© 2010 Deepti Kumar Chivukula

BUS SCHEDULING IMPLEMENTATION ON THE CELL PROCESSOR

BY

DEEPTI KUMAR CHIVUKULA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Adviser:

Associate Professor Marco Caccamo

ABSTRACT

Real-time computing is the study of hardware and software systems that are subject to a "real-time constraint" - i.e., strict deadline guarantees. With uncontrollable cache and front side bus, in the modern computer architectures, the estimation of a tight bound, worst case execution time (WCET) is difficult. The new generation computer architecture, Cell Broadband Engine Architecture (CBEA), has a software controlled front side bus (i.e. Element Interconnect Bus) that helps moderate the unpredictable task execution time problem. The CBEA is a heterogeneous chip system containing one Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs), each having an internal independent local storage memory.

In this thesis, using CBEA as a platform, I implemented an interrupt based scheduling framework that uses Element Interconnect Bus (EIB) in a temporally predictable manner. The framework is built by abstracting away low-level architectural features. Experiments were also performed to show that the real-time transactions of feasible transaction sets are executed before deadline when scheduled according to a real-time scheduling algorithm, while the same transactions can miss their deadlines when scheduled according to an arbitrary (non-real-time) scheduling policy.

To my husband Sujan, for his love, motivation and support

ACKNOWLEDGMENTS

First and foremost, I would like to thank my adviser, Dr. Marco Caccamo, who has been more than just an academic and research adviser over the last two years. He has been a mentor and a valuable guide, giving me complete freedom to choose a problem that I am really excited about, and thus made my research experience fantastic.

For their advice, inspiration, insights and discussions, I would like to thank other professors and fellow graduate students in the Beckman Institute and Siebel Center, including Dr. Narendra Ahuja, Dr. Lui R. Sha, Rodolfo Pellizzoni and Bach D. Bui.

I would also like to thank Shakti Kapoor, Senior Technical Staff Member at IBM Austin, for his timely support in terms of connecting me to the right people to talk.

Thanks are also due to my close friends: Manoj, Ramya, Sreekanth, Siva Kumar Sastry Hari, and many more for making my life at UIUC so enjoyable.

Finally, I would like to thank my parents for their love, encouragement and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A real-time system can be a hardware or a software system, whose application is considered to be *mission critical*. The total correctness of an operation, for such a system, depends not only on its logical correctness but also upon the time in which it is performed. In fact, the classical concept states that in a "hard" real-time system, the completion of an operation after its deadline is considered to be useless. Thus, real-time systems are used when there is a need for the deadlines of all the tasks to be guaranteed: to analyze the temporal behavior of the system, the worst case execution times (WCET) must be reliably estimated. Examples of such hard real-time systems include car engine systems, avionics systems, medical systems (like heart pacemakers) and industrial process controllers.

A significant source of randomness in estimating WCET lies in the *uncontrollability* of interconnection architectures, specifically both shared cache and front side bus (FSB). These architectures are used by CPU and direct memory access (DMA)-enabled peripherals to communicate amongst each other and with the main memory.

This problem, of randomness in estimating WCET, is potentially more severe in multiprocessor systems with a multitasking operating system (OS), as they have more entities which are concurrently competing for bus access. In such a system, execution time of a task can be unexpectedly extended by the execution of other tasks or DMA-enabled peripherals. The authors of [1] observed that extensions in execution time may be as high as 44%.

There have been significant research efforts on interconnection architectures for multi-core processors which have features suited for real-time systems, especially in the field of networks-on-chip (NOC). These works have been surveyed earlier in [2]. Commercial multi-core processors with software-controlled interconnections have also been developed, such as the IBM Cell Broadband Engine Architecture (CBEA) which is well distinguished for its high performance.

The main research issue is to how to provide software designers with: (1) a

practical and accurate abstraction of the real scheduling problem on multiprocessor bus and (2) an effective scheduling methodology that optimizes multiprocessor bus utilization. The present research focuses on addressing this problem on a specific multiprocessor bus architecture, specifically CBEA.

CBEA [3] is a new architecture that extends the 64-bit PowerPC Architecture. It is the result of collaboration between Sony, Toshiba and IBM, popularly known as STI, which was formally started in early 2001. It consists of nine processing elements, including an IBM 64-bit Power Architecture core called the Power Processing Element (PPE), augmented with eight co-processors called Synergistic Processor Elements (SPE) which contain private 256 kB of local storage (LS). All processing units and peripherals communicate with the main memory and other units through the Element Interconnect Bus (EIB) [4], which acts like an FSB. The EIB supports multiple transactions at the same time and its accesses are software controllable. It provides more control for moving or accessing memory data of the processing elements. Also, it provides more control over the cache usage, thus making this architecture intrinsically more predictable. Due to its hardware features, CBEA overcomes three important limitations of modern microprocessor performance - power use, memory use and processor frequency.

Support provided by the unique software features of CBEA also enhances control over EIB and the cache. Some of these features include dedicated instruction sets for each type of processor (PPE and SPEs), flexibility in developing applications in either C/C++/assembly language, good application programming interface (API) support called software development kit (SDK), and enhanced ability for parallelization (PPE and SPE Linux threads). These features are discussed in Chapter 3.

As mentioned earlier, our research group focused on developing an abstraction of the scheduling problem as well as an effective scheduling methodology that predictably schedule the multiprocessor bus. My key contribution to this effort, which is the focus of this thesis, was to build a framework that would allow the team to implement and test the performance of a class of scheduling algorithms for CBEA, while maintaining high throughput. The motivation for building the framework is that CBE's low level arbitration is complex and analyzing how requests compete for access to the bus is not easy. This framework would abstract away from the user the low level physical bus implementation. Further, it would allow table based scheduling through which we can eliminate all contentions for access to the bus and make it completely predictable. A key consideration while

building the framework was to ensure that it is simple to understand and be used by developers.

I have also conducted experiments to show that the real-time transactions of feasible transaction sets are executed before deadline when executed according to the real-time scheduling algorithm, while the same transactions can miss the deadlines if my scheduling framework is not used.

The implemented framework is interrupt based: it receives a timer interrupt at the beginning of each time slot (or quanta), and it makes a scheduling decision for the current slot based on a table that is generated offline by a real-time scheduling algorithm. The framework abstracts away from the low-level physical bus implementation, which includes the data and command arbiter. If a transaction is scheduled in a slot, a DMA packet, of the transaction of a given size, will be transferred from source to destination in that slot through EIB. The size of a DMA packet to be transferred is decided on the basis of bus bandwidth and slot size. The source and destination of DMA packets are SPE units on the chip. Each SPE receives data specific to the transactions it is involved in. The interrupt mechanism is implemented on SPEs, and interrupts are fired periodically depending on the timer value.

Many hardware and software hurdles (Chapters 2 and 3) had to be overcome to successfully run offline generated schedules on the implemented framework. This merits a more detailed discussion covering the unique challenges that were faced during the project.

In terms of hardware, design of EIB as well as its arbiter in CBEA is inherently complex. Thus, during the implementation of the framework I faced several challenges including incorporating multiple degrees of parallelism on the EIB, eliminating initial delay in the first phase of task execution, avoiding delays caused by the cell arbiter as well as incorporating alignment restrictions imposed by DMA. These are explained in more detail below.

The EIB comprises a 4-ring structure (2 clockwise and 2 anticlockwise directions) and has multiple degrees of parallelism: each bus ring can carry up to three concurrent transactions. The bus ring can start a new task after every three cycles, which causes an initial delay in the first phase of the task execution. Details about the different phases of a task on the EIB are covered in Chapter 2. Tasks on the bus can delay each other further if they share the same bus segment. Details of the types of contentions on EIB and how they affect the task execution time are described in Chapter 4.

Central arbiter is responsible for handling the individual data transactions and scheduling them so that they move around the ring and eventually end up on their respective destination units. The data arbiter implements round-robin bus arbitration with two levels of priority - memory interface controller (interface between the main memory and the EIB) has the highest priority and the rest of the units on the chip have a lower priority. The data flows on the EIB in a stepwise manner around the ring. The data arbiter does not allow a data transaction to be transferred along hops if its path is more than half the ring diameter. That is, a data ring is granted to a requester only so long as the circuit path requested is not more than six hops in either direction. An important design consideration was that if the request is greater than six hops, the requester must wait until a data ring operating in the opposite direction becomes free.

Another important hardware challenge was the alignment restrictions imposed by DMA. Non-aligned DMAs (not at 16-byte boundary) are not supported by CBEA. If non-aligned DMA operation is encountered, the MFC command queue processing is suspended and DMA alignment interrupt is generated. Resolving these exceptions takes up a lot of processor time which again causes delays in the execution of tasks on the bus. DMA accesses in the main storage domain are atomic (128-bit) if they meet the requirements of the PowerPC architecture. All other DMA transfers, if greater than a quadword (16-byte) or non-aligned, are performed as a set of smaller (1, 2, 4 and 8 bytes), disjointed atomic accesses. The number and alignment of these accesses are implementation dependent. Further details of the alignment issues have been illustrated in Chapter 3.

There were some hurdles during software implementation as well. My framework was implemented using a PPE-centric type of model called the parallel stage model, as discussed in Chapter 3. The parallel stage model executes different portions of data in parallel by dividing data among different processing elements (SPEs) - the PPE creates software threads (the types of software threads that run on PPE and SPEs are explained in Chapter 3) and puts them on SPEs which execute the data in parallel. Thus, the main issue was to locate exploitable concurrency in a task to successfully divide the data for parallel computation. Other challenges faced while trying to parallelize the data computation were data dependencies and overhead in synchronizing concurrent memory accesses or transferring data between different processor elements (SPEs).

All the above mentioned problems were resolved and I was also able to leverage the low level features of EIB and central arbiter to successfully implement the

described framework. More details on the implementation are provided in later chapters.

1.1 Thesis Organization

This thesis is divided into two main parts. Before describing the framework implementation, it is critical to have a good understanding of CBEA. Thus, in Part 1 of the thesis, I describe the architecture and various software features in detail. Chapter 2 provides an overview of the architecture of CBE. Chapter 3 covers the programming process on PPE and SPEs of the CBE.

Part 2 of the thesis deals with actual implementation of the framework and experimental results. Chapter 4 begins with description of the bus model and explanation of the terminology that will be used in Chapter 5. This chapter also provides a brief overview of a novel dynamic scheduling algorithm: it is important background material needed to understand the scope of the framework and my experimentation work. Chapter 5 explains the actual implementation of the proposed framework. Chapter 6 describes results of the experiments that were conducted, concluding remarks as well as details of future work.

CHAPTER 2

CELL BROADBAND ENGINE ARCHITECTURE

First generation Cell Broadband Engine (CellBE) is the first incarnation of a new family of microprocessors, called *Cell Broadband Engine Architeciture* (CBEA), that extends the 64-bit PowerPC Architecture. CBEA is the result of collaboration between Sony, Toshiba, and IBM, known as STI, formally started in early 2001.

CBEA, as shown in Figure 2.1, consists of 12 core components connected through the Element Interconnect Bus (EIB): One Power Processing Element (PPE) , eight Synergistic Processing Elements (SPEs), one Memory Interface Control Unit (MIC), and two I/O Interface Control Units (IOIF0 and IOIF1).

PPE is more adept at control-intensive tasks and is quicker at task-switching. SPEs are more adept at compute-intensive tasks and are slower at task switching. While either processor is capable of doing both types of tasks, this specialization has increased the efficiency in implementation of both PPE and, especially, SPEs. It is a significant factor in improving CBE's peak computational performance, area and power efficiency, by an order of magnitude (approximately) over conventional PC processors.

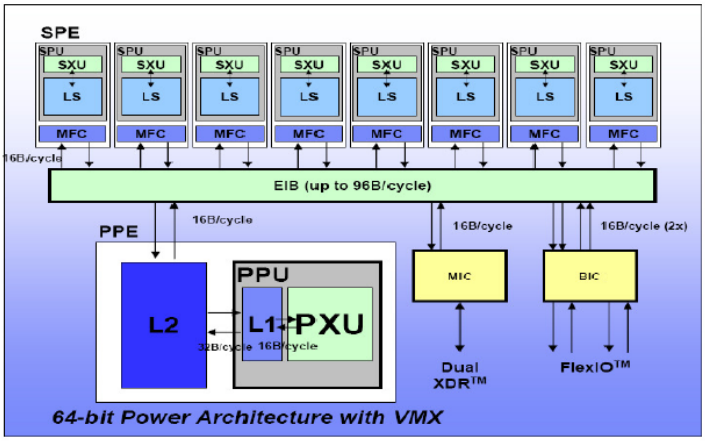

Figure 2.1: The Cell Broadband Engine

This chapter is organized in three sections. Sections 2.1 talks about the limita-

tions of modern microprocessors and how CBEA overcame these hurdles. Section 2.2 talks about the various hardware features of CBEA including Power Processor Element (PPE), Synergistic Processor Elements (SPEs), Memory Interface Controller (MIC), Element Interconnect Bus (EIB) and Cell Broadband Engine Interface (BEI).

2.1 The Three Walls

As mentioned earlier, CBE overcomes three important limitations of modern microprocessor performance: power use, memory use and processor frequency. Following is a detailed description of how CBEA overcomes these hurdles.

2.1.1 Power Wall

Microprocessor performance is often limited by achievable power dissipation, rather than by number of available integrated circuit resources (transistors and wires). Thus, power efficiency has to be improved to significantly increase microprocessor performance. One way to increase power efficiency is to differentiate between processors optimized to run an operating system and control-intensive code, and processors optimized to run compute-intensive applications.

CBE achieves this by providing a general purpose PPE to run the operating system and other control-plane code, and eight SPEs specialized for computing data-rich (data-plane) applications.

2.1.2 Memory Wall

Latency to DRAM memory for multi-gigahertz symmetric multiprocessors, even those with integrated memory controllers, is currently approaching 1,000 cycles. As a result, program performance is dominated by the activity of moving data between main storage (effective-address space that includes main memory) and the processor. Compilers and even application writers must increasingly manage this movement of data explicitly, even though hardware cache mechanisms are supposed to relieve them of this task.

CBE's SPEs use two mechanisms to deal with long main-memory latencies: (a) 3-level memory structure (main storage, local storages in each SPE, and large

register files in each SPE) and (b) asynchronous DMA transfers between main and local storage. These features allow programmers to schedule simultaneous data and code transfers to cover long latencies effectively. As a result, CBE can usually support 128 simultaneous transfers between the eight SPE local storage and main storage. This surpasses the number of simultaneous transfers on conventional processors by a factor of almost 20.

2.1.3 Frequency Wall

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

CBEA, on which the CBE is based, allows both PPE and SPEs to be designed for high frequency without excessive overhead. PPE achieves efficiency primarily by executing two threads simultaneously, rather than by optimizing single-thread performance. Each SPE achieves efficiency by using a large register file, which supports many simultaneous in-flight instructions without the overhead of register-renaming or out-of-order processing. Each SPE also achieves efficiency by using asynchronous DMA transfers, which support many concurrent memory operations without overhead of speculation.

2.1.4 Cell's Solution

CBE takes care of problems posed by power, memory and frequency limitations, by individually optimizing control-plane and data-plane processors. As a result a processor, with the power budget of a conventional PC processor, can theoretically be expected to provide an approximately ten-fold improvement in peak performance compared to a conventional processor.

Of course, actual application performance varies. Some applications may benefit little from SPEs, while others show a performance increase well in excess of ten-fold. In general, compute-intensive applications that use 32-bit or smaller data formats (such as single-precision floating-point and integer) are excellent candidates for the CBE.

2.2 Architectural Overview of CBEA

In the following sections, we provide architectural features and the hardware environment of the CBEA.

2.2.1 PowerPC Processor Element (PPE)

PPE is the main processor that contains a 64-bit PowerPC Architecture reduced instruction set computer (RISC) core with a traditional virtual memory subsystem. It consists of two main units, the Power Processor Unit (PPU) and the Power Processor Storage Subsystem (PPSS).

PPE performs multiple functions, like running the operating system, managing system resources, as well as control processing, including the allocation and management of SPE threads. It can run legacy PowerPC Architecture software and performs well executing system-control code. It supports both the PowerPC instruction set and the Vector/SIMD Multimedia Extension instruction set.

2.2.2 Synergistic Processor Elements (SPEs)

CBE includes eight SPEs that are SIMD processors optimized for data-rich operations which are allocated to them by the PPE. Each of these identical elements contains a RISC core, 256 kB, software-controlled local storage (LS) for instructions and data, and a large (128-bit, 128-entry) unified register file. It consists of two main units (shown in Figure 2.2), the Synergistic Processor Unit (SPU) and the Memory Flow Controller (MFC). Also, each SPE has full access to coherent shared memory, including the memory-mapped I/O space.

SPEs support a special SIMD instruction set, and they rely on asynchronous DMA transfers to move data and instructions between main storage (the effective-address space that includes main memory) and their local storages (LSs). SPE DMA transfers access main storage using PowerPC effective addresses. As on the PPE, address translation is governed by PowerPC Architecture segment and page tables. An SPE is a 128-bit processing unit that has a Memory Flow Control (MFC) unit which controls the memory management unit and the DMA engine. The method considered in this work for moving data between LSs, and between system memory and LSs, is DMA. A software designer operates the DMA engines by issuing commands to its MFC. The command must specify the starting and the

ending addresses of the transaction and its size. Each SPE contains a bus interface unit (BIU) that provides an interface from the SPE to EIB.

SPE is optimized for running compute-intensive applications, and not for running the operating systems. The name *synergistic* for this processor was chosen carefully because there is a mutual dependence between PPE and the SPEs. The latter depend on the former to run the operating system and, in many cases, the top-level control thread of an application. On the other hand, PPE depends on SPEs to provide the bulk of the application performance.

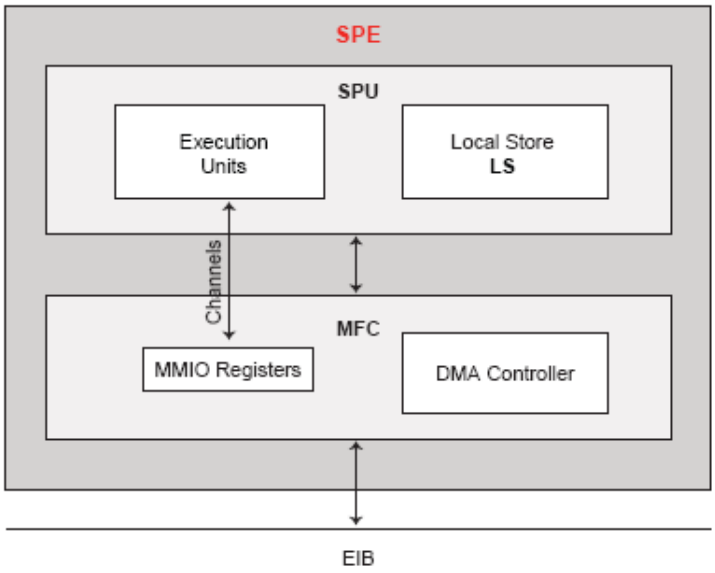

Figure 2.2: SPE Architectural Block Diagram

2.2.3 Memory Flow Controller

As shown in Figure 2.3, each SPU has its own Memory Flow Controller (MFC) that serves as the SPU's interface to main storage, other processor elements and system devices. MFC's primary role is to interface its LS storage domain with the main-storage domain. It does this by means of a DMA controller that moves instructions and data between its LS and main storage. MFC also supports other functions, including storage protection on the main storage side of its DMA trans-

fers, synchronization between main storage and the LS, as well as the communication functions (such as mailbox and signal-notification messaging) with PPE, other SPEs and devices.

In my thesis I will focus on only DMA transfers and mailboxes since I used these two mechanisms in implementing my framework.

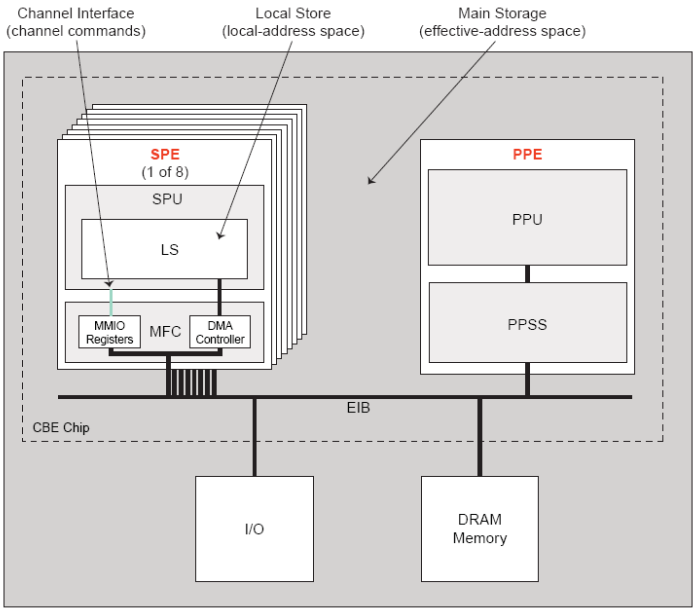

Figure 2.3: Memory Flow Controller

2.2.4 Memory Interface Controller (MIC)

MIC provides the interface between Element Interconnect Bus (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.

2.2.5 Element Interconnect Bus (EIB)

PPE and SPEs communicate coherently with each other, with main storage as well as with I/O through the EIB (see Figure 2.4). The bus includes a 16-byte wide 4-ring structure (two clockwise and two counterclockwise) for data transfer, a data and command arbiter, and a tree structure for commands.

Each participant on the EIB has one 16-byte read port and one 16-byte write port. The bus has multiple degrees of parallelism: each bus ring can carry up to three concurrent transactions. Data flows on an EIB channel stepwise around the ring. Since there are 12 participants or units, the total number of steps around the channel back to the point of origin is 12. Six steps is the longest distance between any pair of participants. An EIB channel is not permitted to convey data requiring more than six steps; such data must take the shorter route around the circle in the other direction. The number of steps involved in sending the packet has very little impact on transfer latency: clock speed driving the steps is very fast relative to other considerations. However, longer communication distances are detrimental to the overall performance of the EIB as they reduce available concurrency. 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.

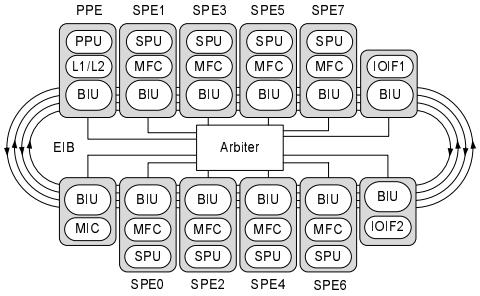

Figure 2.4: CBEA: Element Interconnect Bus

The transfer of a transaction on the bus takes place in four steps or stages - *send phase*, *command phase*, *data phase* and *receive phase*. Each of these steps is detailed in the section below.

2.2.5.1 Sending Phase

Sending Phase, shown in Figure 2.5, is responsible for initiating a transaction. It consists of all processor and DMA controller activities needed before transactions are injected into any components.

At the end of this phase, a command is issued to the command bus to begin the next phase.

