Mapping Multiple Processes onto SPEs of the CELL BE Platform using the SCO Model of Computation

A Master's thesis

Computer Science

Nadia Ramjiawan

January 19, 2009

Abstract

The Cell Broadband Engine platform is a heterogeneous, distributed memory multiprocessor architecture consists of a Power Processor Unit (PPU) and several Synergistic Processor Element (SPE). The Cell platform supports pipeline parallellism making it very suitable for mapping streaming application. The Synergistic Processor Element implemented in the Cell is a single-threaded processor and is limited to execute only a single task. This thesis present a model for mapping stream-processing applications on the Cell in such a way that it provides facilities to map multiple tasks on a Synergistic Processor Element of the Cell processor. The model is designed in the context of the SBF dataflow model. Using the dataflow rather then a thread model we are able to map multiple tasks on an SPE.

Acknowledgments

This thesis is the result of a research work performed at the Leiden Embedded Research Center of the Leiden Institute of Advanced Computer Science (LIACS) - Leiden University. I would like to thank all the people who guided and supported me during my research.

First of all, I would like to thank my supervisor Dr.Ir. Bart Kienhuis for giving me the opportunity to do my Master's research at the Leiden Embedded Research Center (LERC). He always provided guidance and support throughout my Master's research and feedback during the completion of my Master thesis.

I also want to thank Bin Jiang for all his guideness and support he provided during my research. Also for the useful discussions we had and the time he was willing to spend to clarify previous work for me.

Finally, I would like to thank my parents who encourage me during my study. They always stood by me and gave me the strength to continue.

Contents

Acknowledgments **4**

1 Introduction **11**

2 Problem description **13**
2.1 KPN to DataFlow Model 15

3 Related Work **19**

4 Solution Approach **25**
4.1 SCO Model 25
4.1.1 Schedule 25
4.1.2 Communication 27
4.1.3 Observe 28
4.2 Merging with SCO Model 28

5 Working out **31**
5.1 SBF Model 32
5.2 CLooG 32
5.3 Detailed Example 33

6 CellFlow **39**
6.1 The CellFlow Tool 39
6.2 Code generated by CellFlow 41

7 Experiments & Results **47**
7.1 Applications 47

8 Conclusions and Future Work **53**

A Code generated by saas for the schedule of ND_3 **55**

List of Figures

2.1 Cell Structure Diagram . . . 13
2.2 An example of a process network . . . 14
2.3 One-to-one versus many-to-one mapping . . . 14
2.4 Extend Kahn Process Network with firing rules to obtain Dataflow network . . . 16
2.5 Thread versus DataFlow model . . . 17

3.1 Position CellFlow with the CELLCC and ESPAM tool. . . . 21
3.2 Mapping possibilities with CELLCC . . . 21
3.3 Mapping possibilities with ESPAM . . . 22
3.4 Mapping possibilities with CellFlow . . . 23

4.1 Schedule Communication Observe Model . . . 26
4.2 Schedule algorithm . . . 26
4.3 Producer-Consumer Process Network . . . 27
4.4 Mapping Producer/Consumer pair on Cell BE using the SCO Model . . . 28
4.5 The communication is performed in a round robin manner . . . 28
4.6 Cell code in SCO format . . . 29
4.7 Process P1 and P2 with their variants . . . 29
4.8 SPE code in SCO format for Process P1 and P2 . . . 29
4.9 Execution order of process P1 and P2 variants . . . 30

5.1 Tool chain to convert Matlab code into SCO description . . . 31
5.2 Stream-Based Function Object . . . 32
5.3 Matlab code for QRvr matrix decomposition algorithm and the corresponding Kahn Process Network derived by COMPAAN . . . 34
5.4 Process ND_3 iteration space with control statements for input and output ports, selected for the *Vectorize* function . . . 35
5.5 CLooG input and output for ND_3 of QRvr. . . . 36

5.6 Variants domains of ND_3 37
5.7 Variants of ND_3 with active input and output port domains 37

6.1 CellFlow design flow shows the process of translating Matlab code into SCO model specification 40
6.2 Speudo algorithm used by CellFlow 40
6.3 Code in SCO format 41
6.4 init method 41
6.5 Algorithm to select variants 43
6.6 isDataPresent method 44
6.7 execute method 44
6.8 update method 44
6.9 fireVariant method 45
6.10 schedule method 45

7.1 Self created application with different SPE mappings. 48
7.2 Code that is executed on SPE0 for the mappings shown in Fig. 7.1 49
7.3 XML-Mapping specification for the mappings shown in Fig. 7.1. 49
7.4 M-JPEG Process Network 50
7.5 Different mapping strategy for M-JPEG Application on Cell archicture 51

Chapter 1

Introduction

Heterogeneous multi-processor architectures are often mentioned as the hardware platform to be used in modern electronics systems. They are composed of a combination of processor cores of varying type and offers good potential for computational efficiency for many applications. Although providing high performance, such architecture brings new design challenges as well as increased complexity in developing software for these platforms.

Writing efficient parallel applications that utilize the computing capability of many processing cores may be even more challenging. Manually deriving parallel code from a sequential program is very difficult. It is a very error prone and time consuming process. Another approach is required to increase programmer productivity. Compiler support is needed to generate multithreaded parallel code from a single threaded sequential program.

The Cell Broadband Engine (Cell BE) [1] is such a heterogeneous multi-core processor comprised of control-intensive processor and compute-intensive SIMD processor cores, each with its own distinguishing features. To get the most out of the Cells incredible computational capability, it is necessary for each programmer to consider the differences between the two processor cores and utilize them appropriately in a way to suit the intended application.

Streaming application are suitable to execute on the Cell platform because of it's platform characteristics. It can chain its compute-intensive processor cores together to perform streaming operations in a sequence. For example, an processor core reads data from an input into it's local store, performs the processing step and stores the result into it's local store. The second processor reads the output from the first processor local store and processes it and stores it in it's output area. Streaming applications have become increasingly important and widespread. Examples of streaming applications include Internet audio and video streaming, automatic target recognition (ATR) found in radar digital signal processor (DSP) systems. Streaming applications operate on a continuous stream of data and are usually compute-intensive. The Cell architecture increases the performance of streaming applications by introducing system-level parallelism as oppose to instruction level parallelism. It exhibit system-level parallelism by using a CPU and multiple

coprocessors.

