent years, the processor industry has reached a new market of consumer electronics and personal computers. In order to further improve the already high performance of processor, the concept of multi-core on a chip comes out. With the improvement of semiconductor processes, it's possible to put many processing cores onto a single processor chip [1]. This kind of processor is called as multi-core processor or chip multiprocessor. It may be simply named as multiprocessor.

There are some reasons for this trend. First, the processor needs more effective performance per Hz, i.e., the power would become the bottleneck of processor. The utilization of more processors on a system was a common solution in the past. The multi-chip module (MCM) belongs to this category. But with the help of semiconductor technology, the integration of many circuits into a single chip is feasible. Figure 1-1 depicts this trend. The processor designers could now improve the processor performance just by making a single processor chip more compact, i.e., put more processing cores together on a chip. Furthermore, many standards for audio and multimedia have the same functionality but are almost not used simultaneously. These are the reasons why the general-purpose processors but not the application-specific integrated circuits (ASIC) are the solutions of multimedia processing for personal devices. Because the generic processors are broadly applicable, the cost of development and production could be decreased. This is the motivation to design the multi-core processor for performance in order to meet the real-time requirements of multimedia processing.

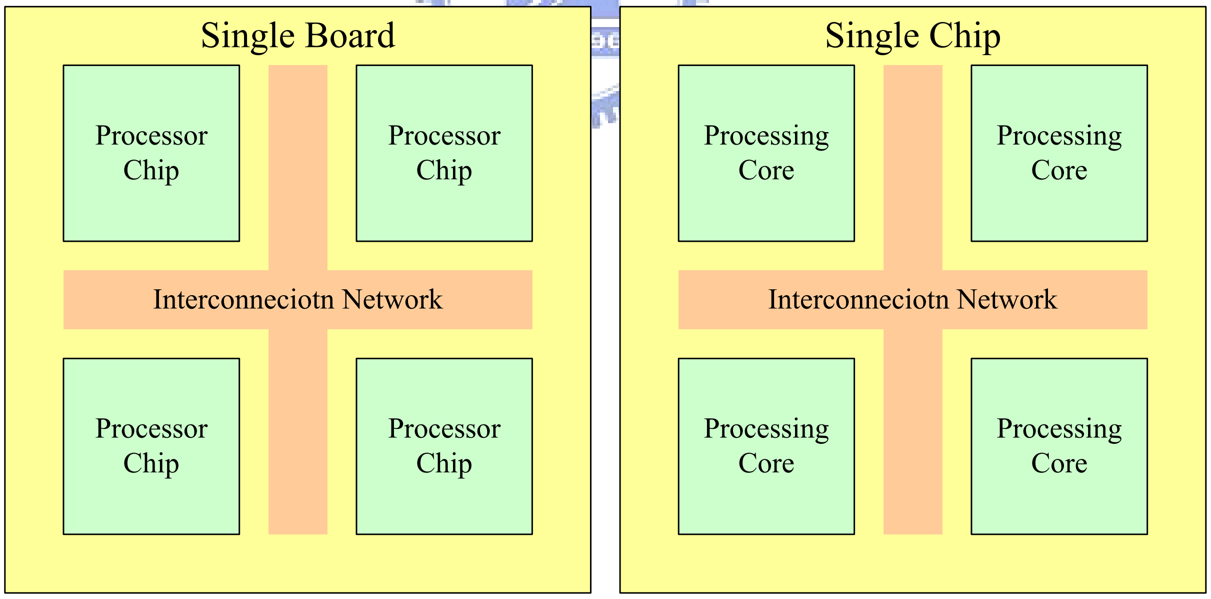


Figure 1-1 Multi-chip module vs. multi-core processor

Cell, also known as the Cell Broadband Engine Architecture (CBEA), is a famous multi-core processor created by Sony, Toshiba, and IBM (called STI). It’s a design project to provide power-efficient and cost-effective high-performance processing for a wide range of applications. It has been used in servers known as Cell blade, game consoles known as PlayStation 3. Cell is the multi-core system where this thesis holds experiment.

With the advent of multi-core processors, the programmers and consumers may simply think that the performance would increase linearly with the number of cores on a single chip. However, it’s usually not the case as we expected. The potential problems are the level of parallelism and the communication between each core. It’s a hard job to find a balanced workload for each core. The communication in a multi-core system may become the bottleneck when the communication time is too much or frequency is too high. Thus the key point to improve the performance of a multi-core system to an acceptable and reasonable level is task partitioning and communication between each core.

1.3 Thesis Organization

This work proposes a data partitioning scenario for media processing on Cell. The goal is to improve the performance with the number of cores utilized on Cell as linearly as possible. Namely, the number of frames per second of the decoding process is the performance index while the number of cores utilized is increased. The rest of this thesis is organized as follows.

Chapter 2 reviews the experimental platform: Cell Broadband Engine (CBE). A brief description of the architecture of Cell processor is the beginning. Two processing units called as the Power Processor Element (PPE) and the Synergistic Processor Element (SPE), the direct memory access (DMA), and the element interconnect bus (EIB) would be in the description. Then the communication mechanisms and associate application programming interface (API) are presented. A flow of porting the multimedia processing onto CBE is shown after the introduction is made.

Chapter 3 proposes a frame-based data partitioning technique for multimedia processing. A simple discussion of functional partitioning and data partitioning is presented. This chapter provides the data management and dataflow planning for multimedia stream decoding on CBE. The proposed technique tries to reduce the number of communication between each core, i.e., it tries to avoid the data dependencies between each partition of data. The optimization of data transfer via DMA is a key point to mitigate the communication burden on EIB of CBE. This is considered in Chapter 3 and discussed in Chapter 4.

Chapter 4 experiments the size of DMA and the allocation of buffer in local store. After the data management and dataflow planning implement as in Chapter 3, the details of data transfer between the local store and shared memory need to be considered. The suitable transfer size per DMA command and the I/O buffer allocation are found here. By the co-working of dataflow planning and the optimization of DMA, the performance boots to an expected level.

Chapter 5 summarizes this thesis and provides the future work.

2 Background

This chapter provides background information on topics related to this thesis. Chapter 2.1 gives an overview of the hardware platform, Cell Broadband Engine (CBE) [7][13][15]. Chapter 2.2 gives a flow of porting the multimedia decoding application onto CBE.

2.1 Cell Broadband Engine

The Cell Broadband Engine is the first incarnation of a new family of microprocessors conforming to the Cell Broadband Engine Architecture (CBEA, or, informally, "Cell"). The CBEA is a new architecture that extends the 64-bit PowerPC Architecture. The CBEA and the Cell Broadband Engine are the result of collaboration between Sony, Toshiba, and IBM, known as STI, formally started in early 2001 [2][3].

Figure 2-1 shows the block diagram of CBE processor hardware. The CBE processor is a multi-core processor with 9 processor elements and a shared coherent memory on-a-chip. The functionality of 9 processors can be specialized into 2 categories: the Power Processor Element (PPE) and the Synergistic Processor Element (SPE). There are 1 PPE and 8 SPEs in the CBE processor. In order to improve the productivity of PlayStation 3, only 6 SPEs are available for the programmers.

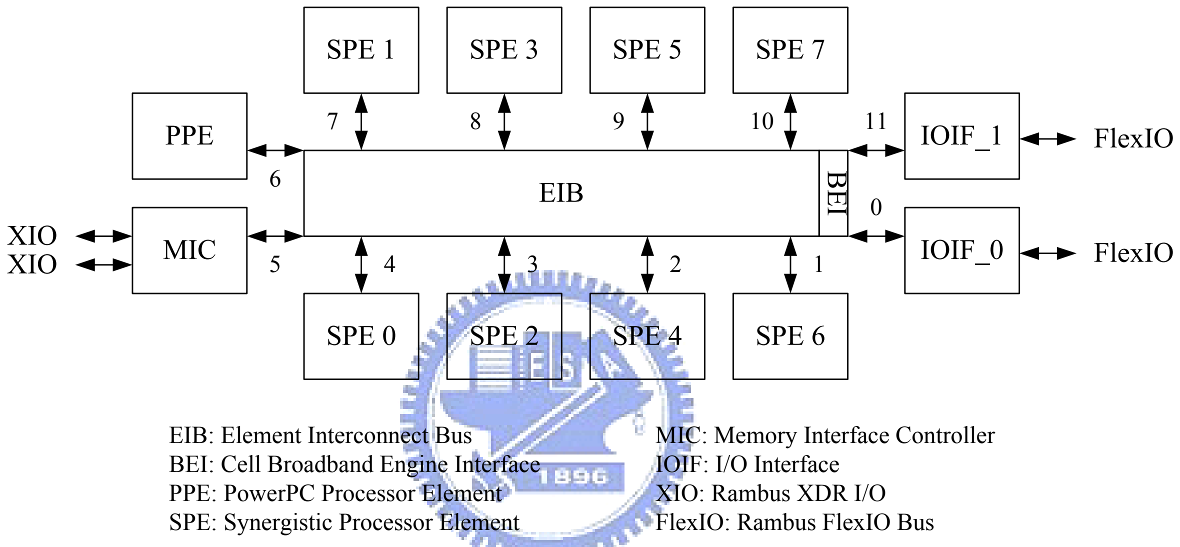

SPE 1
SPE 3
SPE 5
SPE 7
PPE
7
8
9
10
11
IOIF_1
FlexIO
6
EIB
BEI
0
XIO
XIO
MIC
5
4
3
2
1
IOIF_0
FlexIO
SPE 0
SPE 2
SPE 4
SPE 6
EIB: Element Interconnect Bus
BEI: Cell Broadband Engine Interface
PPE: PowerPC Processor Element
SPE: Synergistic Processor Element
MIC: Memory Interface Controller
IOIF: I/O Interface
XIO: Rambus XDR I/O
FlexIO: Rambus FlexIO Bus

Figure 2-1 The block diagram of CBE processor

The PPE complies with the 64-bit PowerPC Architecture. PPE could run 32-bit and 64-bit operating systems and applications. On the other hand, the SPE is optimized for running compute-intensive applications. The PPE and the SPEs could work in a collaborative scenario. The PPE runs the operating system and the top-level thread control for applications. The SPEs provide the computing power to boot the performance of applications. Brief block diagrams of PPE and SPE are shown in Figure 2-2 [4].

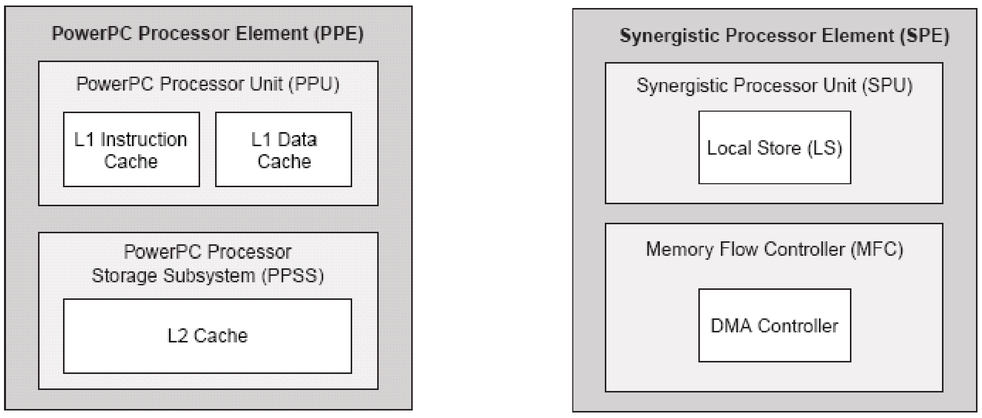

PowerPC Processor Element (PPE)
PowerPC Processor Unit (PPU)
L1 Instruction Cache
L1 Data Cache
PowerPC Processor Storage Subsystem (PPSS)
L2 Cache
Synergistic Processor Element (SPE)
Synergistic Processor Unit (SPU)
Local Store (LS)
Memory Flow Controller (MFC)
DMA Controller

Figure 2-2 Brief block diagrams of PPE and SPE

The Cell processor could be viewed as a 9-way multiprocessor for the application programmer. The PPE is suitable for control-intensive tasks and task switching. The SPEs are suitable for compute-intensive tasks but not task switching. The more significant difference between the SPE and PPE lies in how they access memory. The PPE can access main storage at all 264 memory addresses, also called effective addresses (EA), with the help of caches. The SPEs, in contrast, access main storage with the help of direct memory access (DMA) commands directed explicitly by programmers. Each SPE has its own local store (LS) which contains 256 KB. The LS is a scratchpad memory of each SPE and could be access by PPE or other SPE via DMA. Table 2-1 summarizes some differences between PPE and SPE.

Table 2-1 Differences between PPE and SPE

Feature	**PPE**	**SPE**
Addressability	2^{64} bytes	256-KB LS
Load Latency	variable (cache)	6 cycles [20]
128-bit SIMD Registers	32	128
Doubleword SIMD	no	yes
Usage	control	computation

◆ PowerPC Processing Elements

The PowerPC Processor Element (PPE) is a general-purpose, dual-threaded, 64-bit RISC processor with the vector/SIMD multimedia extensions. The PPE is responsible for overall control of a CBE system and the operating systems. The PPE consists of two main units as shown in Figure 2-3 [4]. The PowerPC processor unit (PPU) is the computation unit, and the PowerPC processor storage subsystem (PPSS) is for the purpose of storage.

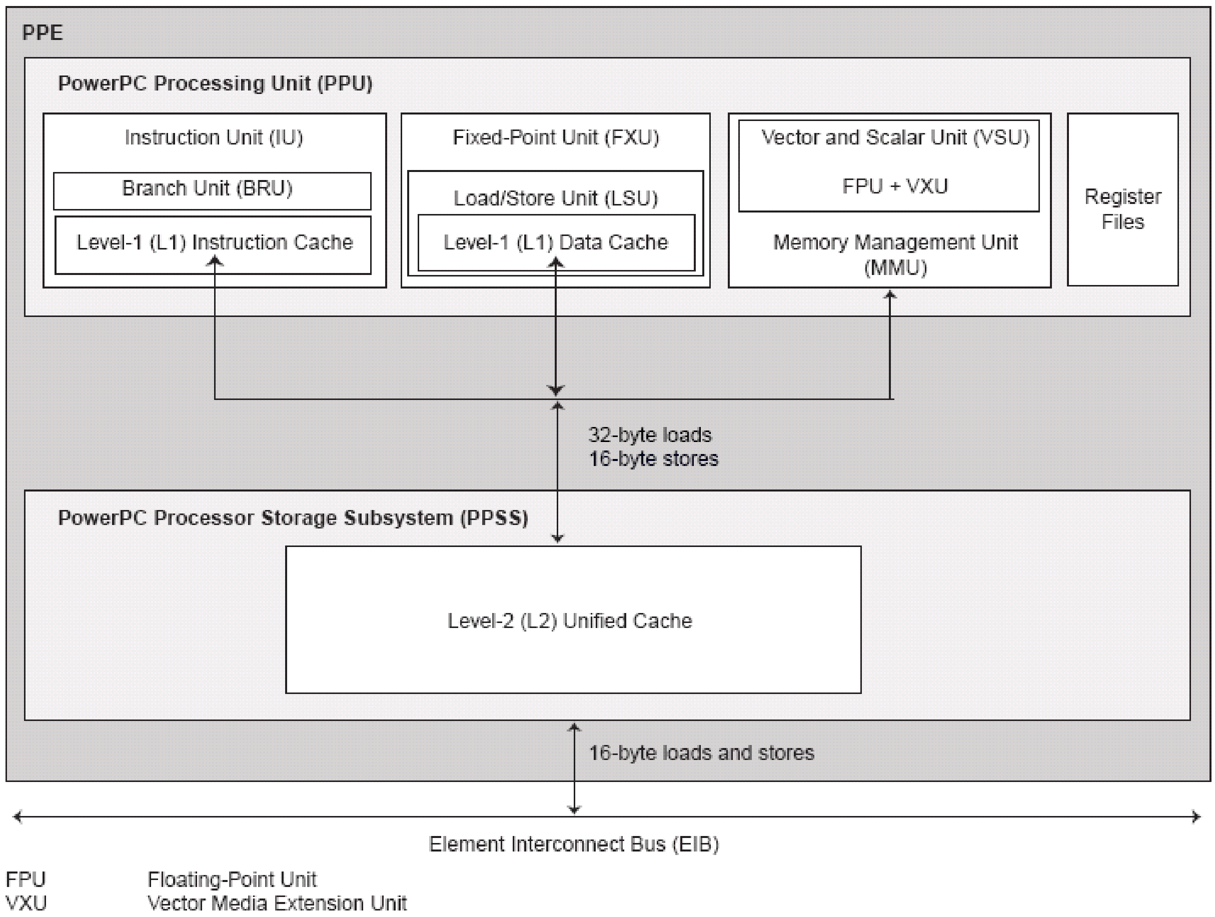

PPE
PowerPC Processing Unit (PPU)
Instruction Unit (IU)
Branch Unit (BRU)
Level-1 (L1) Instruction Cache
Fixed-Point Unit (FXU)
Load/Store Unit (LSU)
Level-1 (L1) Data Cache
Vector and Scalar Unit (VSU)
FPU + VXU
Memory Management Unit (MMU)
Register Files
32-byte loads
16-byte stores
PowerPC Processor Storage Subsystem (PPSS)
Level-2 (L2) Unified Cache
16-byte loads and stores
Element Interconnect Bus (EIB)
FPU Floating-Point Unit
VXU Vector Media Extension Unit

Figure 2-3 The PowerPC Processor Element (PPE).

The PPU could further divided into the following units.

• Instruction Unit (IU)

The IU contains a 2-way set-associative and reload-on-error L1 instruction cache with 32

KB. The cache-line size is 128 bytes. The IU performs the instruction-fetch, decode, dispatch, issue, and completion portions of execution.

• Branch Unit (BRU)

The BRU performs the branch functionality.

• Fixed-Point Unit (FXU)

The FXU performs fixed-point operations, including add, multiply, divide, compare, shift, rotate, and logical instructions.

• Load and Store Unit (LSU)

The LSU contains a 4-way set-associative and write-through L1 data cache with 32 KB. The cache-line size is 128 bytes. The LSU performs all data accesses, including load and store instructions.