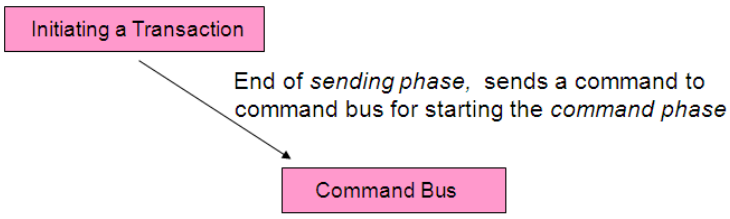

Figure 2.5: Sending Phase

2.2.5.2 Command Phase

Command Phase shown in Figure 2.6 coordinates end-to-end transfers across the EIB. This phase is also responsible for coherency checking, synchronization, and inter-element communication. EIB informs the read or write target element of the transaction in progress to allow the target to set up the transaction (i.e., data fetch or buffer reservation).

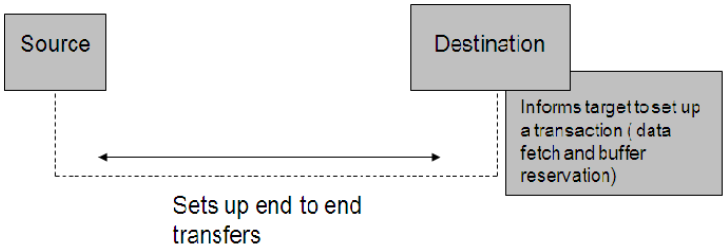

Figure 2.6: Command Phase

2.2.5.3 Arbitration Phase

A low-level round robin scheduler arbitrates bus accesses between contending transactions stored in the post-command phase queue (arbitration phase together with data phase is called post-command phase). This phase occurs when the BIU has to wait for bus access due to the contention between atomic transactions on the bus.

2.2.5.4 Data Phase

As shown in Figure 2.7 *Data Phase* handles data ring arbitration and actual data transfers across the ring. This phase grants access to packets through one of the four data rings when a ring becomes free and ensures that no more than six

hops are needed along the ring. End-to-end transport of packets happens over a pipelined circuit-switched granted EIB Ring.

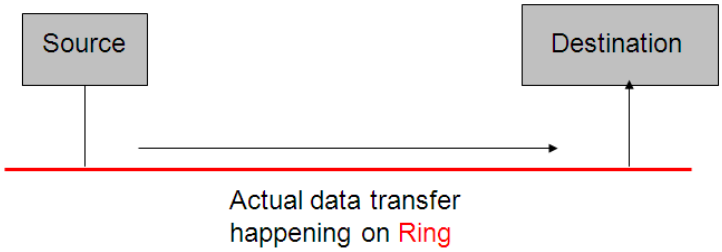

Figure 2.7: Data Phase

2.2.5.5 Receiving Phase

Receiving Phase shown in Figure 2.8 concludes the transaction by transferring received data from the receiving node's BIU to its final destination at that receiver, such as local storage memory or the system memory.

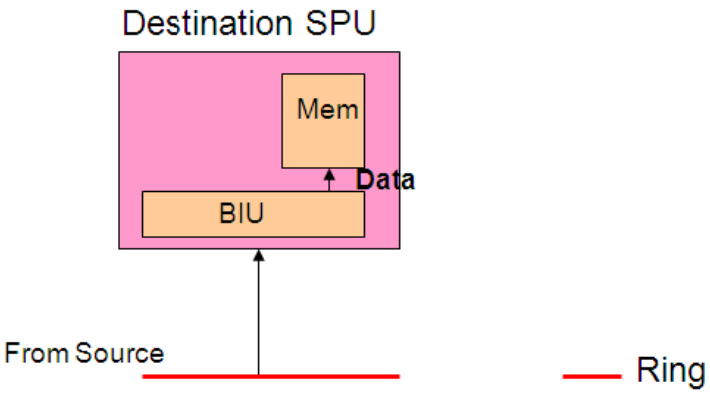

Figure 2.8: Receiving Phase

2.2.6 Cell Broadband Engine Interface (BEI)

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 CBE.

2.3 Chapter Conclusion

In this chapter, I gave a brief overview of the CBE architecture, as well as the functions of each of its hardware components. As discussed earlier, the unique architecture of the CBE allows it to overcome serious limitations of modern microprocessors and deliver approximately ten-fold improvement in performance. In the next chapter I will describe unique and important architecture details, knowledge of which is required for programming on CBEA.

CHAPTER 3

PROGRAMMING ENVIRONMENT ON CBEA

Support provided by the unique software features of CBEA enhances our control over the interconnection architecture. Thus, it is important to understand the programming environment of CBEA to know the complexity involved in abstracting from the low level bus architectural constraints. This chapter gives an overview of programming on the nine processor elements of CBEA, i.e. PPE and eight SPEs. As already discussed in Chapter 2, PPE on the CBEA is optimized to run the operating system and any control-intensive code, while the SPEs are optimized to run compute-intensive applications. Running an application on these processors requires an understanding of how applications partition tasks among different processors as well as how these tasks are actually executed on them. For further reference a detailed description of the programming process is given in [5] and [6].

This chapter is organized in four sections. Partitioning of applications is described in Section 3.1. The next section deals with creation of threads; these are used to send processing requests to different processors. Section 3 describes instruction sets that are used to process these tasks. Finally in the last section, an overview of how SPEs and PPEs interact is provided.

3.1 Partitioning of the Applications

All applications running on the Cell Broadband Engine need to divide the work among the nine processors. The following considerations have to be taken into account while deciding on how to distribute the workload and data.

- Program structure: Any application will usually have both compute intensive tasks as well as control intensive tasks. In CBEA, the two types of tasks are assigned to different types of processors. Thus, the choice of a task partitioning model, i.e. PPE-centric or SPE-centric as shown in

Figure 3.1, will depend on whether the application is more compute-intensive or more control intensive. The model choice ensures that the application is run optimally on the processors.

- Program data access and data flow patterns: After deciding the type of partitioning, the developer will determine if data is going to be sent in a parallel fashion or serially to SPEs, depending on the processor's coded functionality.

- Optimizing cost: Context switching can cause the processing time to increase substantially as SPEs have to be stopped while local storages are reloaded. Thus, the developer needs to be careful to write code to minimize context switching.

Our implementation used a PPE-centric model, which is described below. Details about the justification for using this model are given in Chapter 5.

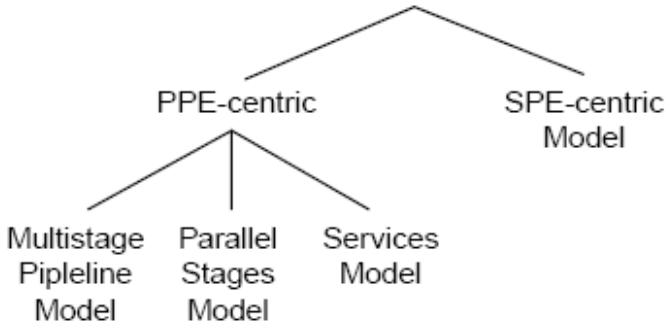

Figure 3.1: Application Partitioning Models

3.1.1 PPE-Centric Model

In this model, the main application runs on the PPE, and individual tasks are offloaded to the SPEs. PPE then waits for, and coordinates, the results returning from the SPEs. This model fits an application with serial as well as with parallel data computation.

The PPE-centric model can be classified into three sub-models depending on how SPEs are used:

- Multistage pipeline model (shown on left of Figure 3.2): It is used when a task requires sequential stages, where the SPEs can act as a multi-stage

pipeline. Each SPE is responsible for one stage of the process. This model is not suitable for parallel processing owing to difficulty in load balancing. Additionally, this model increases the costs due to greater data movement.

- Parallel stage model (shown on right of Figure 3.2): Suitable when tasks involve a large amount of data that can be partitioned and executed in parallel. Thus, each SPE is used to execute different portions of data simultaneously.

- Services model (shown in Figure 3.3): Here, PPE assigns different services to different SPEs, and the PPE's main process calls upon the appropriate SPE when a particular service is needed. For example one SPE processes data encryption, another SPE processes MPEG encoding, and a third SPE processes curve analysis. Fixed static allocation of SPE services should be avoided. These services should be virtualized and managed on a demand-initiated basis.

We used the parallel stage model in our framework. More details for our choice of sub-model are presented in Chapter 5.

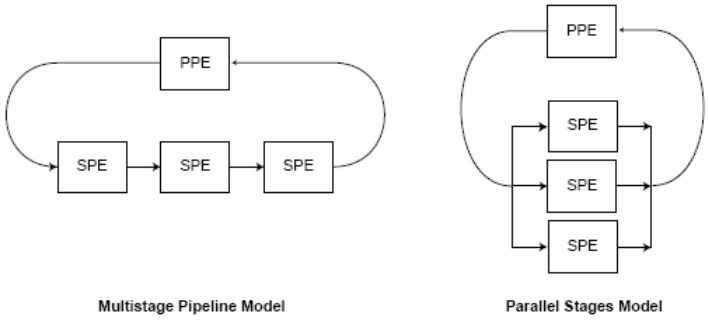

Figure 3.2: PPE-Centric Models: Multistage Pipeline Model and Parallel Stages Model

3.2 Threads on PPE and SPE

A software developer also needs to understand how applications are executed on CBEA. This will give the developer more control and he/she will be able to effectively abstract the low level design.

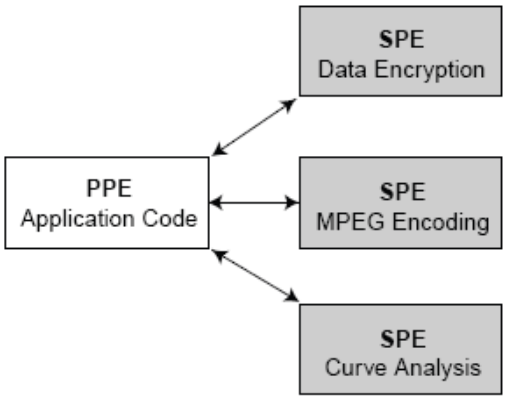

Figure 3.3: PPE-Centric Service Model

The architecture can support several types of operating systems. Our research group used the Linux operating system (Fedora Core 7), which was running on the Sony play station (PS)3.

Every program is a process to the operating system. A process is a task that competes for execution time on the microprocessor, and has resources. Linux allows a program to execute multiple threads of execution. One program can create independent threads, which share the resources of the parent process, but execute independently of each other, and the parent process. A thread running in the Linux OS environment is referred to as Linux thread.

In CBEA, the main thread of a program is a Linux thread running on the PPE. Any Linux thread running on a PPE is called a PPE thread. The main thread can spawn one or more CBE Linux tasks. A CBE Linux task has one or more Linux threads associated with it that may execute on either a PPE or an SPE. All Linux threads within the task share the tasks resources.

The operating system defines the mechanism and policy for scheduling an available SPE. It must prioritize among all the Cell Broadband Engine Linux applications in the system, and it must schedule SPE execution independent from regular Linux threads. It is also responsible for runtime loading, passing parameters to SPE programs, notification of SPE events and errors, and debugger support.

A thread running on SPE is called an SPE thread. It has its own SPE context which includes the 128x128-bit register file, program counter, and MFC command queues. Further, it can communicate with other execution units or with effective address memory through the MFC channel interface. All architectures create SPE threads; however, SPE architecture of CBEA is new and so it needs to create SPE

threads in a specific way. It is thus important to understand how to load, run and destroy threads on SPEs, which is described in the next section.

Additionally, programming on the CBE processor requires an understanding of parallel programming as it is a multi-core system. Section 3.2.2 reviews different styles of parallel programming on the CBE processor.

3.2.1 SPE Thread Creation

Programs to be run on an SPE are often written in C or C++ (or assembly language) and can use the SPE data types and intrinsics defined in the SPU C/C++ Language Extensions. A PPE module starts running an SPE module by first creating a thread on the SPE. It creates SPE threads using the SPE context create, program load, context run, pthread join and context destroy library calls, all of which are provided in the SPE runtime management library except pthread join.

3.2.1.1 Creating Threads

```
spe_context_ptr_t spe_context_create(unsigned int
        flags, spe_gang_context_ptr_t gang)
```

- flags - This is a bit-wise OR of modifiers that is applied when the new context is created. A number of values are accepted for this parameter out of which we use *SPE_MAP_PS* - Request permission for memory-mapped access to the SPE threads problem state area.
- gang - It is a collection of contexts in which the context being created should be made a part of.

Loading the SPEs

```
int spe_program_load(spe_context_ptr spe,
            spe_program_handle_t *program)
```

- spe - The SPE context in which the specified program is to be loaded.
- program - Indicates the program to be loaded into the SPE context.

Running the SPEs

```
int spe_context_run(spe_context_ptr_t spe,
 unsigned int *entry, unsigned int runflags,
 void *argp, void *envp,
 spe_stop_info_t *stopinfo)
```

- spe - The SPE context to be run.
- entry - Initial value of the instruction pointer in which the SPE program should start executing. Upon return from the spe_context_run call, the value pointed to by entry contains the next instruction to be executed upon resumption of the program.
- runflags - This is a bit-wise OR of modifiers which request specific behavior when the SPE context is run. Flags include: - 0 - Default behavior. No modifiers are applied.
- argp - An optional pointer to application specific data. It is passed as the second parameter of the SPU program.
- envp - An optional pointer to environment specific data. It is passed as the third parameter of the SPU program.
- stopinfo - An optional pointer to a structure of type spe_stop_info_t that provides information as to the reason why the SPE stopped running.

Destroy the SPE threads

```
spe_context_destroy (spe_context_ptr_t spe)
```

Destroys the context for the SPE context *spe*. The SPE threads that were initially created are destroyed.

3.2.2 Parallel Programming

For efficient computation it is important to understand how tasks can be executed in parallel, on the eight SPEs. The key to parallel programming is to locate exploitable concurrency in a task. The basic steps for parallelizing any program are:

- Locate concurrency: Time spent analyzing the program, its algorithms and data structures will be repaid many-fold in the implementation and coding phase. The most important question that we need to ask ourselves is - Will the anticipated speedup from parallelizing a program be greater than the effort to parallelize a program, which includes any overhead for synchronizing different tasks or access to shared data? Another question we can ask ourselves is - Which parts of the program are the most computationally intensive? It is worthwhile to do initial performance analysis on typical data sets, to be sure the hot spots in the program are being targeted. When you know which parts of the program can benefit from parallelization, you can consider different patterns for breaking down the problem. Key elements to examine are:
  - Function calls
  - Loops
  - Large data structures that could be operated on in chunks
- Structure the algorithm(s) to exploit concurrency: Ideally, you can identify ways to parallelize the computationally intensive parts:
  - Break down the program into tasks that can execute in parallel.
  - Identify data that is local to each subtask.
  - Group subtasks so that they execute in the correct order.
  - Analyze dependencies among tasks.

The major challenges faced during parallelization of the programs are:

- Data dependencies exist.
- Overhead in synchronizing concurrent memory accesses or transferring data between different processor elements and memory might exceed any performance improvement.
- Partitioning work is often not obvious and can result in unequal units of work.
- What works in one parallel environment might not work in another, due to differences in bandwidth, topology, hardware synchronization primitives, and so forth.

All levels of parallelism are already available with the CBE. These features can be used to our advantage to mitigate some challenges posed by software parallelization. The CBE processor provides a foundation for many levels of parallelization. Starting from the lowest, fine-grained parallelization SIMD processing up to the highest, course-grained parallelization networked multiprocessing with the CBE processor provides many opportunities for concurrent computation. The levels of parallelization include:

- Dual-issue superscalar microarchitecture
- Multithreading
- Multiple execution units with heterogeneous architectures and differing capabilities
- Shared-memory multiprocessing
- SIMD processing

3.2.3 Instruction Sets

As mentioned earlier, in CBEA the PPE is focused on control-intensive tasks and the SPEs are focused on computation-intensive tasks. The two types of processors have different instruction sets due to their different functionality.

Based on our previous discussion in Chapter 2 the instruction set for the PPE is an extended version of the PowerPC instruction set. The extensions consist of the Vector/SIMD Multimedia Extension instruction set plus a few additions and changes to PowerPC instructions. The instruction set for the SPEs is a new SIMD instruction set, the Synergistic Processor Unit Instruction Set Architecture, with accompanying C/C++ intrinsics. It also has a unique set of commands for managing DMA transfer (gets and puts - discussed in later sections of this chapter), external events, interprocessor messaging (mailboxes), and other functions.

Although the PPE and the SPEs execute SIMD instructions, their instruction sets are different, and programs for the PPE and SPEs must be compiled using different compilers. These compilers generate code streams for two completely different instruction sets.

To conclude, even though a high-level language such as C or C++ code can be used for the CBE processor, an understanding of the PPE and SPE machine in-

structions adds considerably to a software developer's ability to produce efficient, optimized code.

3.3 Communication between PPE and SPE

Before understanding how processors communicate with each other, it is important to know about the types of storage domains the processors use to move data. In CBE there are three types of storage domains - one main-storage domain, eight SPE local storage domains, and eight SPE channel domains, as shown in Figure 3.4. The main-storage domain, which is the entire effective-address space, can be configured by the PPE operating system to be shared by all processors and memory-mapped devices in the system (all I/O is memory-mapped). However, the local-storage and channel problem-state (user-state) domains are private to the SPU, LS, and MFC of each SPE.

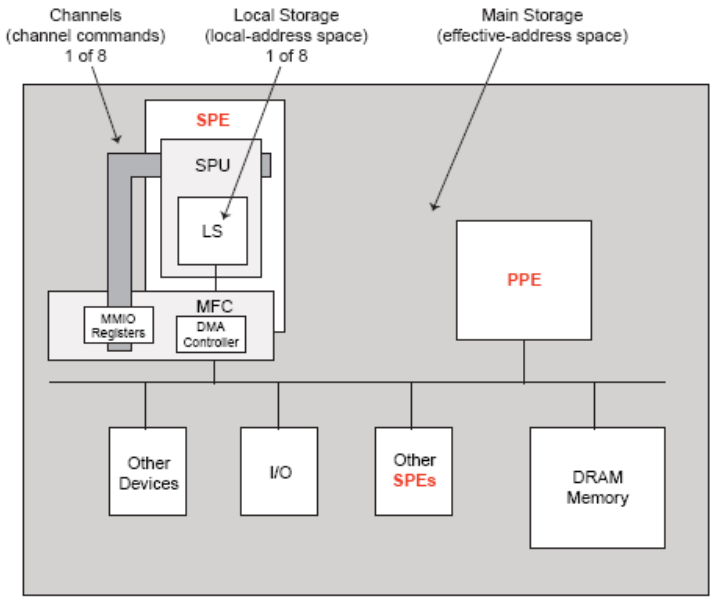

Figure 3.4: The Cell Storage Domains

Finally, using the defined communication mechanism implemented in hardware between the processors, we move data to enable interaction between them.

The three primary communication mechanisms between the PPE and SPEs are mailboxes, signal notification registers, and DMAs. This thesis focuses on understanding only mailboxes and DMA transfers, because only these two mechanisms

have been used in implementing the framework. They are explained in the next two sub-sections.

3.3.1 Mailboxes

Mailboxes are the best way to send fast dedicated 32-bit messages between processors. Thus, whenever the processors need to communicate some information that is less than or equal to 32-bits, mailboxes are our best bet; for example, in the case of our framework, addresses of certain structures and variables, sending acknowledgements on receiving specific data, etc., are communicated using the mailboxes. Each SPE has three mailboxes, for sending, receiving and buffering 32-bit messages from the SPE to the PPE. Two mailboxes (the *SPU Write Outbound Mailbox* and the *SPU Write Outbound Interrupt Mailbox*) are provided for sending messages from the SPE to the PPE. One mailbox (the *SPU Read Inbound Mailbox*) is provided for sending messages from PPE to the SPE.

PPE is often used as an application controller, managing and distributing work to the SPEs. A large part of this task may involve loading main storage with data to be processed, then notifying an SPE by means of a mailbox. The SPE can also use its outbound mailboxes to inform the PPE that it has finished with a task. An SPE sends a mailbox message by writing the 32-bit message value to either of its two outbound mailbox channels. The PPE can read a message in an outbound mailbox by reading the MMIO register in the SPE's MFC that is associated with the mailbox channel. Likewise, the PPE sends messages to the SPE's inbound mailbox by writing the associated MMIO register.

Following is a detailed explanation of how *SPU* Inbound and Outbound Mailboxes work along with examples of how to use the functions. This is essential because: (1) it enables us to follow the low level code of framework, and (2) knowing the mechanism of how mailboxes work may also help developers to implement their own mailbox messages.

3.3.1.1 *SPU* Outbound Mailboxes

The MFC provides two one-entry mailbox channels - the *SPU* Write Outbound Mailbox and the *SPU* Write Outbound Interrupt Mailbox - for sending messages from the SPE to the PPE. We use the write outbound mailbox channel in our

implementation for communication. Further details about the channel as well as the functions that are used at the PPE and SPE side are described below.

***SPU* Write Outbound Mailbox Channel:** SPE software writes to the *SPU* Write Outbound Mailbox channel to put a mailbox message in the *SPU* Write Outbound Mailbox. This write-channel instruction will return immediately if there is sufficient space in the *SPU* Write Outbound Mailbox Queue to hold the message value. If there is insufficient space, the write-channel instruction will stall the SPU until the PPE reads from this mailbox.

***SPE* Side:** On the SPE side the function *spu_write_out_mbox()* writes to the *SPU* Write Outbound Mailbox and stalls until space is available. Once the *mbox_data* is successfully written to the *SPU* Write Outbound Mailbox on the SPE side, the next step is for PPE to read the value. The function used by SPE to send the data is given below:

```
unsigned int mbox_data;
spu_write_out_mbox(mbox_data);
```

***PPE* Side:** Before PPE software can read data from one of the *SPU* Write Outbound Mailboxes, it must first read the Mailbox Status Register to determine that unread data is present in the *SPU* Write Outbound Mailbox; otherwise, stale or undefined data may be returned. To determine that unread data is available in the SPU Write Outbound Mailbox, PPE software reads the Mailbox Status Register and extracts the count value from the SPU Out Mbox Count field. If the count is non-zero, then at least one unread value is present. If the count is '0', PPE software should not read the *SPU* Write Outbound Mailbox Register because it will get incorrect data and should poll the Mailbox Status Register. The function used to read the Mailbox Status Register is int_spe_out_mbox_status (spe_context_ptr_t spe) where the parameter spe specifies the SPE context for which the *SPU* Outbound Mailbox has to be read. PPE polls to read the SPU channel.

The function used by PPE to read *SPU* Write Outbound Mailbox Channel is:

```
#include <libspe2.h> - \\library containes app
                       \\programs to access the SPEs
int spe_out_mbox_read (spe_context_ptr_t spe,
              unsigned int *mbox_data, int count)
```

The description of parameters of the function is as follows:

- spe - Specifies the SPE context for which the *SPU* Outbound Mailbox has to be read.
- mbox_data - A pointer to an array of unsigned integers of size count to receive the 32-bit mailbox messages read by the call.
- count - The maximum number of mailbox entries to be read by this call.

This function returns an integer value - 0, >1 or <1.

- >0 - the number of 32-bit mailbox messages read
- 0 - No data to be read
- -1 - error condition and errno is set appropriately

3.3.1.2 *SPU* Inbound Mailboxes

The MFC provides one mailbox for a PPE to send information to an SPU: the *SPU* Read Inbound Mailbox. This mailbox has four entries; that is, the PPE can have up to four 32-bit messages pending at a time in the *SPU* Read Inbound Mailbox. More details about the channel and the functions used at the SPE and PPE side are given in the following subsections.