The aim of this thesis project is to build a design environment for mapping stream-based application on the Cell architecture, with the focus on mapping multiple tasks on the very high performance processors of the Cell, called SPE. The motivation for exploiting the SPEs capabality is that the Cell may get close to it's theoretical maximum performance when the SPEs processors compute heavy streaming applications. For mapping streaming applications on the Cell we need a model that fits well into it's platform specification. We used the Kahn Process Network (KPN) Model of Computation, which is a widely used model to specify task level parallelism in streaming application. Since the KPN model is a thread based model and an SPE is a single threaded processor, using the KPN model, we can map only a single task on a SPE. To be able to map multiple tasks on a single threaded SPE, we need to translate the KPN into a dataflow model. We use the SBF model [8] to express processes in a dataflow format. Using the dataflow model specification, we are able to map mutiple tasks on an SPE.

This thesis offers the following research contributions:

- A new model of computation to express streaming applications in a dataflow format. This model offers the possibility to execute multiple processes on a single threaded processor by exploiting the dataflow concept as opposed to the thread concept.
- Showed that we can use SBF model to realize SCO concept.
- The design and implementation of a fully automated tool for efficiently mapping streaming applications in SCO format onto the Cell architecture.

The rest of the thesis is organized as follows. Section 2 describes the problem we are investigating in this thesis project. Section 3 shows other approaches that are mainly related to our work. In Section 4 we briefly explain our approach to map streaming applications on the Cell architecture and give an in-depth example in Section 5. The example shows the steps required to translate a process from KPN format into SCO format and ready to be mapped on the Cell architecture. Section 6 shows some experiments and results for the M-JPEG application using different mappings. Finally, we conclude the thesis in Section 7.

Chapter 2

Problem description

Cell technology has been developed as a solution to the need for higher performance. The Cell is a heterogeneous multi-core processor comprised of one control-intensive processor core (PPE) and eight compute-intensive processor cores (SPEs) serving different functions. A picture of the Cell is shown in Fig. 5.1. A high-speed bus called the Element Interconnect Bus (EIB) is used for connecting these processor cores within the Cell. The EIB also provides connections to main memory and external I/O devices, making it possible for processor cores to access data from various sources. The PPE shares program processing with the SPEs. Program processing is further shared among the SPEs that perform Single Instruction Multiple Data (SIMD) calculations to maximize the Cells computational performance.

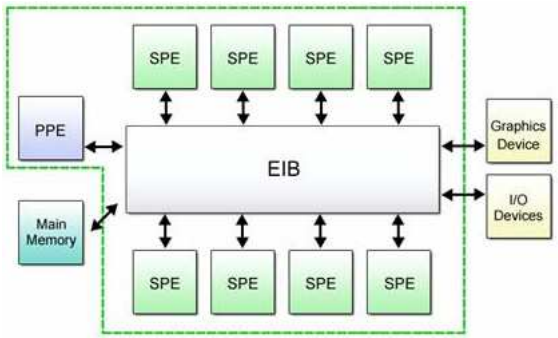


Figure 2.1: Cell Structure Diagram

Like conventional processors, the PPE allows execution of the operating system and applications. It also performs input/output control when the operating system accesses the main memory and external devices, and provides control over the SPEs. The PPE is designed to handle multithreading. The SPEs are less complex processing units than the PPE, in that they are not designed to perform control-intensive tasks. The SPEs are designed to iterate simple operations necessary

for processing multimedia data. The Cell delivers an exceptional computational capability by combining these compute-intensive processor cores. This comprises the focus of our research. How can we utilize the compute power of these compute-intensive processor cores as efficient as possible for stream based applications? The SPEs doesn't support multithreading and as consequence it can't execute more than one task. This causes a situation where most of the SPE compute power remains unused as there is not enough workload in a single task. The Cell will be working the hardest when the SPEs are working on compute heavy applications. Stream based applications are heavy in compute and the type of applications on which we focus in the remainder of this thesis. It's in these applications that the Cell may get close to it's theoretical maximum performance. Thus, improving the assignment of tasks to these SPEs is very important.

Fig. 5.2 shows a process network with six processes, p1, p2, p3, p4, p5 and p6, that are connected with each other. This is a typical configuration when processing stream based applications. If we map these processes onto the Cell processor using the thread concept that is supported by the Cell, then the mapping would look like the left picture of Fig. 2.3, where only single processes are mapped on SPEs. We actually want go to the mapping of the right picture, where multiple tasks can be mapped on single SPEs. As the picture depicts, p2 and p3 are mapped on one SPE, p4 and p5 are mapped on another SPE and p1 and p6 are mapped on the PPU. The problem is how to map multiple tasks on the single threaded processors of the Cell platform?

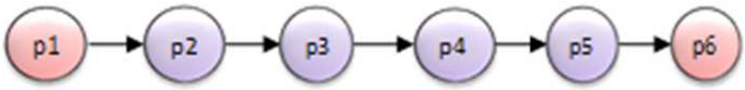


Figure 2.2: An example of a process network

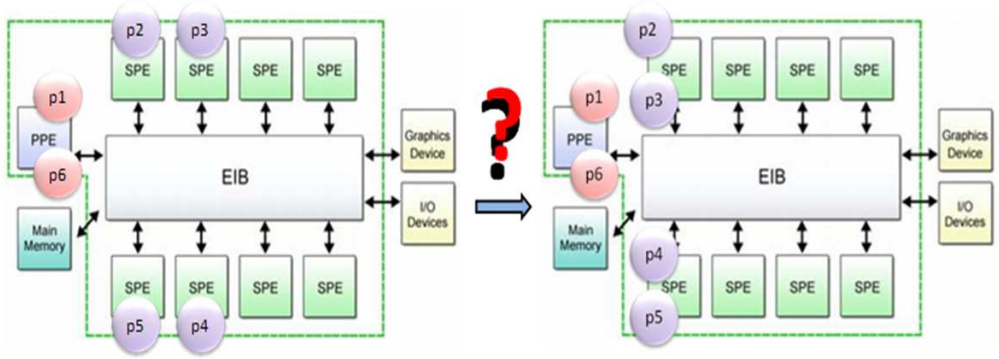


Figure 2.3: One-to-one versus many-to-one mapping

2.1 KPN to DataFlow Model

Applications that have to be executed on multi-processor architectures are typically specified in a imperative language, like C or Matlab. To obtain parallel code that can be mapped on the Cell architecture we use the COMPAAN compiler [7]. The COMPAAN tool automatically extracts parallelism from a single threaded program written in Matlab. It transforms a nested loop program written in Matlab into a Kahn Process Network specification. In the KPN specification an application is modelled as a collection of concurrent processes communicating through FIFO channels. The processes communicate with each other using blocking read and non-blocking write operations. Writing to a channel is non-blocking because the FIFO sizes are infinite. Since the Cell architecture doesn't have infinite memory, we use a blocking write primitive in our approach. Processes are arbitrary sequential programs. If a process tries to read from an empty input it is suspended until enough input data is available. At any given point, a process is either executing code or blocked waiting for data on one of its channels.