• Vector/Scalar Unit (VSU)

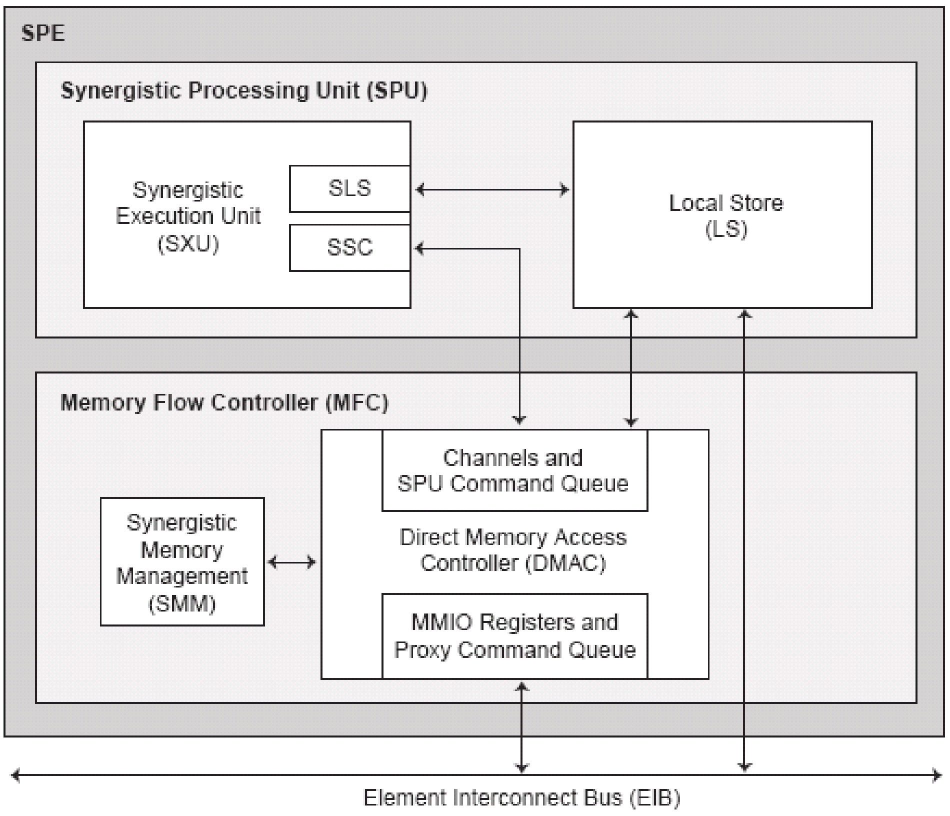
The VSU contains a floating-point unit (FPU) and a 128-bit vector/SIMD multimedia extension unit (VXU), which together execute floating-point and vector/SIMD multimedia extension instructions.

• Memory Management Unit (MMU)

The MMU contains a 64-entry segment look-aside buffer (SLB) and 1024-entry, unified, parity protected translation look-aside buffer (TLB). The MMU manages address translation for all memory accesses.

The PPSS contains a unified, 512-KB, 8-way set-associative, write-back L2 cache with error-correction code (ECC). The cache-line size for the L2 is 128 bytes as the same as L1 cache-line size. The PPSS handles all memory accesses by the PPU and memory-coherence (snooping) operations from the element interconnect bus (EIB). The PPSS performs data-prefetch for the PPU and bus arbitration and pacing onto the EIB. There are MMU, L1 instruction cache, and L1 data cache of PPU getting data from PPSS by a shared 32-byte load port. There are MMU and L1 data cache of PPU putting data to PPSS by a shared 16-byte store port. The interface between the PPSS and EIB supports 16-byte load and 16-byte store buses.

◆ Synergistic Processor Elements


SPE
Synergistic Processing Unit (SPU)
Synergistic Execution Unit (SXU)
SLS
SSC
Local Store (LS)
Memory Flow Controller (MFC)
Channels and SPU Command Queue
Direct Memory Access Controller (DMAC)
MMIO Registers and Proxy Command Queue
Synergistic Memory Management (SMM)
Element Interconnect Bus (EIB)

MMIO	Memory-Mapped I/O Registers
SLS	SPU Load and Store Unit
SSC	SPU Channel and DMA Unit

Figure 2-4 The synergistic processor elements (SPE)

Each SPE is a 128-bit RISC processor for data-rich, compute-intensive applications. It consists of two main units, the synergistic processor unit (SPU) and the memory flow controller (MFC), as shown in Figure 2-4 [4]. The data interface consists of a 128-bit read bus and a 128-bit write bus. The MFC can send up to 16 outstanding MFC commands. It supports atomic requests and snoop requests (read and write) of the SPU's LS memory and the MFC's

MMIO registers.

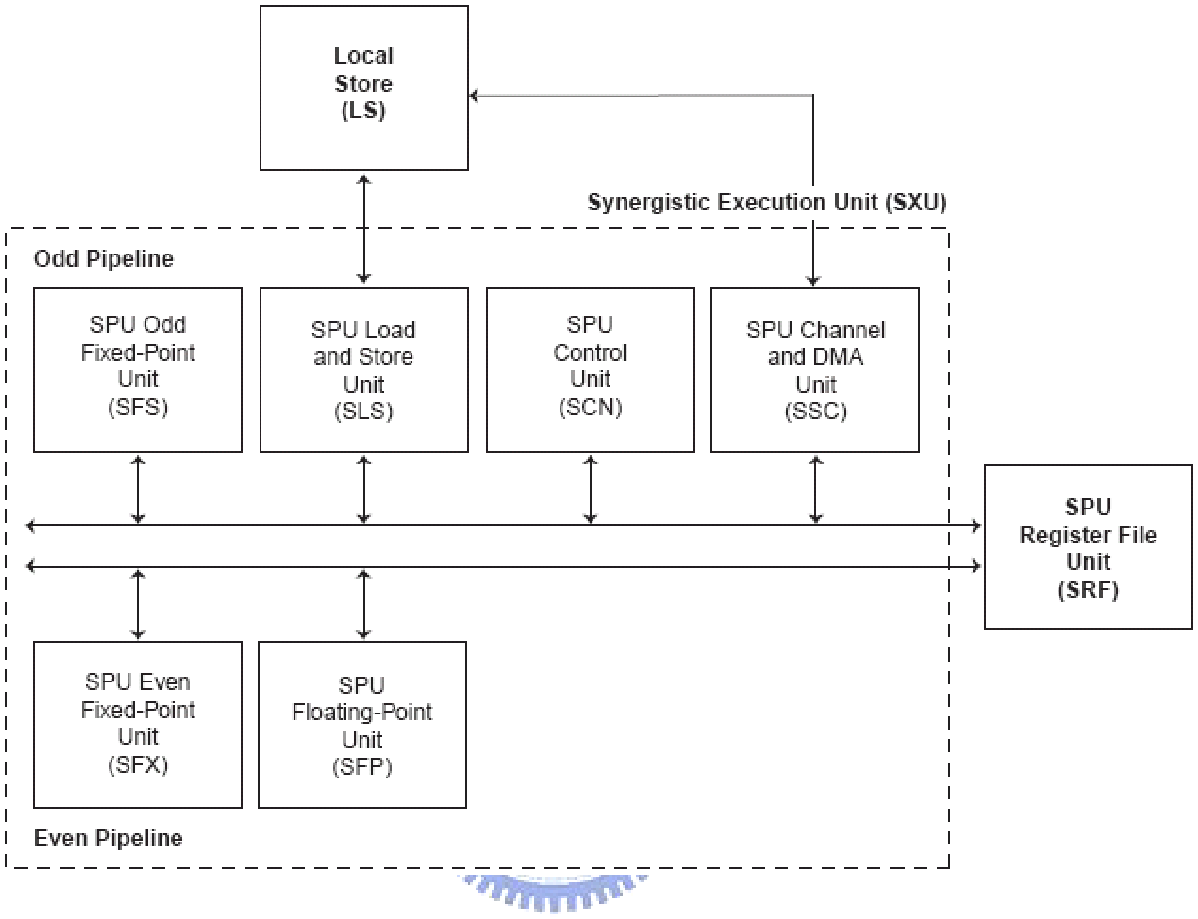

Local Store (LS)
Synergistic Execution Unit (SXU)
Odd Pipeline
SPU Odd Fixed-Point Unit (SFS)
SPU Load and Store Unit (SLS)
SPU Control Unit (SCN)
SPU Channel and DMA Unit (SSC)
SPU Register File Unit (SRF)
SPU Even Fixed-Point Unit (SFX)
SPU Floating-Point Unit (SFP)
Even Pipeline

Figure 2-5 The functional units in SPU

Figure 2-5 shows the functional units in SPU. The SPU issues two instructions to its two execution pipelines respectively. The pipelines are referred to as even (pipeline 0) and odd (pipeline 1). The units in SPU could be pointed out as follows.

• SPU Odd Fixed-Point Unit (SFS)

The SFS executes byte shift, rotate mask, and shuffle operations on quadwords.

• SPU Load and Store Unit (SLS)

The SLS executes load and store instructions and hint for branch instructions. It also handles DMA requests to the LS.

• SPU Control Unit (SCN)

The SCN fetches and issues instructions to the two pipelines. It performs control functions such as branch instructions, arbitration of access to the LS and register file, etc.

• SPU Channel and DMA Unit (SSC)

The SSC manages communication, data transfer, and control into and out of the SPU.

• SPU Even Fixed-Point Unit (SFX)

The SFX executes arithmetic instructions, logical instructions, word SIMD shifts and rotations, floating-point comparisons, and floating-point reciprocal and reciprocal square-root estimations.

•SPU Floating-Point Unit (SFP)

The SFP executes single-precision and double-precision floating point instructions, 16-bit integer multiplies and conversions, and byte operations. The 32-bit multiplies are implemented in software using 16-bit multiplies.

◆ Element Interconnect Bus

Figure 2-6 [5] shows the element interconnect bus (EIB), the heart of the Cell processor's communication architecture, which enables communication among the PPE, the SPEs, main system memory, and external I/O. The EIB has separate communication paths for commands and data. The EIB data network consists of four 16-byte data rings: two running clockwise and the other two counterclockwise. Each ring allows up to three concurrent data transfers, as long as their paths don't overlap.

Bus elements request data bus to initiate a data transfer. The data bus arbiter gives the first priority to requests coming from the memory controller to minimize the stalls of reading. It treats all others equally in round-robin fashion. The arbiter receives these requests and decides which ring should handle each request. It selects one of the two rings that travel in the same direction of the shortest transfer to ensure that the data won't need to travel more than halfway around the ring to its destination. The arbiter also schedules the transfer to avoid the interferences with other in-flight transactions. The EIB operates at the speed of half the processor-clock. Each bus element could simultaneously send and receive 16 bytes of data every bus cycle.

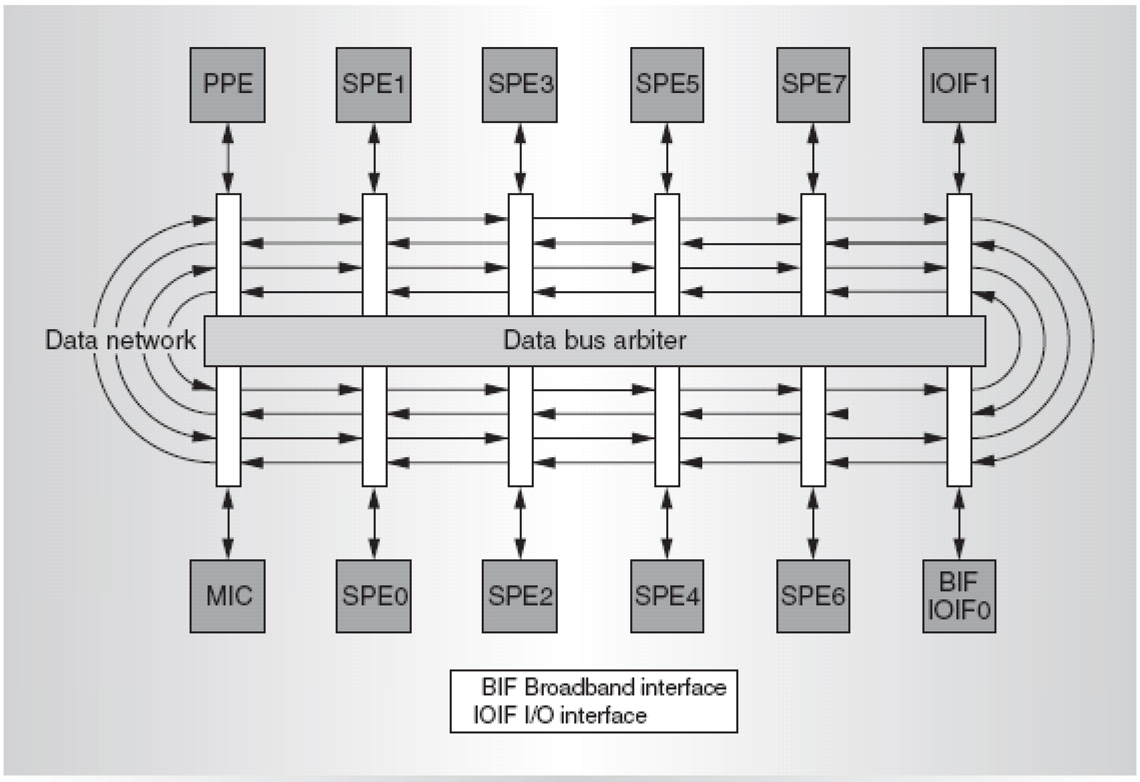


Figure 2-6 The element interconnect bus (EIB)

The EIB's maximum data bandwidth is limited by the rate at which addresses are snooped across all units in the system. The rate is one address per bus cycle. Each snooped address request can potentially transfer up to 128bytes, so in a 3.2GHz Cell processor, the theoretical peak data bandwidth on the EIB is 128 bytes * 1.6 GHz = 204.8 Gbytes/sec. The maximum bandwidth of Cell processor is summarized in Figure 2-7. However, the actual data bandwidth depends on several factors: the relative locations of destination and source, the new transfer's interferences with in-flight transfers, and the efficiency of data arbiter, etc.

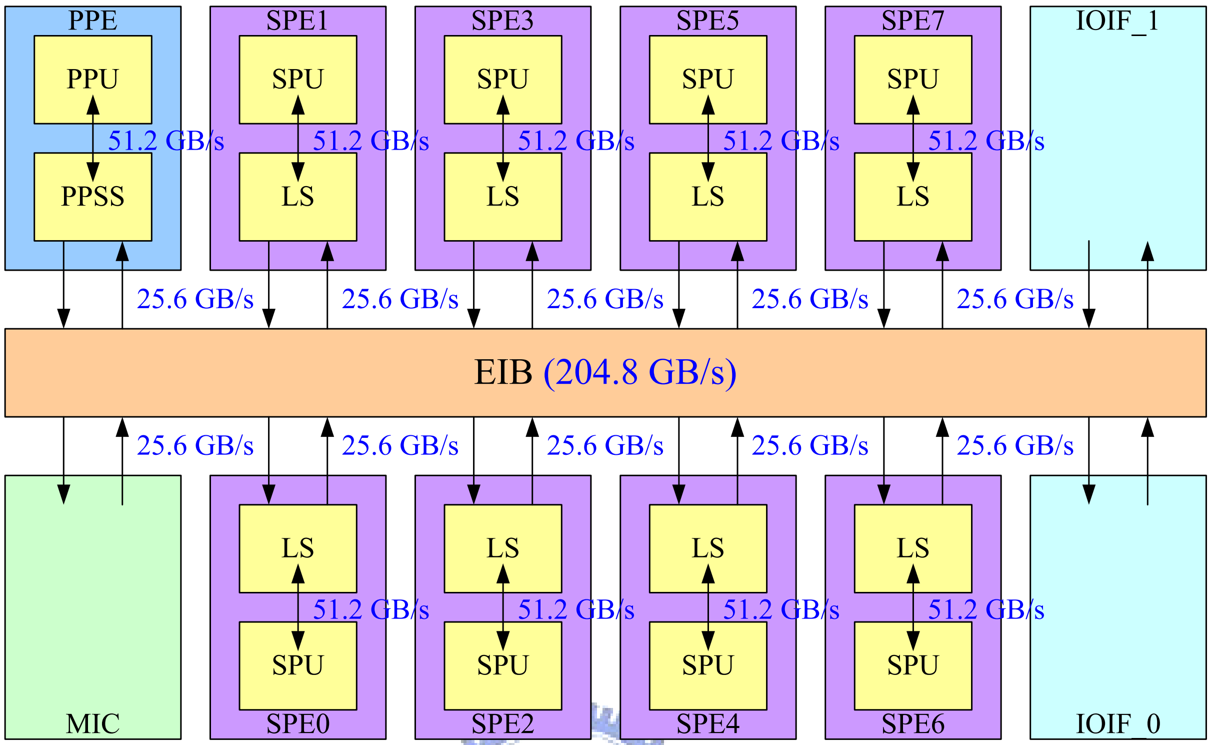


Figure 2-7 Maximum bandwidth of Cell processor

◆ Inter-processor Communication

There are many attributes of the shared-memory system. The PowerPC processor element (PPE) and all synergistic processor elements (SPEs) have coherent access to the main storage. All communication mechanisms are implemented and controlled by the SPE's memory flow controller (MFC). The SPEs must explicitly use the following three communication mechanisms: DMA transfers, mailbox messages, and signaling messages in order to communicate with other bus elements in the system. Table 2-2 summarizes the three mechanisms mentioned above.

Table 2-2 Three primary mechanisms of interprocessor communication

Mechanism	Description
Signaling	Used for control communication from the PPE or other devices. Signaling utilizes 32-bit registers for one-sender-to-one-receiver signaling or many-senders-to-one-receiver signaling.
Mailboxes	Used for control communication between an SPE and the PPE or other devices. Each SPE has two mailboxes for sending and one mailbox for receiving 32-bit messages.
DMA Transfers	Used for data communication between main storage and an LS of the SPE. The asynchronous DMA transfers of MFC hide the memory latency and transfer overhead by moving data in parallel with SPU computation.