***SPU* Read Inbound Mailbox Channel:** *SPU* Read Inbound Mailbox Channel: If the *SPU* Read Inbound Mailbox channel has a message, the value read from the mailbox is the oldest message written to the mailbox. If the inbound mailbox is empty, the *SPU* Read Inbound Mailbox channel count will read as '0'.

***PPE* Side:** This function writes up to count messages to the SPE inbound mailbox for the SPE context *spe*. This call may be blocking or non-blocking, depending on behavior. The blocking version of this call is particularly useful to send a sequence of mailbox messages to an SPE program without further need for synchronization. The non-blocking version may be advantageous when using SPE events for synchronization in a multi-threaded application. *spe_in_mbox_status* can be called to ensure that data can be written prior to writing the *SPU* inbound mailbox. *spe_in_mbox_status(spe_context_ptr_t spe)* function fetches the status of

the *SPU* Inbound Mailbox for the SPE context specified by the *spe* parameter. A *0* value is returned if the mailbox is full. A non-zero value specifies the number of available (32-bit) mailbox entries. The function to write to the *SPU* Read Inbound Mailbox is given below:

```
#include <libspe2.h>
int spe_in_mbox_write (spe_context_ptr_t spe,
    unsigned int *mbox_data, int count,
    unsigned int behavior)
```

- spe - Specifies the SPE context of the *SPU* Inbound Mailbox to be written.
- mbox_data - A pointer to an array of count unsigned integers containing the 32-bit mailbox messages to be written by the call.
- count - The maximum number of mailbox entries to be written by this call.
- behavior - Specifies whether the call should block until mailbox messages are written.

There are four possible values for behavior, given in Table 3.3.1.2

Table 3.1: Values of Behavior

Value	Description
SPE_MBOX_ALL_BLOCKING	The call blocks until all count mailbox messages have been written
SPE_MBOX_ANY_BLOCKING	The call blocks until at least one mailbox message has been written
SPE_MBOX_ANY_NONBLOCKING	The call writes as many mailbox messages as possible up to a maximum of count without blocking

After the PPE writes the mbox_value, the SPE has to read the same from the *SPU* Read Inbound Mailbox.

***SPE* Side:** SPE software can use a read-channel funcion on the *SPU* Read Inbound Mailbox channel to read the contents of its *SPU* Read Inbound Mailbox. This channel read will return immediately if any data written by the PPE is waiting in the *SPU* Read Inbound Mailbox. This read-channel function will cause the SPU to stall if the *SPU* Read Inbound Mailbox is empty.

```
unsigned int mbox_data;
mbox_data  = spu_read_in_mbox();
```

Although the mailboxes are primarily intended for communication between PPE and SPEs, they can also be used for communication between an SPE and other SPEs, processors, or devices.

3.3.2 Direct Memory Access

When the PPEs and SPEs have to transfer larger amounts of data or instructions they use DMAs. Architecturally the DMAs are implemented to support the transfer of large amounts (maximum 16 kB) of data or instructions between the processors at a single instant. As our framework needs to support data transfers of size 10 kB, knowing to how to use and implement DMAs properly is essential.

MFC's DMA Controller (DMAC) implements DMA transfers of instructions and data between the SPUs LS and main storage. Programs running on the associated SPU, or the PPE, can issue the DMA commands. The MFC executes DMA commands autonomously, which allows the SPU to continue execution in parallel with the DMA transfers. Each DMAC can initiate up to 16 independent DMA transfers to or from its LS.

3.3.2.1 DMA Transfers

To initiate a DMA transfer, software on an SPE uses a channel instruction to write the transfer parameters to the MFC command queue channels. An SPE can only fetch instructions from its own LS. An SPE or PPE performs data transfers between the SPE's LS and main storage primarily using DMA transfers controlled by the MFC DMA controller for that SPE. Software on the SPE's SPU interacts with the MFC through channels, which enqueue DMA commands (length of the queue being 16) and provide other facilities, such as mailboxes and signal notification. An SPE program accesses its own LS using a local storage address (LSA). The LS of each SPE is also assigned a real address (RA) range within the systems memory map. This allows privileged software to map LS areas into the effective address (EA) space, where the PPE, SPEs, and other devices that generate EAs can access the LS. Each SPE's MFC serves as a data-transfer engine. DMA trans-

fer requests contain both an LSA and an EA. Thus, they can address both a SPE's LS and main storage and thereby initiate DMA transfers between the domains.

The MFC accomplishes this by maintaining and processing an MFC command queue. The queued requests are converted into DMA transfers. Each MFC can maintain and process multiple in-progress DMA command requests and DMA transfers. The MFC can also autonomously manage a sequence of DMA transfers in response to a DMA-list command from its associated SPU. Each DMA command is tagged with a 5-bit Tag Group ID. Software can use this identifier to check or wait on the completion of all queued commands in one or more tag group.

DMA commands: The majority of MFC commands initiate DMA transfers; these are called DMA commands. The basic DMA commands are the *get* and *put*. Since the LSs of the SPEs and the I/O subsystems are typically mapped into the effective address space, DMA commands can transfer data between the LS and these areas as well. Regardless of the initiator (SPU, PPE, or other device), DMA transfers up to 16 kB of data between LSs and main memory or between LSs. An MFC supports naturally aligned DMA transfer sizes of 1, 2, 4, 8, and 16-bytes and multiples of 16-bytes. The performance of a DMA transfer can be improved when the source and destination addresses have the same quadword offsets within a 128-byte cache line.

- *put* (put*[s]*) command transfers the number of bytes specified by the transfer size parameter from the local storage address of the corresponding SPU to the effective address (EA).

  ```
  (void) mfc_put(volatile void *ls, uint64_t ea,
  uint32_t size,  uint32_t tag, uint32_t tid,
  uint32_t rid)
  ```

  The arguments to this function correspond to the arguments of the spu_mfcdma64 command: *ls* is the local-storage address, *ea* is the effective address in system memory, *size* is the DMA transfer size (maximum is 16 kB), *tag* is the DMA tag, *tid* is the transfer class identifier, and *rid* is the replacement class identifier.

- *get* command transfers the number of bytes specified by the transfer size parameter from the effective address to the local storage address of the corresponding SPU.

  ```
  (void) mfc_get(volatile void *ls, uint64_t ea,
   uint32_t size,  uint32_t tag, uint32_t tid,
  uint32_t rid)
  ```

  The arguments to this function correspond to the arguments of the spu_mfcdma64 command: *ls* is the local-storage address, *ea* is the effective address in system memory, *size* is the DMA transfer size (maximum is 16 kB), *tag* is the DMA tag, *tid* is the transfer class identifier, and *rid* is the replacement class identifier, same as above.

When MFC commands are entered into the command queue, each command in the queue is tagged with a 5-bit tag-group identifier, called the *MFC command tag identifier*. The identification tag can be any value between 0 and 31. The same identifier can be used for multiple MFC commands to create a tag group containing all the commands currently in the queue with the same command tag. Software can use the MFC command tag to check the completion of all queued commands in a tag group. In our implementation the transfer class identifier and replacement class identifier are both 0 and more information on them is not required.

After MFC commands are entered into the command queue, we might want to know the status of these MFC commands. There are certain MFC DMA command functions that can be used to check the completion of MFC commands or the status of entries in the MFC DMA queue. The function used to determine the status of the MFC DMA command is *mfc_read_tag_status*. Each bit of a returned value indicates the status of each tag group. If set, the tag group has no outstanding operation (that is, commands completed) and is not masked by the query.

In our implementation, out of the many provided MFC DMA status functions, we use *mfc_write_tag_mask* and *mfc_read_tag_status_all()*. We explain these two functions in more detail below.

- *mfc_write_tag_mask* - A tag mask is set to select the MFC tag groups to be included in the query operation, where the parameter *mask* in the function is the DMA tag group query mask. Each bit of the mask indicates the tag group. Implementation:

```
(void) mfc_write_tag_mask (uint32_t mask)
```

For example in an *mfc_get()* command suppose the tag group identifier is given as 7, then the 7th bit in the tag group identifier is set to 1. Thus, the *mfc_write_tag_mask* function is implemented as *mfc_write_tag_mask (1<<7)*.

- *mfc_read_tag_status_all()* - A request is sent to update tag status when all enabled MFC tag groups have a "no operation outstanding" status. The processor waits for the status to be updated. Implementation:

  ```
  (uint32_t) mfc_read_tag_status_all(void)
  ```

  Thus, the MFC waits till all the DMA commands in the tag group (tag mask is already set for all the commands with the same tag group) have completed.

3.4 Chapter Conclusion

In this chapter, I explained the various unique software features of CBEA that enhance our control over the interconnection architecture. This helps to understand the complexity involved in abstracting away from the low level bus architectural constraints to successfully implement the scheduling framework.

I also described in detail the different application partitioning models, creation of threads that are used to send processing requests to different processors, the PPE and SPE instruction sets and finally an overview of how PPE and SPEs interact.

CHAPTER 4

BACKGROUND WORK: BUS SCHEDULING ALGORITHMS

Our research group focused on developing an abstraction of the scheduling problem as well as an effective methodology that predictably schedules the multiprocessor bus. My key contribution to this effort was to build a framework that would allow the team to implement and test the performance of a class of scheduling algorithms for CBEA, while maintaining high throughput. While the focus of this thesis is to understand the framework; it is also important to know the scheduling algorithm, as it is important background material needed to appreciate the overall solution. In this chapter, I briefly describe the real-time algorithm that was designed.

The research team proposed to employ a software-controllable Multi-Domain Ring Bus (MDRB) architecture to increase system predictability and tighten WCET estimation. The problem of scheduling periodic real-time transactions on MDRB is challenging because the bus allows multiple non-overlapping transactions to be executed concurrently, and because the degree of concurrency depends on the topology of the bus and of executed transactions. The team proposed a practical abstraction for the scheduling problem together with novel scheduling algorithms. The first algorithm is optimal for transaction sets under restrictive assumptions while the second induces a competitive sufficient schedulable utilization bound for more general transaction sets.

This chapter is divided into three sections. The first section briefly reviews important terminology that is needed to understand the algorithms and implementation. This will also be helpful in understanding the description of the implementation of the framework (Chapter 5). The second section describes Real-Time Bus Transaction and Scheduling Model. The last section presents the scheduling algorithm for the proposed real-time transaction sets on the ring buses. Also, the relevant proofs that demonstrate the effectiveness of these algorithms have been included.

4.1 Terms and Terminology

4.1.1 Transaction

An event in which data is transferred from one processor to another is referred to as a transaction. We model three types of transactions in our implementation, which are described below.

4.1.1.1 Atomic Transaction

An *atomic* transaction (see Figure 4.1) is defined as the smallest non-interruptible transaction on the bus. The size of an *atomic* transaction on the EIB is 128 bytes.

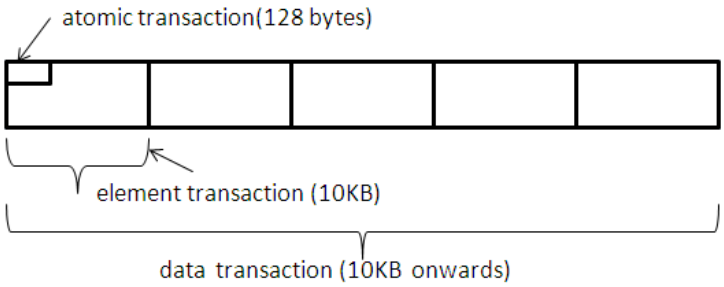

Figure 4.1: Types of Transactions

4.1.1.2 Element Transaction

An *element* transaction is defined as a transaction that is started by a transfer command issued by the bus scheduler. In other words, an *element* transaction is a sequence of *atomic* transactions that are scheduled without interrupts from the bus scheduler's standpoint. The size of an *element* transaction is defined in terms of the number of *atomic* transactions it is composed of.

4.1.1.3 Data Transaction

It represents a request made by an application for transferring a certain amount of data between CBEA's components. A *data* transaction comprises one or more *element* transactions. The size of a *data* transaction is defined in terms of the number of *atomic* transactions it is composed of.

The bus scheduler starts a transaction by first putting the data of the transaction into a DMA buffer of the MFC and then issues a *sending command* to the MFC. The transaction is transferred by the MFC in a hop-by-hop manner from the source to the destination through the shortest possible bus segment. For example, in Figure 2.4, the route of a transaction from SPE1 to SPE2 is SPE1-PPEMIC-SPE0-SPE2 on either of the two counterclockwise rings. We define two transactions to be overlapping if they share a segment of their route on the same ring.

Let us call a transaction that is started by a *sending command* an *element* transaction. A 128-byte chunk is an *atomic* transaction which operates in a non-interrupted manner. However, between *atomic* transactions of an *element* transaction, the bus may transfer *atomic* transactions of other *element* transactions on the same bus segment. In other words, an *element* transaction can be interrupted between its *atomic* transactions.

As already mentioned previously in Chapter 2, every *atomic* transaction goes through five phases sequentially, out of which three phases have a higher significance in terms of building the schedules: the *command*, the *arbitration*, and the *data* phase. Consequently, the remaining two phases, the *arbitration* and *data* phases, are called the *post-command* phase. During the *command* phase of an *atomic* transaction, the arbiter sets up a route for the transaction and puts it into a *post-command* phase queue. Let the command phase latency be L^c. The command phase latency can be hidden by pipelining.

4.1.2 Contention

A bus interface unit (BIU) can issue one command every cycle even while the command phase of its previous atomic transaction has not been completed. The arbitration phase starts at the end of the command phase. Due to some bus constraints, contention between *atomic* transactions may occur in their post-command phases. A low-level round robin scheduler arbitrates bus accesses between contending transactions stored in the post-command phase queue. This phase occurs when the BIU has to wait for bus access due to the contention between atomic transactions on the bus. There are three types of contentions described in the following.

4.1.2.1 Overlap Contention

Overlap contention occurs when there is an *atomic* transaction finishing the command phase while there are other overlapping transactions in their post-command phases. The *atomic* transaction has to wait at least until the other active overlapping transactions complete. Suppose if on one of the clockwise rings two *element* transactions, SPE0 to SPE1 and SPE3 to SPE5 (Figure 4.2), are in their postcommand phases. An *atomic* transaction SPE 1 to SPE 5 on the same ring has to wait for the overlapping transaction SPE3 to SPE5 to finish. Even though there are currently two transactions and according to the bus architecture, there can be three atomic transactions simultaneously on the ring. The *atomic* transaction from SPE1 to SPE5 has to wait due to overlap on the bus segment.

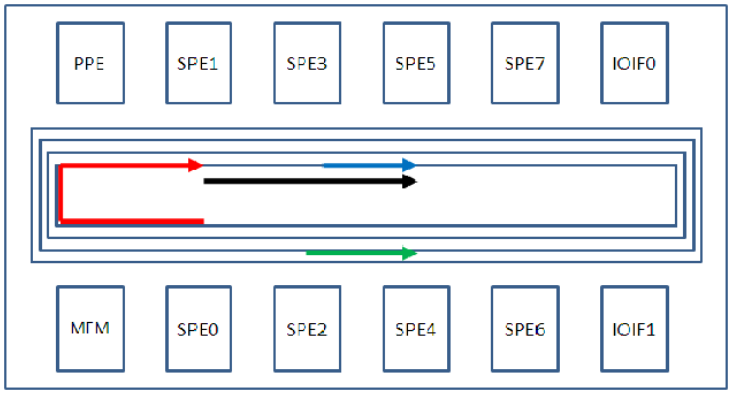

Figure 4.2: Overlap Contention

4.1.2.2 Overload Contention

Overload contention occurs when there is an *atomic* transaction finishing the command phase while, on the same ring, there are at least three non-overlapping transactions in their post-command phases (see Figure 4.3). Because each bus ring can support at most three *atomic* transactions at a point of time, the *atomic* transaction has to wait at least until any one of the previously active atomic transactions completes. For example, suppose there are three transactions SPE0 to SPE1, SPE3 to SPE5 and SPE4 to SPE2; if you want to start another transaction SPE2 to SPE0, you need to wait for any one of the above transactions to complete because at any given time there can be only three non-overlapping simultaneous transactions on the bus ring.

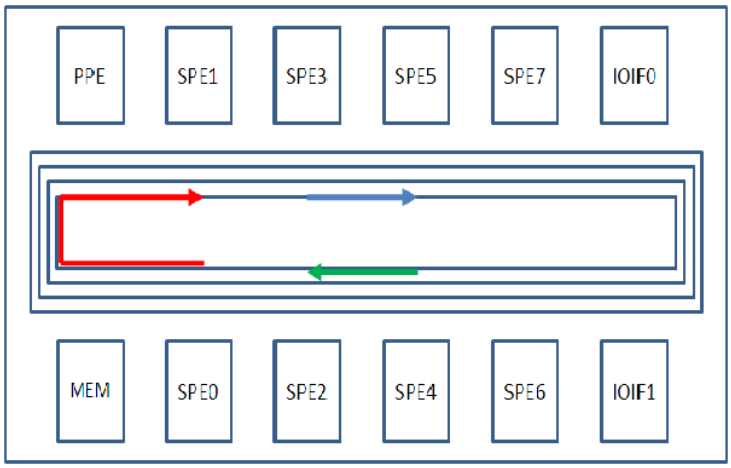

Figure 4.3: Overload Contention

4.1.2.3 Start-Time Contention

Start-time contention occurs when there is an *atomic* transaction finishing the command phase while on the same ring there are $k < 3$ other non-overlapping transactions which have been in their post-command phases for less than three cycles. Since a bus ring can only start the data phase of one *atomic* transaction every $L^o = 3$ cycles, the *atomic* transaction has to wait for at most $k * L^o$ cycles. Let us define the delay due to the start-time contention to be the *start-time latency*.

4.2 Real-Time Bus Transaction and Scheduling Model

We consider a scheduling problem where applications request periodic data transfers (data transactions) on the bus. A data transaction comprises of an infinite number of periodic jobs.

Without loss of generality, let the bus elements be indexed clock-wise. We define $\mathcal{T}$ as the set of data transactions $\mathcal{T} = \{\tau_i : i = [1, N]\}$. A data transaction τ_i is characterized by a tuple $\tau_i = (e_i, p_i, \epsilon_i^1, \epsilon_i^2)$ where e_i is the time that the bus spends to transmit a job of τ_i, p_i is the period of τ_i, and $\epsilon_i^1, \epsilon_i^2$ are the two indexes of the two endpoints which are called the first and the second index of τ_i, respectively. Each job must complete within its period, i.e. relative deadlines are equal to periods. A transaction has two endpoints ϵ_i^1 and ϵ_i^2 if it uses all bus elements in $[\epsilon_i^1, \epsilon_i^2]$ in the clockwise direction. Transaction τ_i is said to *go through* element ϵ if $\epsilon \in [\epsilon_i^1, \epsilon_i^2)$ (excluding the second endpoint of τ_i). The bus utilization

u_i of τ_i is calculated as $u_i = e_i/p_i$. We assume that all data transactions arrive at time 0. Let hyper-period h of $\mathcal{T}$ be the least common multiple of the periods of all transactions in $\mathcal{T}$.

Two transactions are said to *overlap* and cannot be transferred concurrently on the bus if they use the same bus segment between any two elements. Based on the endpoint definition, it is obvious that two transactions overlap if and only if they go through the same element. Given a data transaction set $\mathcal{T}$, we define an overlap indicating function $OV : \mathcal{T} \times \mathcal{T} \mapsto \{0, 1\}$ where $OV(\tau_i, \tau_j) = 1$ if τ_i and τ_j overlap, and 0 otherwise.

A *pairwise overlap set* (PO-set) $\mathcal{D}$ is defined as a maximal subset of $\mathcal{T}$ such that $\forall \tau_i, \tau_j \in \mathcal{D} : OV(\tau_i, \tau_j) = 1$. For convenience, we consider that a non-overlapping transaction belongs to a PO-set that contains only that transaction. In general a transaction may belong to more than one PO-set. Figure 4.4 shows an example of a transaction set with four PO-sets: $\mathcal{D}_1 = \{\tau_1, \tau_2, \tau_3, \tau_4\}$, $\mathcal{D}_2 = \{\tau_2, \tau_4, \tau_5\}$, $\mathcal{D}_3 = \{\tau_4, \tau_5, \tau_6\}$, $\mathcal{D}_4 = \{\tau_7, \tau_8\}$. Let the total number of PO-sets in a transaction set be $N^{\mathcal{D}}$. Notice that although τ_2 and τ_6 have one common endpoint (element 5), they do not overlap because they do not share any bus segment. Since each PO-set contains at least one element different from each other PO-set and transactions are arranged in an one dimension space, $N^{\mathcal{D}} \leq N$.

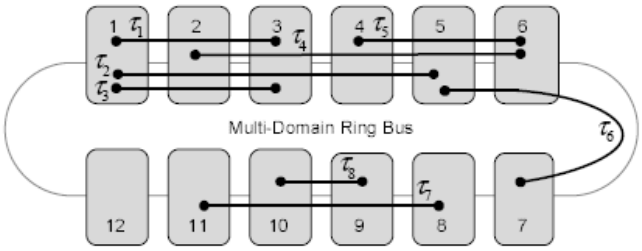

Figure 4.4: Non-circular Transaction Set

A transaction set is said to be *circular* if its overlapping transactions create a cycle on the bus and to be *non-circular* otherwise. Figures 4.4 and 4.5 show an example of a non-circular and a circular transaction set, respectively. A non-circular transaction set can be represented as a set of overlapping intervals on an indexed straight line where each interval corresponds to a transaction and the straight line is indexed by the indexes of the bus elements. Figure 4.6 shows the indexed straight line representation of the non-circular transaction set shown in

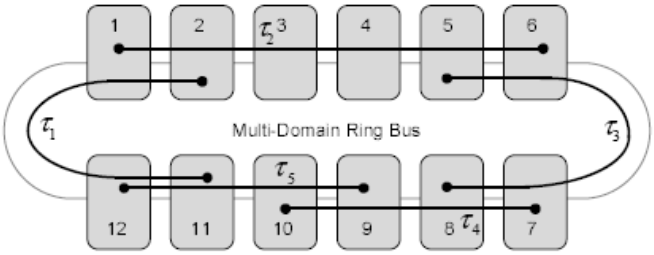

Figure 4.5: Circular Transaction Set

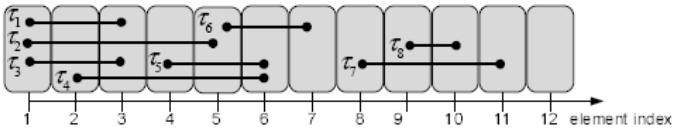

Figure 4.6: Indexed Straight Line Representation

Figure 4.4. Let the left-most transaction on the straight line be the *first* transaction. If there are more than one left-most transactions, any one of them can be the first transaction. For simplicity, we index the bus elements such that the first transaction has the first index to be the smallest index.