The Kahn Process Network model of computation fits well with the characteristics of the Cell architecture. They both have distributed memory and control. The Kahn Process Network is composed of processes and the Cell architecture is composed of multiple processing components, making the mapping of Kahn processes on the Cell processing components easier. The processes in the Kahn Process Network are running concurrently and atomically in the same way an SPE executes. According to these features expressing streaming applications as kahn process networks are very suitable for mapping on the Cell platform.

Although the KPN model is a very good model for the Cell architecture, it has some limitations. First, processes in an KPN model are thread based and are controlled by the thread scheduler. Each process in a kahn process network executes as a thread. That means that when a process start executing we can not interrupt until it ends. If for some reason a process blocks (waiting for data) while executing we can not do anything only the thread scheduler can yield control to another thread. Second, because processes executes as threads and the SPE is a single threaded processor, using the KPN model we can only map a single process on an SPE. We could develop a multithreading environment for the SPE. But this would be a lot of work and will lead to a performance lost. Since we are interested in mapping more than one process on a single SPE, we need another model rather then the KPN model to express the target applications. Therefore, we translate the thread model into a dataflow model. Each Kahn's process becomes an actor with define firing rule and function as shown in Fig. 2.4. The execution of a process then depends on these firing rules instead on the thread control.

In the dataflow model, an application is represented as a directed graph, where nodes represent processes (actors) and arcs represent the FIFOs that connect these processes. Each actor or process has a set of firing rules and can fire only if these firing rules are satisfied. Furthermore, specifying processes in a dataflow format allows to explicitly define a scheduler for the firing rules, which will evaluate the firing order of these firing rules. The execution of a process is then

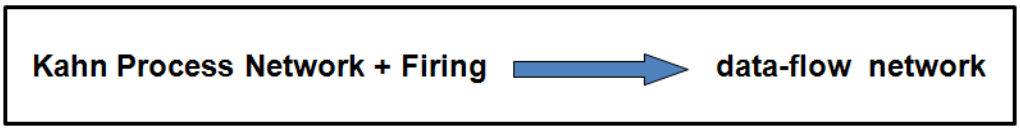


Figure 2.4: Extend Kahn Process Network with firing rules to obtain Dataflow network

handled by this scheduler. By using firing rules with a self modeled scheduler we can guaranteed to get the thread control during execution back and prevent the blocking of a processor. Instead of idling we can do some other calculations which reduce the stalling time drastically leading to more performance.

The key difference between the thread and dataflow model is that the thread model only model the control. In this model a process don't know when data arrive. The time of arrival often matters more than the data. Instead in the datalow model no control is modeled, data arrive in regular streams and the data matters the most. In the dataflow model the data availability is checked before it is read. The thread model on the other hand doesn't performed this check, it directly tries to read the data without being aware if the data is available. If there is no data available the process blocks. The execution of a process in handled by the thread control. The dataflow model defines firing rules, an execution occurs if and only if a firing rule is satisfied, means if all data for the particular firing is available.

The thread and dataflow model behavior is depicted in Fig. 2.5. The left picture shows the thread model and the right picture shows the dataflow model behavior. In the dataflow model we read the data, execute the function and exit. We can't block on the read since data availability is checked before reading by the scheduler. This scheduler checks for availability of data until a firing rule is satisfied. If a rule is satisfied, the scheduler calls the fire function that is guaranteed to terminate giving back control to the scheduler. This is referred as the fire-exit behavior. In the KPN model, the read blocks, we get the thread back if all loop iterations are performed. If the read blocks in each iteration then we can imagine the stalling time this process creates for an processor.

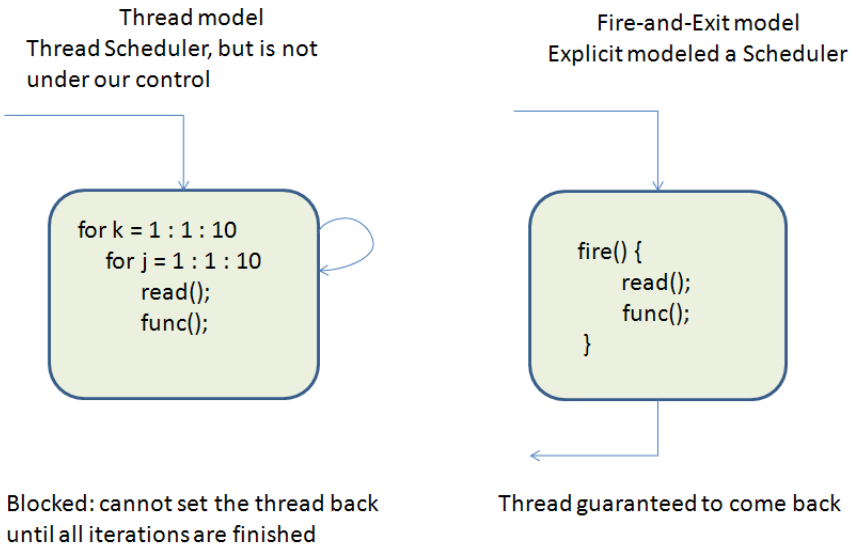


Figure 2.5: Thread versus DataFlow model

Chapter 3

Related Work

In this section, we discuss related work for mapping streaming applications on the Cell architecture and also mapping multiple processes on a single threaded processor.