■ DMA

An MFC supports naturally aligned DMA transfer sizes of 1, 2, 4, 8, and 16 bytes and multiples of 16 bytes. For naturally aligned 1, 2, 4, and 8-byte transfers, the source and destination addresses must have the same 4 least significant bits (LSB). A single DMA command could transfer up to 16 KB between an LS and shared memory storage.

The throughput of a DMA transfer when the source and destination addresses are 128-byte aligned is double as compared to that of a mis-aligned transfer within a cache line. It’s because that the mis-aligned transfer is a partial cache-line transfer, and actually there may be two bus requests for this transfer. Peak performance is achieved when the size of the transfer is a multiple of 128 bytes and both the effective address (EA) and the local store address (LSA) of the DMA transfer are 128-byte aligned. The following performance guidelines for DMA commands in CBE could be made.

• Minimize small transfers

• Align source and destination addresses to a 128-byte cache-line boundary.

• Minimize the use of synchronizing and data-ordering commands.

• Have SPEs (not PPE) initiate DMA transfers. The reasons state in Table 2-3.

Table 2-3 The reason why we have SPE initiate DMA commands

Feature	**SPE**	**PPE**
Processor Amount	8	1
MFC Command Queue	16	8
Synchronization	easy	hard
# of Cycles to Initiate a DMA transfer	smaller	larger

■ Mailbox

Mailboxes take charge of the 32-bit messages between an SPE and other devices. There are three mailbox channels of each SPE: two one-entry mailbox channels and one four-entry mailbox channel. The SPU Write Outbound Mailbox and the SPU Write Outbound Interrupt Mailbox which belong to one-entry mailbox channels are used for sending mails from the SPE to the PPE or other bus elements. The SPU Read Inbound Mailbox which belongs to four-entry mailbox channel is used for sending messages from the PPE or other bus elements to the SPE. Table 2-4 gives details about the mailbox channels and their associated MMIO registers.

Table 2-4 The mailbox channels and their associated MMIO registers

Feature	**SPU Write Outbound Mailbox**	**SPU Read Inbound Mailbox**	**SPU Write Outbound Interrupt Mailbox**
Channel Interface			
Mnemonic	SPU_WrOutMbox	SPU_RdInMbox	SPU_WrOutIntrMbox
# of Entries	1	4	1
R/W	W	R	W
Width (bits)	32	32	32
MMIO Register Interface			
Mnemonic	SPU_Out_Mbox	SPU_In_Mbox	SPU_Out_Intr_Mbox
# of Entries	1	4	1
R/W	R	W	R
Width (bits)	32	32	64

2.2 Programming on PPE and SPEs

Figure 2-8 shows the application programming interface (API) of DMA utilized in this thesis and the associated direction. The SPE could use the API “mfc_write_tag_mask” and “mfc_read_tag_status_all” to wait for the completion of DMA commands. It’s noted that the waiting time of DMA commands could be reduced by programmer, because the operation of DMA and the computation of SPU are asynchronous.

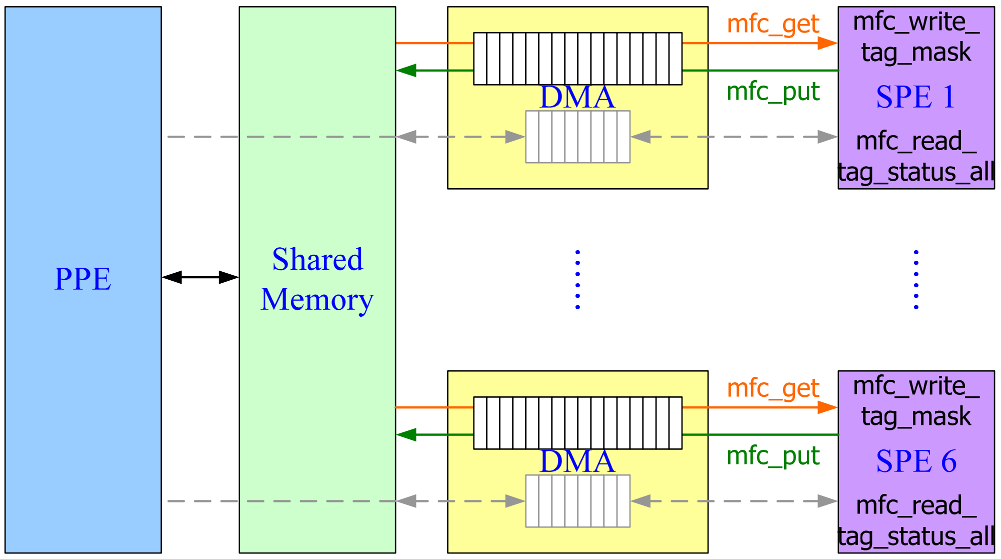


Figure 2-8 The API of DMA

Figure 2-9 shows the API of mailbox utilized in this thesis and the associated direction. The PPE could use the API "spe_out_mbox_status" to gather the information of SPU Write Outbound Mailbox for each SPE. Usually this API is bundled with "spe_out_mbox_read". PPE use "spe_out_mbox_status" to wait the update of SPU Write Outbound Mailbox and receive the message via "spe_out_mbox_read".

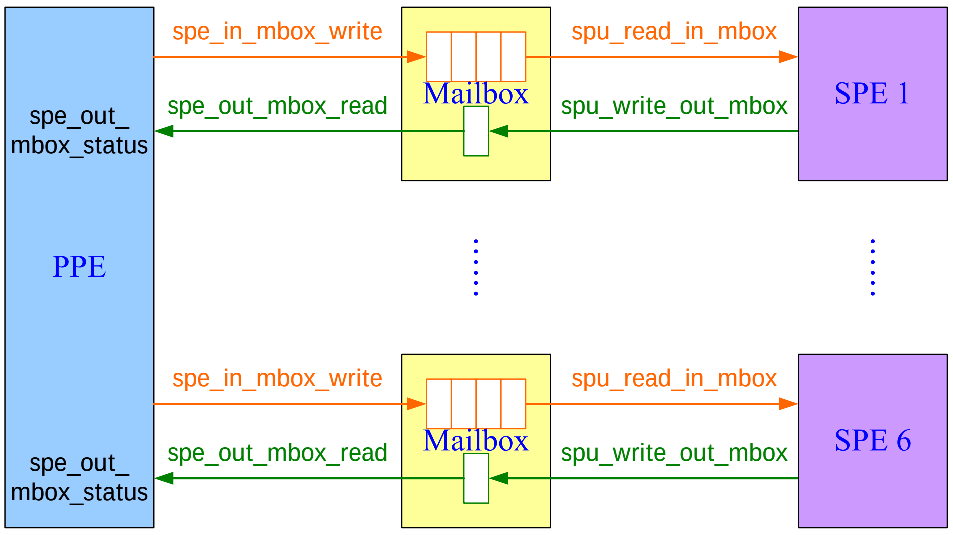
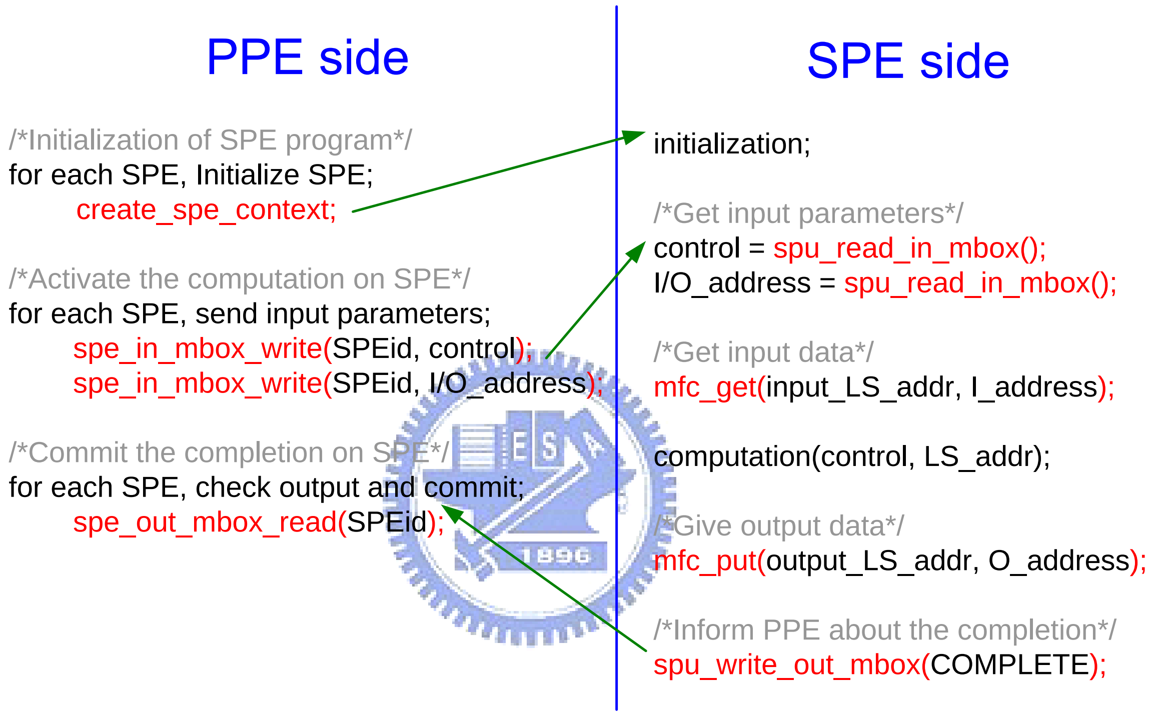


Figure 2-9 The API of mailbox

Figure 2-10 shows a common form of program which utilizes the PPE and SPEs on CBE. The communication between PPE and SPEs is a significant factor of the system performance, namely, the mailbox and DMA commands. It's noted that the algorithm of application is almost unchanged. The major works are the function offload of SPE's code and the building of communication scenario.



```
/*Initialization of SPE program*/
for each SPE, Initialize SPE;
        create_spe_context;

/*Activate the computation on SPE*/
for each SPE, send input parameters;
        spe_in_mbox_write(SPEid, control);
        spe_in_mbox_write(SPEid, I/O_address);

/*Commit the completion on SPE*/
for each SPE, check output and commit;
        spe_out_mbox_read(SPEid);
```

```
initialization;

/*Get input parameters*/
control = spu_read_in_mbox();
I/O_address = spu_read_in_mbox();

/*Get input data*/
mfc_get(input_LS_addr, I_address);

computation(control, LS_addr);

/*Give output data*/
mfc_put(output_LS_addr, O_address);

/*Inform PPE about the completion*/
spu_write_out_mbox(COMPLETE);
```

Figure 2-10 The cooperation of PPE and SPEs

It's noted that the APIs used here are mentioned above. As shown in the pseudo code the PPE maintains the thread-control and I/O behavior, and SPEs take charge of the computing tasks. The starting and ending of computation on SPEs are activated by the mails between the PPE and the SPEs.

The PPE is the manager of threads, and it creates the context for SPE. The program for SPEs would be loaded and executed in this simplified step. Then the SPEs run to the waiting for mails from the PPE. SPEs get what they want via mailboxes or DMA transfers from PPE.

The completing message would be send to PPE when the computing on SPE is done. The PPE receives this completing message and knows that the SPE with SPEid is idle and ready for next task.

3 Frame-based Data Partitioning on CBE

This chapter focuses on the data management of multimedia stream on multi-core system. The platform is CBE, and the demonstration software is JPEG decoder. For a multi-core processor, the data communication efficiency is a key factor of the overall performance. Although the algorithm optimization and the special instruction of processor could improve the system computing power, they are too specific and irrelevant to the architecture and design concepts of multi-core. High performance can’t be achieved without the optimization of data partitioning and dataflow on a multi-core processor.

3.1 Port the Multimedia Applications to CBE

For Cell, such a multi-core processor, a common flow of porting multimedia application could be summarized as in Figure 3-1. At the beginning, a single-thread program is the most original program in the first implementation. In this original program on the PPE, the programmers could access all 2^{64} memory address and the program runs sequentially. The main purpose of the original program is to verify the functionality. It has a little chance to meet the performance requirements of multimedia applications nowadays without optimization.

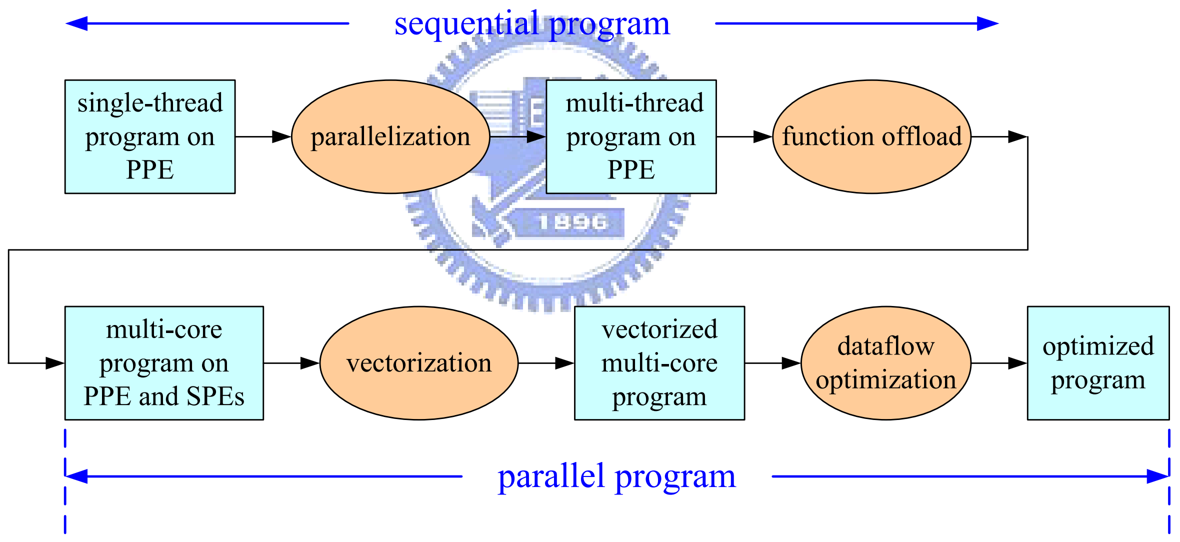

sequential program
single-thread program on PPE
parallelization
multi-thread program on PPE
function offload
multi-core program on PPE and SPEs
vectorization
vectorized multi-core program
dataflow optimization
optimized program
parallel program

Figure 3-1 Port the Multimedia Applications to CBE

The program could be divided into threads and then becomes a multi-thread program. This parallelization step should be done in a very careful way. In order not to reduce performance, the threads should be load-balanced. In other words, the computing time of each thread should be kept as close as possible. The processor core could then manipulate the threads equally and easily. For advanced single-core processors, the multiple functional units

(FUs) in the processors could take charge of different threads simultaneously which is called simultaneous multi-thread (SMT) processors. This approach exploits the thread-level parallelism (TLP) to boots the performance by parallel processing among FUs.

For CBE, the threads could be allocated to the SPEs [6][21]. It's noted again that the SPEs are the computing engines in CBE. In this stage, the load-balanced threads which found previously get significance. The SPEs are all homogeneous with same computing power. Thus the threads should have equal computing time to achieve the balanced loading. There are 6 available SPEs in PlayStation 3 which in turn means that the number of load-balances threads could be 6. This technique is called "function offload" and the key point is load balancing. After function offloading, the multi-thread program becomes the multi-core program and each thread on different processors runs in parallel [22][23].

There are commonly used optimizations, namely, single instruction multiple data (SIMD) [17][18][19] and loop unrolling. This process vectorizes the code to exploit the data-level parallelism (DLP) and instruction-level parallelism (ILP). The vectorized program executes on multiple data in a single instruction (SIMD) or consumes multiple instructions with multiple functional units (FUs). The vectorization is a technique to reduce the computing time of a single processor. In a load-balanced multi-core environment, the amount of performance improvement in single processor by vectorization equals that in entire multi-core system. For example, the 200% improvement gained from SIMD in single processor core within a multi-core system, the overall maximum improvement of the entire system would never exceed 200%. Even worse, the improvement of computation in a multi-core system would be bounded by the communication between each processor core. The communication scenario is a very important performance factor in multi-core system with limited communication resource such as CBE.

The last step is to make a good dataflow plan to reduce the communication overhead within the CBE. This stage could boot the performance a lot while the bandwidth of communication resource is low. For example, if all components connect to a low bandwidth bus, all traffic would appear in this single bus. The components wait for the data transfers to continue their own computing tasks. In the worst case, component A could wait for all other transfers until the last for itself. Then component B needs the computation result and would wait even longer. The communication overhead would accumulate if there is a traffic jam in the system. This thesis would utilize the frame-based data partition method and optimize the dataflow to reduce such burden. After the effort from programmers, the resulting optimized program is a vectorized program utilizing multi-core processor and running in parallel.