Due to the discrete nature of transactions' execution times and periods, we adopt the discrete scheduling model used in [7]. More specifically, we assume that every transaction's execution time and period are integral values. Scheduling decisions are also made at integral values of time, starting from 0. The real interval between time $t \in \mathbb{N}$ and time $t+1$, i.e. $[t, t+1)$, is called *slot* t. A schedule S is defined as a function S: $\Gamma \times \mathbb{N} \mapsto \{0, 1\}$ where $S(\tau_i, t) = 1$ if and only if τ_i is scheduled at slot t. A schedule S is *valid* if and only if according to S, it *never* happens that a transaction is scheduled in the same slot together with one or more other transactions that overlap with it.

Given the constraint on overlapping transactions, a necessary condition on the schedulability of a transaction set can be easily derived as in Theorem 4.2.1.

Theorem 4.2.1. *A transaction set $\mathcal{T}$ is schedulable only if:*

$$\forall \mathcal{D} \subset \mathcal{T} : u^{\mathcal{D}} = \sum_{\forall \tau_i \in \mathcal{D}} u_i \leq 1. \tag{4.2.1}$$

Proof. Since, by definition, no two transactions of a PO-set $\mathcal{D}$ can be scheduled

concurrently, all transactions of $\mathcal{D}$ must be scheduled in sequence. In other words, the transactions of $\mathcal{D}$ can be considered to be sharing one resource. Therefore, Inequality 4.2.1 must be satisfied. □

Let $\mathcal{E}(k)$ be a set of all transactions in $\mathcal{T}$ that go through same bus element indexed k. The following lemma is necessary for later discussion.

Lemma 4.2.1. *Given a transaction set $\mathcal{T}$ that satisfies the necessary condition, the following inequality holds:*

$$\sum_{\forall \tau_i \in \mathcal{E}(k)} u_i \leq 1.$$

Proof. Since transactions in $\mathcal{E}(k)$ pairwise overlap, there exists $\mathcal{D}$ such that $\mathcal{E}(k) \subseteq \mathcal{D}$. Therefore the lemma is implied by Theorem 4.2.1. □

4.3 Scheduling Algorithms for Ring Buses

In this section we present our scheduling algorithms for the proposed real-time transaction sets on the ring buses. The discussion is divided into three parts.

First, we propose an algorithm, namely POBase, which schedules every non-circular transaction set whose transactions have the *same* period. We will prove that the necessary condition (Theorem 4.2.1) is also the sufficient condition for same-period non-circular transaction set to be schedulable by POBase. Therefore, POBase is optimal for these transaction sets.

Second, a scheduling algorithm, namely POGen, is proposed to schedule non-circular transaction sets whose transactions do not have the same period. POGen, which is built based on POBase, can schedule all transaction sets for whose PO-set utilizations satisfy the following utilization bound:

$$\forall \mathcal{D} \subset \mathcal{T} : u^{\mathcal{D}} \leq \frac{\mathsf{L} - 1}{\mathsf{L}}, \tag{4.3.1}$$

where L is defined as the greatest common divisor of all transaction periods. Although the utilization bound is sufficient, it approximates 1 when L is large. We believe that this assumption holds in most practical real-time applications [8]. As we will show in the implementation section, with the speed of the state of the art

multicore chip buses [4], the practical time slot size is about 1 μs to 1 μs. Meanwhile, the period granularity in practical real-time applications [8] is at the level of milliseconds. That means L has practical values ranging from 10 to 100 time units.

Finally, we will discuss the issue of scheduling circular transaction sets and our proposed initial solution.

4.3.1 The POBase Algorithm

The problem of scheduling a non-circular same-period transaction set is similar to the problem of interval graph vertex coloring [9]. However, the optimal coloring algorithm in [9] can only schedule transactions which all have the same execution time. POBase is a modification of this algorithm to handle the problem at hand. POBase is a first-fit algorithm with respect to a transaction ordering. More specifically, in POBase, the transactions are ordered by their first indexes. Then in ascending order, each transaction is assigned to the earliest slots where no smaller-ordered overlapping transaction has been already assigned to[1]. Figure 4.7 shows an example of the schedule generated by POBase for the transaction set shown in Figure 4.4 whose transactions have period equal to 8 and execution times $e_1 = 2, e_2 = 1, e_3 = 2, e_4 = 3, e_5 = 4, e_6 = 1, e_7 = 4, e_8 = 4$. Consider the schedule of transactions of $\mathcal{D}_2 = \{\tau_2, \tau_4, \tau_5\}$. Transaction τ_5 is scheduled in slots $\{0, 1, 3, 4\}$ because its smaller-ordered overlapping transactions τ_2 and τ_4 are scheduled in slots $\{2, 5, 6, 7\}$.

[1]The transactions can also be order by their second indexes and their schedules are generated in descending order of the order list.

Algorithm 1 POBase

Input: transaction set $\mathcal{T}$ such that $\forall \tau_i \in \mathcal{T} : p_i = p$ where p is a constant

Output: schedule S for period p

1: $\mathcal{L} \leftarrow$ the list of all $\tau_i \in \mathcal{T}$ ordered according to ϵ_i^1
2: **for** each $\tau_i \in \mathcal{L}$ in ascending order **do**
3: **for** each $t \in [0, p)$ **do**
4: **if** $\sum_{x \in [0,p)} S(\tau_i, x) < e_i$ **then**
5: **if** $\forall \tau_j \in \mathcal{L} : OV(\tau_i, \tau_j) = 0$ or $S(\tau_j, t) = 0$ **then**
6: $S(\tau_i, t) \leftarrow 1$
7: **end if**
8: **end if**
9: **end for**
10: **end for**

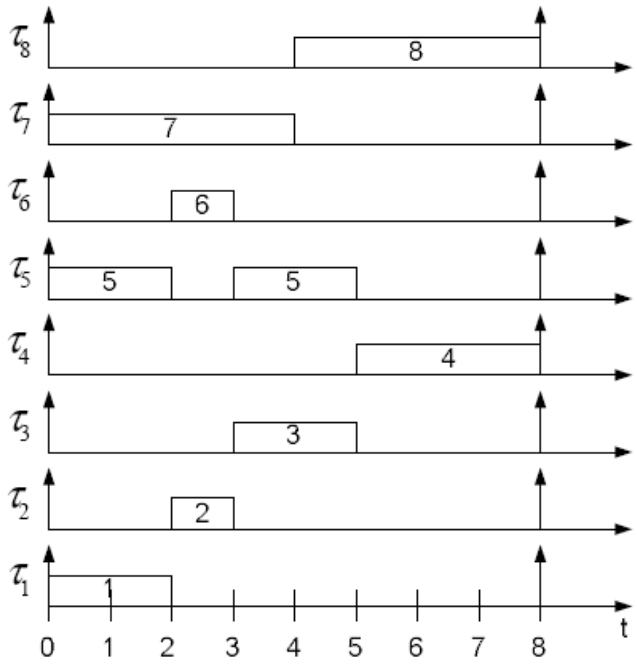

Figure 4.7: An Example of the POBase Algorithm

Theorem 4.3.1. POBase *is optimal for non-circular transaction sets*

Proof. The generated schedule is valid because the condition at Step 5 guarantees that a transaction is not scheduled in the same slot with its overlapping transac-

tions. It remains to show that if the transaction set satisfies the necessary condition, then at the end of the algorithm,

$$\forall \tau_i : \sum_{x \in [0,p)} S(\tau_i, x) = e_i. \tag{4.3.2}$$

We will prove this by induction.

Base case: Consider the first iteration of the for-loop starting at Step 2. In this iteration, the schedule of τ_1 in $\mathcal{L}$ is generated. Since τ_1 is the first transaction whose schedule is generated and $e_1 \leq p$, at the end of the iteration, we have $\sum_{x \in [0,p)} S(\tau_1, x) = e_1$. Furthermore, since $\epsilon_1^1 \leq \epsilon_2^1$, the following induction condition also holds at the end of the iteration: $\forall \tau_i \in \mathcal{T}$ if $\sum_{x \in [0,p)} S(\tau_i, x) > 0$ then $\epsilon_i^1 \leq \epsilon_2^1$.

Induction case: Assume after iteration k of the for-loop starting at Step 2, Equation 4.3.2 holds for all transactions $\{\tau_i : i \in [1, k]\}$ and the following induction condition also holds at the end of iteration k: $\forall \tau_i \in \mathcal{T}$ if $\sum_{x \in [0,p)} S(\tau_i, x) > 0$ then $\epsilon_i^1 \leq \epsilon_{k+1}^1$. Consider iteration $k+1$. By contradiction, assume that at the end of the iteration, $\sum_{x \in [0,p)} S(\tau_{k+1}, x) < e_{k+1}$. Let $\mathcal{E}(\epsilon_{k+1}^1)$ be the set of transactions that go through ϵ_{k+1}^1. Since $\mathcal{T}$ is non-circular, by the induction condition, we have $\forall \tau_i \in \mathcal{T}$ if $OV(\tau_i, \tau_{k+1}) = 1$ and $\sum_{x \in [0,p)} S(\tau_i, x) > 0$ then $\tau_i \in \mathcal{E}(\epsilon_{k+1}^1)$. In other words, among all the transactions that overlap with τ_{k+1}, only transactions in $\mathcal{E}(\epsilon_{k+1}^1)$ have their schedule generated. Therefore, the contradiction assumption occurs only when:

$$\sum_{\tau_i \in \mathcal{E}(\epsilon_{k+1}^1)} \sum_{x \in [0,p)} S(\tau_i, x) = p. \tag{4.3.3}$$

Since the following is true:

$$\forall \tau_i \in \mathcal{E}(\epsilon_{k+1}^1) \setminus \{\tau_{k+1}\} : \sum_{x \in [0,p)} S(\tau_i, x) \leq e_i,$$

by the contradiction assumption and Equation 4.3.3 we have: $\sum_{\tau_i \in \mathcal{E}(\epsilon_{k+1}^1)} e_i > p$. This contradicts Lemma 4.2.1 which implies that $\sum_{\tau_i \in \mathcal{E}(\epsilon_{k+1}^1)} e_i \leq p$. Therefore, at the end of the iteration, Equation 4.3.2 must hold for τ_{k+1} and the induction condition also holds. This completes the proof.

□

Algorithm analysis: An efficient sorting algorithm has time complexity $O(N)$.

In addition, Step 5 can be implemented to have a time complexity of $O(N)$. Therefore the time complexity of POBase to build a schedule of p slots for N transactions is $O(N^2 * p)$.

4.3.2 The POGen Algorithm

In this subsection we propose a scheduling algorithm (POGen) for non-circular transaction sets whose transactions do not have the same period. In POGen, the execution timeline from 0 to the hyper-period h, i.e. $[0, h)$, is divided into a set of consecutive *scheduling intervals*: $\{\mathsf{int}^k = [t^k, t^{k+1}) : k \in \mathbb{N} \wedge 0 \leq t^k < t^{k+1} < h\}$. Let $|\mathsf{int}^k| = t^{k+1} - t^k$. In each scheduling interval int^k, each transaction τ_i is assigned an *interval load* l_i^k which is the number of slots in the interval allocated to schedule τ_i. The interval loads, of each transaction are calculated such that at the end of each interval, the transaction's execution approximates its execution in the fluid scheduling model [10]. The interval load of a PO-set is the sum of the interval loads of its transactions. Given the interval loads of all transactions in interval int^k, POBase is used to generate the schedule of int^k. As shown in the previous subsection, the interval schedule given by POBase will be feasible if and only if:

$$\forall \mathcal{D} \subset \mathcal{T} : \sum_{\tau_i \in \mathcal{D}} l_i^k \leq |\mathsf{int}^k|.$$

A schedule of a transaction set, which is generated by POGen, is feasible if it satisfies the following two conditions:

- Condition 1: for each transaction τ_i, the sum of the interval loads over each of the transaction's period is equal to e_i.
- Condition 2: there is a feasible schedule for every scheduling interval.

In the following paragraphs, we will discuss our solution to identify the scheduling intervals and the interval loads, which induces a feasible schedule.

Our proposed solution is inspired by the work in [11, 12]. However, since neither of these works uses the transaction overlap assumption, their proposed algorithms cannot be used for the problem at hand. In POGen, a *scheduling interval* is defined as the interval between two closest arrival times (also deadlines) of any two transactions. Figure 4.8 shows an example of the scheduling intervals induced by the set of three transactions $\tau_1 = \{e_1 = 1, p_1 = 2, \epsilon_1^1 = 1, \epsilon_1^2 = 3\}$, $\tau_2 = \{e_2 = 1, p_2 = 3, \epsilon_2^1 = 1, \epsilon_2^2 = 4\}$ and $\tau_3 = \{e_3 = 1, p_3 = 6, \epsilon_3^1 = 2, \epsilon_3^2 = 5\}$.

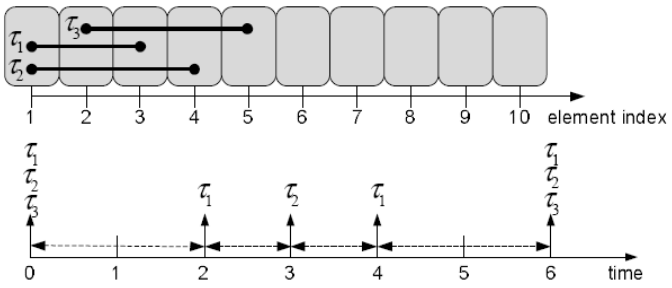

Figure 4.8: Scheduling Intervals on the Execution Timeline

With regard to the interval loads, we define for each transaction τ_i and scheduling interval int^k a lag function:

$$\mathsf{lag}(\tau_i, \mathsf{int}^k) = u_i * t^{k+1} - \sum_{x \in [0, t^k)} S(\tau_i, x).$$

The function calculates how much time τ_i must be executed in interval int^k such that at the end of int^k it is scheduled as if by the fluid scheduling model [10]. We also define for each PO-set $\mathcal{D}$ a similar lag function:

$$\mathsf{lag}(\mathcal{D}, \mathsf{int}^k) = u^{\mathcal{D}} * t^{k+1} - \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^k)} S(\tau_i, x).$$

In POGen, at each interval int^k, all interval loads must satisfy Inequality 4.3.4. The lower bound and the upper bound of the interval loads in Inequality 4.3.4 are the closest integral values of the lag functions.

$$\forall \tau_i \in \mathcal{T} : \lfloor \mathsf{lag}(\tau_i, \mathsf{int}^k) \rfloor \leq l_i^k \leq \lceil \mathsf{lag}(\tau_i, \mathsf{int}^k) \rceil. \tag{4.3.4}$$

Note that if all loads satisfy the lower bounds of Inequality 4.3.4, then the generated schedule satisfies Condition 1. The reason is as follows. Consider the last scheduling interval of a period of transaction τ_i: $\mathsf{int} = [t, a * p_i)$ where t and a are some integers; the lag function of τ_i is:

$$\mathsf{lag}(\tau_i, \mathsf{int}) = a * u_i * p_i - \sum_{x \in [0, t)} S(\tau_i, x).$$

Since $u_i * p_i = e_i$ is an integer, and so is $S(\tau_i, x)$, $\lfloor \mathsf{lag}(\tau_i, \mathsf{int}) \rfloor = \mathsf{lag}(\tau_i, \mathsf{int})$. That means the total interval loads of τ_i up to slot $a * p_i$, which is calculated as

$$\lfloor \mathsf{lag}(\tau_i, \mathsf{int}) \rfloor + \sum_{x \in [0,t)} S(\tau_i, x),$$

are equal to $a * e_i$ and satisfy Condition 1. However using only the lower bound loads does not guarantee the existence of a feasible schedule in each scheduling interval (Condition 2). This is also true if only upper bound loads are used. The following example illustrates this point. Consider again the example of the transaction set in Figure 4.8. If the algorithm runs with interval loads to be their lower bound loads, then the schedule of interval $[4, 6)$ is not feasible because the total load in this interval is 3. If, on the other hand, only the upper bound loads are used, then the schedule of interval $[0, 2)$ is also not feasible because the total load in this interval is 3. An algorithm that generates feasible schedules must use a combination of these values, and computing this is not trivial. For the ease of presentation, we split our discussion into two parts. First, we assume to have the GenerateLoad procedure (used in Step 2 of POGen) which satisfies the following proposition:

Proposition 4.3.1. *Assume that all PO-sets satisfy the utilization bound in Inequalities 4.3.1. If the following inequalities hold before the execution of* GenerateLoad *for an interval* int^k*:*

$$\forall \mathcal{D} \subset \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}, \mathsf{int}^k) \rfloor \leq |\mathsf{int}^k|, \tag{4.3.5}$$

$$\forall \mathcal{D} \subset \mathcal{T} : \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^k)} S(\tau_i, x) \geq \lfloor u^{\mathcal{D}} * t^k \rfloor, \tag{4.3.6}$$

then GenerateLoad *generates a set of interval loads for* int^k *which satisfy both Inequalities 4.3.4 and:*

$$\forall \mathcal{D} \subset \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}, \mathsf{int}^k) \rfloor \leq \sum_{\tau_i \in \mathcal{D}} l_i^k \leq |\mathsf{int}^k|. \tag{4.3.7}$$

Inequalities 4.3.7 set conditions on the total interval load of each PO-set. The right side of Inequality 4.3.7 guarantees that each PO-set with the generated interval loads is schedulable in int^k by POBase. Let us call a set of interval loads of a scheduling interval that satisfies both Inequality 4.3.4 and 4.3.7, and therefore

Conditions 1 and 2, a *feasible load set*. We will prove in Lemma 4.3.1 that, if Proposition 4.3.1 is true, then the conditions in Inequalities 4.3.5 and 4.3.6 are indeed always satisfied for every interval int^k. Given the defined GenerateLoad procedure and Lemma 4.3.1, it is then obvious that POGen generates a feasible schedule of $\mathcal{T}$. In the second part (Section 4.3.3), we will detail how to construct GenerateLoad and prove that Proposition 4.3.1 holds.

Algorithm 2 POGen

Input: transaction set $\mathcal{T}$

Output: schedule S

1: **for** each scheduling interval int^k **do**
2: $\{l_i^k : \forall i \in [1, N]\} \leftarrow \mathsf{GenerateLoad}(\mathcal{T}, \mathsf{int}^k)$
3: $\mathcal{T}' \leftarrow \left\{\{l_i^k, |\mathsf{int}^k|, \epsilon_i^1, \epsilon_i^2\} : \forall i \in [1, N]\right\}$
4: S for interval $\mathsf{int}^k \leftarrow \mathsf{POBase}(\mathcal{T}')$
5: **end for**

Lemma 4.3.1. *If Proposition 4.3.1 is true, then Inequalities 4.3.5 and 4.3.6 hold before the execution of* GenerateLoad *for every interval* int^k.

Proof. We prove by induction.
Base step: Consider the first scheduling interval $\mathsf{int}^0 = [0, t^1)$. Inequalities 4.3.5 for this interval hold because

$$\forall \mathcal{D} \subset \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}, \mathsf{int}^0) \rfloor = \lfloor u^{\mathcal{D}} * t^1 \rfloor \leq |\mathsf{int}^1|,$$

and Inequalities 4.3.6 hold because

$$\forall \mathcal{D} \subset \mathcal{T} : \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0,0)} S(\tau_i, x) = 0 = \lfloor u^{\mathcal{D}} * 0 \rfloor.$$

Induction step: Assume that Inequalities 4.3.5 and 4.3.6 hold in every scheduling interval up to int^k. We prove that Inequalities 4.3.5 and 4.3.6 also hold before the execution of GenerateLoad at interval int^{k+1}. Since Inequalities 4.3.5 and 4.3.6 are satisfied at interval int^k, GenerateLoad generates a feasible load set and POBase generates a feasible schedule for the interval. Therefore after Step 4, we have:

$$\forall \mathcal{D} \subset \mathcal{T} : \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [t^k, t^{k+1})} S(\tau_i, x) = \sum_{\tau_i \in \mathcal{D}} l_i^k.$$

Then by the left side of Inequalities 4.3.7, we obtain the following which proves that Inequalities 4.3.6 hold for int^{k+1}.

$$\begin{aligned}
&\forall \mathcal{D} \subset \mathcal{T} : \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^{k+1})} S(\tau_i, x) \\
&= \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^{k})} S(\tau_i, x) + \sum_{\tau_i \in \mathcal{D}} l_i^k \\
&\geq \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^{k})} S(\tau_i, x) + \\
&\left\lfloor u^{\mathcal{D}} * t^{k+1} \right\rfloor - \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^{k})} S(\tau_i, x) \\
&= \lfloor u^{\mathcal{D}} * t^{k+1}. \rfloor
\end{aligned}$$

Now consider Inequalities 4.3.5. Notice that since $S(\tau_i, x)$ is integer, we have:

$$\forall \mathcal{D} \subset \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}, \mathsf{int}^{k+1}) \rfloor = \left\lfloor u^{\mathcal{D}} * t^{k+2} \right\rfloor - \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^{k+1})} S(\tau_i, x).$$

Since Inequalities 4.3.6 hold for int^{k+1}, Inequalities 4.3.5 also hold because:

$$\begin{aligned}
&\forall \mathcal{D} \subset \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}, \mathsf{int}^{k+1}) \rfloor \\
&= \left\lfloor u^{\mathcal{D}} * t^{k+2} \right\rfloor - \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^{k+1})} S(\tau_i, x) \\
&\leq \lfloor u^{\mathcal{D}} * t^{k+2} \rfloor - \lfloor u^{\mathcal{D}} * t^{k+1} \rfloor \\
&\leq \lceil u^{\mathcal{D}} * (t^{k+2} - t^{k+1}) \rceil \leq |\mathsf{int}^{k+1}|.
\end{aligned}$$

This completes the proof. □

4.3.3 The GenerateLoad Procedure

As we mentioned, procedure GenerateLoad searches for a feasible load set of each scheduling interval. There are two questions that have to be answered: (1) Is there a feasible load set? (2) Is there an efficient algorithm to find it? We will show that the problem at hand is equivalent to the problem of circulations in graphs with loads and lower bounds [13]. This is the problem of finding a feasible circulation flow in a directed graph where each edge has a capacity and

a lower bound. Furthermore, we will prove that if the utilization of each PO-set is smaller than the utilization bound expressed by Inequalities 4.3.1, there always exists a feasible solution, therefore answering Question 1. Then, since the Ford-Fulkerson algorithm [13] can be used to solve the problem, Question 2 is also answered.

In the following, we will intuitively describe the construction of a directed graph from the input of GenerateLoad. Each vertex of the constructed graph represents a PO-set $\mathcal{D}_j$. For each vertex, a *PO-set edge* $g_j^{\mathcal{D}}$ is defined which exits from the vertex and whose flow value $f_j^{\mathcal{D}}$ represents the interval load of the corresponding PO-set. A lower bound value $b_j^{\mathcal{D}}$ and a capacity $c_j^{\mathcal{D}}$ are defined for each of the PO-set edges such that Inequalities 4.3.7 are imposed on their flow values:

$$\forall \mathcal{D}_j \subset \mathcal{T} : b_j^{\mathcal{D}} = \lfloor \mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) \rfloor \leq f_j^{\mathcal{D}} \leq c_j^{\mathcal{D}} = |\mathsf{int}^k|. \tag{4.3.8}$$

Furthermore, for each transaction τ_i, a *transaction edge* is defined whose flow value f_i represents the interval load of the corresponding transaction. A lower bound value b_i and a capacity c_i are defined for each of the transaction edges such that Inequalities 4.3.4 are imposed on their flow values:

$$\forall \tau_i \in \mathcal{T} : b_i = \lfloor \mathsf{lag}(\tau_i, \mathsf{int}^k) \rfloor \leq f_i \leq c_i = \lceil \mathsf{lag}(\tau_i, \mathsf{int}^k) \rceil. \tag{4.3.9}$$

The flow of a transaction edge entering a vertex represents the contribution of the corresponding transaction's interval load to the corresponding PO-set's interval load. The endpoints and the direction of each edge are defined in such a way that the values of the flows in and out a vertex preserve the relationship between the interval load of the corresponding PO-set and that of its transactions. The graph has a feasible circulation flow which represents a feasible load set.