The approach presented in [2] developed the automated tool, CELLCC, that maps streaming applications specified as Kahn Process Network (KPN) on the Cell platform. This tool is capable of mapping single tasks on the compute intensive processors (SPEs) of the Cell platform. Each process in the KPN model corresponds to a thread and since the SPEs are single threaded processors only one-to-one mapping is possible. The KPN model uses blocking read primitive to synchronize the communication between processes. Thus, if an process tries to read from an empty FIFO it blocks until data is available. This can cause a situation where an SPE spend most of the time waiting for data instead of processing data. Since we are interested in optimizing the workload of an SPE, mapping multiple processes simultaneously on an SPE with the thread approach implemented in CELLCC is not possible.

Our work is different from the approach mentioned above. Our approach relies on a dataflow model for specifying streaming applications in a dataflow format in order to map on the Cell platform. Using the dataflow model as opposed to the KPN model, we are able to map multiple processes on an SPE.

Another approach to map streaming applications on the Cell platform is presented in [3]. In this approach the Synchronous DataFlow (SDF) model [5] is used to specify streaming applications in a dataflow format. The StreamIT [4] programming language is used to implement the synchronous dataflow programming model. StreamIT is an explicitly parallel programming language for streaming applications and includes stream-specific abstractions and representations that are designed to improve programmer productivity. The SDF model consist of actors that communicate exclusively through FIFOs. The SDF actors have a single firing rule. They consume and produce a fixed number of data tokens in each firing. The SDF model is a more restrictive model then the KPN model. As a result, an optimal compile-time scheduling can be found. However, the SDF model has limited expressiveness, it can not express the control of a

process as in the KPN model. As a consequence, an actor in the SDF model can't switch between different FIFOs to select input and output FIFO since it communicate through only one FIFO. Furthermore, this approach implements one-to-one mapping, only a single actor can be mapped on an SPE. In contrast, our approach use the SBF dataflow model to specify streaming applications in a dataflow format, which is capable to express the control of a process. The SBF model has more expressiveness then the SDF model since it is a combination of dataflow and process network models.

The mapping of streaming applications on a FPGA hardware platform using the ESPAM tool is presented in [6]. We relate our work to this work since an FPGA also has multiple processing elements as the Cell Broadband Engine and since this approach present single and multiple task mapping on the MicroBlaze processors of an FPGA. This approach describes streaming applications as Kahn process networks as we do and uses the ESPAM tool to automatically map these applications on the target FGPA platform. The ESPAM tool allows one-to-one mapping (one process per processor) as well as many-to-one (more than one process per processor). In this approach three different mappings are implemented for the MicroBlaze processors:

1. Thread model: map single process on a processor.

2. Merged thread model: multiple processes are merged into one process and mapped on a processor. In this case a valid schedule need to be found before merging.

3. OS supported thread model: execute an operating system on a processor. For this mapping multithread support is needed and no valid schedule is required at compile time.

Our approach differs from this approach in the sense that the ESPAM tool uses dedicated hardware FIFO but our CellFlow tool uses a software modelled FIFO library. Our FIFO library uses DMA, messages and signals to realize the communication between processes. Since the Cell architecture doesn't provide any hardware FIFOs we have to model the required FIFOs in software. Furthermore, in the ESPAM tool multithreading support is implemented for an MicroBlaze processor, making it capable to run an operating system. As a result, multiple processes can be mapped on an MicroBlaze processor. To tackle the problem of our reserach, we could also implement multithreading support for an SPE. Since multithreading support creates overhead and affect the performance of an processor, we don't want to create multithreading support for an SPE. In our approach, multiple process mapping relies on the way we specify streaming applications rather than implementing extra functionality for an processor that isn't provided by the hardware platform. We use a dataflow model with specialized characteristics, which offers the possibility to map multiple processes on an single threaded SPE.

Figure 3.1 illustrates the mapping possibility of our CellFlow tool compared to the CELLCC and ESPAM tool. As depicted in the picture, the CELLCC tool implement the tread model where only single tasks can be assigned to SPE processors. As an example Figure 3.2 show the mapping of a producer/consumer pair for the CELLCC tool. The picture shows the matlab code with the

process network that is derived from it. Each process is mapped on different processors and executes different part of the matlab code.

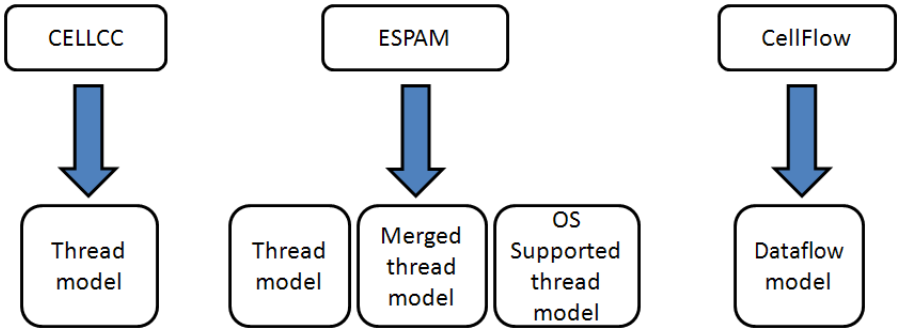


Figure 3.1: Position CellFlow with the CELLCC and ESPAM tool.