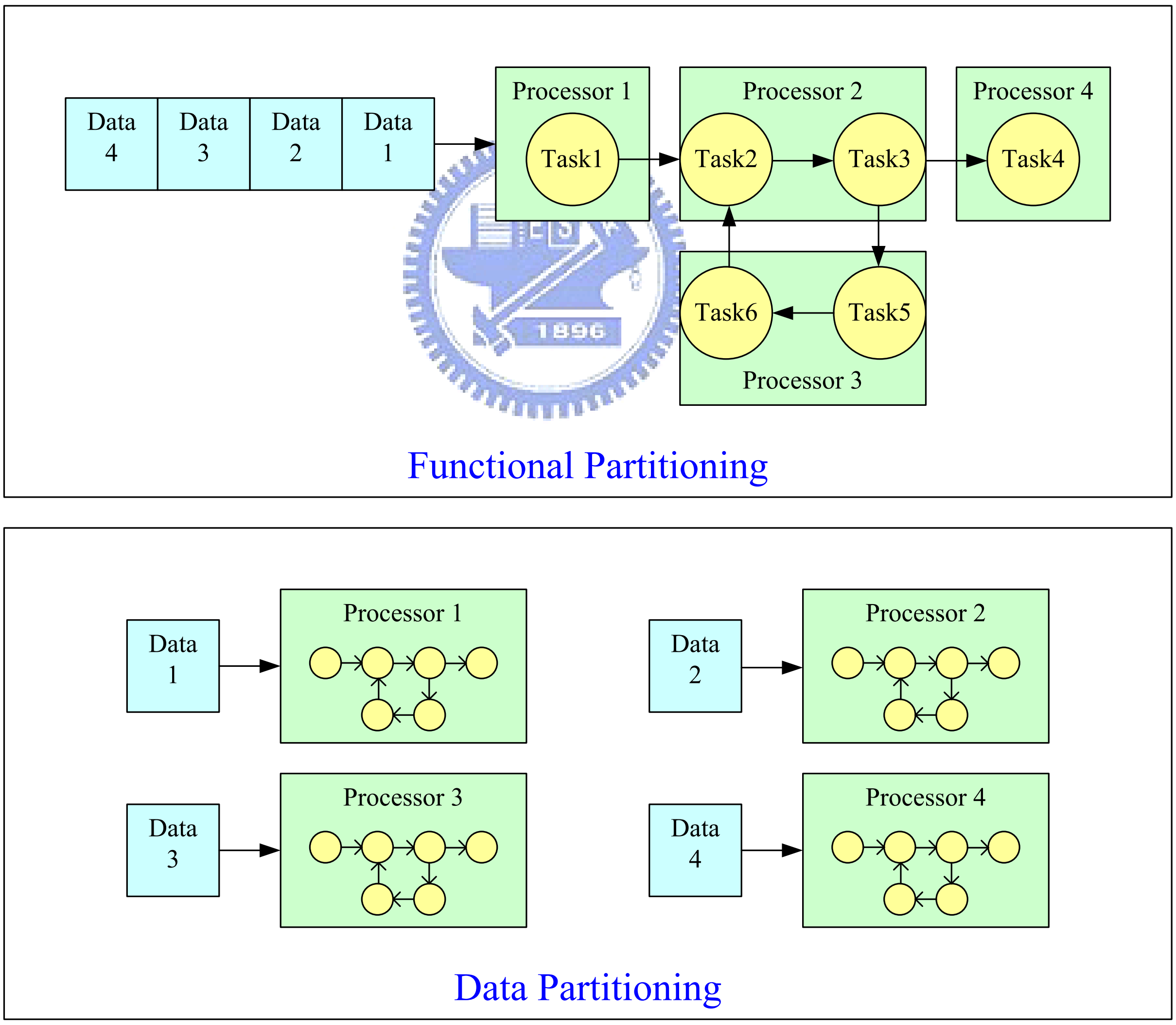


Figure 3-2 Functional partitioning and data partitioning

3.2 Functional Partitioning and Data Partitioning for Multi-Core Processor

Intuitively, there are two ways to partition applications over a multi-core environment, i.e., data partitioning [8][9] and functional partitioning [10][11][12] as illustrated in Figure 3-2. The left hand side of Figure 3-2 is function partitioning. A function is decomposed into tasks, and tasks are grouped and allocated to single core of the processor. The right hand side of Figure 3-2 is data partitioning. For example, we can divide a picture into partitions and process each partition on a single processor. For a streaming application running on multi-core processor, the bottleneck of system performance is generally determined by the communication between each core.

The locality of data plays an important role in such an environment. At first, data should be loaded to a local memory. Then all operations are performed at local processor. Finally, the result is transferred back to the shared memory at higher level. The input and output data packets flowing into the local processor and the intermediate results should be no larger than the comparatively small size of local memory. When the small local memory is occupied by data and instruction together, and the application deals with large data sets and performs complex algorithm, the data partitioning and management becomes a big challenge.

3.3 Comparison between Data and Functional Partitioning

There are many points of view about the pros and cons of the twos types of application partitioning. Some comparisons could be made between data and functional partitioning [8].

•Depending on the application, different approaches result in different communication behavior. For data partitioning, communication overhead would occur because of the data dependencies between the partitions; for functional partitioning, communication overhead would occur between individual tasks on different processor cores.

• In the case of data partitioning, task-to-task communication remains locally on the core if sufficient local memory size is available. Thus, data partitioning inherently results in locality of data.

•It's very common that the number of data partitions is larger than the number of processor cores. Given sufficient data partitions, load balancing between processor cores comes in a nature way; for functional partitioning, it's not uncommon that a certain task becomes a system bottleneck due to imbalanced loads of the processor. It strongly depends on the granularity of the functional decomposition into tasks and takes a lot of work to find a clear balanced partitioning point.

• Data partitioning provides scalability of software. For instance, a Standard-
Definition (SD) multimedia sequence can be decoded using 2 CPUs, whereas for HD resolution 8 CPUs are needed. For functional partitioning, different throughput requirements would affect the overall partitioning of the application, which results in laborious rewriting of the software.

• In order to fully exploit the computational power of the specific processors chosen by programmer, the application software needs to be optimized for instruction-level parallelism. For functional partitioning, this purpose would take a great deal of effort in the way of partitioning and the restructuring the software; for data partitioning, the partitioning remains unchanged, since each processor cores executes the complete function. The comparison is summarized in Table 3-1.

Table 3-1 Data partitioning vs. functional partitioning

Feature	Data Partitioning	Functional Partitioning
Communication between Partitions	data dependency	task dependency
Load Balancing	nature balance	need extra effort
Scalability	good	bad
Optimization	partitions remain unchanged	re-partitioning

The main issue that needs to be resolved for data partitioning, is the minimization of the communication overhead for data dependencies between partitions. Furthermore, the scheduling of the data partitions has to be considered, since inter-dependencies impose restrictions on the order in which the partitions can be processed. The best way is making the data partitions fully independent, although there should be a cost of occupying a room on shared memory. This is the concept we proposed on a desk-top level system like PlayStation3. It is obviously that the advantages of data partitioning would become larger when the application is very complex and the amount of processing data is huge. It's just the case while the multimedia coding standard becomes more aggressive with time goes by.

3.4 The Partitioning and Management of Multimedia Decoding on CBE

First of all, the decision about what should be left in SPE and what should be left in PPE must be made. The usual and recommended way is to move all computing part of an application to SPE and leave the memory management part and system control task in PPE. The most computing-intensive part is the decoding algorithm, so it is reasonable to put all decoding function into SPE. For the input process, such as file reading, data packet management and the formation of one frame should be put into PPE. The display process, usually the last step in real time multimedia decoding, utilizes the frame buffer of shared memory in PS3. This is absolutely a PPE task which manages memory and the peripheral.

The way of data management between encoded multimedia stream and decoded multimedia frame is an important issue, especially when the local store (LS) is comparatively too small for the stream or frame. The smoothness of dataflow is a basic requirement for high performance and must be treated carefully. It’s depicted in Figure 3-3. Thus the design of dataflow and allocation of buffers become the major concern in this chapter [16].

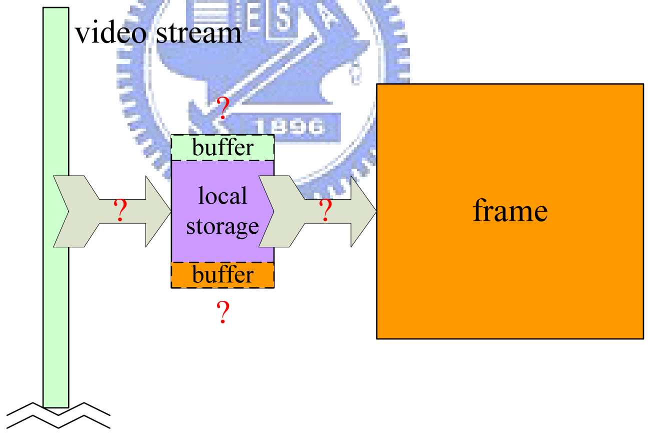


Figure 3-3 The situation of LS in the multimedia decoding

For SPE, the program and data both reside in a single 256 Kbytes LS, so the usage of program memory earns the first glance. Thus the starting work is to minimize the code size of which part would be placed into the SPE. This means that the loop unrolling and some other techniques which trade the code size for the performance should not be used before the porting on SPE.

After some surveys of open source code, we insure that the algorithm part in the program of simple multimedia decoder usually would not excess the limited size of the LS. Typically the resulting execution file with no compiler's optimization on SPE is about 130 Kbytes. It's worthy to note that the CBE SDK provides a mechanism called "SPU Overlay" to support the access of SPU overlay code sections located in the main memory. It can decompose the large code into small segments and load with programmer's will. However, this mechanism degrades the performance due to the overhead of the loading of the code sections. Next, the consideration moves from program to static data on SPE.

There is an amount of static data in the program of multimedia stream codec. Take JPEG for example, the quantization table and the Huffman tree table are necessary. For the quantization table, the luminance table and chrominance table are needed. For the Huffman tree table, the luminance DC and AC table, the chrominance DC and AC table are needed. Furthermore, the code lookup table and the code size table should be built from the above Huffman tree table and the header information. The total size of tables used in JPEG is summarized below:

Table 3-2 Total size of table used in JPEG

unit: bytes	Luminance		Chrominance		Total
Table	DC	AC	DC	AC	
Quantization	64*1		64*1		128
Huffman Tree	16+12	16+162	16+12	16+162	412
Huffman Code	512*2	512*2	512*2	512*2	4096
Huffman Size	512*1	512*1	512*1	512*1	2048
Total	792		792		6684

There are still some heap data in the program. The heap size should also be minimized by the programmer. However, heap data is highly application-dependent and not the significant part of the local store. Thus the important thing of the programming in CBE is to keep the size of program and heap small. In the roadmap of CBE, the size of LS may be enlarged with the improvement of semiconductor technology. It's good news for the methodology of data partitioning. The programmers are more comfortable for the larger LS because that the more processing data units could be pulled in and out from the shared memory each time.

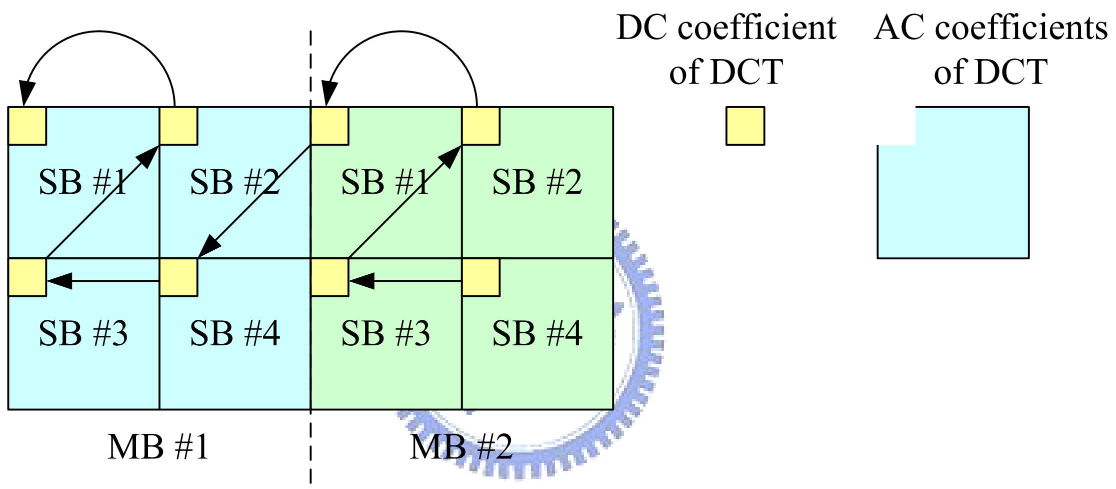


Figure 3-4(a) Data dependency in JPEG

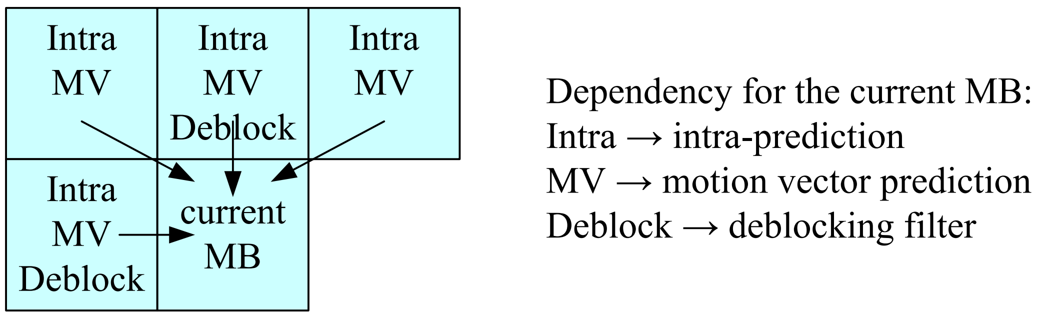

Intra MV
Intra MV Deblock
Intra MV
Intra MV Deblock
current MB
Dependency for the current MB:
Intra → intra-prediction
MV → motion vector prediction
Deblock → deblocking filter

Figure 3-4(b) Spatial data dependency in H.264

The basic processing data unit in the multimedia processing is called "macroblock" (MB) as described in section 2.2. The MB could be partitioned into sub-block. There are many dependencies between MBs and/or sub-blocks. Figure 3-4 shows two examples. In the simple case of JPEG, the DC value in DCT of this sub-block comes from the DC value of the previous sub-block and the difference value which is encoded in this sub-block. Furthermore, in H.264 the spatial data dependencies are the left, upper left, upper, and upper right MBs. A total of 4 reference MBs are involved in the decoding of this MB. This dependent data should keep in the LS until it's useless to avoid the massive data reloading from the shared memory. This is a very critical step after the minimization of program and static data. The number of times of DMA in the multi-core system should be as small as possible. After the first compilation of application, the remaining room in LS should be smartly utilized with these basic data units.

In multimedia decoding the input is in the form of stream, and the variable length encoding scheme makes the parallel processing at input very difficult. For data partitioning technique, the parallel processing is at data level. To fully utilize the parallelism that data partitioning has, the data level parallelism should be shift to a level higher than MB. The appropriate level of data parallelism is frame-level, and only in this way the variable length decoding could be put into SPE. The entire decoding functions are now in SPEs, and the input and output data management left in PPE as expected.

Move from the single PPE program to the SPE program, the MB-based multimedia decoding algorithms are unchanged. The only difference is that SPEs cannot directly access the shared memory. The shared memory of all system could be accessed by SPEs only with the help of DMA. The scenario of data communication between each LS in SPE and the shared memory is shown in Figure 3-5. The partitions at shared memory are divided into 2 categories: the multimedia stream and the frames. The partitions at LS are divided also into 2 categories: the input buffer and the output buffer. It's noted that there are still program and data storage in the LS which is not shown in the figure. The I/O buffer utilized the remaining part of LS as described above. SPE reads the multimedia stream with input buffer and write to the frame through output buffer. The reading and writing process between the LS and shared memory carries out through DMA command (dashed lines in Figure 3-5). For decoding of one frame, the I/O process at SPE runs many times because the size of multimedia stream and frame is much bigger than the I/O buffer in size. Each SPE takes charge of the decoding of one frame. The stream of one frame is represented as many chunks. Each chunk has the size which equals the size of input buffer. Put it in another way, the input stream of one frame is cut into pieces by the input buffer. Input buffer gets one piece per DMA command. The relationship between output buffer and the frame in shared memory is similar. The details of dataflow and buffer allocation are discussed in following sections.

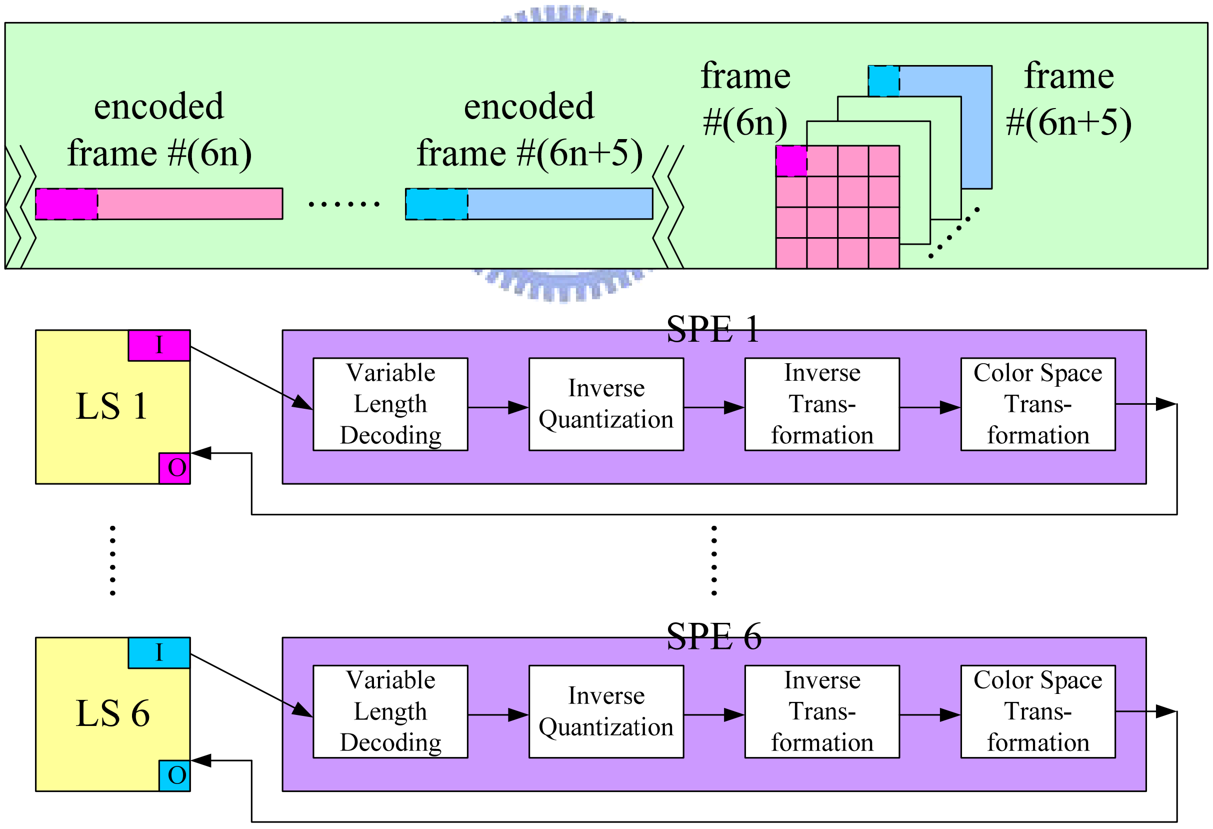


Figure 3-5 Data communication between LS and shared memory

A basic multimedia decoding flow is placed in the SPE part of the figure. Variable length decoding is the 1st function of decoder. The implementation of variable decoding often involves lookup tables which must be stored in LS. The 2nd function is inverse quantization which is very common in many multimedia applications and involves quantization tables. The 3rd function is inverse transformation which is usually a type of inverse discrete cosine transformation (IDCT). The final function is color space transformation for the purpose of display. The transformation from YCbCr to RGB is the most common color space transformation being used. For advanced multimedia application, there would be more functions between inverse transformation and color space transformation. The complexity of a multimedia decoding algorithm largely depends on the processes in this interval. However, the complexity and optimization of multimedia decoding algorithm are not the concern and discussion in this article, i.e., the performance of decoding on a single core is not the issue here. The focus is the data partitioning and dataflow planning for multi-core system.