The following definition is necessary for the graph construction. Let the *index PO-set order* of a transaction set $\mathcal{T}$ be an ordered list of all PO-sets in $\mathcal{T}$ where PO-set $\mathcal{D}$ with smaller $\min_{\tau_i \in \mathcal{D}_j} \epsilon_i^2$ has smaller index. Ties are broken arbitrarily. Since each PO-set has only one value $\min_{\tau_i \in \mathcal{D}_l} \epsilon_i^2$, the order is well-defined. The transaction set in Figure 4.4 has the index PO-set order be $\{\mathcal{D}_j : j \in [1, 4]\}$ where $\mathcal{D}_1 = \{\tau_1, \tau_2, \tau_3, \tau_4\}$, $\mathcal{D}_2 = \{\tau_2, \tau_4, \tau_5\}$, $\mathcal{D}_3 = \{\tau_4, \tau_5, \tau_6\}$, $\mathcal{D}_4 = \{\tau_7, \tau_8\}$. Figure 4.9 shows the graph G constructed from the transaction set in Figure 4.4. Transaction edges are represented by solid lines while PO-set edges are represented by dotted lines.

Graph construction: let us define a tuple $G = (V, E)$ as follows:

- For each PO-set $\mathcal{D}_j$ in the index PO-set order, define a vertex v_j.
- For each PO-set $\mathcal{D}_j$ in the index PO-set order, define a directed edge $g_j^{\mathcal{D}}$ with capacity $c_j^{\mathcal{D}} = |\mathsf{int}^k|$ and lower bound $b_j^{\mathcal{D}} = \lfloor \mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) \rfloor$. Let $g_j^{\mathcal{D}}$ be a *PO-set edge*.
- For each transaction τ_i, define a directed edge g_i with capacity $c_i = \lceil \mathsf{lag}(\tau_i, \mathsf{int}^k) \rceil$, and lower bound $b_i = \lfloor \mathsf{lag}(\tau_i, \mathsf{int}^k) \rfloor$. Let g_i be a *transaction edge*.
- $\{g_i : \tau_i \in \mathcal{D}_1\}$ are edges that enter v_1; $g_1^{\mathcal{D}}$ are edges that exit v_1.
- $\forall j : 1 < j \leq N^{\mathcal{D}}$, $\{g_i : \tau_i \in \mathcal{D}_j \setminus \mathcal{D}_{j-1}\}$ and $g_{j-1}^{\mathcal{D}}$ are edges that enter v_j; $\{g_i : \tau_i \in \mathcal{D}_{j-1} \setminus \mathcal{D}_j\}$ and $g_j^{\mathcal{D}}$ are edges that exit v_j. This construction step deals with the situation where two PO-sets $\mathcal{D}_{j-1}, \mathcal{D}_j$ share some transactions. Intuitively, to preserve the relationship between the interval loads of the PO-sets and that of its transactions, the transaction edge of a transaction common to the two PO-sets would have to enter the two corresponding vertexes v_{j-1}, v_j. Since in a qualified graph, each directed edge can enter at most one vertex, this situation must be avoided. This can be accomplished by representing the interval loads of the common transactions on the second PO-set (v_j) as the interval load of the first PO-set (i.e., $g_{j-1}^{\mathcal{D}}$ enters v_j) minus the interval load of the transactions that are only in the first set (i.e., $\{g_i : \tau_i \in \mathcal{D}_{j-1} \setminus \mathcal{D}_j\}$ exit v_j). Lemma 4.3.2 will detail the proof of this argument.
- $V = \{v_j : j \in [1, N^{\mathcal{D}}]\}$.
- $E = \{g_i : j \in [1, N]\} \cup \{g_j^{\mathcal{D}} : j \in [1, N^{\mathcal{D}}]\}$.

Finally, the graph flow is subject to the flow conversation constraint [13] in which given a vertex, the sum of the flow values entering it minus the sum of the flow values exiting it is zero.

As a graph construction example, consider vertex v_2 that represents PO-set $\mathcal{D}_2$. The vertex has an output PO-set edge $g_2^{\mathcal{D}}$ which represents the interval load of $\mathcal{D}_2$. Since $\mathcal{D}_1$ has τ_2 and τ_4 in common with $\mathcal{D}_2$ but not τ_1 and τ_3, v_2 has an input PO-set edge $g_1^{\mathcal{D}}$ which represents the interval load of $\mathcal{D}_1$ and two output transaction edges g_1 and g_3 that represent the interval loads of τ_1 and τ_3, respectively. Finally

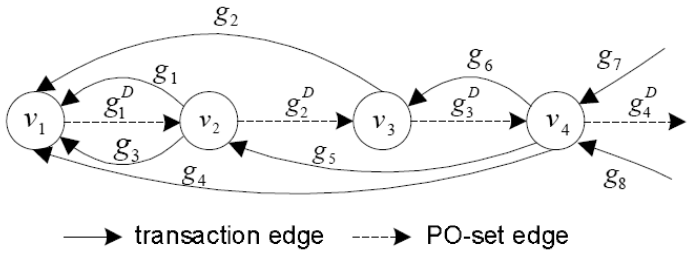

Figure 4.9: Constructed Graph G

v_2 has an input transaction edge g_5 that represents the interval load of τ_5. Lemma 4.3.2 shows that G is indeed a directed graph.

Lemma 4.3.2. *G is a directed graph.*

Proof. Since every edge of G is directed, it remains to show that each edge has only one or two indexes. There is one edge defined for each PO-set and one edge defined for each transaction.
For each PO-set $\mathcal{D}_j$, the PO-set edge g_j^D exits only v_j. In addition, g_j^D enters only v_{j+1} when $j < N^{\mathcal{D}}$. Therefore each PO-set edge exits exactly one vertex and enters at most one vertex.
By the index PO-set ordering, if $\tau_i \in \mathcal{D}_j \setminus \mathcal{D}_{j-1}$, then $\tau_i \notin \mathcal{D}_k \setminus \mathcal{D}_{k-1}$ where $j < k \leq N^{\mathcal{D}}$. Therefore, the elements of the following set are disjoint: $\mathcal{A} = \big\{\{g_i : \tau_i \in \mathcal{D}_1\}, \{g_i : \tau_i \in \mathcal{D}_j \setminus \mathcal{D}_{j-1}\} : j \in (1, N^{\mathcal{D}}]\big\}$. By definition, $\mathcal{A}$ contains the transaction edges of G that enter some vertices. Also the union of the elements of $\mathcal{A}$ is $\{g_i : \tau_i \in \mathcal{T}\}$. Therefore, each transaction edge enters exactly one vertex. By a similar proving technique, we can show that each transaction edge exits at most one vertex. Due to space constraints, we skip the detailed proof. In conclusion, every edge of G has at most two endpoints and is directed. □

It remains to show that Proposition 4.3.1 holds and therefore POGen generates a feasible schedule for all transaction sets that satisfy the utilization bound of Inequalities 4.3.1. For simplicity of exposition, we split the proof in multiple lemmas. Lemma 4.3.3 shows that a feasible load set can be found from an integral feasible flow of the correspondent graph G. Then, Lemma 4.3.5 proves that graph G has a feasible flow if Inequalities 4.3.5 and 4.3.6 are satisfied for interval int^k and furthermore all PO-sets satisfy an utilization constraint based on $|\mathsf{int}^k|$. Note that we know from [13] that if graph G has a feasible flow, then it has an integral

feasible flow which can be found by the Ford-Fulkerson algorithm [13]. Finally, we will show that the utilization bound of Inequalities 4.3.1 implies the utilization bound used in Lemma 4.3.5. Hence, Proposition 4.3.1 holds.

Lemma 4.3.3. *If there is an integral feasible flow in graph G, then there is a feasible load set where $\forall \tau_i \in \mathcal{T} : l_i^k = f_i$.*

Proof. Given an integral feasible flow, $\forall \tau_i \in \mathcal{T}$ let $l_i^k = f_i$. The following inequality holds:

$$\forall \tau_i \in \mathcal{T} : \lfloor \mathsf{lag}(\tau_i, \mathsf{int}^k) \rfloor \leq l_i^k \leq \lceil \mathsf{lag}(\tau_i, \mathsf{int}^k) \rceil.$$

Thus the interval loads satisfy Inequality 4.3.4. We now have to prove that the interval loads also satisfy Inequality 4.3.7. We prove this by induction over the ordered set of vertices.
Base case: By the flow conservation constraint at vertex v_1, we have

$$\sum_{\tau_i \in \mathcal{D}_1} l_i^k = \sum_{\tau_i \in \mathcal{D}_1} f_i = f_1^{\mathcal{D}}.$$

Then, by the edge constraints of PO-set edge $g_1^{\mathcal{D}}$, the Inequalities 4.3.7 of $\mathcal{D}_1$ hold.

$$\lfloor \mathsf{lag}(\mathcal{D}_1, \mathsf{int}^k) \rfloor \leq \sum_{\tau_i \in \mathcal{D}_1} l_i^k \leq |\mathsf{int}^k|.$$

Induction case: Assume Inequality 4.3.7 is satisfied up to $\mathcal{D}_{j-1}$ and the following induction condition holds:

$$\sum_{\tau_i \in \mathcal{D}_{j-1}} f_i = f_{j-1}^{\mathcal{D}}.$$

We prove that Inequality 4.3.7 and the induction condition are also satisfied for $\mathcal{D}_j$. Given the induction assumption and the flow conservation constraint at vertex v_j, the following equalities hold:

$$\begin{aligned}
&\sum_{\tau_i \in \mathcal{D}_j} l_i^k = \sum_{\tau_i \in \mathcal{D}_j \setminus \mathcal{D}_{j-1}} f_i + \sum_{\tau_i \in \mathcal{D}_j \cap \mathcal{D}_{j-1}} f_i \\
&= \sum_{\tau_i \in \mathcal{D}_{j-1} \setminus \mathcal{D}_j} f_i + f_j^{\mathcal{D}} - f_{j-1}^{\mathcal{D}} + \sum_{\tau_i \in \mathcal{D}_j \cap \mathcal{D}_{j-1}} f_i \\
&= f_{j-1}^{\mathcal{D}} + f_j^{\mathcal{D}} - f_{j-1}^{\mathcal{D}} = f_j^{\mathcal{D}}.
\end{aligned}$$

Then, by the edge constraints of PO-set edge $g_j^{\mathcal{D}}$, Inequality 4.3.7 holds for $\mathcal{D}_j$.

$$\lfloor \mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) \rfloor \leq \sum_{\tau_i \in \mathcal{D}_j} l_i^k \leq |\mathsf{int}^k|.$$

Furthermore, the induction condition also holds:

$$\sum_{\tau_i \in \mathcal{D}_j} f_i = f_j^{\mathcal{D}}.$$

This completes the proof.

□

The following lemma is necessary for later discussion and is a direct result from the induction condition in the proof of Lemma 4.3.3.

Lemma 4.3.4. *The following equalities hold for every vertex v_j:*

$$\sum_{\tau_i \in \mathcal{D}_j} f_i = f_j^{\mathcal{D}}.$$

Lemma 4.3.5. *There exists a feasible flow in graph G if Inequalities 4.3.5 and 4.3.6 are satisfied for interval int^k and furthermore the PO-set utilizations satisfy the following condition:*

$$\forall \mathcal{D}_j \subset \mathcal{T} : u_j^{\mathcal{D}} \leq \frac{|\mathsf{int}^k| - 1}{|\mathsf{int}^k|}. \tag{4.3.10}$$

Proof. First note that Inequalities 4.3.5 are necessary for the edge constraints on each PO-set edge (Inequality 4.3.8) to be satisfied. Let us construct a flow as follows:

$$\forall \tau_i \in \mathcal{T} : f_i = \mathsf{lag}(\tau_i, \mathsf{int}^k)$$
$$\forall \mathcal{D}_j \subset \mathcal{T} : f_j^{\mathcal{D}} = \mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k).$$

We will have to prove that the constructed flow satisfies the edge constraints and the flow conservation constraints. Given the constructed flow, it is easy to verify that the edge constraints of each transaction edge (Inequality 4.3.9) and the left-side edge constrains of each PO-set edge (Inequality 4.3.8) are satisfied. The right-side edge constraints of each PO-set edge are satisfied because, by the definition

of the lag function and by Inequalities 4.3.6, before the execution of GenerateLoad for interval int^k we have the following:

$$\begin{aligned}
&\mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) = u_j^{\mathcal{D}} * t^{k+1} - \sum_{\tau_i \in \mathcal{D}_j} \sum_{x \in [0, t^k)} S(\tau_i, x) \\
&\leq u_j^{\mathcal{D}} * t^{k+1} - \lfloor u_j^{\mathcal{D}} * t^k \rfloor \\
&< u_j^{\mathcal{D}} * t^{k+1} - u_j^{\mathcal{D}} * t^k + 1.
\end{aligned}$$

Now by Inequalities 4.3.10, the following holds:

$$\mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) < u_j^{\mathcal{D}} * t^{k+1} - u_j^{\mathcal{D}} * t^k + 1 \leq |\mathsf{int}^k|.$$

It remains to verify that the flow conservation constraints at each vertex also hold. By Lemma 4.3.4, the total flow value entering vertex v_j can be calculated as:

$$\begin{aligned}
&\sum_{\tau_i \in \mathcal{D}_j \setminus \mathcal{D}_{j-1}} f_i + f_{j-1}^{\mathcal{D}} \\
&= f_j^{\mathcal{D}} - \sum_{\tau_i \in \mathcal{D}_j \cap \mathcal{D}_{j-1}} f_i + f_{j-1}^{\mathcal{D}} \\
&= f_j^{\mathcal{D}} - \sum_{\tau_i \in \mathcal{D}_j \cap \mathcal{D}_{j-1}} f_i + \sum_{\tau_i \in \mathcal{D}_{j-1}} f_i \\
&= f_j^{\mathcal{D}} + \sum_{\tau_i \in \mathcal{D}_{j-1} \setminus \mathcal{D}_j} f_i,
\end{aligned}$$

which equals to the total flow value exiting v_j, i.e.:

$$f_j^{\mathcal{D}} + \sum_{\tau_i \in \mathcal{D}_{j-1} \setminus \mathcal{D}_j} f_i.$$

□

We can finally state our main theorem.

Theorem 4.3.2. *Transaction set $\mathcal{T}$ is schedulable by* POGen *if:*

$$\forall \mathcal{D}_j \subset \mathcal{T} : u_j^{\mathcal{D}} \leq \frac{\mathsf{L} - 1}{\mathsf{L}}.$$

Proof. Since $\mathsf{L} \leq \min_k(|\mathsf{int}^k|)$, Inequalities 4.3.10 hold. Assume that Inequalities 4.3.5 and 4.3.6 hold for a specific interval int^k. Then by Lemma 4.3.5 and [13],

the constructed graph G has an integral feasible flow. Hence, by Lemma 4.3.3 algorithm GenerateLoad computes a feasible load set, which proves Proposition 4.3.1. Since furthermore, according to Lemma 4.3.1, Inequalities 4.3.5 and 4.3.6 hold for every interval int^k, it follows that Inequalities 4.3.4 and 4.3.7, and therefore feasibility Conditions 1 and 2, also hold for every interval. This concludes the proof. □

Algorithm analysis: The time complexity of the Ford-Fulkerson algorithm, which is also the time complexity of GenerateLoad, is $O(N * |\mathsf{int}^k|)$. Therefore, the time complexity of POGen is $O(N^2 * h)$.

4.3.4 Scheduling Circular Transaction Sets

Unfortunately, neither algorithm POBase nor POGen can be applied to circular transaction sets. This is because the necessary condition of Theorem 4.2.1 is not a sufficient condition for same period circular transactions. As an example, consider the transaction set in Figure 4.5. Assume that all transactions have transmission times equal to 1 and periods equal to 2.

The utilization of each PO-set of this transaction set is 1, hence it satisfies the necessary condition. However, it is easy to see that there exists no feasible schedule. In fact, any valid schedule can have at most four transactions scheduled in the first two slots; therefore, the fifth transaction misses its first deadline.

It is important to stress that in most cases, engineering approaches can be used to replace a circular transaction set with a functionally equivalent non-circular one. We believe that in many applications, software designers can avoid the transaction cycle by selecting the endpoints of transactions. That is possible because endpoints are determined by the placement of software tasks which produce or use the transaction data.

When this technique cannot be applied, we propose a solution to convert a circular transaction set into a non-circular one as follows: select an element ϵ on the ring; then for each transaction τ_i that goes through ϵ, split τ into two transactions such that one of them has endpoints $\{\epsilon_i^1, \epsilon\}$ and the other one has endpoints $\{\epsilon, \epsilon_i^2\}$. Note that the two transactions do not overlap, hence the cycle is broken at ϵ. However, to respect the ordering between transactions it might be necessary to buffer the transaction data at ϵ for one full transaction period.

4.4 Chapter Conclusion

In this chapter I described the bus scheduling algorithm that was developed along with its mathematical proof. Real-Time Bus Transactions and Scheduling Model were also discussed, along with the basic terms and terminology required to understand the algorithm. Since the real-time scheduling algorithm and its derivation were not my contribution, I have included only a brief discussion as background material. For a more detailed overview, one can contact my research group (in particular Bach D. Bui and Rodolfo Pellizzoni) in the Real-Time Laboratory at the Department of Computer Science, University of Illinois.

CHAPTER 5

IMPLEMENTATION

This chapter explains the implementation of the bus scheduling framework for real-time applications. It describes the rationale and functions used in each step of the framework. As mentioned earlier, we chose to implement this framework on a CBEA. The components of this architecture including PPE, SPEs, IOIFs, and MIF are synchronized to an accuracy of nanoseconds, which is good enough for most of the critical real-time applications. Further, as discussed earlier, due to its hardware features CBEA overcomes important limitations of modern architectures.

The CBEA implementation platform used in this work is the Sony Play Station 3 (PS3) which has 6 SPEs enabled out of 8. The 7th SPE on the chip is blasted for power purposes and the last SPE is used for running the hypervisor. It is to be noted that all 8 SPEs are enabled on the Cell Blades. Of the two application-partitioning models mentioned in Chapter 2 our implementation is based on the PPE-centric model. In this type of model the PPE runs the main application, and the individual jobs are loaded onto the SPEs for execution. The PPE waits while the SPEs execute, and then coordinates the results returned by them. Furthermore, within this framework I implemented the sub-model called *parallel stages model*, the one with serial data and parallel computation.

Keeping parallel stage model in mind I developed four versions of code (see Figure 5.1), and from these I picked the interrupt based model where PPE sends the complete schedule table to all SPEs. This version was optimized to conduct experiments.

In this chapter I describe the selected code in more detail. This scheduling framework code is divided into two main parts - PPE side and SPE side, which are explained in the following sections. Within each section, rationale and functions for each step are discussed at length. In the end, using a flowchart, I explain the flow of the implementation code, and provide an overview of the build environment of this project.

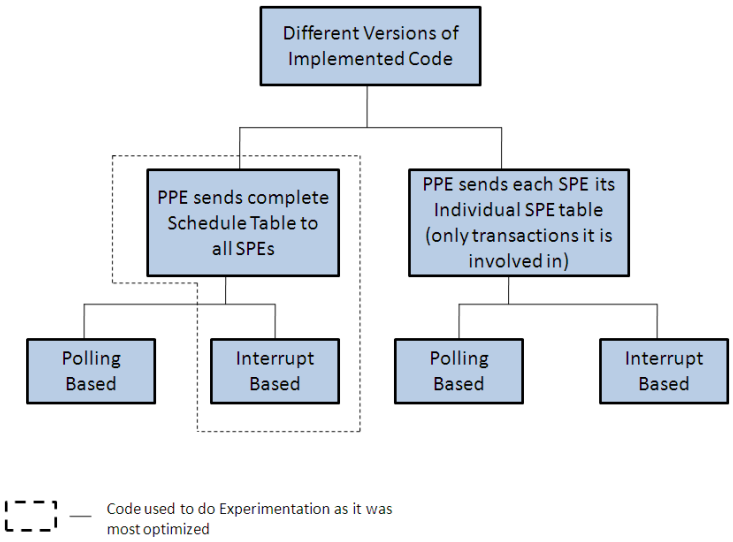

Figure 5.1: Different Versions of Implemented Code

5.1 Scheduling Framework: PPE Side

On the PPE side, first all the input arrays and variables are initialized. Then schedules, in table format, are solicited via the User Interface, referred to as *PPE Table*. Next the PPE creates SPE contexts and loads them with this *PPE Table*.

5.1.1 Input Arrays and Variables

Before understanding the User Interface let us take a look at the parameters required for building the table based schedule using an example. A data transaction of size 100 kB is to be transferred from SPE-X (source SPU id) to SPE-Y (destination SPU id). Based on the experimentation done, which will be described in Chapter 6, the slot size we use is 10 kB, so the 100 kB data transaction has to be divided into a series of 10 element transactions each of size 10 kB. The SPE-X is considered as the *source_spu_id* from which the data is sent, while SPE-Y is the *destination_spu_id* which receives the data. The amount or size of data to be DMAed is 10 kB, which basically means the size of *element* transaction is 10 kB. The above example represents only one SPE to SPE DMA. When we place three simultaneous data transactions of sizes 100 kB, 200 kB and 150 kB transferring data from SPE-X to SPE-Y, SPE-A to SPE-B and SPE-I to SPE-J respectively, together in a table, it is called the *PPE table*.

5.1.2 User Interface

The technique of picking the various combinations of SPEs for sending and receiving the data in specific slots is defined as a scheduling algorithm. In our framework the schedules are built based on an algorithm that is computed offline. The user is required to enter some details of the scheduling framework into the arrays on the PPE side. They are described as follows:

```
/**** Initialize Schedule Table ****
***********************************/

/**** PLEASE ENTER THE FOLLOWING ****
***********************************/

1. Number of SPEs (2-6):
************************
```

Number of SPEs, as the name suggests, is the number of SPEs for which you want to create contexts; it can either be 2, 3, 4, 5 or 6. This number means that many different SPEs are part of the schedule and they all need to be initialized. The variable that holds this number is called *num_spes*.

```
2. Array Size (Number of entries in the
                            schedule table (1-30))
***********************************************
```

Array Size is the total number of entries you will be entering into the *PPE Table*. The value of the variable *array_size* defines the number of data transactions the user wants to schedule on the SPEs. The maximum number of entries you can have in the User Interface is 30, as all the arrays defined in this implementation are static arrays and thus they need to be within the provided range for the application to work properly.

```
3. Data Size in Kilobytes (minimum being 10KB)
*********************************************
```

Data Size is the size of the *data* transaction DMA*ed* from the source SPU id to the destination SPU id. To make the User Interface *user friendly*, we ask the

user to enter the data size in kB (where, 1 kB = 1024-bytes). According to our implementation we have a lower limit on the size of the *data* transaction, which is 10 kB (size of the element transaction) and it is advisable to keep the *data* transaction a multiple of 10, as it helps in analyzing and scheduling the initialization and completion of *element* transactions. Through the User Interface the data sizes are entered into an array called *data_size[i]*. These sizes are later converted from kilobytes to bytes in the function *size_conversion()* and stored in the variable *spu_dma_data*. The size of *element* transaction is always 10 kB, irrespective of the size of the *data* transaction entered in the User Interface. *Data* transactions are divided into *element* transactions of size 10 kB in the SPU side. The *element* transactions are scheduled in the interrupt handler function (described in later section). For example, if a *data* transaction of size 50 kB is to be transferred from SPE1 to SPE3, then it is divided into five *element* transactions each of size 10 kB, and transferred based on its schedule from SPE1.