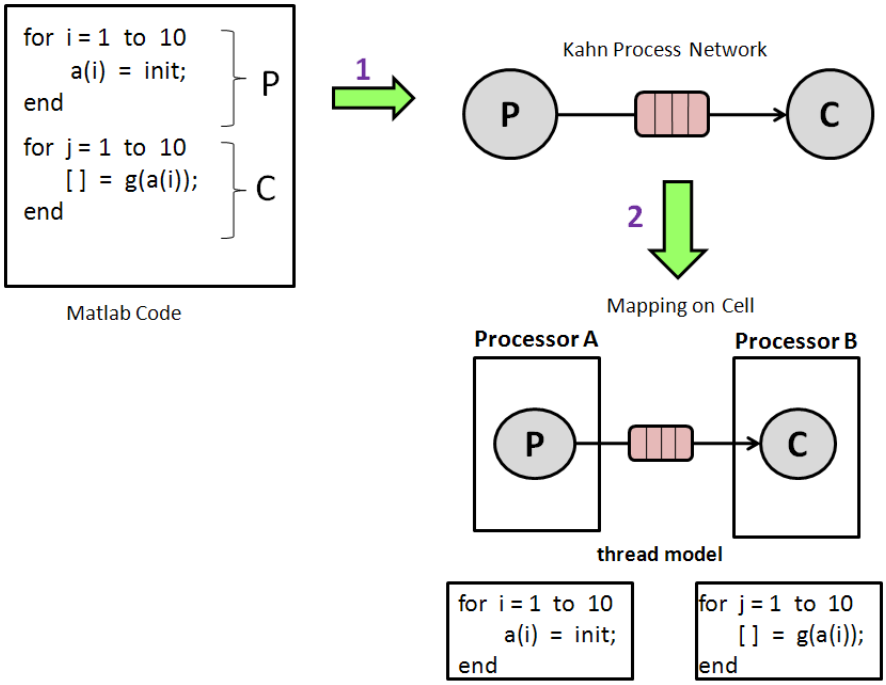


Figure 3.2: Mapping possibilities with CELLCC

As described above, the ESPAM tool is capable to implement three different mappings such as the thread model, merged thread model and OS suported thread model. In Figure 3.3 the same producer/consumer example is used to illustrate the different mapping possiblities. The thread model mapping is the same as implemented by CELLCC. In the merged thread model, the piece of code of both processes are merged into one progam and mapped on a processor. The execution

of the merged code is handled by a single thread. In the OS thread model each process executes a thread. In this model multiple threads runs in parallel.

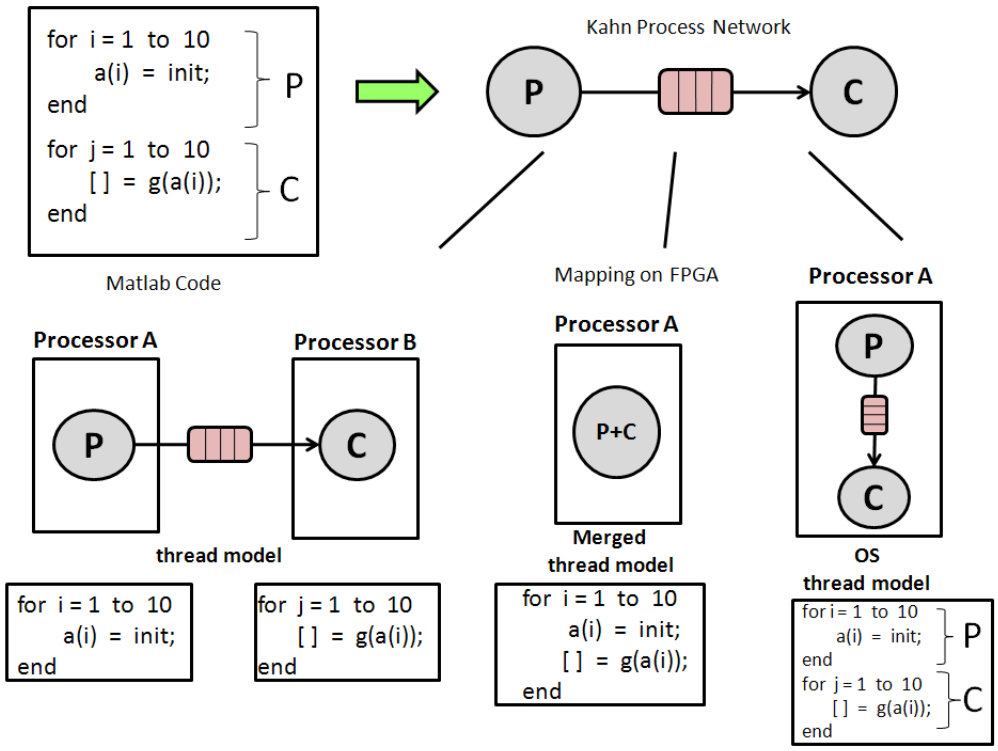


Figure 3.3: Mapping possibilities with ESPAM

Our CellFlow tool offers multiple process mapping on a single threaded processor by exploiting the dataflow concept of our dataflow model of computation. The dataflow model allows one-to-one as many-to-one mapping as depicted by Figure 3.4.

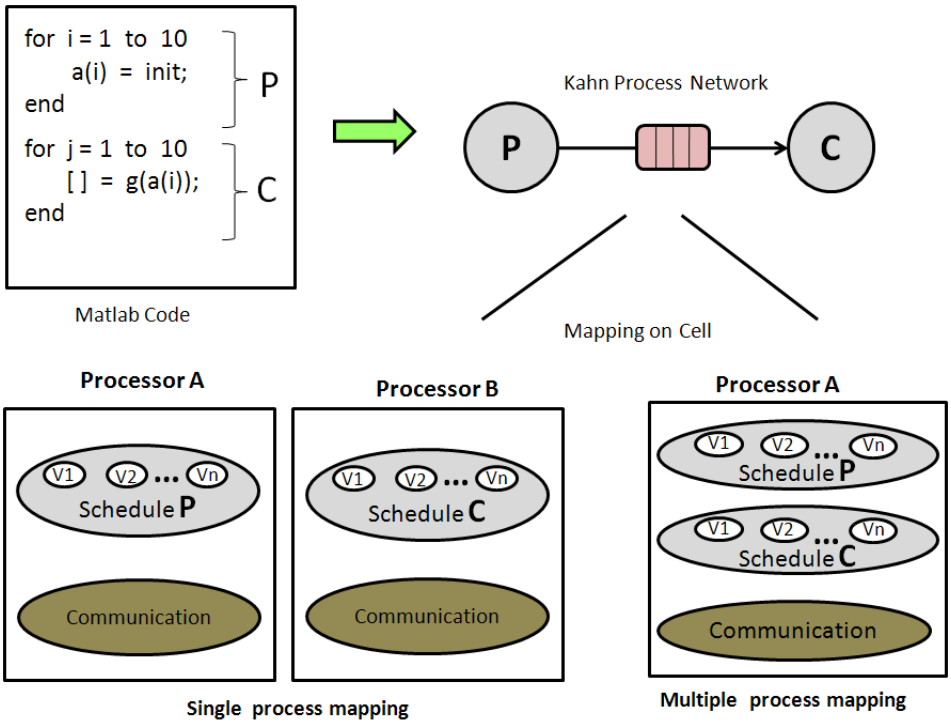


Figure 3.4: Mapping possibilities with CellFlow

Chapter 4

Solution Approach

The problem that is investigated in this thesis is how to map multiple processes on the single threaded SPE processors of the Cell architecture. In this section we propose the model of computation that allows many-to-one mapping on a SPE.