3.5 Dataflow Planning for Multimedia Decoding on CBE

The proposed dataflow planning for frame-based multimedia decoding on Cell is summarized in Figure 3-6 below. Start from the input multimedia stream and end at the display of frames. Partitioning of multimedia stream into encoded frame and dispatch to each SPE is the first task of PPE.

The encoded frames in a multimedia stream are divided into groups. There are 6 frames in each group. The reason why the number “6” is chosen here is obvious: there are 6 available SPEs in PS3. The encoded frames are allocated to SPEs in a round-robin fashion, i.e., the frame 1, 7, and 13… are allocated to SPE 1, the frame 2, 8, and 14 are allocated to SPE 2, … etc. It should be noted that the actually data size flowing in and flowing out of a single SPE depends on the I/O buffer in LS. An entire frame cannot be pulled in or out the LS once a time undoubtedly. Thus the I/O buffering is a significant issue in the design of multi-core programming. This would be discussed in section 3.6.

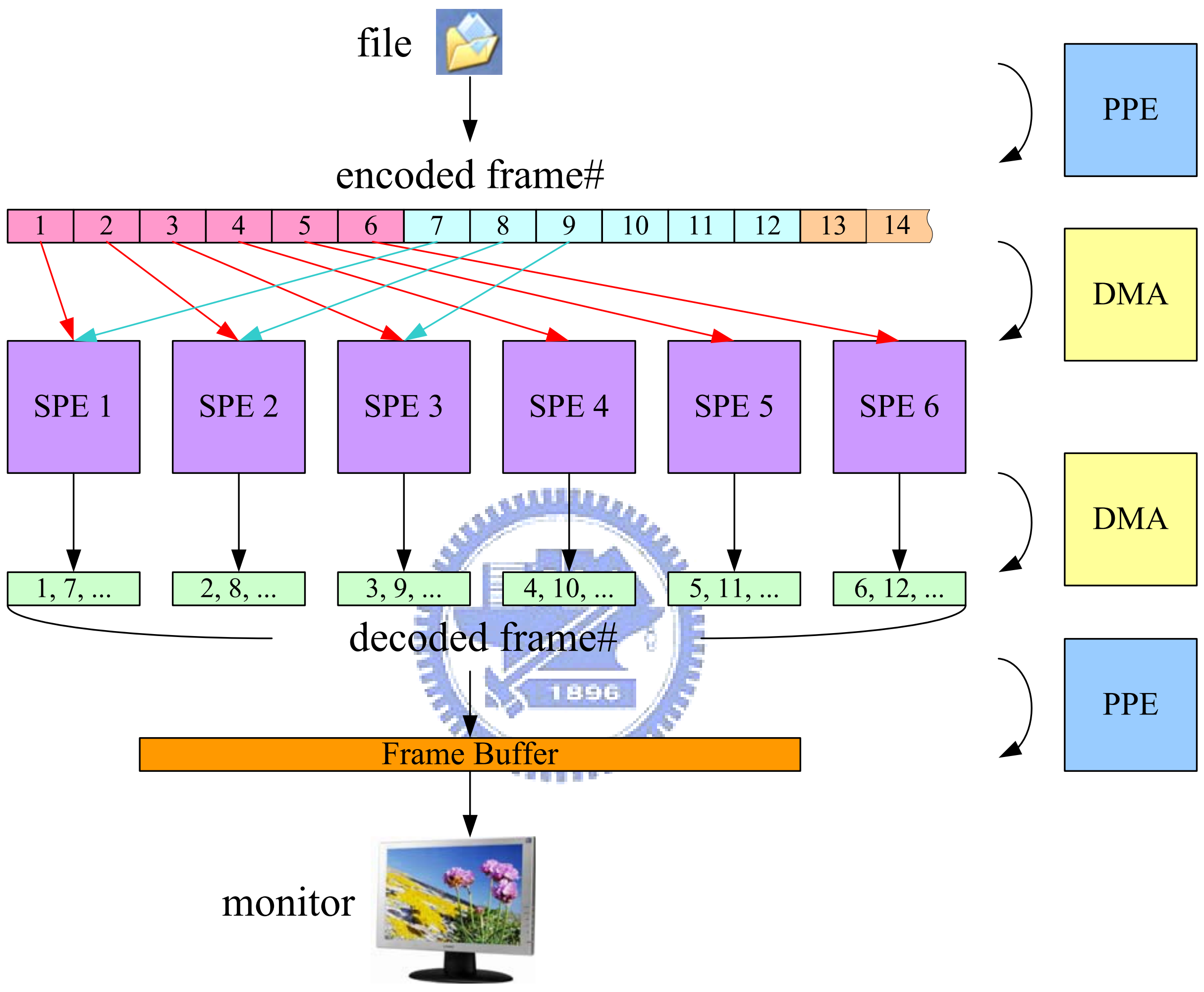


Figure 3-6 Dataflow planning for multimedia decoding on Cell

The decoding mechanism needs 6 frame repositories in shared memory. Each SPE utilizes its own frame repository thorough all decoding of frames. Each frame repository occupies one RGB frame in size. The content in frame repository could be updated right after the display of current frame. In some aggressive multimedia decoding standards such as H.264, the data communications would cross the frames, i.e., inter-prediction or motion compensation. In such case this decoding mechanism needs 6 larger repositories in shared memory to accommodate the longer range of data dependency. It is feasible in desktop-level computers such as PS3.

Finally the display process takes each frame from 6 repositories also in round-robin and shows the frame on the monitor through the frame buffer in PS3. The PPE takes charge of the frames' movement from the 6 frame repositories to the frame buffer.

3.6 DMA and Buffer Allocation on CBE

For MFC of SPE, the transfer size of a single DMA command ranges from 1 byte to 16 K bytes. It would be a confusing problem for programmers to decide on how large the size should be with DMA commands. Therefore, the beginning of optimization is the transfer size per DMA command. We keep the size of input and output buffers constant and tune the transfer size of DMA commands. The detail of experiment on different DMA size is shown in the next chapter. Now, the buffer allocation and structure is considering.

The processing of each code is variable in length, thus a basic implementation is to read one byte from the input stream each time until the code is valid. This method is feasible on a single core system undoubtedly, but it's dangerous for the multi-core system where massive data communication holds. For CBE the DMA is not efficient enough. To get one byte each time from the input stream at shared memory is a huge burden and would become one performance bottleneck. This would be shown in the next chapter. Thus input buffer is implemented as a ring structure as shown in Figure 3-7 and read from the file block-wise. Now the reading of one byte each time accesses the input buffer in LS, not the shared memory through DMA. The term "ring" means that the data in buffer is continuous across the half ring boundary, and the access of this ring structure could be simply in a smooth way.

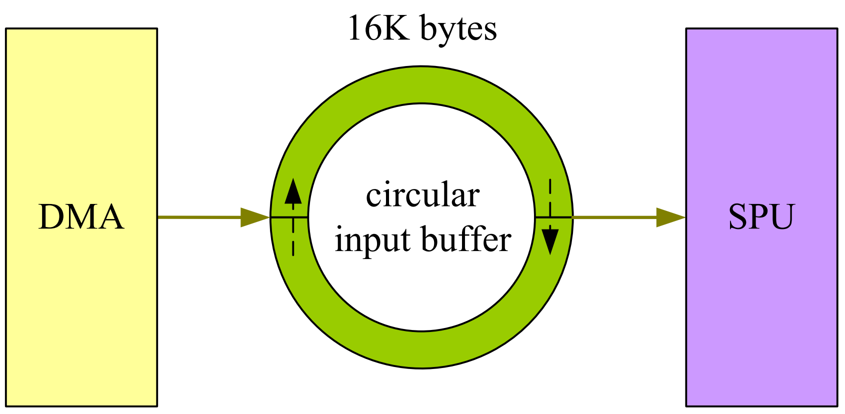


Figure 3-7 The input buffer is a ring structure

It's noted that the update of half ring is an input DMA command that could occur anywhere depending on where the boundary checking and update places. The simplest way of boundary checking and update is to check right after each reading of input buffer in LS. However the boundary checking and update could be move to a bigger block, namely, the MB. It could be placed where the processing of one MB is complete. In this way the size of half ring must be larger than the size of one MB in bytes, and the number of boundary checking could be reduced. It's a common case because that the MB size is usually not more than tens of bytes.

For a decoding application, the output bandwidth is many times larger than input bandwidth. The decoding procedure could simply be viewed as a decompression of a tightly compressed data stream.

Because multimedia decoding is a MB-based algorithm, the output basic unit is MB. The MB size commonly used is 16*16 = 256 points. Thus the output size is 256*3 = 768 bytes if the red, green and blue colors are stored as 1 byte each (256 levels). There are 16 rows and 16*3 = 48 columns in one output MB. For a single DMA command, the access address should be consecutive. To be more specific, the DMA command contains only the starting address and the access size in bytes. The size of a single DMA command for output is set at 128 bytes. The packing of pixels for display is done on PPE. The typical form of output buffer for MB-based multimedia decoding algorithm is shown in Figure 3-8. The term “n macroblocks” in the figure means the width of the output buffer in terms of the number of MBs. There are 16 pixels in each row of one MB, i.e., 48 bytes in each row for RGB 3 colors. The discussion of how to set “n macroblocks” is left in the next chapter.

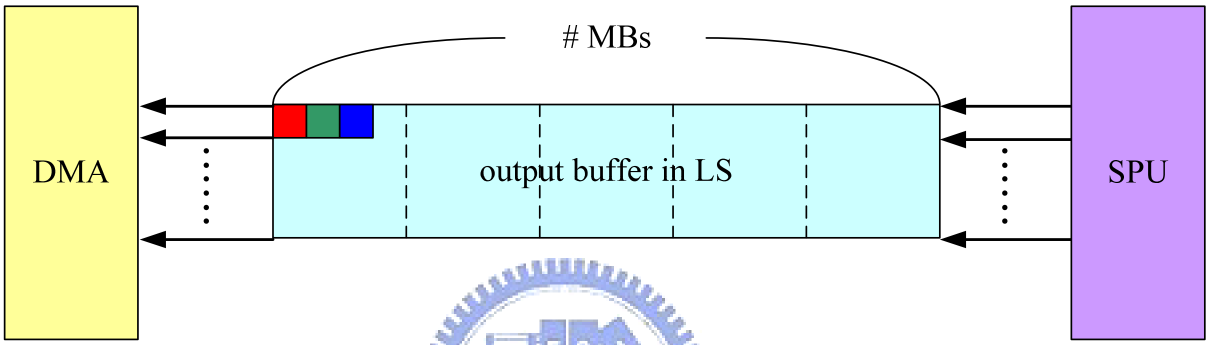


Figure 3-8 The single output buffer

An improvement at output buffer could be made in this decoding scenario. The double buffering is a technique to overlap the data transfer and the computation: the current result is written to one buffer by SPE, while the previous result residing in the other buffer is sent to the shared memory through DMA. The difference between the single buffering and double buffering is shown in Figure 3-9. “MFC” refers to the memory flow controller and “SPU” to the synergistic processing unit as mentioned in Chapter 2. SPU is busy at decoding while MFC is busy at output DMA commands. There are two buffers in the double-buffering case, in which MFC and SPU utilize buffer A and buffer B interchangeably.

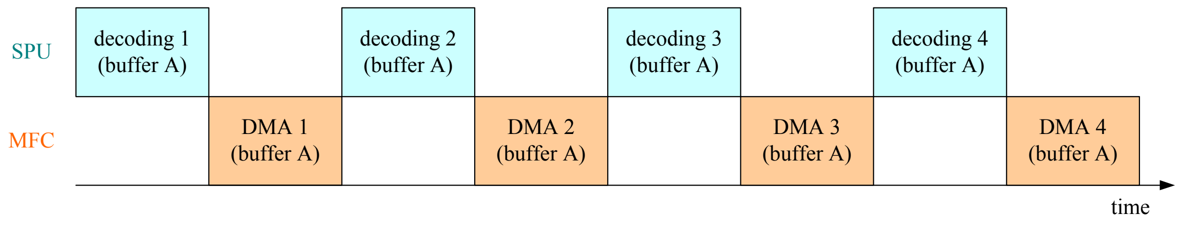


Figure 3-9(a) Single buffering

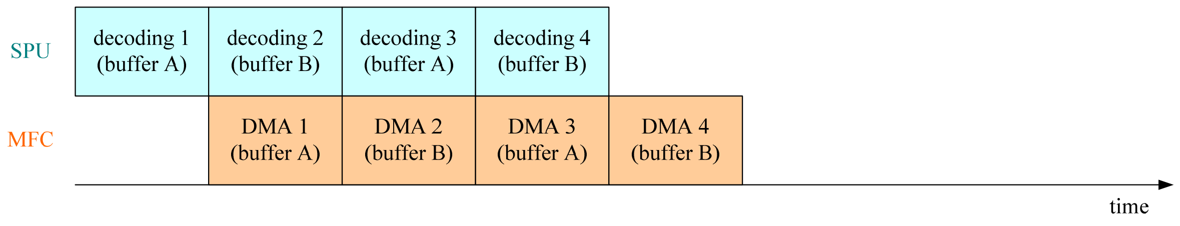


Figure 3-9(b) Double buffering

However, there are 2 situations in which the effects of double buffering are not so attractive. Namely, the 2 asynchronous processes are very imbalanced in consuming-time. These situations are shown in Figure 3-10. In Figure 3-10(a), the computation time of SPU is much less than the DMA time of MFC. This is what this thesis focuses on. It is noted that the DMA time of the multi-core system increases in a multiple trend with the number of cores. There are 6 SPEs in CBE, and the DMA time should be 6-fold as compared to 1 SPE. The total system performance is bounded if the DMA time is much larger than the computation time, i.e., the utilization of 6 SPEs couldn’t have 6-fold speedup as compared to that of 1 SPE. This is the critical problem which must be solved in the multi-core system. The DMA and the associated buffer allocation would be discussed in the followings.

The other situation is shown in Figure 3-10(b), where the computation time of SPU is much more than the DMA time of MFC. This situation is solved by the optimization of decoding algorithm and the utilization of chosen processor. The commonly used techniques are single-instruction-multiple-data (SIMD), loop unrolling, and some special extended instructions. It’s not the point discussed here.

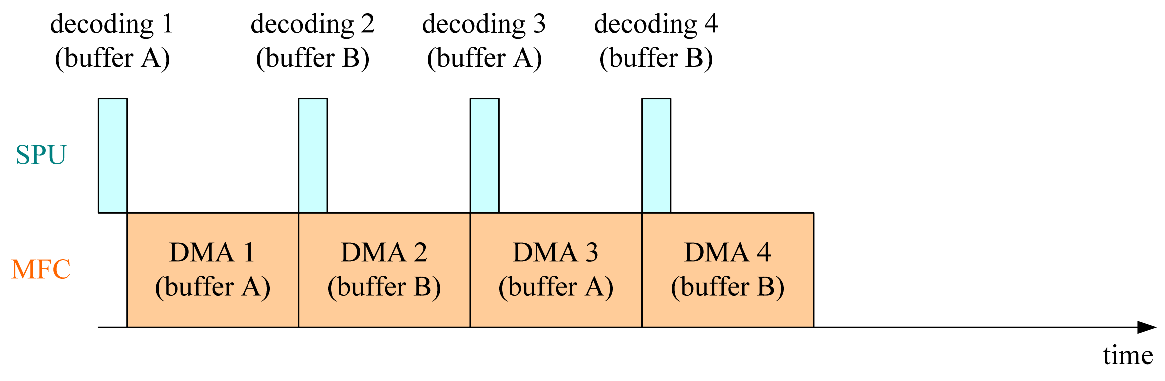


Figure 3-10(a) The computation time of SPU is much less than the DMA time of MFC

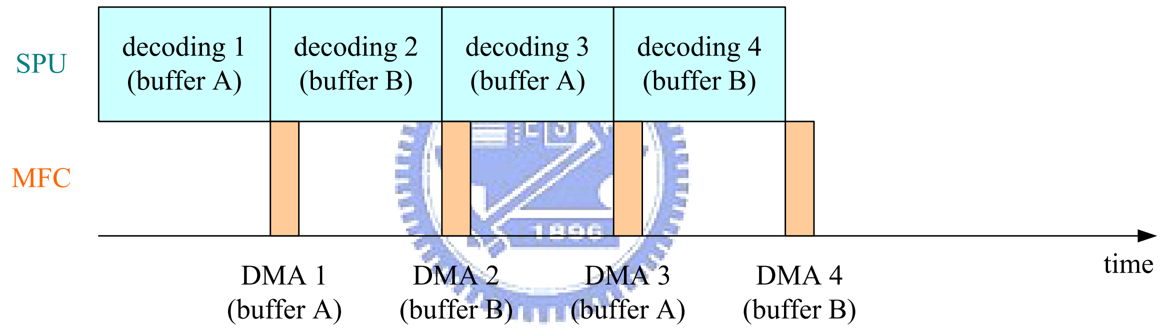


Figure 3-10(b) The computation time of SPU is much more than the DMA time of MFC

The implementation of double buffering for multimedia decoding is shown in Figure 3-11. There are 2 pairs of data movement depicted as the arrows. Pair A is the upper left and the bottom right arrows and Pair B is the upper right and the bottom left arrows. These 2 pairs take place in a ping-pong fashion, and the computation and DMA could utilize 2 buffers respectively at the same time.