If in the previous entry we entered the *Array Size* as 3, it implies that we have three *data* transactions in our PPE Table. Thus, for all the three *data* transactions the *data_size* in kB has to be entered, which is later converted to bytes in the function *size_conversion()* (1024 * *data_size*).

```
4. Enter the Source SPU ids (0-5)
*********************************
```

As mentioned earlier, the *source SPU ids* are the SPEs that send the data. The number of *source SPU ids* to be entered is equal to the *Array Size* value already entered previously. The array *spu_source_id[i]* holds the ids of the source SPEs.

```
5. Enter the Destination SPU IDs (0-5)
**************************************
```

Destination SPU ids are the SPEs that receive the data. For every *source SPU id* entered above, we need to enter a *destination SPU id* respectively. The array that holds the destination SPU ids is the *spu_destn_id[i]*.

5.1.3 Creating SPE Contexts

After entering all the details of the data transactions into the User Interface, the PPE creates contexts for the number of SPEs the user wants (*num_spes*). As seen

in the code snippet below, the *for* loop goes from 0 to num_spes, where num_spes is the number of SPEs for which you want to create contexts.

```
/*** CREATE CONTEXT ***/
for(i = 0; i < num_spes; i++){
    if ((id[i] = spe_context_create(SPE_MAP_PS, NULL))
                                                == NULL){
            perror ("Failed creating context");
            exit (1);
    }
}
```

5.1.3.1 Loading the Program on SPE Contexts

After the contexts are created, the program to be run on the SPE is loaded onto the context (refer to the piece of code below). In our implementation, all the SPEs execute the same program which is - *simple_spu*. However, it is to be noted that, though they run the same program and the contain the same information, the SPEs are scheduled to execute different schedules.

```
/*** LOAD THE SPES WITH THE PROGRAM ***/
for(i = 0; i < num_spes; i++){
       spe_program_load (id[i], &simple_spu) ;
}
```

5.1.3.2 Running the SPEs

Once the program is loaded, the next step is to start running the SPEs. The PPE intrinsic *spe_context_run()* function serves the purpose of running the SPES for which the contexts have already been created.

```
/*** SPE CONTEXT RUN FUNCTION ***/
    if (spe_context_run(ctx, &entry,
                    0, NULL, NULL, NULL) < 0){
        perror ("Failed running context");
        exit (1);
    }
    pthread_exit(NULL);
```

5.1.3.3 Wait for SPEs

After the SPEs have completed their execution, the control returns back to the PPE, where the PPE is waiting for them to complete their job. The function *pthread_join()* waits for all the SPEs that are running.

```
for (i = 0; i<num_spes; i++) {
      if (pthread_join (threads[i], NULL)){
          perror("Failed pthread_join");
          exit (1);
      }
  }
```

5.1.3.4 Destroy SPE Contexts

The function *spe_context_destroy* destroys all the SPE contexts created. Before we detroy the contexts, we need to wait for all the SPEs to complete their execution and return the control to the PPE. Once that is complete, the PPE then destroys all the contexts.

```
/*** Destroy context ***
  ************************/
  for (i = 0; i<num_spes; i++){
      if (spe_context_destroy (id[i]) != 0){
          perror("Failed destroying context");
          exit (1);
      }
  }
```

5.1.4 Control Block Structures

The next important section of the PPE code is the *Control Block* (CB) structure. There are two CB structures - *CONTROL_BLOCK_MAIN* and *CONTROL_BLOCK_DATA_ADDR*. The *Control Block* structure provides information about *SPE tables* to their respective SPEs. SPEs use this information to execute the *data transactions*.

5.1.4.1 CONTROL_BLOCK_MAIN

The *CONTROL_BLOCK_MAIN*, as the name suggests, is the main CB structure that holds important information like the transactions (source and destination SPU ids), the size (size of the DMA between them) of these transactions, DMA exist value etc. After the respective arrays are filled through the User Interface, PPE creates tables for the SPEs with complete information of all the entered transactions. In addition to creating the CB for the SPEs, the PPE also writes the CB addresses into the LS of the SPEs. Providing the CB address to the SPEs makes it convenient for them to access all the information pertaining to its schedule.

5.1.4.2 CONTROL_BLOCK_DATA_ADDR

The *CONTROL_BLOCK_DATA_ADDR* structure is sent to all the SPES from the PPE. This structure has only one member *dma_addr_array[i]* which holds the local store address (LSA) of all the SPEs. *Destination SPU ids* require the *LSA* of the *source SPU ids* to enable them to DMA data from their respective *source SPU id's LS*.

5.1.5 DMA Exist Array

DMA exist array is an array of type *int* and is of size 6, as the maximum number of SPEs that can be enabled is 6. The first element of the array is the DMA exist value of SPE0, the second element of the array is the DMA exist value of SPE1 and so on till SPE5. According to our implementation, the SPEs that do not receive any data (i.e., not a destination for any transaction in the framework) are not required to enter a certain part of the SPE code (interrupt handler section). If the SPEs are required to send data, then they do their job and return the control back to the PPE. Thus, the SPEs which *do not* receive any data *need not* execute the complete code on SPE. If a SPE receives data, then its DMA exist value is loaded into the respective array element as 7, else it is loaded as 0. In other words, DMA exist value tells the SPE it is a destination for a certain *data transaction*. On the PPE side the *DMA exist array* is filled based on the PPE table entered through the User Interface. All SPEs receive their DMA exist array value (either 7 or 0) through their respective CBs.

Another version of the implementation was also developed where PPE creates

SPE contexts for the SPEs and loads them with their individual *SPE Tables*. These *SPE Tables* are created using *PPE Table* where they contain only those schedules in which they are involved.

5.2 Scheduling Framework: SPE Side

SPEs are responsible for compute-intensive tasks. It is critical to implement a framework that ensures data transfer among the SPEs is fast and efficient. Keeping this in mind, I developed two types of framework (see Figure 5.1) - polling based and interrupt based. I focused on the latter because after experimentation it was found to be faster, better optimized and efficient.

5.2.1 Types of Framework

Initially I implemented a polling based scheduling framework. However, after analyzing its performance several disadvantages were identified. Consequently, I later switched to an interrupt-based scheduling framework. These are briefly discussed below.

5.2.1.1 Scheduling Framework : Polling Based

In the initial implementation of the scheduling framework, the program on the SPE side was polling based, where the SPEs poll at the *time base register*. The SPEs poll at *time base register* waiting for the correct time to execute DMA. Thus, the concept of polling results in some disadvantages: (1) as the program polls on the time base register the SPEs cannot perform any work, (2) the SPEs waste a lot of computation time in waiting for correct time to start the DMAs. In the newer version of the implementation we have enhanced the code so that programs on the SPE side, besides running the schedule (transactions) and transferring data at the precise time required, can also do their work. This has been accomplished by making the SPE program *interrupt based*. In addition to the SPEs doing their work, the framework of the interrupt handler also increases the predictability and robustness of the code and the application as a whole.

5.2.1.2 Scheduling Framework: Interrupt Based

As mentioned above, the scheduling framework implemented is *interrupt* based. To implement the interrupts we need to take advantage of certain hardware and software features of the CBEA like the interrupt handlers, barrier function touching all pages of target buffer etc. Making the framework interrupt based enables the SPEs to do their work (described in the work function below) when they are not scheduled to execute any transaction. This prevents the SPEs from going into an idle state (in Cell Broadband Engine Architecture idle state means going into a sleep mode). If SPEs go into an idle state they need to be awakened at the required time to execute the transactions, and reawakening of the SPEs is considered an expensive job on CellBE's platform. Thus, we should try to avoid it under all circumstances.

Another important aspect that needs to be ensured while transferring data among SPEs is to guarantee all SPEs begin executing data transactions at the same time in the required slot. To enable this I have developed a barrier function (described in sections below) that provides a barrier to SPEs until they are all are ready to execute simultanously. The Scheduling Slot structure, as the name suggests, provides the SPEs with the information of its schedule (when they need to transfer data and when not). The work SPEs are required to do is provided in the Work Function.

While transferring data from one SPE to another SPE, sometimes accesses to the main memory are made, which is not only time consuming but also results in TLB and cache misses. To avoid these misses the code was enhanced which is described in Section 5.2.1.6.

Interrupt Handlers The CBE's software development kit's (SDK) SPU Timer Library [14] provide services like timer and virtual clock for SPU programs to help us implement the interrupt handler.

The *Interval Timers* in the *SPU Timer Library* enable us to register a user-defined handler that can be called at a specific interval, whose value is also set by the user. The specific interval is the time after which the *Interval Timer* expires and it sends an interrupt to the SPE. In our framework the Interval Timer is set to 1 ms value; that is, SPE receives an interrupt every 1 ms (called slot). When the SPE receives an interrupt, it goes into its interrupt handler and executes the next *data* transaction based on the schedule. After the *data* transaction is executed,

the SPE restores the state and resumes its work. This continues till all the *data* transactions on the SPE side are completed. Virtual clock is a simple, software managed, monotonically increasing *time base* counter. It can count up to a value of 79 800 000 in one second. This number is also called the time base frequency (79 800 000 Hz).

For servicing the timer requests, the clock and timer services require the use of first and second level interrupt handlers (FLIH and SLIH, respectively). The SPU library provides both FLIH and SLIH for handling decrementer interrupt. The use of the library-supplied SLIH is required for using the clock and timer services.

First Level Interrupt Handler (FLIH) For implementing our framework, we have used the library supplied FLIH. To use it we need to call the provided spu_slih_reg() service to register spu_clock_slih() as the SLIH for the MFC_DECREMENTER_EVENT. This service is a part of the library FLIH and the symbol reference to it causes it to be linked into the application. However, if we wish to use our own FLIH, we must register the spu_clock_slih() using our own mechanism. In our implementation, we have used our own FLIH, and I have registered it in the spu_slih_reg(). The decrement counter has been given a value of 1 ms. Thus for every 1 ms the program gets an interrupt and it goes into the FLIH and executes the code in the handler. After executing it, the decrement count is reset to the original value and the timer is restarted and the execution comes out of the handler. This continues until the interrupts are disabled. It can be done using the i_disable() function. Similarly, when we initialize the decrement counter and start the timer, we enable the interrupts as well (for the program to get interrupts) using i_enable() function.

Second Level Interrupt Handler (SLIH) The *spu_slih_reg*, as mentioned above, is the SPU second level interrupt handler (SLIH) manager. This file *spu_slih_reg.h* consists of an interrupt handler dispatch table (spu_slih_handlers), an interrupt handler registration routine (spu_slih_reg), and a default interrupt handler (spu_default_slih). This file is readily available in the CBEA platform.

5.2.1.3 Barrier Function

As the name suggests, the barrier_heavy function provides a barrier, a compulsory wait on SPEs from progressing until all the SPEs are ready to start executing at

the same time. Even if one SPE is delayed for some reason, all the other ready SPEs have to wait for it. This function is important because it is neccessary to make sure that all the SPEs start at the same instant (have all SPEs start their DMAs at the same time) and also as it helps calculate the time taken by these DMAs more accurately. Thus, the purpose of the barrier_heavy function is to provide robustness and to maintain synchrony in order to make sure all SPEs start simultaneously. The file *barrier_heavy.h* is a part of the SPE code, and the function barrier_heavy receives the value of *nprocs* or *num_spes* from the PPE.

5.2.1.4 Work Function

Besides the interrupt handler, the work function *work()* is an important part of the SPE code. In this function the SPE performs its actual work. Every 1 ms the SPE receives an interrupt from the interrupt handler. In response to the interrupt, the SPE stops doing its work, there is a context switch, and the code for the interrupt is loaded and executed. After servicing the interrupt, the execution returns back to the original state at which it was interrupted. The SPE resumes doing its work from there (in the *work()* function) till it is interrupted again after 1 ms.

5.2.1.5 SCHEDULING_SLOT Structure

SCHEDULING_SLOT structure holds the schedule for all *destination SPU ids*. The structure has slots which are scheduled based on the algorithm that is computed offline. It might happen that some slots are scheduled while some are not. The size of data scheduled in each slot is called the *element transaction*, which is fixed at 10 kB.

5.2.1.6 Touch all Pages of Target Buffer

According to processor architecture, whenever there is a translation lookaside buffer (TLB) miss or a page table miss, the application has to fetch the data from the main memory, which takes a large number of cycles in comparison to the fetch from the cache. This will delay the task at hand and cause deadline misses. The same applies to our case as well, where a TLB miss can cause a delay in the execution of an *element* transaction, which in turn can cause deadline misses. To mitigate this problem we made sure that the scheduling framework touches all the

pages of the target buffer (data to be DMA*ed* from source SPE) well ahead of time before the actual access to the data happens. This way the data will sit in the TLB well ahead of time, and an access to this data will always be a hit and unnecessary delays can be avoided.

5.3 Development Structure

The development structure or the code structure describes how the application code is arranged. The application is divided into two main directories - PPU (the main directory containing the PPE code) and SPU (a subdirectory of PPU). The PPU directory contains the file simple.c which creates the contexts for the SPEs, loads the program (simple_spu) onto the SPE contexts and starts running them. The SPU subdirectory contains file simple_spu.c which has the code for executing *data transactions* (transferring of data at the scheduled time). It also contains three more files - barrier_heavy.h, spu_slih_reg.c and spu_slih_reg.h - whose purposes have already been described in detail above. There are two main Makefiles in the application - one is the PPU Makefile and the other is the SPU Makefile.

5.3.1 Compiling and Running

Once you are in the main directory (PPU) to compile the application, you need to type *make clean; make* and press enter. This creates a binary ELF (Extensible Linking Format) image named *simple*. Additionally, the name of the ELF image can be anything of your choice. As PPE and SPE have different instruction sets, they are compiled using separate libraries. The SPU directory is first compiled and its executable is added to the PPU library. Then the PPU Makefile compiles the PPU directory and generates the final ELF image - *simple*. To run the application you need to type *./*$< binaryELFimagename >$ which in our case is *simple*. Typing *./simple* runs the application.

5.4 Flow of Implementation Code

We now explain the flow of implementation code using an example. This example executes the schedules on the framework from the schedule table that was

generated offline using the real-time algorithm. Please refer to Figure 5.2 to better follow the example.

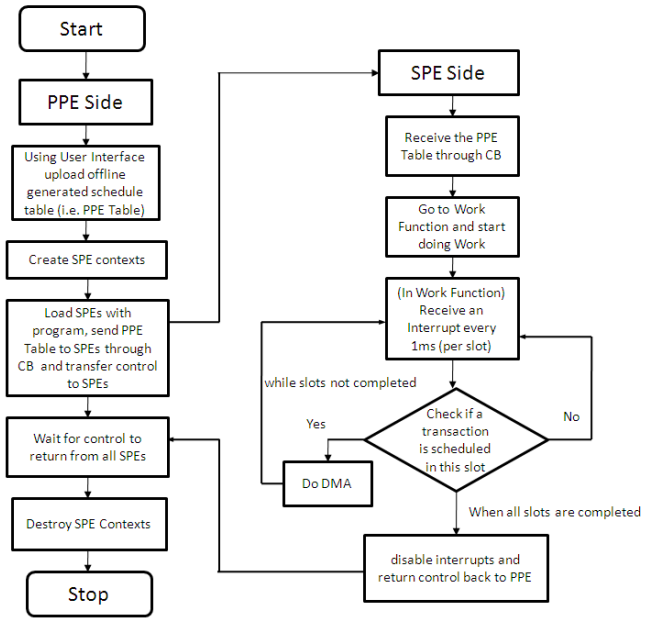

Figure 5.2: Flow of Implementation Code

On the PPE side, the schedule table that is generated offline is uploaded into PPE through the User Interface as shown in Figure 5.3.

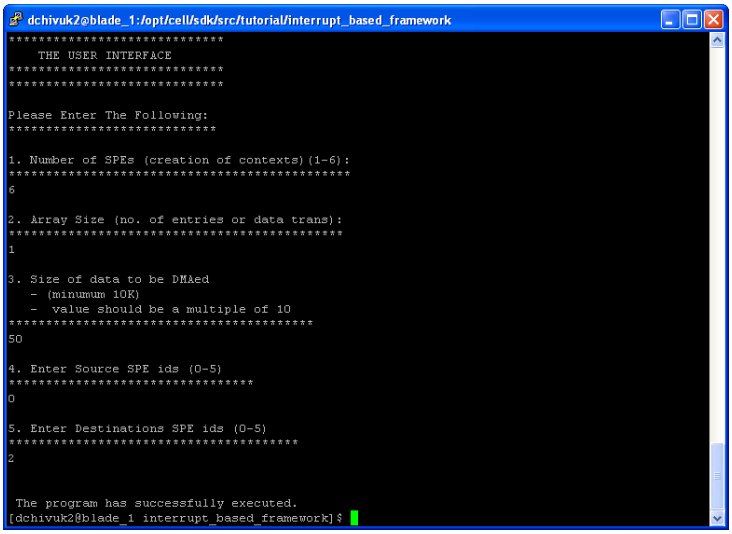

Figure 5.3: Snapshot of User Interface

After uploading the schedule table, PPE creates contexts for SPEs and loads the program context (contains code and data) into them. Next PPE sends CB

Table 5.1: Schedule Table

SNo.	Source SPU id	Destn SPU id	Execution time(slots)
1.	0	2	3

structures (MAIN and DATA_ADDR) containing the PPE table and details for DMAing to all the SPEs.

Coming to SPE side, they receive the CB structures containing the schedule table (see Table 5.1 as an example) and go into their work functions, after source SPEs load data into their LS and destination SPEs touch all pages of the target buffer. In work function all SPEs do their work until the Interval Timer expires (every 1 ms) and SPEs go into their own interrupt handlers. The scheduling slot structure contains the required schedule details for transferring data by SPEs in every slot.

After going into interrupt handler, the SPE checks the schedule table to see whether it has a DMA scheduled. If yes, the timer is first reset and SPE0 does an mfc_get of 10 kB from SPE2 and returns control to the work function. If it does not need to do a DMA, the timer is reset and control still returns to work function where the SPE continues doing its work, until it is interrupted again. This repeats for all the slots according to the executed schedule. Once scheduling is complete, SPEs return the control back to PPE, where it waits for all the SPEs and destroys their contexts before exiting.

5.5 Chapter Conclusion

In this chapter, I described the interrupt based scheduling framework in detail. I talked about the different versions of code that were implemented during the entire project. I also illustrated the flow of code by using an example.

CHAPTER 6

EXPERIMENTAL RESULTS, CONCLUSION AND FUTURE WORK

In this chapter, we discuss the experiments conducted with the bus scheduling engine on a real system and analyze the obtained results. More importantly, we show the effect of scheduling enforcement on the real-time behavior of bus transactions.

6.1 Experimental Setup

As already mentioned in Chapter 5, I chose Sony Play Station 3 (PS3) as the experimental platform. I implemented a scheduling engine, one instance of which runs on each processing element (SPEs). The engine is in fact a timer based interrupt handler. When an interval timer expires it fires an interrupt at the beginning of each time slot. The scheduling engine makes a scheduling decision for the current slot. The scheduling decision is made based on a scheduling table which is generated off-line by the POGen algorithm or by executing the algorithm itself. I chose a table based approach in the following experiments. If a transaction is scheduled in a slot, a DMA packet of the transaction with a given size will be transferred in that slot. The size of a DMA packet is dictated by the bus bandwidth and the slot size.

Although the CBEA bus has two rings in each direction, there exist transaction layouts that use only one ring in each direction. I chose one of those layouts for my experiments as the algorithm assumes EIB has one ring in each direction. In addition, all transactions in the experiments have endpoints that are SPEs.

In the first experiment I determined the slot size and the amount of data to be transferred in each slot. This was measured by transmitting packets of various sizes. The next experiment I conducted was to show the effect of real-time scheduling enforcement.

Table 6.1: DMA Transmission Time

DMA size (KB)	8	10	12	14	16
Transmission time (ticks)	34	41	48	54	60

6.2 Experimental Results

The first experiment conducted was to determine the slot size and the amount of data to be transferred in each slot. We determined it by measuring the transmission time of DMA packets with various sizes. The results are shown in Table 6.1, where a tick is a SPE real-time clock tick and is equal to 12.5 ns. The transmission times are rounded up to the smallest integral number of ticks. Referring to Table 6.1, we can see that the bus achieves a higher bandwidth when the DMA packet size is bigger. Based on the above measurement, we can also show that the time a SPE spends to execute a timer interrupt handler to transfer a DMA packet is 2 ticks. Thus, in the following experiments I chose a slot size to be 43 ticks and DMA packet size in each slot to be 10 kB. The slot size includes 41 ticks of a 10 kB DMA packet transmission time and 2 ticks of the interrupt handler processing overhead.

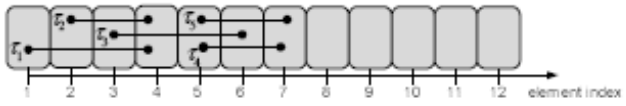

Figure 6.1: Experimental Transaction Set

Next, to show the effect of the real-time scheduling enforcement, I performed an experiment with a transaction set with five transactions whose layout is shown in Figure 6.1 and with values (the time unit is the number of slots) of timing parameters equal to $\tau_1 : e_1 = 4, p_1 = 20; \tau_2 : e_2 = 6, p_2 = 10; \tau_3 : e_3 = 6, p_3 = 60; \tau_4 : e_4 = 6, p_4 = 10; \tau_5 : e_5 = 4, p_5 = 20$. The transaction set has $\mathsf{L} = 10$ and two PO-sets: $\mathcal{D}_1 = \{\tau_1, \tau_2, \tau_3\}$ and $\mathcal{D}_2 = \{\tau_3, \tau_4, \tau_5\}$. The utilization of each PO-set is $9/10 = (\mathsf{L}-1)/\mathsf{L}$. In the first experiment, we initiate each periodic transaction as soon as it arrives in the system; hence the transactions are scheduled by the low-level round-robin bus arbiter. As a result, transactions τ_2 and τ_4 *miss* their deadline with the maximum relative completion time of 12 slots as opposed to their relative deadline of 10 slots. When the transaction set is scheduled by

POGen, all transactions *meet* their deadlines.