4.1 SCO Model

To specify streaming applications in a format that allows may-to-one mapping on a SPE we propose the SCO model. The SCO model is a dataflow model, that is composed of three separate components as shown in Fig. 4.1:

- Schedule
- Communication
- Observe

In the SCO model, processes consists of firing rules that fire only if their firing rules are satisfied. This is controlled by the scheduler. The communication between processes is done in the communication phase. In the SCO model, the computation and communication of processes are seperated. This is the key to map multiple processes on a single threaded processor. The following sections describes the several parts of the SCO model into more detail.

4.1.1 Schedule

The schedule phase of the SCO model controls the computation of a process. In this phase the function that needs to be executed in a process is executed, based on function variants [10]. It defines the execution order for the function variants of a process. In each function variant, the same function is executed, only different input ports are used to read the data that is needed to

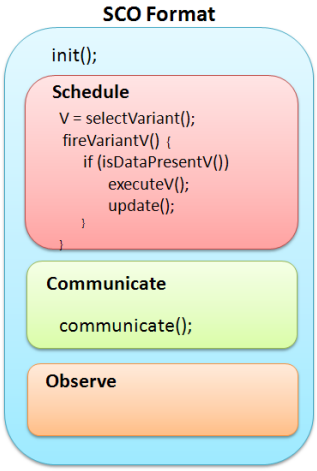


Figure 4.1: Schedule Communication Observe Model

execute the function and different output ports are used to write the result that is produced by the function.

```
V = selectVariant();
fireVariantV() {
   if (isDataPresentV()) {
        executeV();
        update();
    }
}
```

Figure 4.2: Schedule algorithm

The structure of the shedule algorithm is listed in Fig. 4.6. The first step is to select which function variant to execute. This is done by the *selectVariant* method. After a variant is selected it can be fired using the *fireVariant* function. Before a variant can fire, the scheduler has to check if all data is available for the inputs of the function and room is available for writing away produced data. The checking of data is performed by the *isDataPresent* function. This method will iterate over all buffers connected to the read and write ports active for variant *V*. For the buffers connected with the read ports it will checks the availability of data. At the write ports it will checks for space in the output buffers. This function returns a boolean value and if it returns the value *true* then the scheduler can executes variant *V* and updates the state. In the *execute* method, the function reads data from the input buffers and produces the output value that is written to the output buffers. After each execution the scheduler has to update the state, which

is done by the *update* method.

4.1.2 Communication

The communication part transfers data that is produced by the schedule part. The schedule produces tokens and store them in local buffers. The communication part reads the tokens from the schedule local buffers and transfer it to the communicating process's local buffers. In Fig. 4.3 a producer-consumer process network is shown. The processes are communicating through three channels, ED_1, ED_2 and ED_3. These channels are realized using the Cell's infrastructure involving DMA (Direct Memory Access) and messages. Suppose, that the producer is mapped on the PPU and the consumer is mapped on a SPE. The result is given in Fig. 4.4. ED_1, ED_2 and ED_3 are local buffers for the schedule part of both the producer and the consumer. The producer process uses these local buffers to store the produced tokens and the consumer process uses the local buffers to consume tokens from it. The PPU will sent the content of a local buffer via DMA to the other local buffer on the SPE. The synchronization between the PPU and SPE is done using message passing.

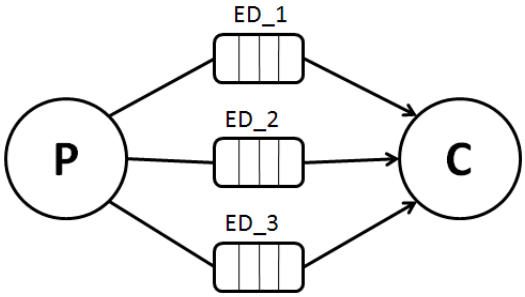


Figure 4.3: Producer-Consumer Process Network

To realize the FIFO implementation, we have developed a communication library. Since the PPU and SPE have slightly different synchronization mechanisms, four different communication types need to realized, PPUSPE, SPESPE, SPESELF and SPEPPU. For the producer-consumer example given above, the PPUSPE communication type is used, since there is only outgoing edges from the producer (PPU) to the consumer (SPE).

The communication is performed by iterating in a round robin manner over all ports as depicted in Figure 4.5. At each port data is transferred if the data is available. For input ports we check if data is to be received and for output ports if data is to be send. The schedule phase produces the data and the communication phase transfer these data but there is no relation between the schedule and communication. They operate independently from each other.

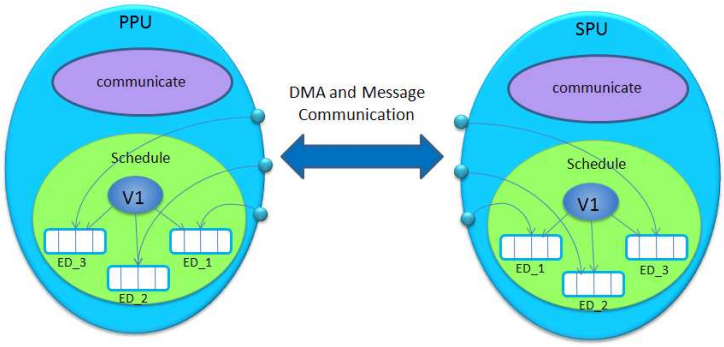


Figure 4.4: Mapping Producer/Consumer pair on Cell BE using the SCO Model

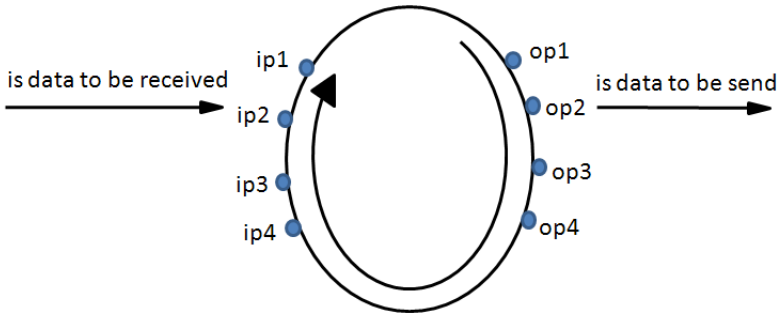


Figure 4.5: The communication is performed in a round robin manner

4.1.3 Observe

The purpose for the observe part in SCO model is to detect deadlock situations. Since the observe phase show as less relevant for our research we will not consider it in this thesis.