The 2 halves of input ring buffer operates cooperative. When the address of this ring buffer excesses the total size, it wraps around and starts at the beginning of ring. The ring buffer updates half each time. When the access address excesses the half boundary, the other half which is non-accessed updates by reading half size of data from the input stream. The basic unit of the size of input ring buffer is 128 bytes. The reason and the associate discussion are provided in Chapter 4.

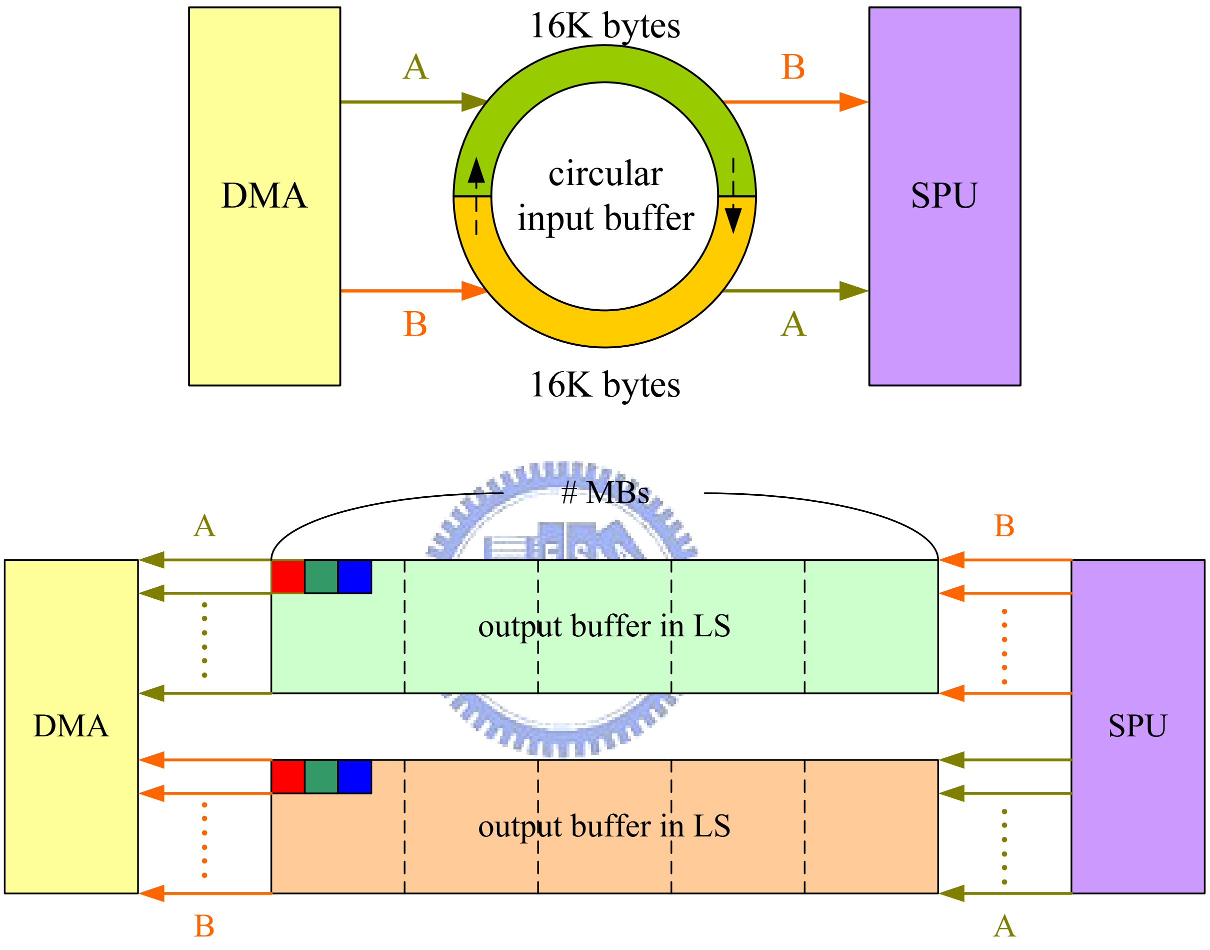


Figure 3-11 The I/O double buffer

The double buffering matches the property that DMA transfers are asynchronous to SPE computation. It's noted that this technique double increase the output buffer size. There may be a situation that double buffering isn't feasible in such limited LS in SPE. It's a tradeoff issue of memory storage and the performance.

4 Experimental Results

This chapter provides the experimental results of different size of DMA transfers and the allocation of I/O buffer. The experimental environment is shown below.

- ◆ Instrumentation
  - ■ PlayStation 3
  - ■ Linux kernel: 2.6.16 (Fedora) [14]
  - ■ HD monitor
- ◆ System input
  - ■ 1080p motion JPEG sequence
- ◆ System output
  - ■ 1080p decoded frames displayed on monitor

The code structure of experimental program is shown in Figure 4-1. The PPE controls the decoding task on each SPE, and the SPEs take charge of the decoding of frames. The PPE first create the spe_context for each SPE to start the running of SPE program. SPE freely runs to the location waiting for the I/O address from PPE. The PPE sends the I/O address to SPE needed for the decoding of one frame. The SPE continues computing after the reception of mail from PPE. When the decoding task is done the SPE sends a mail to PPE informing about the completion. The PPE display the decoded frames after the reception of completion mail and send another I/O address to the idle SPE. The detail operation for PPE and SPE side is made below.

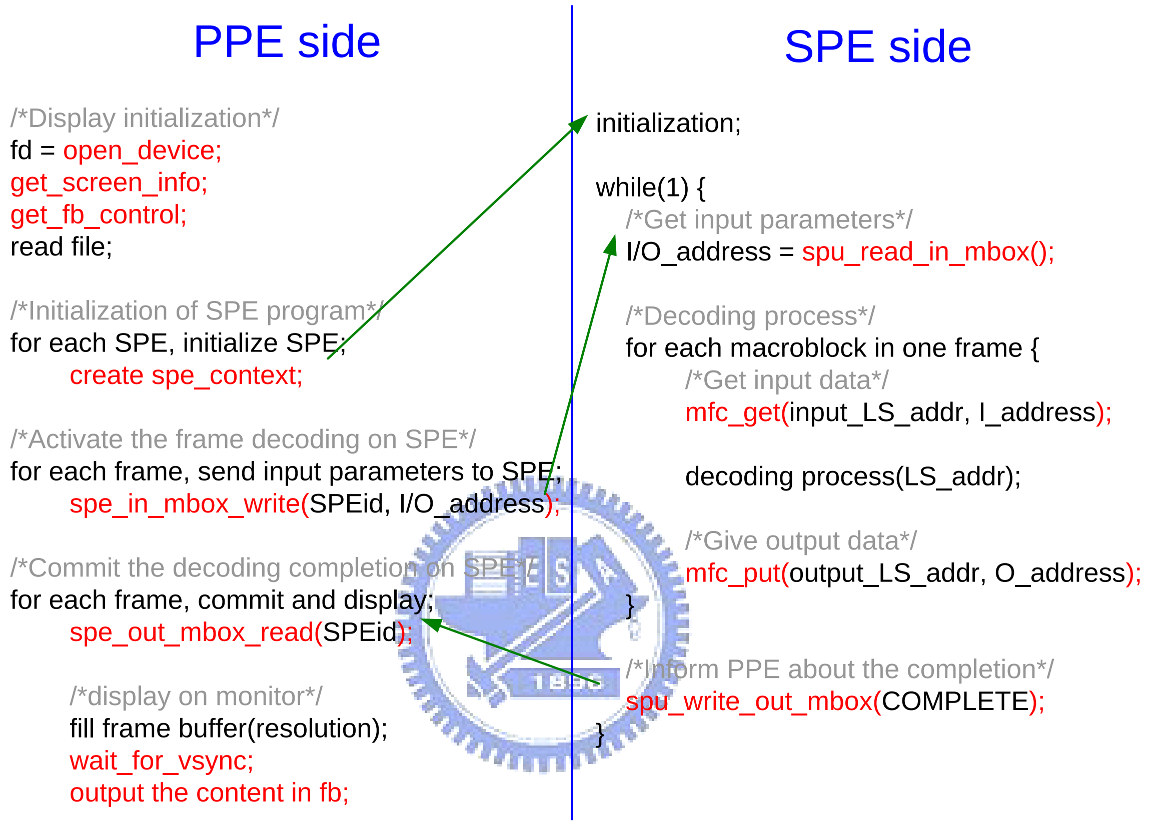
At first, the PPE side is considered. PPE accumulates 6 encoded frames from the input stream and allocates them to 6 different SPE threads. The PPE mail the memory pointers of 6 encoded frames and the 6 frame repositories to individual 6 SPEs. After this initial sending of mails PPE waits for the response of one SPE. The actual meaning of response is the completion mail sent by SPE to PPE. When PPE knows the decoding of one frame is done, the display procedure follows.

The display procedure fills the RGB color pixels into the frame buffer of PS3 system. This procedure takes about 1/120 seconds per 1920*1080p frame. Thus the display procedure would only slightly degrade the performance and not be the critical part of multimedia decoding.

After the display of frame, the returned SPE which completes its decoding gets another encoded frame pointer. The returned SPE continues on the decoding job activating by this mail from PPE.

If there are still frames need to be decoded, PPE waits for another comeback of one SPE and repeats the flow described above. If all frames are sent to SPEs, PPE goes to the final step. PPE waits the completion mail from all SPEs and display the last frames. The overall decoding ends.

As for the SPE side, SPEs could be viewed as servants of PPE. The practical implementation in SPE is an infinite loop starting from the reception of mail and return to the loop at the sending of completion mail. SPEs get the mail containing the input encoded frame pointer and the output decoded frame pointer from PPE. Then SPE knows where to get the input and put the output in shared memory. SPE starts running the decoding function when the PPE sends this information to it. At the end of decoding of one frame SPE sends a mail to PPE to tell PPE the completion.



```
/*Display initialization*/
fd = open_device;
get_screen_info;
get_fb_control;
read file;

/*Initialization of SPE program*/
for each SPE, initialize SPE;
        create spe_context;

/*Activate the frame decoding on SPE*/
for each frame, send input parameters to SPE;
        spe_in_mbox_write(SPEid, I/O_address);

/*Commit the decoding completion on SPE*/
for each frame, commit and display;
        spe_out_mbox_read(SPEid);

        /*display on monitor*/
        fill frame buffer(resolution);
        wait_for_vsync;
        output the content in fb;
```

```
initialization;

while(1) {
    /*Get input parameters*/
    I/O_address = spu_read_in_mbox();

    /*Decoding process*/
    for each macroblock in one frame {
        /*Get input data*/
        mfc_get(input_LS_addr, I_address);

        decoding process(LS_addr);

        /*Give output data*/
        mfc_put(output_LS_addr, O_address);
    }

    /*Inform PPE about the completion*/
    spu_write_out_mbox(COMPLETE);
}
```

Figure 4-1 The code structure of experimental program

The frame buffer is a double buffer and it could be viewed in 3 layers. The first layer is application layer which is controlled by programmers at top level. The control registers could be modified by users manually to enhance the controllability of frame buffer. The performance is improved by this way without the usage of application programming interface (API). The second layer is kernel layer which is handled by operating systems (OS). The communication and operation mechanisms between Cell processor and graphic processing unit (GPU) are automatically controlled by kernel. The frame buffer is a virtual memory, and it's managed by kernel. The third layer is GPU layer which is the heart of graphic computing and display. There are double frame buffer in GPU, and the content is delivered by kernel via DMA. GPU provides the display on monitor. The schematic of these 3 layers is shown in Figure 4-2.

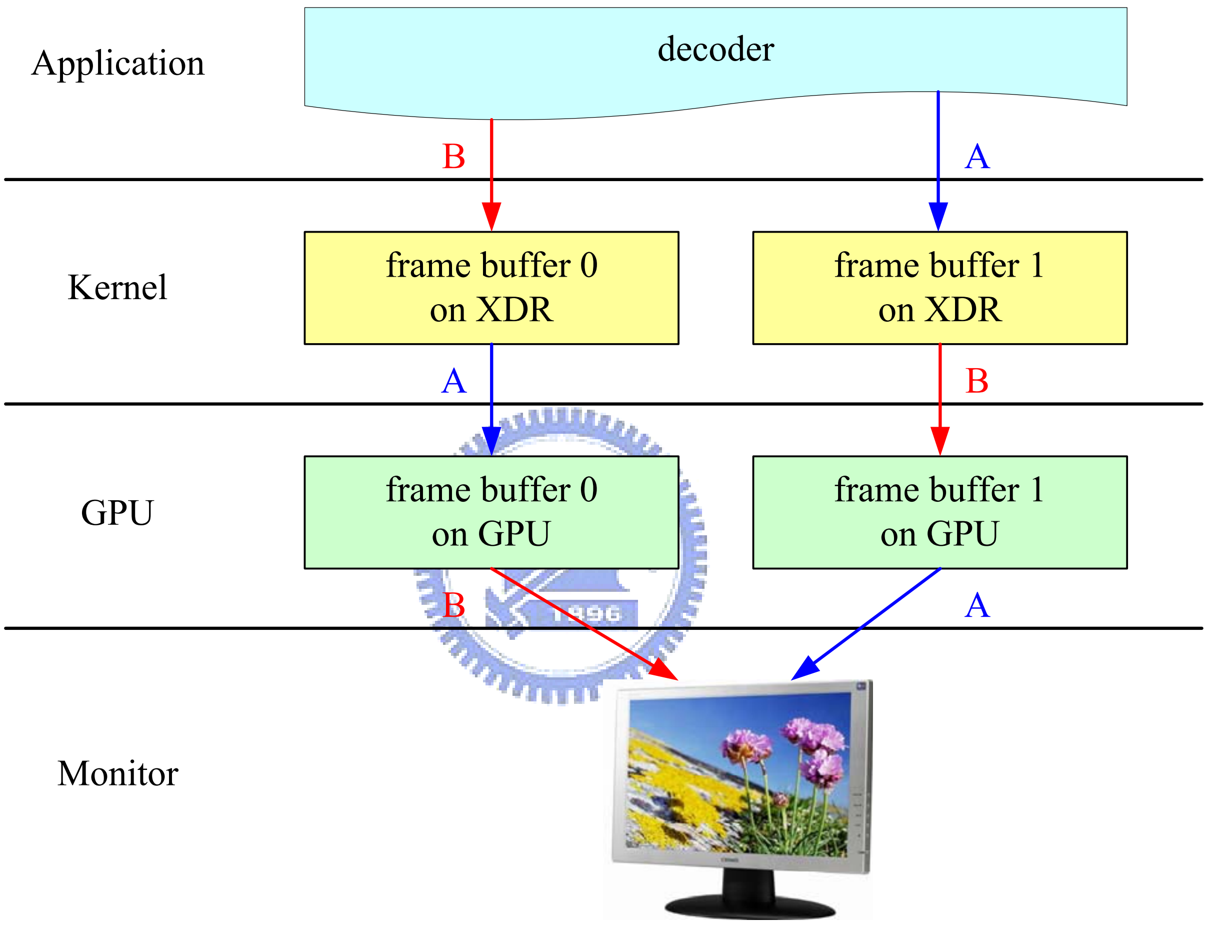


Figure 4-2 The double frame buffer

The arrows A and B represent the double buffering operation mechanism. When application is busy writing results to frame buffer 0 on shared-memory (Rambus, XDR), the frame buffer 1 on XDR moves its content to frame buffer 1 on GPU via DMA, and the pixels displayed on monitor comes from frame buffer 0 on GPU. Then the frame buffer flips to the other one and follows the mechanism described above. It's noted that the computing time of display isn't included in the decoding statistics below.

There are 3 optimization techniques applied in this thesis. They are vectorization, parallelization, and dataflow optimization. The performance with combinations of these techniques is shown in Figure 4-3. The unit of y-axis is frames per second (fps), and the x-axis represents the optimization techniques.

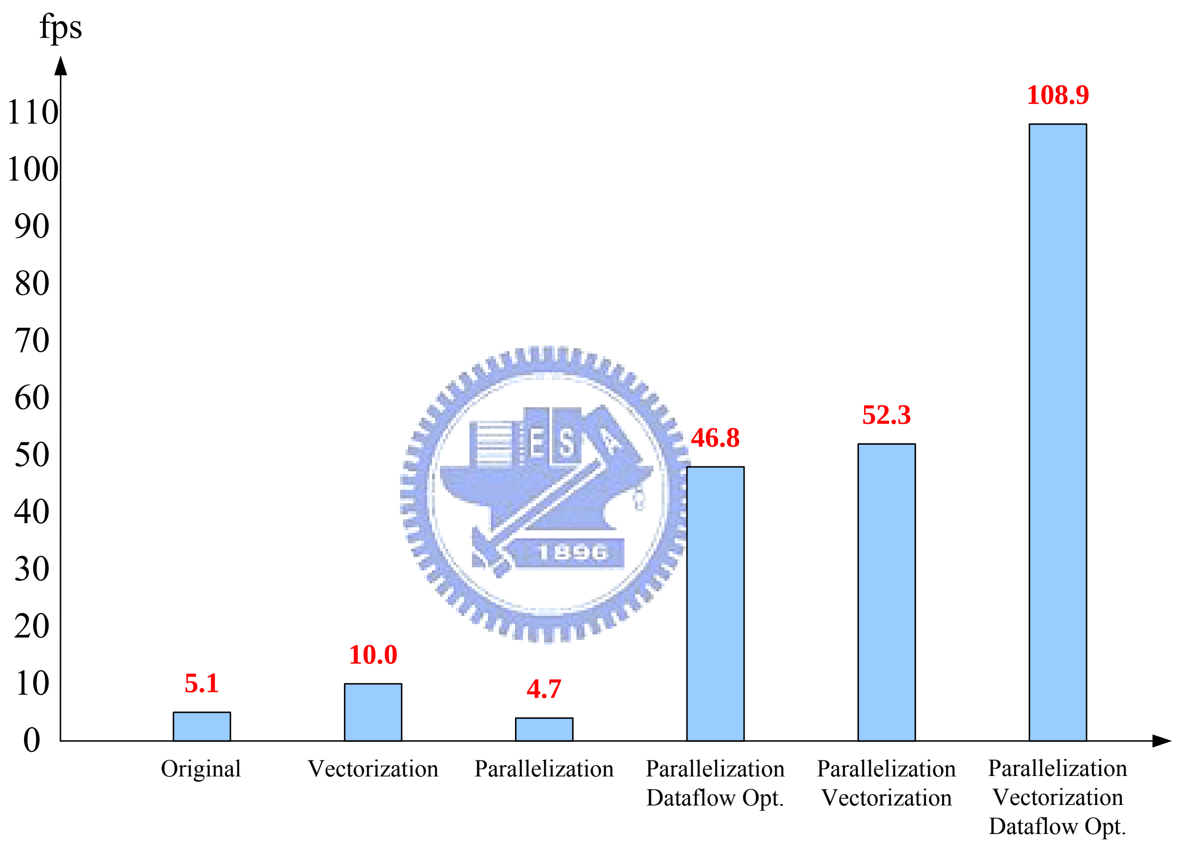


Figure 4-3 The performance with combinations of optimization techniques

The experiment is divided into 4 categories as the followings.