6.3 Conclusion

The main research issue is to how to provide software designers with: (1) a practical and accurate abstraction of the real scheduling problem on multiprocessor bus and (2) an effective scheduling methodology that maximizes multiprocessor bus utilization. The present research focuses on addressing this problem on a specific multiprocessor bus architecture, specifically CBEA. In this thesis, I developed an interrupt based scheduling framework that abstracts away from the low-level physical bus implementation, to provide a platform to implement and test the performance of a class of scheduling algorithms. I have also conducted experiments to show that the real-time transactions of feasible transaction sets are executed before deadline when scheduled according to a real-time scheduling algorithm, while the same transactions can miss their deadlines when scheduled according to an arbitrary (non-real-time) scheduling policy.

I also presented a detailed discussion about hardware and software features of the Cell processor and its advantages when compared to other modern computer architectures.

6.4 Future Work

In our future work we would like to (1) schedule on-line the *data* transactions instead of having the algorithm computed offline, so we would like to have a scheduling algorithm implemented into the scheduling framework itself; (2) address the issue of transforming circular transaction sets into non-circular ones and in order to apply the proposed scheduling framework to circular transaction sets.

APPENDIX A

IMPLEMENTATION CODE AND BIT ORDERING IN CELL PROCESSOR

A.1 PPU: simple.c

```
/* ------------------------------------------------------------ */
/* All Rights Reserved.                                         */
/*                                                              */
/*      University of Illinois at Urbana Champaign              */
/*          Department of Computer Science                      */
/*           Real-Time Systems Laboratory                       */
/*                                                              */
/*                                                              */
/* - Deepti Kumar Chivukula - dchivuk2@illinois.edu             */
/*                                                              */
/*                                                              */
/*                                                              */
/*          BUS SCHEDULING ON THE CELL PROCESSOR                */
/*                                                              */
/* - File Name: simple.c                                        */
/*                                                              */
/* This is the PPE side code                                    */
/*  - initializes all the arrays,                               */
/*  - includes the 'User Interface' for the user                */
/*    to enter the parameters of the tasks.                     */
/*  - creates, loads and runs the SPEs threads                  */
/*    and wait for them to complete. Also destroys              */
/*    them in the end.                                          */
/*  - contains the Control Block structures to                  */
/*    create SPE tables                                         */
/*                                                              */
/* ------------------------------------------------------------ */
/* PROLOG END TAG zYx                                           */

#include <stdlib.h>
```

```
#include <stdio.h>
#include <stdbool.h>
#include <errno.h>
#include <libspe2.h>
#include <pthread.h>
#include <sys/types.h>
#include <libmisc.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <sched.h>
#include <string.h>
#include <sys/time.h>
#include <dirent.h>


/*** Constants ***/
#define ARRAY_SIZE 30  //All arrays are static and of type int -
    max value is 30
#define NUM_ELEMENTS 4096  //Maximum DMA size is 16KB [(4096 *
   4) = 16KB]
#define NUM_SPES 6  //Max number of SPEs that can be enabled is
   6
#define CACHE_LINE_SIZE 128  //Required for file barrier_heavy.h


/*** Name of SPE thread created - simple_spu ***/
extern spe_program_handle_t simple_spu;


/**** DECLARING MAIN Control Block (CB)  ****/
typedef struct {
unsigned long* spu_source_ls;
unsigned long* spu_destn_ls;
long spu_dma_data_array[ARRAY_SIZE];
long spu_source_ls_array[ARRAY_SIZE];
long spu_destn_ls_array[ARRAY_SIZE];
long spu_source_id_array[ARRAY_SIZE];
long spu_destn_id_array[ARRAY_SIZE];
long spu_destn_id_numbers[ARRAY_SIZE];
long spu_source_id_numbers[ARRAY_SIZE];
long periods[ARRAY_SIZE];
```

```
long spu_num_jobs[ARRAY_SIZE];
int spu_array_size;
int task;
int dma_exist;
int nprocs;
int speid_cb;
int *barptr;
int id_spu;
unsigned char pad[36];
}CONTROL_BLOCK_MAIN;


/*** CB structure for Local Store Address ***/
typedef struct {
   long dma_addr_array[ARRAY_SIZE];
   unsigned char pad[8];
}CONTROL_BLOCK_DATA_ADDR;


/*** Make CB address - 128-byte aligned ***/
CONTROL_BLOCK_MAIN cb __attribute__((aligned(128)));
CONTROL_BLOCK_DATA_ADDR cbdata __attribute__((aligned(128)));


/*** Cache-line sized blocks for use in barrier calls - 128-byte
     aligned ***/
static unsigned int bar[CACHE_LINE_SIZE/
             sizeof(unsigned int)] __attribute__((aligned (128))
                 );

/*** Declaring Global Variables ***/
unsigned int id[NUM_SPES];
pthread_t threads[NUM_SPES];
int array_size;
int num_spes;
int num_destination_ids;


/**** Source and Destn SPU values for the schedule table ****
************************************************************/
long spu_source_ls[ARRAY_SIZE] =
   {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
         -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
```

```
long spu_destn_ls[ARRAY_SIZE] =
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
            -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
long spu_source_control[ARRAY_SIZE]=
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
            -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
long spu_destn_control[ARRAY_SIZE] =
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
            -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
long spu_dma_data[ARRAY_SIZE] =
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
            -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
long data_size[ARRAY_SIZE] =
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
            -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
long spu_dma_data_array[ARRAY_SIZE] =
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
            -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};

long dma_addr[ARRAY_SIZE] =
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
            -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
long spu_source_id[ARRAY_SIZE] =
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
            -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
long source_ids[ARRAY_SIZE] =
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
            -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
long destn_ids[ARRAY_SIZE] =
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
            -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
long spu_destn_id[ARRAY_SIZE] =
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
            -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
long ppe_periods[ARRAY_SIZE] =
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
            -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
long num_jobs[ARRAY_SIZE] =
    {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
            -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
long dma_exist_array[NUM_SPES]= {-1, -1, -1, -1, -1, -1};
```

```
/*** SPE CONTEXT RUN FUNCTION ***
*******************************/
void *ppu_pthread_function(void *arg) {
  spe_context_ptr_t ctx;
  unsigned int entry = SPE_DEFAULT_ENTRY;
  ctx = *((spe_context_ptr_t *)arg);
  if (spe_context_run(ctx, &entry, 0, NULL, NULL, NULL) < 0) {
    perror ("Failed running context");
    exit (1);
  }
  pthread_exit(NULL);
}


/*** Initialize all the arrays ***
*********************************/
void initialize() {
  int i;
  for( i = 0; i<array_size; i++ ){
    cb.spu_dma_data_array[i]= -1;
    cb.spu_source_ls_array[i]= -1;
    cb.spu_destn_ls_array[i] = -1;
    cb.spu_source_id_array[i]= -1;
    cb.spu_destn_id_array[i]= -1;
    cb.spu_destn_id_numbers[i]= -1;
    cbdata.dma_addr_array[i] = -1;
    cb.periods[i] = -1;
    cb.spu_num_jobs[i] = -1;
  }
}


/*** To convert the Size of data from kilobytes to bytes ***
************************************************************/
void size_convert() {
  int i = 0;
  for(i = 0; i<array_size; i++){
    spu_dma_data[i] = (data_size[i] * 1024) ;
  }
}


/*** Period conversion from  millisecs to number of ticks ***
```

```
*************************************************************/
void period_conversion(){
  int i;
  //one ms has 79800 ticks
  for(i = 0; i<array_size; i++){
    ppe_periods[i] = (79800/ppe_periods[i]);
  }
}


/*** Function to calculate number of jobs in transactions ***
*************************************************************/
void num_of_jobs_calc(){
  int r;
  for(r = 0; r < array_size; r++){
    num_jobs[r] = (unsigned int) (data_size[r]/10) ;
    //we have assumed size of job to be 10KB
  }
}


/*** THE MAIN FUNCTION ***
************************/
int main()
{
  int i=0, l=0, j=0, k=0;
  int temp_source ;
  int temp_destn ;
  int rc;
  unsigned int cb_array[1];
  unsigned int cb_data_addr[1];
  unsigned long garbage;
  unsigned long spe_dma_data_addr[NUM_SPES] ;
  unsigned long spu_source_ls_dma[NUM_SPES];
  unsigned long spe_dma_addr[NUM_SPES];
  int start_schedule[1]={1000};
  unsigned int temp_id[NUM_SPES];
  int c_garbage = 0;
  int c_garbage_cblsa = 0;
  int getBarID;


  /***      Fill in CB        ****
```

```
 - with addr of the data array
 - with the address of the cb
*****************************/
cb_array[0]=&cb;
cb_data_addr[0]=&cbdata;


/*** USER INTERFACE FOR FILLING PARAMETERS OF THE TASKS ***
************************************************************/
printf("****************************\n");
printf("****************************\n");
printf("    THE USER INTERFACE  \n");
printf("****************************\n");
printf("****************************\n\n");

printf("Please Enter The Following:\n");
printf("****************************\n\n");


printf("1. Number of SPEs(creation of contexts)(1-6):\n");
printf("*********************************************\n");
scanf("%d",&num_spes);
fflush(stdin);
printf("\n");
fflush(stdout);

printf("2. Array Size(no. of entries or data trans):\n");
printf("*********************************************\n");
scanf("%d",&array_size);
fflush(stdin);
printf("\n");
fflush(stdout);

printf("3. Size of data to be DMAed\n");
printf("   -(minumum 10K) \n");
printf("   -  value should be a multiple of 10 \n");
printf("*************************************\n");
for (i = 0; i<array_size; i++) {
scanf("%d", &data_size[i]);
fflush(stdin);
}
printf("\n");
fflush(stdout);
```

```
printf("4. Enter the Source SPE ids (0-5)\n");
printf("********************************\n");
for (i = 0; i<array_size; i++) {
scanf("%d", &spu_source_id[i]);
fflush(stdin);
}
printf("\n");
fflush(stdout);

printf("5. Enter the Destinations SPE ids (0-5)\n");
printf("************************************\n");
for (i = 0;i <array_size; i++) {
scanf("%d", &spu_destn_id[i]);
fflush(stdin);
}
printf("\n");
fflush(stdout);



/*** Filling the DMA EXIST array for the Destn Spus ***
*****************************************************/
for(i = 0; i<num_spes; i++) {
  for(j=0;j<array_size;j++){
    if(spu_destn_id[j]== i)
      dma_exist_array[i]= 7;
  }
}


/*** Initialize all the arrays used to fill CB ***/
initialize();


/*** Convert Size in KB to Number of elements for array ***/
size_convert();


/*** To calculate number of jobs ***
***** in every data transaction ***/
num_of_jobs_calc();
```

```
/*** Do SPE context create and load it into SPE ids (0-6) ***
************************************************************/
for(i = 0; i<num_spes; i++) {
  /*** Create contexts ***/
  if ((id[i] = spe_context_create (SPE_MAP_PS, NULL)) == NULL)
      {
    perror ("Failed creating context");
    exit (1);
  }

  /*** load the SPEs with the program - simple_spu ***/
  spe_program_load (id[i], &simple_spu) ;
}

/*** Pthread Creation ***
************************/
for(i = 0; i < num_spes; i++) {
  /*** Do PPU thread_create and context run ***/
  if (pthread_create (&threads[i], NULL, &ppu_pthread_function
      , &id[i])){
    perror ("Failed creating thread");
    exit (1);
    }
}


/*** Fill the Source and Destination arrays according
************** to Schedule Table ********************/
for(i = 0; i<array_size; i++)
{
  temp_source = spu_source_id[i];
  source_ids[i] = spu_source_id[i];
  spu_source_id[i] = id[temp_source];
  temp_destn= spu_destn_id[i];
  destn_ids[i] = spu_destn_id[i];  //required for comparison
     on SPE side to do mfc_get on destn ids
  spu_destn_id[i] = id[temp_destn];

  /*** load the array with ls of the spes according to
  ***************** schedule table  *****************/
  spu_source_ls[i] = spe_ls_area_get(spu_source_id[i]);
```

```
  spu_destn_ls[i] = spe_ls_area_get(spu_destn_id[i]);
  spu_source_control[i] = (spe_spu_control_area_t *)
      spe_ps_area_get
          (spu_source_id[i], SPE_CONTROL_AREA);
  spu_destn_control[i] = (spe_spu_control_area_t *)
      spe_ps_area_get
          (spu_destn_id[i], SPE_CONTROL_AREA);
}


/*** Loading SPEs with its respective CB ***
*****************************************/
cb.speid_cb = -1;
for(i = 0; i<num_spes; i++)
{
  k = 0;
  cb.barptr = &bar;
  cb.id_spu = i;

  /*** We assume that Destination is not repeated ***/
  cb.nprocs = num_destination_ids;
  getBarID=0;
  for(j = 0; j<array_size; j++)
  {
    cb.spu_dma_data_array[j]= spu_dma_data[j];
    cb.spu_source_ls_array[j] = spu_source_ls[j];
    cb.spu_destn_ls_array[j] = spu_destn_ls[j];
    cb.spu_source_id_array[j] = spu_source_id[j];
    cb.spu_destn_id_array[j] = spu_destn_id[j];
    cb.spu_destn_id_numbers[j] = destn_ids[j];
    cb.spu_source_id_numbers[j] = source_ids[j];
    cb.periods[j] = ppe_periods[j];
    cb.spu_num_jobs[j] = num_jobs[j];
    cb.task = i;

    //init bar params
    if(!getBarID){
    cb.speid_cb++;
    getBarID=1;
    }
  }
```

```
  /*** Fill the array size for each individual SPE ***/
  cb.spu_array_size=array_size;
  cb.dma_exist = dma_exist_array[i];


  /*** Initializing other entries in arrays to -1 ***/
  for(l = array_size; l<ARRAY_SIZE; l++)
  {
    cb.spu_dma_data_array[l]= -1 ;
    cb.spu_source_ls_array[l] = -1 ;
    cb.spu_destn_ls_array[l] = -1 ;
    cb.spu_source_id_array[l] = -1 ;
    cb.spu_destn_id_array[l] = -1 ;
    cb.spu_destn_id_numbers[l] = -1 ;
    cb.spu_source_id_numbers[l] = -1 ;
    cb.periods[l] = -1 ;
    cb.spu_num_jobs[l] = -1 ;
  }
  /*** Write CB to Local Storage ***/
  rc = spe_in_mbox_write(id[i], &cb_array, 1,
     SPE_MBOX_ANY_BLOCKING );


  /***      Wait for SPE to read CB      ***
  ***(read garbage value on PPE side) ***/
  while( ! spe_out_mbox_status(id[i]));
  spe_out_mbox_read(id[i], &garbage, 1);
  c_garbage ++;

}


/*** To get Local Store Data Array ***
***** addresses of all source SPES ***/
for(i = 0 ;i<array_size; i++){
  temp_source = source_ids[i];
  spu_source_ls_dma[i] = spe_ls_area_get(id[temp_source]);


  while( ! spe_out_mbox_status(id[temp_source]));
  spe_out_mbox_read(id[temp_source], &spe_dma_data_addr[i], 1)
     ;
```

```
  /*** put it into a different array ***/
  spe_dma_addr[i] = spu_source_ls_dma[i] + spe_dma_data_addr
      [i];
}


for(i = 0; i<num_spes; i++){
  for(j = 0; j<array_size; j++){
    cbdata.dma_addr_array[j] = spe_dma_addr[j];
  }


  /*** Initialiing other entries in arrays to -1 ***/
  for(l = array_size; l<ARRAY_SIZE; l++) {
    cbdata.dma_addr_array[l] = -1 ;
  }


  /*** Write CB_LSA to Local Storage Area of destn SPU's only
      ***/
  rc = spe_in_mbox_write(id[i], &cb_data_addr, 1,
      SPE_MBOX_ANY_BLOCKING);


  /*** Wait for SPE to read CB (read garbage value on PPE side
      ) ***/
  while( ! spe_out_mbox_status(id[i]));
    spe_out_mbox_read(id[i], &garbage, 1);
    c_garbage_cblsa ++;
}


if(c_garbage_cblsa == num_spes){
  for(i = 0; i<num_spes; i++){
    /*** Send mailbox with start_schedule value to all spes
        ***/
    rc = spe_in_mbox_write(id[i], &start_schedule, 1,
        SPE_MBOX_ANY_BLOCKING);
  }
}
```

```
  /*** Wait for SPU-thread to complete execution ***
  **********************************************/
  for (i = 0; i<num_spes; i++) {
    if (pthread_join (threads[i], NULL)) {
      perror("Failed pthread_join");
      exit (1);
    }
  }


  /*** Destroy context ***
  **********************/
  for (i = 0; i<num_spes; i++) {
    if (spe_context_destroy (id[i]) != 0) {
      perror("Failed destroying context");
      exit (1);
    }
  }


  return (0);
  /*** END OF CODE ***/
}
```

A.2 PPU: cFile.c

```
/* ------------------------------------------------------------ */
/* All Rights Reserved.                                         */
/*                                                              */
/*       University of Illinois at Urbana Champaign             */
/*          Department of Computer Science                      */
/*            Real-Time Systems Laboratory                      */
/*                                                              */
/*                                                              */
/* - Deepti Kumar Chivukula - dchivuk2@illinois.edu             */
/*                                                              */
/*                                                              */
/*                                                              */
/*        BUS SCHEDULING ON THE CELL PROCESSOR                  */
/*                                                              */
/* - File Name: cFile.c                                         */
```

```
/*                                                                  */
/* ---------------------------------------------------------------- */
/* PROLOG END TAG zYx                                               */

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <libspe2.h>
#include <pthread.h>
#include <sys/types.h>
#include <libmisc.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <sched.h>
#include <string.h>
#include <sys/time.h>
#include <dirent.h>

#define ARRAY_SIZE 30
#define NUM_ELEMENTS 4096
#define NUM_SPES 6


#define CACHE_LINE_SIZE 128

extern spe_program_handle_t simple_spu;
```

A.3 PPU: Makefile

```
# ----------------------------------------------------------------
#  All Rights Reserved.
#
#       University of Illinois at Urbana Champaign
#           Department of Computer Science
#            Real-Time Systems Laboratory
#
#
# - Deepti Kumar Chivukula - dchivuk2@illinois.edu
#
```

```
#
#         BUS SCHEDULING ON THE CELL PROCESSOR
#
# - File Name: makefile.c
#
# ----------------------------------------------------------------
# PROLOG END TAG zYx

##########################################################
#                         Subdirectories
##########################################################

DIRS            :=      spu


##########################################################
#                         Target
##########################################################

PROGRAM_ppu     :=      simple


#OBJS_spu_interrupt      := spu_flih.o           \
                           spu_interrupt.o      \
                           spu_slih_reg.o

#OBJS_spu_interrupt_fast := spu_handler_fast.o   \
                           spu_interrupt_fast.o \




##########################################################
#                         Local Defines
##########################################################

IMPORTS         = -lspe2 spu/lib_simple_spu.a -lpthread

INSTALL_DIR     = \$(EXP_SDKBIN)/tutorial
INSTALL_FILES   = \$(PROGRAMS_ppu)

##########################################################
```

```
#                                buildutils/make.footer
#########################################################################

ifdef CELL_TOP
  include \$(CELL_TOP)/buildutils/make.footer
else
        include ../../../buildutils/make.footer
endif
```