4.2 Merging with SCO Model

The SCO model provides the possibility the map multiple processes on a single threaded processor because of the way it expresses processes. In this model processes consists of firing rules and can fire only if their firing rules are satisfied. After checking each firing, we jump out of the process computation phase and execute the communication phase of the particular process as shown by the code given in Fig 4.6. This behavior allows us to put more than one process next to each other into a single thread. We can simply add more firing rules to accomodate a new process. Similarly, we can simply add more communication ports to accomodate additional communication. Extending the set of firing rules, means that if one of the firing rule of a process does not satisfy, the firing rules of the other process may be fired, reducing in this way the processor time

```
while(1) {
   Schedule();
   Communicate();
}
```

Figure 4.6: Cell code in SCO format

spent on idling. By exploiting the dataflow concept of interleaving computations, we are able to map more than one process on a single thread processor. For example, if we have two process P1 and P2 as shown in Fig. 4.7 and we want to map both processes on one SPE, then the SPE code in SCO format resemble the code listed in Fig. 4.8. If the data for process P1 is available, it

```
P1=> V1, V2, V3, V4, V5, V6
P2=> B1, B2, B3, B4
```

Figure 4.7: Process P1 and P2 with their variants

```
while(1) {
  V = selectVariantP1;
  fireVariantV();
  B = selectVariantP2;
  fireVariantB();
  Communicate;
}
```

Figure 4.8: SPE code in SCO format for Process P1 and P2

can execute one of its variant. Otherwise the data availability of process P2 is checked. If data is available the execution of its variant is performed. A possible execution order that may result at runtime for both processes is listed in Fig. 4.9. In our approach we merge process P1 and P2 at runtime. However this could also be done at compile time for the class of nested loop programs. This would make the “selectVariant” code much simpler and faster. At the same time, the firing shown in Figure 4.8 could be influenced at run-time by information gathered in the observation phase of the SCO model. For example, if the observer phase detects that a particular resource like the bus is heavily loaded between processes, it could decided to give one process a higher priority until the bus gets less loaded.

To realize the SCO model concept for the Cell architecture, we have build the CellFLow tool, which is described in Section 6. This tool is capable to perform runtime merging on the single

```
V1, B1, V2, B2, V3, B3, V4, B4, V5, V6
```

Figure 4.9: Execution order of process P1 and P2 variants

threaded SPE processors. The number of processes that can be mapped on an SPE depends on the SPE memory storage and the memory usage of the processes.

Chapter 5

Working out

To convert Matlab code into the SCO model specification, we developed the tool chain as shown in Fig. 5.1. First, the COMPAAN tool is used to convert Matlab code into a Kahn Process Network. Next, the Kahn Process Network is converted into the SBF model specification by the CLooG tool. Finally, the CellFlow tool converts the SBF model specification into the SCO model specification.

Figure 5.1: Tool chain to convert Matlab code into SCO description

We already know how to translate Matlab code into a Kahn Process Network specification. The COMPAAN tool is available to us and can automatically perform this translation. The problem is how can we translate a Kahn Process Network specification into the SCO model specification. We use the SBF dataflow model as an intermediate model to close the translation gap between Kahn Process Network specification and SCO model specification. The schedule phase of the SCO model is implemented using the SBF model specification. We first describe this model in the next section, and describe how the SCO schedule is derived from this SBF model using the CLooG tool. We also give an example, which shows how Matlab code is translated into the SCO model specification. In Section 6, we present our CellFlow tool that automatically translate streaming application written in Matlab into SCO model specification.

5.1 SBF Model

The Stream-Based Function (SBF) model is used to close the gap between Kahn Process Network specification and SCO model specification. The SBF model is a dataflow model that is well suited to specify task parallelism in stream-based applications. The SBF model is composed of two components, called Stream-Based Function Objects and Channels. It describes stream-based applications as a network of SBF objects that communicate concurrently with each other using channels. SBF objects transfer data by using blocking read and non-blocking write. An SBF object contains three components: a set of function, a controller and a state. The set of functions also called the repertoire determines the functionality of an SBF object. This set must be finite and should contain at least an initialization function. The functions inside an SBF object are evaluated in a sequential order determined by the controller. The controller enables the functions of an SBF object and handle the order in which these functions are fired. After a function is fired, the current function informs the controller that the next function state can be computed and the next function can be enabled. The controller has a *transition function* and a *binding function*. The transition function determines the next function state. At each state, a specific function needs to be evaluated, which is determined by the binding function. This binding function associates a function f from the function repertoire with a particular function state. Using the transition function and the binding function, the controller repeatedly invokes and enables a function from the function repertoire.

Fig. 5.2 illustrates an SBF Object that has two read and one write port. These ports are connected with the input and output buffers respectively. We also see that the SBF object has two functions, f_a and f_b. Function f_a reads input data from the two read ports and writes output data to the write port. When function f_a terminates its firing, the controller enables the next function, which is f_b. Function f_b also reads input data from the two read ports and writes output data to the write port.