- The transfer size per DMA command
- The size of I/O buffer
- The double buffering
- The utilization of different number of SPEs

The purpose of this experiment is to find the appropriate transfer size of DMA commands and the associate buffer. In the multi-core system, the communication between each core plays an important role. The performance is highly dependent on the behavior of DMA. The optimized results summarize in the final section with the number of utilized SPEs going from zero to six.

4.1 The Transfer Size per DMA Command

In this experiment, the purpose is to find the appropriate transfer size of DMA command on CBE for multimedia decoding applications. The input end is a variable-length reading process and the output end is a block writing process. The input ring buffer reads the stream if the data pointer crosses the boundary of half ring, and the output buffer writes the result to shared memory after the decoding of each MB.. The detail is described in Chapter 3.

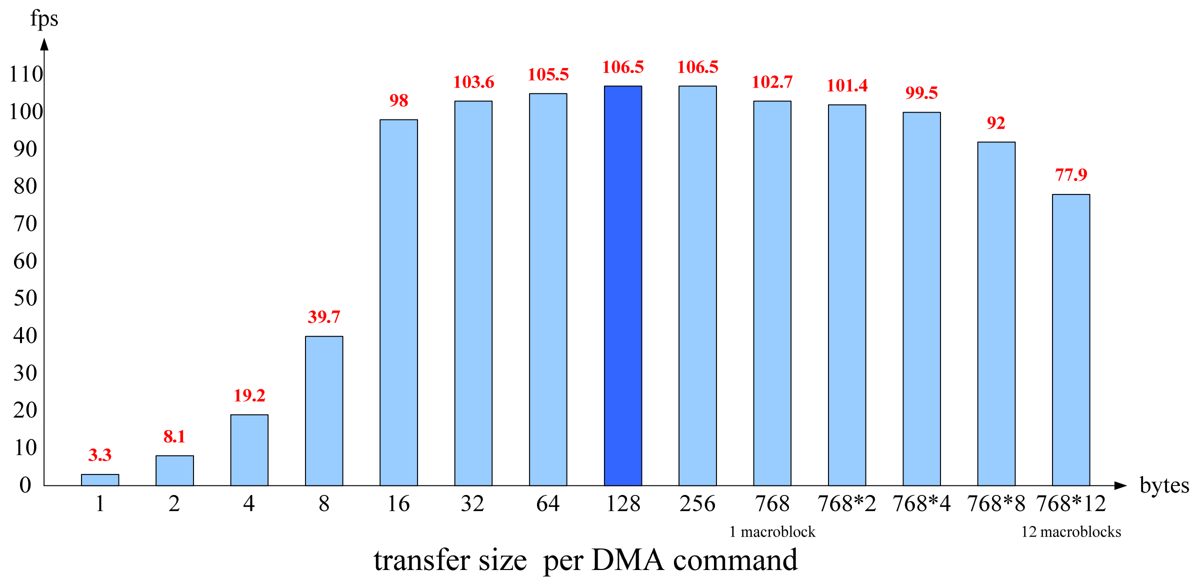


Figure 4-4 The transfer size per DMA command

The unit of y-axis is frames per second (fps), and the unit of x-axis is byte. The size of input buffer size is set at 16 K bytes, and size of output buffer is set at 768*12 = 9216 bytes (the size of 12 output macroblocks). The transfer size per DMA command is changed from 1 byte to 768*12 = 9216 bytes as shown in Figure 4-4.

The considerably poor performance comes from many DMA transfers with small size of data. The issuing of DMA commands and the data shuffling consume much time in data movement by small transfers. When the transfer size per DMA command is 32 bytes, the time spent on DMA comes to an acceptable level because the computation time on SPU dominates the overall performance.

This figure shows that the performance improvement by DMA size saturates at the location "128 bytes". This result makes sense for the CBE. The DMA commands favor the 128-byte transfers. This size is the same as the L2 cache-line size. IBM suggested that, for high performance, the access of memory should be 128-byte alignment, and the size should be an even multiple of 128 bytes. The little degradations at 768 bytes and its multiples conform to the fact that 768 is not a multiple of (128×2^n). The degradation should be noted. Even though the number of DMA command issuing is decreased after 768 bytes, the performance doesn't scale up but drops. Thus the DMA transfer command with size not a multiple of (128×2^n) should be avoided as possible as we can.

It could be summarized that the issuing of DMA commands with transfer size larger than 128 bytes gets nothing better. A DMA command with transfer size larger than 128 bytes would be divided into transfers whose sizes are all 128. Put it in another way, the basic unit of DMA transfers is 128 bytes. Although the improvement among 32~128 bytes is not high, the improvement in the multi-core system could become obvious when the DMA rate of advanced application with higher data compression is increasing. In other words, the improvement of system performance would be multiplied if the communication burden becomes heavier or data rate becomes higher. The issues of DMA commands with small transfer size should be avoided as many as possible. The number of DMA commands would significantly influence the system performance, so the programmers must reduce the number of DMA commands to a suitable amount.

4.2 The Size of I/O Buffer

In this experiment, the purpose is to find the appropriate size of I/O buffer in a multimedia decoding application. It's noted that the input is variable-length. Thus the input buffer is set at a fixed size and the size must bigger than one input macroblock. The size of one input macroblock is usually tens of bytes, as compared to the size of output macroblock which is 768 bytes. The update of input buffer could take place in every beginning of decoding of one MB. For simplicity and safety, at first the input buffer size is set at 16K bytes as the same as that in section 4.1. The transfer size per input DMA command is fixed at 16K bytes which is the appropriate size explored in the first experiment, to be more specific, 16K is a multiple of (128×2^n). The transfer size per output DMA command is 128 bytes if the size of output buffer is less than or equal to 128 bytes. Otherwise, the transfer sizes of DMA commands approach the multiple of (128×2^n). For example, 768 bytes should be divided into 512 bytes and 256 bytes and transferred with 2 DMA commands separately. The reason why the division is made comes from the observation in the 1st experiment.

The unit of y-axis is frames per second (fps), and the unit of x-axis is byte. The size of output buffer is changed from 16 bytes to 9216 bytes as shown in Figure 4-5. It should be noted that the DMA transfers are aligned transfers in CBE. To be more specific, the 4 least significant bytes (LSB) of the source and destination address must be the same. Thus the minimum buffer size could be used is 16 bytes. The size of one MB is 768 bytes contains RGB pixels. The maximum buffer size chosen here, namely, 9216 bytes comes from 12 macroblocks. In Figure 4-4, we found that the performance saturates when output buffer size is 768 bytes. The bad performance at 16, 32, 64 bytes comes from the same reason stated in previous experiment. The number of DMA commands issued is larger, and the SPU should wait for the completion of DMA transfers each time while the buffer is full within the decoding of one MB. The latter is also the reason why the performance at 128 and 256 bytes is worse than that at 768 bytes, even though the number of DMA commands issued is the same.

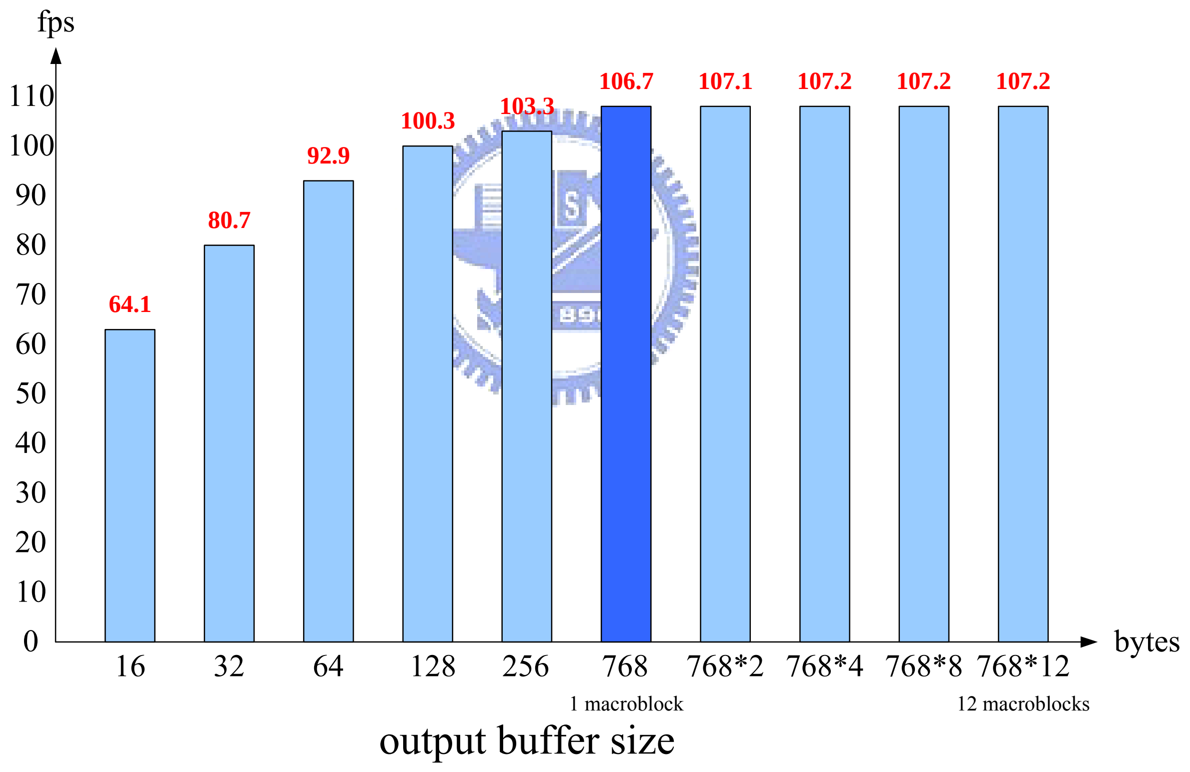


Figure 4-5 The size of output buffer

The performance doesn’t scale with larger size of output buffer. The performance saturates at 768 bytes. The reason is that the time of data transfer is almost unchanged for size of output buffer greater than 768 bytes. The range of data locality is limited within a single

MB, i.e., the computing of this output MB doesn't need the information from the other output MB. The DC value of DCT from previous MB is kept right after the Huffman decoding. On the other side, the performance drops because that the number of DMA commands issued exceeds the depth of command queue with smaller output buffer. The memory flow controller (MFC) would stall the execution until all issuing DMA commands are queued. The summary could be made. The issuing of DMA commands should never exceed the maximum capacity of DMA command queue. The associate buffer size needs not to be the largest size occupying all the remaining room in the small LS. The programmers would not have to leave a considerably huge buffer size of an application. The bandwidth of CBE and the DMA efficiency achieves its maximum capability at a point earlier than we expect. Although the programmers could simply assign all the rest of unused LS to the buffer, it is unnecessary and would make a large modification with the growing of application, i.e., the usage of LS for algorithm is increased and the rest of LS which left for buffer allocation is decreased therefore.

As for the size of input buffer, we set the size of output buffer to 768 and 768*2 and find the best combination of I/O buffer. It's reasonable that we search the size of input buffer after output buffer. Note that the output bandwidth is almost 20-fold larger than input bandwidth. The input buffer size is a minor effect on system performance. The experimental results not shown here present that the size of input buffer almost has no influence on the system performance. As for we know the fact that the maximum transfer size of a DMA command is 16K bytes, it's reasonable that the size of input buffer is simply set as 16K bytes. The input buffer could be filled up by a single DMA command and only occupies one slot maximally of DMA command queue.

In the multimedia decoding applications, the summary is that the entries of DMA command queue should be never overflowed. The decoding time of each MB is almost equaled, so the issuing rate of DMA commands could be amortized as long as the queue is not overflow. That's the reason why the performance results at 768 bytes or lager are almost the same. The arbitration mechanism in EIB is round-robin, so the 6 queues in MFC of 6 SPEs would get the EIB resource in a fair way. The best way is always trying to fill up the DMA queues and simply waiting for the arbitration granted next time. It's should be noted again that the I/O buffer doesn't have the necessity to occupy all remaining capacity of the small local store (LS). This utilization of LS by brute force is neither an easy nor good idea.

4.3 The Double Buffering

The concept of double buffering is that the content of one part is updating while the content in the other part is transferring to outside place. The overlapping of computation and transportation is the goal of double buffering technique. The detailed description is provided in Chapter 4.

The input buffer is a ring structure. The ring is divided into two halves. The output buffer is a ping-pong buffer, i.e., the two parts of output buffer operates interchangeably. In the previous experiment, the input and output buffer structures are single buffer. The purpose of this experiment is to find the performance improvement by double buffering. The experiment divides into 2 parts.

In Figure 4-6, the experiment holds on different transfer size per DMA command. The size of input buffer is fixed at 32K bytes, and the size of output buffer is fixed at 18432 bytes. It's just the double size in 1st experiment. The figure reveals 2 facts. Although the data communication time is very huge for transfer size among 1~8 bytes, the double buffering can't do any help. The huge DMA time here comes from the queuing delay because the DMA command queue overflows. The double buffer couldn't ease the queuing delay. On the other hand, the data shuffling time could be reduced by the double buffering effect as shown for transfer size bigger than 768 bytes. The reason is that the data shuffling is handled by MFC asynchronous to the computation handled by SPU.

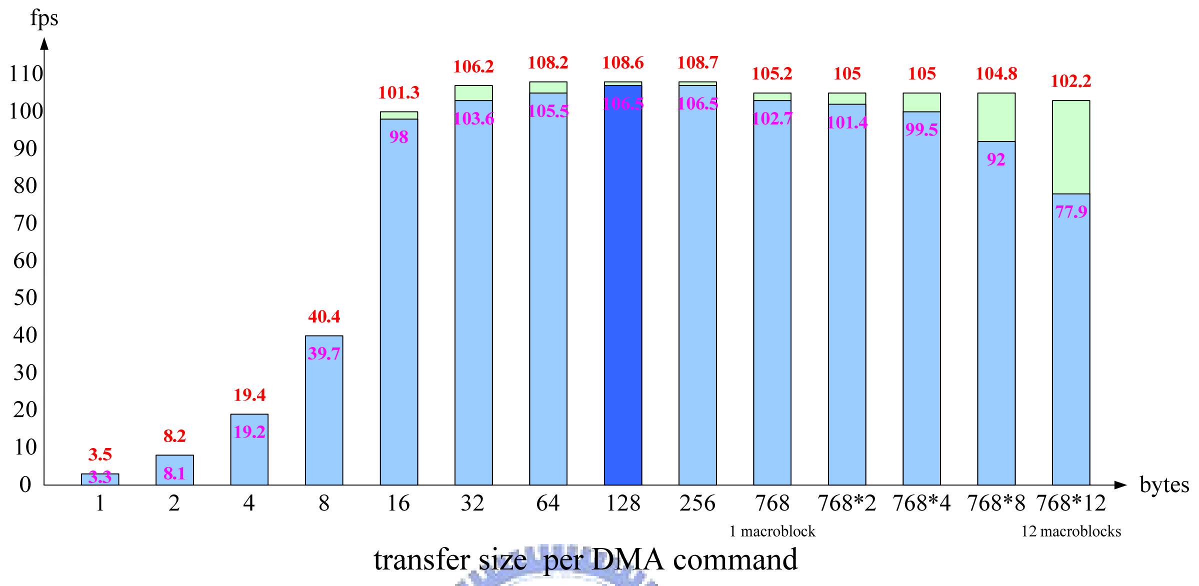


Figure 4-6 The double buffering for different transfer size per DMA command

In Figure 4-7, the experiment holds on different output buffer size. From the summary in 1st experiment, the transfer size per DMA command is set at 128 bytes. The size of input buffer is fixed at 32K bytes. The size of output buffer is changed from 32 bytes to 18432 bytes. It's just the double size in 2nd experiment. The result of previous experiment on single output buffer is also in this figure for the purpose of comparison. The unit of y-axis is frames per second (fps), and the unit of x-axis is byte. It's noted that the x-axis keeps the same as Figure 4-5 for clarity, and the actual buffer size is doubled since the double buffering is utilized. The performance saturates at 1536 bytes as the case in the 2nd experiment (768 bytes for single buffer). The improvement provided by double buffering ranges from about 25% to 1.6% as shown in Figure 4-8.