A.4 SPU: simple_spu.c

```
/* ------------------------------------------------------------ */
/* All Rights Reserved.                                         */
/*                                                              */
/*        University of Illinois at Urbana Champaign            */
/*            Department of Computer Science                    */
/*              Real-Time Systems Laboratory                    */
/*                                                              */
/*                                                              */
/* - Deepti Kumar Chivukula - dchivuk2@illinois.edu             */
/*                                                              */
/*                                                              */
/*                                                              */
/*          BUS SCHEDULING ON THE CELL PROCESSOR                */
/*                                                              */
/* - File Name: simple_spu.c                                    */
/*                                                              */
/* This is the SPE side code                                    */
/*  - includes the interrupt based scheduling framework         */
/*  - contains the scheduling slot structure                    */
/*  - does DMA("mfc_get") from the source SPE ids               */
/*                                                              */
/* ------------------------------------------------------------ */
/* PROLOG END TAG zYx                                           */

#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <sys/types.h>
#include <spu_intrinsics.h>
#include <barrier_heavy.h>
```

```
#include <spu_slih_reg.h>
#include <spu_mfcio.h>
#include <malloc_align.h>
#include <libmisc.h>
#include <unistd.h>


/*** Constants ***/
#define NUM_ELEMENTS 1024 * 100
#define ARRAY_SIZE 30
#define NUM_SPES 6
#define CACHE_LINE_SIZE 128
#define MAX_COUNT 0x100000ULL
#define DECR_COUNT 79800
#define GB_to_B (1024*1024*1024)
#define KB_to_B (1024)
#define SIZE_LOCAL 10240
#define NUMBER_SLOTS 20


/*** Required to read the Time Base value ***/
spu_read_decrementer;
spu_write_decrementer;


/*** DECLARING MAIN Control Block (CB) ***/
typedef struct {
  unsigned long* spu_source_ls;
  unsigned long* spu_destn_ls;
  long spu_dma_data_array[ARRAY_SIZE];
  long spu_source_ls_array[ARRAY_SIZE];
  long spu_destn_ls_array[ARRAY_SIZE];
  long spu_source_id_array[ARRAY_SIZE];
  long spu_destn_id_array[ARRAY_SIZE];
  long spu_destn_id_numbers[ARRAY_SIZE];
  long spu_source_id_numbers[ARRAY_SIZE];
  long periods[ARRAY_SIZE];
  long spu_num_jobs[ARRAY_SIZE];
  int spu_array_size;
  int task;
  int dma_exist;
  int nprocs;
  int speid_cb;
```

```
  int *barptr;
  int id_spu;
  unsigned char pad[36];
}CONTROL_BLOCK_MAIN;


/*** CB structure for LSA ***/
typedef struct {
  long dma_addr_array[ARRAY_SIZE];
  unsigned char pad[8];
}CONTROL_BLOCK_DATA_ADDR;

/*** Schedule structure for Tasks***/
typedef struct{
  bool task[2]; // right now we have two tasks
  unsigned char pad [126];
}SCHEDULING_SLOT;

/*** Array of scheduling slots for schedule table***/
SCHEDULING_SLOT scheduling_table[20];

/*** Declaring arrays and making them 128-byte aligned ***/
char data_total[NUM_ELEMENTS] __attribute__((aligned(128)));
char data_receive[NUM_ELEMENTS] __attribute__((aligned(128)));
static char lsbuf[128] __attribute__ ((aligned (128)));

/*** Make CB address - 128-byte aligned ***/
CONTROL_BLOCK_MAIN cb1 __attribute__((aligned(128)));
SCHEDULING_SLOT ss1 __attribute__((aligned(128)));
CONTROL_BLOCK_DATA_ADDR cbdata1 __attribute__((aligned(128)));


/*** Filling the Scheduling Table ***/
void schedule_table(){
  scheduling_table[0].task[0] = true;
  scheduling_table[1].task[0] = false;
  scheduling_table[2].task[0] = false;
  scheduling_table[3].task[0] = true;
  scheduling_table[4].task[0] = false;
  scheduling_table[5].task[0] = false;
  scheduling_table[6].task[0] = false;
  scheduling_table[7].task[0] = true;
  scheduling_table[8].task[0] = false;
```

```
  scheduling_table[9].task[0] = false;
  scheduling_table[10].task[0] = true;
  scheduling_table[11].task[0] = false;
  scheduling_table[12].task[0] = false;
  scheduling_table[13].task[0] = false;
  scheduling_table[14].task[0] = true;
  scheduling_table[15].task[0] = false;
  scheduling_table[16].task[0] = false;
  scheduling_table[17].task[0] = false;
  scheduling_table[18].task[0] = false;
  scheduling_table[19].task[0] = false;
  scheduling_table[0].task[1] = true;
  scheduling_table[1].task[1] = false;
  scheduling_table[2].task[1] = false;
  scheduling_table[3].task[1] = true;
  scheduling_table[4].task[1] = false;
  scheduling_table[5].task[1] = false;
  scheduling_table[6].task[1] = true;
  scheduling_table[7].task[1] = false;
  scheduling_table[8].task[1] = true;
  scheduling_table[9].task[1] = false;
  scheduling_table[10].task[1] = false;
  scheduling_table[11].task[1] = true;
  scheduling_table[12].task[1] = false;
  scheduling_table[13].task[1] = false;
  scheduling_table[14].task[1] = false;
  scheduling_table[15].task[1] = false;
  scheduling_table[16].task[1] = false;
  scheduling_table[17].task[1] = false;
  scheduling_table[18].task[1] = false;
  scheduling_table[19].task[1] = false;
}


/*** Declaring Global variables ***
********************************/
int i=0, k=0;
int interrupt_id=0;
bool wait;
unsigned int destn_index = 0;
unsigned int source_index = 0;
unsigned int intrhdlr_index = 0;
unsigned int ticks_index = 0;
```

```
unsigned int counter = 0;
unsigned int ticks_1[NUMBER_SLOTS];
unsigned int ticks_2[NUMBER_SLOTS];
long size_temp=0;//temp variable
long size_array_temp[2];//temp variable
int incr_size = 0;


/*** Interrupts and Timers declaration ***
****************************************/
unsigned int decr_handler(unsigned int);


/*** Profiled Work Function ***
*****************************/
void work (unsigned long long id) {
  int nprocs, speid;
  void *barptr;

  /***Load the Scheduling Table ***/
  barptr = cb1.barptr;
  nprocs = cb1.nprocs;
  speid = cb1.speid_cb;

  /*** Calling schedule table before
  *** starting interrupt handler ***/
  schedule_table();

  /*** use SPU timer library FLIH and SLIH to implement
     Interrupts***/
  spu_slih_reg(MFC_DECREMENTER_EVENT, decr_handler); //every 1
     ms
  spu_writech(SPU_WrEventMask, MFC_DECREMENTER_EVENT);
  spu_writech(SPU_WrDec, DECR_COUNT);


  /*** wait until all SPEs are done ***/
  _barrier_heavy((unsigned int)barptr, speid, lsbuf, nprocs);


  /*** Enable the interrupts ***/
  spu_ienable();
```

```
  /*** Do work until interrupt comes ***/
  while(1){
    if(wait == true){
      mfc_write_tag_mask(1 << (cb1.task));
      mfc_read_tag_status_all();
      /*** Record end tick value ***/
      ticks_2[ticks_index] = spu_readch(SPU_RdDec);  //SPU timer
          library function
                                    //used to read the channel
                                       for number of ticks
      wait = false;
      ticks_index++;
    }
    if(intrhdlr_index == NUMBER_SLOTS+1) {
      break;
    }
  }

  /*** To print the number of ticks the DMA take ***/
  if(cb1.dma_exist == 7){
    for(i=0;i<ticks_index;i++){
      printf("\n ID = %d\t ticks[%d]= %d\t",
                        cb1.id_spu,i,ticks_1[i]-ticks_2[i]);
      fflush(stdout);

    }
  }
  return 0;
}


/*** THE MAIN FUNCTION ***
************************/
int main(unsigned long long id, void* argp, unsigned int envp) {
  unsigned long start_schedule;
  unsigned long *dma_data_addr;
  unsigned long garbage=1234;
  CONTROL_BLOCK_MAIN *cb_addr;
  CONTROL_BLOCK_DATA_ADDR *cb_data_addr;
  long tgt_addr;
  long tgt_incr;
  int j = 0;
```

```
int p = 0;
int q = 0;


/*** Wait for control block address from PPE ***/
cb_addr  = (CONTROL_BLOCK_MAIN *)spu_read_in_mbox();


/*** DMA over control block  & wait until done ***/
mfc_get(&cb1, (unsigned int)cb_addr, sizeof(cb1), 5, 0, 0);


/*** Mask out tag we are interested in ***/
mfc_write_tag_mask(1 << 5);


/*** Wait for DMA completion ***/
mfc_read_tag_status_all();

/*** SPE sends to PPE garbage value after receiving CB address
     ***/
spu_write_out_mbox(garbage);


/*** Creating Array of numbers to be DMAed
******  between SEND and RECEIVE SPE ***/
for(i=0;i<cb1.spu_array_size;i++){
  size_array_temp[i] = cb1.spu_dma_data_array[i];

}

/*** Determining the destn_id_index ***/
for (p=0;p<cb1.spu_array_size;p++) {
  if(cb1.spu_destn_id_numbers[p]== cb1.id_spu){
    destn_index = p;
    break;
  }
  else
    destn_index = -1;
}

/*** Determining the source_id_index ***/
for (q=0; q<cb1.spu_array_size; q++) {
```

```
  if(cb1.spu_source_id_numbers[q]== cb1.id_spu) {
    source_index = q;
    break;
  }
  else
    source_index = -1;
}

/*** Filling the Source SPEs with SPEID
***** specific data that is DMAed ***/
if(cb1.spu_source_id_numbers[source_index] == cb1.id_spu){
  for(i = 0; i < size_array_temp[source_index]; i++){
    data_total[i] = ((id & (0x000000f0))>> 4 )+ 1 ;
  }

  /*** Address of SPE data array ***/
  dma_data_addr = (unsigned long)&data_total[0];

  /*** Send mailbox with SEND SPE's
  **** data_array_addr to PPE ***/
  spu_write_out_mbox(dma_data_addr);
}


/*** Wait for control block adress from PPU ***/
cb_data_addr  = (CONTROL_BLOCK_DATA_ADDR *)spu_read_in_mbox();


/*** DMA over control block  & wait until done ***/
mfc_get(&cbdata1, (unsigned int)cb_data_addr, sizeof(cbdata1),
    5, 0, 0);


/*** Mask out tag we are interested in ***/
mfc_write_tag_mask(1 << 5);


/*** Wait for DMA completion ***/
mfc_read_tag_status_all();


/*** SPE sends to PPE garbage value
*** after receiving CB address ***/
```

```
  spu_write_out_mbox(garbage);


  /*** Read start_schedule value from the PPE ***/
  start_schedule  = spu_read_in_mbox();


  /*** Function to Call the Work Function ***/
  if(start_schedule == 1000){
    if(cb1.spu_destn_id_numbers[destn_index]== cb1.id_spu){

      /*** Touch each page of the target buffer so that ****
      ******* the page tables and TLBs are all loaded up ***/
      for(i=0 ;i<cb1.spu_array_size;i++){
        tgt_addr= (unsigned long)cbdata1.dma_addr_array[i];
        tgt_incr = 4096;        /*** one page ***/
        size_temp = cb1.spu_dma_data_array[i];
        if(size_temp < 4096)
          size_temp = 4096;
        else
          size_temp = cb1.spu_dma_data_array[i];

        for (j=0; j<=size_temp; j+=tgt_incr) {
          mfc_get(&data_receive[0], (unsigned long)tgt_addr,
              128, 5, 0, 0);
          mfc_write_tag_mask(1 << 5 );
          mfc_read_tag_status_all();    /*** Wait for DMA to
              complete ***/
          mfc_put(&data_receive[0], (unsigned long)tgt_addr,128,
               5, 0, 0);
          mfc_write_tag_mask(1 << 5);
          mfc_read_tag_status_all();    /*** Wait for DMA to
              complete ***/
          tgt_addr += tgt_incr;
        }
      }
    }
    /*** SPE work ***/
    work(id);
  }
  return 0;
}
```

```
/*** Interrupt Handler ***
*************************/
unsigned int decr_handler(unsigned int status) {
  status &= ~MFC_DECREMENTER_EVENT ;


  /*** Resetting interrupt as soon as it enters the handler ***/
  /*** Reset Counter ***/
  spu_writech(SPU_WrDec, DECR_COUNT);

  /*** ALGORITHM FUNCTION ***
  *************************/

  if((cb1.spu_destn_id_numbers[destn_index]== cb1.id_spu) ) {
    if ((scheduling_table[intrhdlr_index].task[destn_index] ==
        1) &&
                                        (intrhdlr_index <
                                           NUMBER_SLOTS)){
      wait = true;

      /*** Record start tick value ***/
      ticks_1[ticks_index] = spu_readch(SPU_RdDec);

      /*** mfc_get - do DMA from source SPE ***/
      mfc_get(&data_receive[0]+ incr_size,
            (unsigned long)(cbdata1.dma_addr_array[destn_index]+
               incr_size),
                 SIZE_LOCAL, cb1.task, 0, 0);
      incr_size += SIZE_LOCAL; //incr to the next 10KB of data
    }
  }
  intrhdlr_index++ ;

  return (status);
}
```

A.5 SPU: Makefile

```
# ------------------------------------------------------------
#  All Rights Reserved.
```

```
#
#       University of Illinois at Urbana Champaign
#          Department of Computer Science
#           Real-Time Systems Laboratory
#
#
# - Deepti Kumar Chivukula - dchivuk2@illinois.edu
#
#
#         BUS SCHEDULING ON THE CELL PROCESSOR
#
# - File Name: makefile_spu.c
#
# ----------------------------------------------------------------
# PROLOG END TAG zYx

##########################################################
#                          Target
##########################################################

#OBJS              := simple_spu0.o simple_spu1.o
PROGRAMS_spu      := simple_spu #simple_spu0 simple_spu1
LIBRARY_embed     := lib_simple_spu.a


OBJS_simple_spu       := spu_flih.o            \
                         simple_spu.o       \
                        spu_slih_reg.o

#OBJS_spu_interrupt_fast := spu_handler_fast.o    \
                           spu_interrupt_fast.o \


INSTALL_DIR       = \$(EXP_SDKBIN)/tutorial
INSTALL_FILES     = \$(PROGRAM_spu)




#INCLUDE = -I\$(SDKPRINC)
#LDFLAGS += -L\$(SDKPRLIB)

##########################################################
```

```
#                               Local Defines
#############################################################

#IMPORTS            := -lmisc -lsputimer

#############################################################
                              buildutils/make.footer
#############################################################

ifdef CELL_TOP
        include \$(CELL_TOP)/buildutils/make.footer
else
        include ../../../../buildutils/make.footer
endif
```

A.6 SPU: barrier_heavy.h

```
/* ------------------------------------------------------------ */
/* All Rights Reserved.                                         */
/*                                                              */
/*      University of Illinois at Urbana Champaign              */
/*          Department of Computer Science                      */
/*            Real-Time Systems Laboratory                      */
/*                                                              */
/*                                                              */
/* - Deepti Kumar Chivukula - dchivuk2@illinois.edu             */
/*                                                              */
/*                                                              */
/*                                                              */
/*          BUS SCHEDULING ON THE CELL PROCESSOR                */
/*                                                              */
/* - File Name: barrier_heavy.h                                 */
/*                                                              */
/* function: _barrier_heavy(ea, id, lsbuf, total)               */
/* - This function implements a specialized barrier             */
/*   function that is robust and ensures that all               */
/*   parties leave the barrier at as close to the same          */
/*   time as possible.                                          */
/* - The barrier uses a system memory cache-line                */
/*   buffers 'ea'.                                              */
/* - The ea buffer contains an array flags that are             */
```

```
/*    set by each of the SPEs when they enter the       */
/*    barrier.                                          */
/* - SPE with an id of 0, is consider the master.       */
/*    He waits until all the flags are set by the       */
/*    slave SPEs, and then clears the flags to releases */
/*    the slave SPEs.                                   */
/* - 'total' is the number of SPEs participating in     */
/*     the barrier.                                     */
/*                                                      */
/* Restrictions: 'ea' must point to a 16-byte aligned   */
/* address in system memory. No other variables should  */
/* reside in this cacheline. The 'lsbuf' must point to  */
/* a 128-byte aligned address in local store memory.    */
/*                                                      */
/* -------------------------------------------------------- */
/* PROLOG END TAG zYx                                   */

#ifndef _barrier_heavy_h_
#define _barrier_heavy_h_

#include <spu_mfcio.h>


static __inline void _barrier_heavy(unsigned int ea,
      unsigned int id, volatile void *lsbuf, unsigned int total)
{
  int i;
  unsigned int flag;
  unsigned int mask;
  volatile unsigned int *ls_ptr;

  ls_ptr = (volatile unsigned int *)lsbuf;

  /* Save the callers tag mask */
  mask = spu_readch(MFC_RdTagMask);
  spu_writech(MFC_WrTagMask, 1);

  if (id == 0) {
    /* Master SPE */

    /* Wait for all the slave SPEs to enter the barrier. */
    if(total>1){
            do {
```

```
            spu_mfcdma32(lsbuf, ea, 128, 0, MFC_GETLLAR_CMD);
            (void)spu_readch(MFC_RdAtomicStat);

            for (i=1, flag=1; i<(int)total; i++) flag &=
                ls_ptr[i];

          } while (flag == 0);


          /* Clear the flags for all the slave SPEs. */
          for (i=1; i<(int)total; i++) ls_ptr[i] = 0;

          spu_mfcdma32(lsbuf, ea, 128, 0, MFC_PUT_CMD);

          /* Read the buffer to affect an equivalent delay
           * on the master SPE.
           */
          spu_mfcdma32(lsbuf, ea, 128, 0, MFC_GETB_CMD);
          spu_mfcstat(MFC_TAG_UPDATE_ALL);
              }

  } else {
    /* Slave SPE */

    /* Write to its flag word to signal that
                it has reached the barrier */
    ls_ptr[id] = 1;
    spu_mfcdma32(&ls_ptr[id], ea + id*sizeof(unsigned int),
                                  4, 0, MFC_PUT_CMD);
    spu_mfcstat(MFC_TAG_UPDATE_ALL);

    do {
      spu_mfcdma32(lsbuf, ea, 128, 0, MFC_GETLLAR_CMD);
      (void)spu_readch(MFC_RdAtomicStat);
    } while (ls_ptr[id]);
  }

  /* Restore the callers tag mask */
  spu_writech(MFC_WrTagMask, mask);
}

#endif /* _barrier_heavy_h_ */
```

A.7 SPU: spu_slih_reg.c

```
/* ------------------------------------------------------------ */
/* All Rights Reserved.                                         */
/*                                                              */
/*     University of Illinois at Urbana Champaign               */
/*         Department of Computer Science                       */
/*          Real-Time Systems Laboratory                        */
/*                                                              */
/*                                                              */
/* - Deepti Kumar Chivukula - dchivuk2@illinois.edu             */
/*                                                              */
/*                                                              */
/*                                                              */
/*        BUS SCHEDULING ON THE CELL PROCESSOR                  */
/*                                                              */
/*                                                              */
/* - File Name: spu_slih_reg.c                                  */
/*                                                              */
/*  Example SPU second level interrupt handler                  */
/*  slih) manager. This example consists of an                  */
/*  interrupt handler dispatch table                            */
/*  (spu_slih_handlers), a interrupt handler                    */
/*  registration routine spu_slih_reg), and a default           */
/*  interrupt handler spu_default_slih). For this               */
/*  implementation, a second level interrupt handler            */
/*  takes as its input, the current event status word.          */
/*  The handler can assume that it was dispatched to            */
/*  by the most significant non-zero event bit. The             */
/*  second level handler is assumed to process any or           */
/*  all events and return a new event status back to            */
/*  the first level interrupt handler for further               */
/*  event processing. The first level interrupt                 */
/*  handler has already acknowledged all received               */
/*  events before calling any slih. A slih should               */
/*  only perform subsequent acknowledgements if it is           */
/*  determined that additional events have been                 */
/*  received while in the slih.                                 */
/*                                                              */
/* ------------------------------------------------------------ */
/* PROLOG END TAG zYx                                           */

#include <spu_slih_reg.h>
```

```
#include <spu_intrinsics.h>


#define SPU_EVENT_ID(_mask)     \\
            (spu_extract(spu_cntlz(spu_promote(_mask, 0)), 0))

/* spu_default_slih
 * ----------------
 * This function is called whenever an event occurs for which
 * no second level event handler was registered. The default
 * event handler does nothing and zeros the most significant
 * event bit indicating that the event was processed (when
 * in reality, it was discarded..
 */
static unsigned int spu_default_slih(unsigned int events)
{
  unsigned int mse;

  mse = 0x80000000 >> SPU_EVENT_ID(events);
  events &= ~mse;

  return (events);
}


/* spu_slih_handlers[]
 * -------------------
 * Here we initialize 33 default event handlers.  The first
 * entry in this array corresponds to the event handler for
 * the event associated with bit 0 of Channel 0 (External
 * Event Status).  The 32nd entry in this array corresponds
 * to bit 31 of Channel 0 (DMA Tag Status Update Event). The
 * 33rd entry in this array is a special case entry to handle
 * "phantom_events" which occur when the channel count for
 * Channel 0 is 1, causing an asynchronous SPU interrupt,
 * but the value returned for a read of Channel 0 is 0.
 * The index calculated into this array by spu_flih() for
 * this case is 32, hence the 33rd entry. */
  spu_slih_func  spu_slih_handlers[33] __attribute__ ((aligned
     (16))) = {
  spu_default_slih,  spu_default_slih, spu_default_slih,
     spu_default_slih,
```

```
  spu_default_slih, spu_default_slih, spu_default_slih,
      spu_default_slih,
  spu_default_slih, spu_default_slih, spu_default_slih,
      spu_default_slih,
  spu_default_slih, spu_default_slih, spu_default_slih,
      spu_default_slih,
  spu_default_slih, spu_default_slih, spu_default_slih,
      spu_default_slih,
  spu_default_slih, spu_default_slih, spu_default_slih,
      spu_default_slih,
  spu_default_slih, spu_default_slih, spu_default_slih,
      spu_default_slih,
  spu_default_slih, spu_default_slih, spu_default_slih,
      spu_default_slih,
  spu_default_slih,
};

/* spu_slih_reg
 * ------------
 * Registers a SPU second level interrupt handler for
 * the events specified by mask. The event mask consists of a
 * set of bits corresponding to the event status bits
 * (see channel 0 description). A mask containing multiple 1
 * bits will set the second level event handler for each of
 * the events.
 */
void spu_slih_reg(unsigned int mask, spu_slih_func func)
{
  unsigned int id;

  while (mask) {
    id = SPU_EVENT_ID(mask);
    spu_slih_handlers[id] = func;
    mask &= ~(0x80000000 >> id);
  }
}
```

A.8 SPU: spu_slih_reg.h

```
/* ------------------------------------------------------------ */
/* All Rights Reserved.                                         */
```

```
/*                                                                  */
/*     University of Illinois at Urbana Champaign                   */
/*         Department of Computer Science                           */
/*           Real-Time Systems Laboratory                           */
/*                                                                  */
/*                                                                  */
/* - Deepti Kumar Chivukula - dchivuk2@illinois.edu                 */
/*                                                                  */
/*                                                                  */
/*                                                                  */
/*          BUS SCHEDULING ON THE CELL PROCESSOR                    */
/*                                                                  */
/* - File Name: spu_slih_reg.h                                      */
/*                                                                  */
/* ---------------------------------------------------------------- */
/* PROLOG END TAG zYx                                               */

#ifndef _SPU_SLIH_REG_H_
#define _SPU_SLIH_REG_H_        1

typedef         unsigned int (*spu_slih_func)(unsigned int);

extern          spu_slih_func spu_slih_handlers[];
extern          void spu_slih_reg(unsigned int, spu_slih_func);
extern          void spu_flih(void);
#endif /* _SPU_SLIH_REG_H_ */
```

A.9 Bit Ordering and Numbering

Bit order is an important consideration in network programming, since two computers with different byte orders may be communicating. Failure to account for varying *endianness* when writing code for mixed platforms can lead to bugs that can be difficult to detect. This section explains big-endian ordering used by the CBE.

Storage of data and instructions in the Cell Broadband Engine is *big-endian*. Big-endian ordering is shown in Figure A.1 and has the following characteristics:

- Most-significant byte is stored at the lowest address, and least-significant byte is stored at the highest address.

- Bit numbering within a byte goes from most-significant bit (bit 0) to least-significant bit (bit n). This differs from some other big-endian processors.

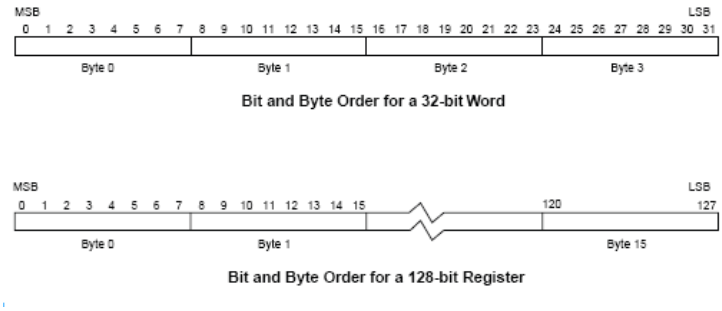

Figure A.1: Big Endian Ordering

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

REFERENCES

[1] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha, "Coscheduling of cpu and i/o transactions in cots-based embedded systems," in *Proceedings of the 29th IEEE Real-Time Systems Symposium*, 2008, pp. 221–231.

[2] A. Agarwal, C. Iskander, and R. Shankar, "Survey of network on chip architectures and contributions," in *Journal of Engineering Computing and Architecture*, vol. 3, no. 1, 2009.

[3] T. Chen, R. Raghavan, J. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementation: A performance view," in *IBM Journal of Research and Development*, vol. 51, no. 5, 2005, pp. 559–572.

[4] T. W. Ainsworth and T. M. Pinkston, "Characterizing the cell eib on-chip network," *IEEE Micro*, vol. 27, no. 5, pp. 6–14, 2007.

[5] *Cell Broadband Engine Programming Tutorial*. IBM Corporation, 2007.

[6] *Cell Broadband Engine Programming Handbook*. IBM Corporation, 2006.

[7] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, pp. 600–625, 1996.

[8] C. D. Locke, D. R. Vogel, L. Lucas, and J. B. Goodenough, "Generic avionics software specification," Carnegie-Mellon Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-8, 1990.

[9] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs Annals of Discrete Mathematics*. Amsterdam, The Netherlands: North-Holland Publishing Company, 2004, vol. 57.

[10] P. Holman and J. H. Anderson, "Adapting pfair scheduling for symmetric multiprocessors," *Journal of Embedded Computing*, vol. 1, no. 4, pp. 543–564, 2005.

[11] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *The 27th IEEE International Real-Time Systems Symposium*, 2006, pp. 101–110.

[12] D. Zhu, D. Mosse, and R. Melhem, "Multiple-resource periodic scheduling problem: How much fairness is necessary?" in *The 24th IEEE International Real-Time Systems Symposium (RTSS'03)*, 2003, p. 142.

[13] J. Kleinberg and E. Tardos, *Algorithm Design*. Reading, MA: Addison-Wesley, 2005.

[14] *SPU Timer Library Programmer's Guide and API Reference*. IBM Corporation, 2007.