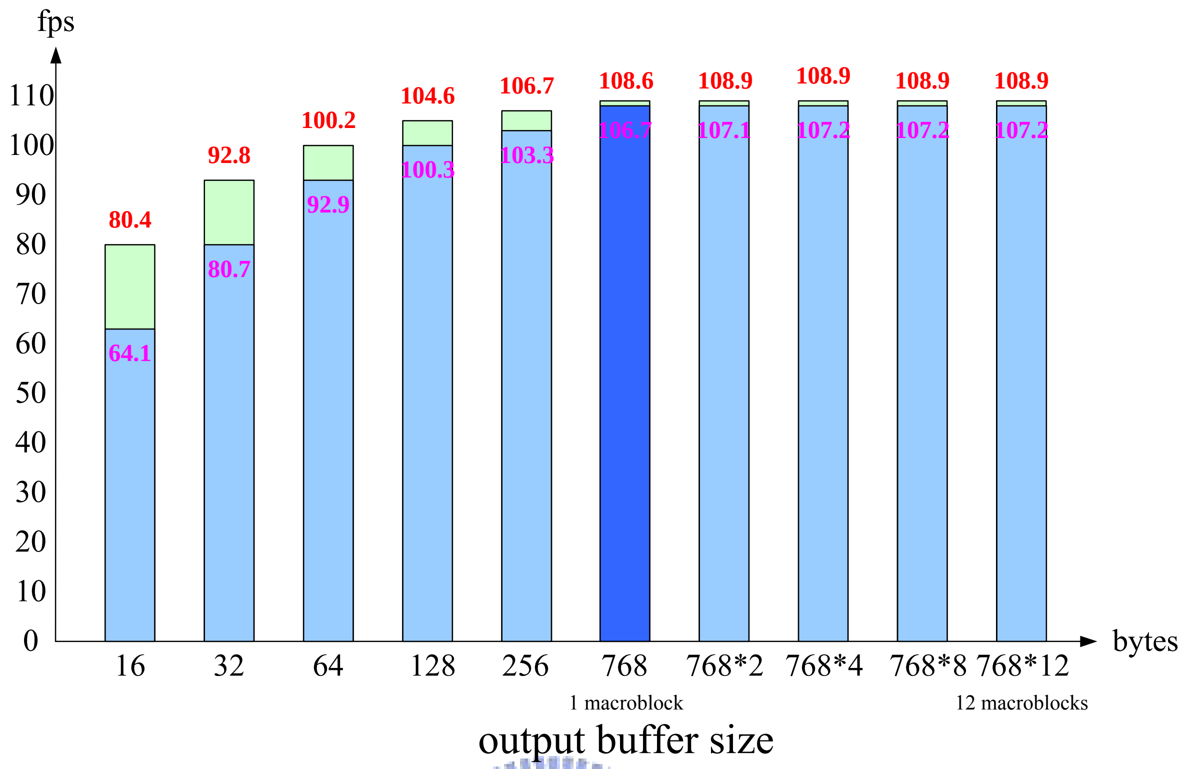


Figure 4-7 The double buffering for different output buffer size

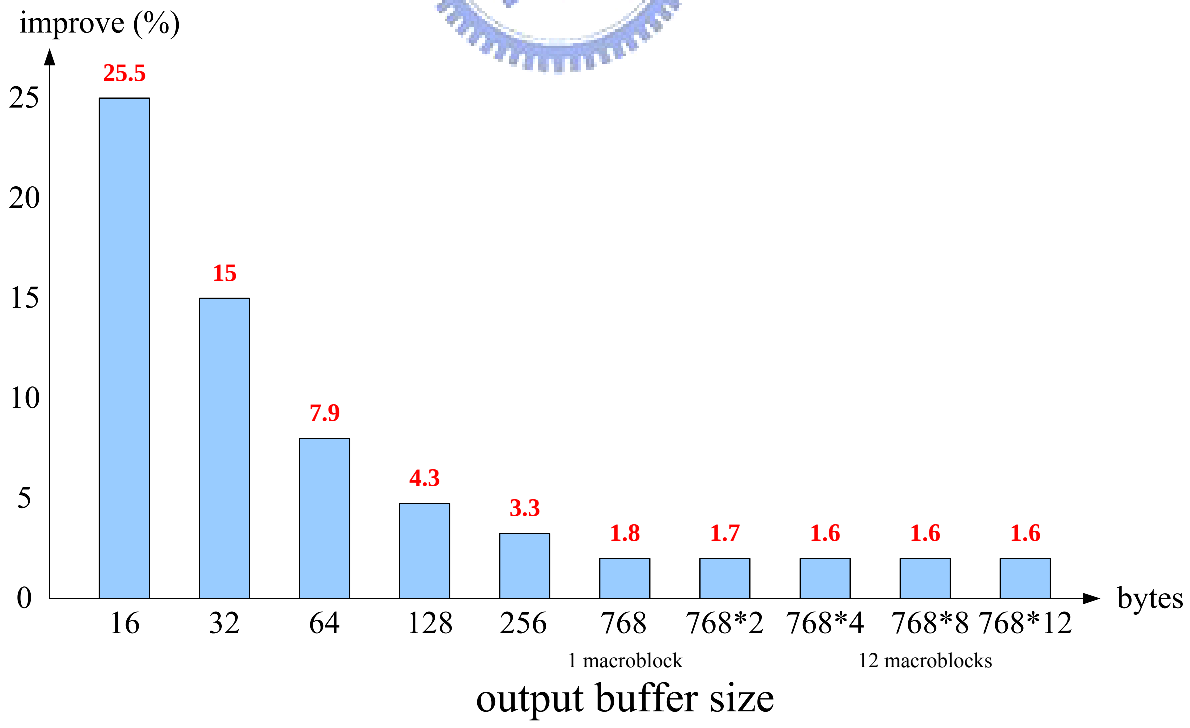


Figure 4-8 The improvement of double buffering compared to single buffering

The improvement descends as the size of output buffer rising. This shows the actual contribution where the double buffering gives. The time spent on DMA could be divided into 2 parts. The 1st part is the time from the issuing to the committing of DMA command, and the 2nd part is the time that actual data transfer spent on EIB. The double buffering technique only improves the 2nd part. It overlaps the time of data transferring and computation. As the statement before, if the size of output buffer and associate transfer size per DMA are greater than or equal to 128 bytes, the transfer size is automatically divided into units of 128 bytes. This means that the time spent on EIB is the same for the cases where the sizes of output buffer are greater than 768 bytes. When the output buffer size is less than 128 bytes, the time of data transferring is multiplied. For example, the time of data transferring whose output buffer size is 64 bytes doubles that whose size is 128 bytes. The poor improvement for sizes of output buffer greater than 768 bytes comes from the fact that the DMA time is very small in these settings.

The improvement at peak performance (768 bytes) is less than 2%. This result tells that the output double buffer actually works but not so attractive. The tradeoff should be considered. In order to gain the about 2% improvement, the memory usage raises to 200% for output buffer. However, it's reasonable to make this tradeoff if remaining capacity in LS at last is still large enough.

4.4 The Utilization of Different Number of SPEs

This experiment shows the performance scaling with the number of SPEs. The first experiment in this section is an original code. This original code is the beginning of multi-core program mentioned in Chapter 4. The concept of memory is still the shared-memory model. The access to memory is masked by the API of Cell, i.e., it's a simple

DMA command without any buffering structure. The transfer size of input and output DMA command equals to the size of memory access in single-core program on PPE. The accumulation of DMA transfer is not applied here, and the size of DMA command is the same as that of the output variable. The experiment result is shown in Figure 4-9. The performance with 0 SPEs means the decoding is a single core program on PPE. It's obvious that the performance grows slowly and isn't direct proportional to the number of SPEs. Even worse, the performance with 6 SPEs is not better than that with single PPE.

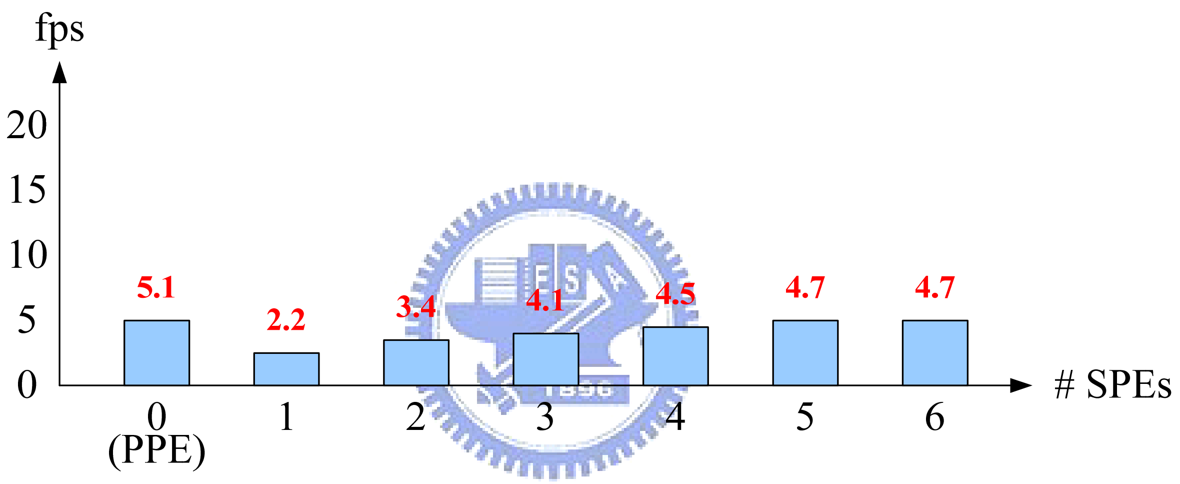


Figure 4-9 The utilization of different number of SPEs (original version)

The result in Figure 4-9 reveals a common problem in multi-core programming, namely, the multiple of improvement is much less than the number of utilized cores. However, this is not surprised in the multi-core system if we look inside the architecture. The communication resource is a limiting factor of system performance because that the bandwidth is limited in all multi-core systems. For CBE the data communication is supported by element interconnect bus (EIB) mentioned in Chapter 2. All data transfers should go through the EIB to other elements. It's almost impossible to achieve the best performance without dataflow planning and optimization. The performance with 6 SPEs is no better than that with a single PPE in this experiment.

The result after all the optimization experiment in previous section is shown in Figure 4-10. The size of input double ring buffer is 32K bytes, and that of output double buffer is 1536 bytes. The transfer size of input DMA command is 16K bytes, and the output DMA commands are 2 commands with 512 and 256 bytes respectively. As shown in the figure, the performance with 6 SPEs is 6-fold with respect to 1 SPE. It's what we expect in a multi-core system. The final results boots the performance about 20-fold as compared to the case of single PPE.

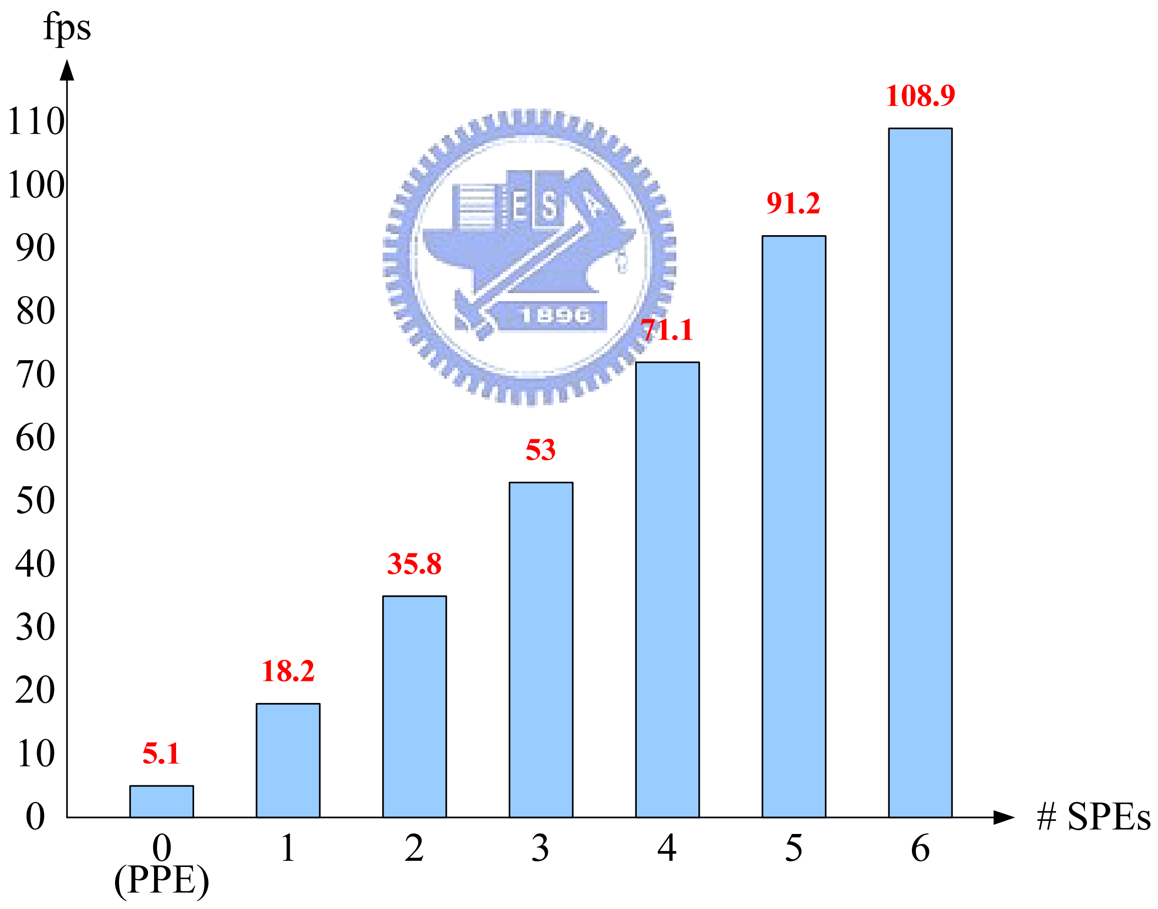


Figure 4-10 The utilization of different number of SPEs (optimized version)

5 Summary and Future Work

In this thesis, a frame-based data partitioning for multimedia processing on PlayStation3 is utilized. The dataflow planning starts from the input stream dividing by the PPE. The PPE allocates the encoded frames to the 6 SPEs in round-robin fashion. Each SPE is responsible for the decoding of an entire frame. The SPE returns the decoded frame and the PPE display the contents in the frame buffer. When all frames are returned, the PPE ceases the decoding process and destroys the threads.

The communication scenario is very important in a multi-core system. There are two mechanisms: mailbox for message passing and DMA commands for data transfers. The number of times of message passing between the PPE and the SPEs is reduced to 2 per frame.

The discussion of transfer size per DMA command and the according buffer allocation is experimented. The appropriate transfer size per DMA command is 128 bytes which conforms to the design of DMA controller on CBE. It's found that an output buffer with 16*16*3 = 768 bytes which is the size of 1 output macroblocks is good enough. The input buffer is a ring structure whose size is 16k bytes. The I/O buffer is implemented as single buffer if the local storage is not big enough, and the tiny profit of performance with double buffering could be ignored. The key point is that the DMA command queue has 16 entries. It's important to note that the overflow of command queue would incur the queuing delay. The marginal utility is negligible if the size of the I/O buffers is increased.

In this thesis, it's supposed that the data dependencies only exist within a single frame, i.e., the inter-frame issues aren't discussed here. However, the proposed concept and optimization could be extended. For some advanced multimedia applications, such as H.264, the inter-frame prediction is the most complex part. The data dependency between frames is bounded in the so-called group of pictures (GOP). The GOP-based data partitioning could be implemented in a desktop level system such as PS3. The optimization of DMA commands and I/O buffer allocations is also applicable for this advanced decoding standard.

REFERENCE

[1] L. Benini and G. De Micheli, "Networks on chips: a new SoC paradigm," *Computer*, 35, pp. 70-78, 2002.

[2] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy, “Introduction to the Cell multiprocessor,” in *IBM J. RES. & DEV.* VOL. 49 NO. 4/5, 2005.

[3] Cell broadband engine programming tutorial version 2.1, IBM, 2007.

[4] *Cell broadband engine programming handbook version 1.1*, IBM, 2007.

[5] M. Kistler, M. Perrone, and F. Petrini, “Cell multiprocessor communication network: built for speed,” published by the IEEE Computer Society, 2006.

[6] *SPE runtime management library version 2.1*, IBM, 2007.

[7] *Cell broadband engine SDK libraries overview and users guide version 2.1*, IBM, 2007.

[8] E. van der Tol, E. Jaspers, and R. Gelderblom, “Mapping of H.264 decoding on a multiprocessor architecture,” *Proceedings of SPIE*, volume 5022, 2003.

[9] Y. Pu, C. Long, and R. Jianhua, “Porting practices: compute-intensive applications,” IBM, 2007.

[10] J. Xu, W. Wolf, J. Henkel, and S. Chakradhar, “H.264 HDTV decoder using application-specific networks-on-chip,” *Multimedia and Expo*, 2005.

[11] F. Petrini, G. Fossum, J Fernandez, A.L. Varbanescu, M. Kistler, and M. Perrone, ”Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the CellBroadband Engine,” in Parallel and Distributed Processing Symposium, 2007

[12] S. Olivier, J. Prins, J. Derby, and K. Vu, ” Porting the GROMACS Molecular Dynamics Code to the Cell Processor,” in Parallel and Distributed Processing Symposium, 2007

[13] T. Chen, R. Raqhavan, J. Dale, and E. Iwata, ” Cell Broadband Engine Architecture and its first implementation: a performance view,” IBM, 2005

[14] J. Bartlett, ” Programming high-performance applications on the Cell BE processor, part 1: an introduction to Linux on the PLAYSTATION 3,” IBM, 2007

[15] S. K. Krewell, "Cell moves into the limelight," Microprocessor,2005

[16] D. Brokenshire, “Maximizing the power of the Cell Broadband Engine processor: 25 tips to optimal application performance,” IBM, 2006

[17] *C/C++ language extensions for Cell Broadband Engine architecture version 2.4*, IBM, 2007

[18] *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension technology programming environments manual version 2.07*, IBM, 2006

[19] *Cell Broadband Engine architecture version 1.01*, IBM, 2006

[20] D. Pham et al, “Overview of the Architecture, Circuit Design, and Physical Implementation of a First-Generation Cell Processor,” *IEEE Journal of solid-state circuits*, 2006

[21] M. Ohara et al, "MPI Microtask for programming the cell broadband engine processor,” *IBM Systems Journal Volume 45*, 2006

[22]M. Gschwind, "The Cell broad engine: exploiting multiple levels of parallelism in a chip multiprocessor," *IBM research report*, 2006

[23]Barry Wilkinson and Michael Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall, 1999

作者簡歷

陳慶至，1982 年 12 月 22 日出生於台北市。2005 年取得國立交通大學電子工程學系學士學位，並繼續在國立交通大學電子工程研究所攻讀碩士。2007 年在劉志尉教授指導下，取得碩士學位。本篇論文「在 PlayStation 3 上的即時多媒體處理」為其碩士論文。