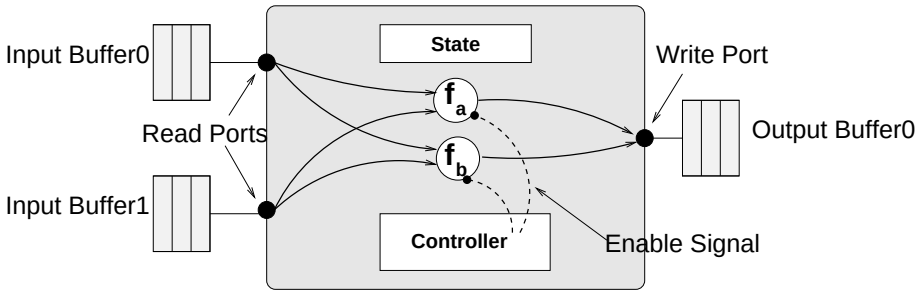


Figure 5.2: Stream-Based Function Object

5.2 CLooG

To convert a piece of Matlab code into the SBF model, we convert the KPN model in the SBF model using the Chunky Loop Generator (CLooG) software [9]. This software takes as input a set of iteration spaces and scans these space to produce the intersections between the spaces. The Streaming Application Analysis and Synthesis (Saas) tool implements a datamodel to specify CLooG output program as a parse tree. We can walk through this parse tree to find different functions. From the parse tree, we construct a controller that enables and defines an execution order for these functions. We create three functions to implement the controller:

- *selectVariant*, used to select function variants from CLooG's output (binding function).
- *evaluateData*, used to check for data availablity on read ports and buffer space for write ports.
- *execute*, used to execute a function.

We define an other function to keep track of the SBF object state, called *update*. This function is used to move from the current function state *c* to the next function state *c'* whenever the current function has fired and implements the transition function of the SBF model. An SBF object must contains an initialization function. Therefore we also define an initialization function, *init*. This function initialize the state of the SBF object.

5.3 Detailed Example

In this section we show for the QRvr example how the flow shown in Figure 5.1 can convert a Matlab program, describing the QRvr matrix decomposition algorithm [11], into a SCO realization. For this example we focus on process ND_3 of QRvr.

Figure 5.3 shows the Matlab code for the QRvr application and the process network that is derived by COMPAAN. COMPAAN generates a process for each statement in the Matlab program. The third statement in the Matlab program(i.e *Vectorize*) is executed in process ND_3. COMPAAN adds control statements to the iteration domain of ND_3 to select input and output ports. These ports are the result of the distribution of control over the various processes as shown in Fig. 5.4.

When we zoom into process ND_3, we see the picture as illustrated in the right part of Fig. 5.4. Process ND_3 has four input ports, named *A, B, C, D* and three output port, called *E, F, G*. It executes the *Vectorize* function, which has two input arguments, in_0, in_1 and three output arguments out_0, out_1, out_2. Data for input argument in_0 is selected from input ports *A* and *B*. And the data for input argument in_1 is selected from input ports *C* and *D*. At a given iteration, one of the portcombination: *AC, AD, BC, BD* is selected as input for in_0 and in_1 The result

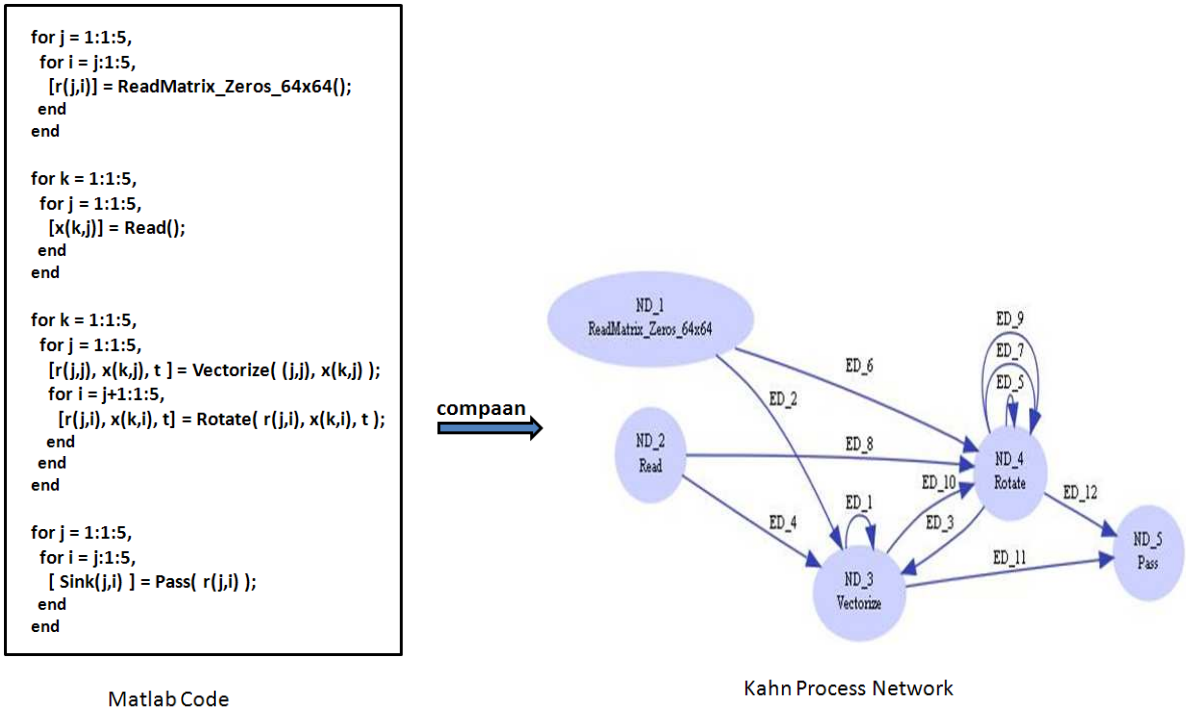


Figure 5.3: Matlab code for QRvr matrix decomposition algorithm and the corresponding Kahn Process Network derived by COMPAAN

of *Vectorize* is written to output arguments out_0, out_1 and out_2. These values are passed to output ports *E, F and G*.

Process ND_3 has seven control statements, where each control statement represents a subset of ND_3 iteration domain. The CLooG software describes these subdomains as matrices and takes these matrices as input and produce the intersections of the subdomains. The CLooG input and output for ND_3 are listed in Fig. 5.5. The CLooG input shows the seven control statements as matrices and the CLooG output shows the intersections of the subdomains. We use the SAAS tool to generate the CLooG input and output. The CLooG software is included in the SAAS tool and SAAS has a datamodel that represent the output generated by CLooG. In left part of Figure 5.6 the iteration domain of each control statement is depicted. Input and output port domains are seperated. The intersection is depicted in the right part. By taking the itersection, the iteration domain of ND_3 gets divided into nine subdomains. In each subdomain the same function is executed using different input and output ports to read and write data. Figure 5.7 list the active input and output ports for the variants for ND_3.

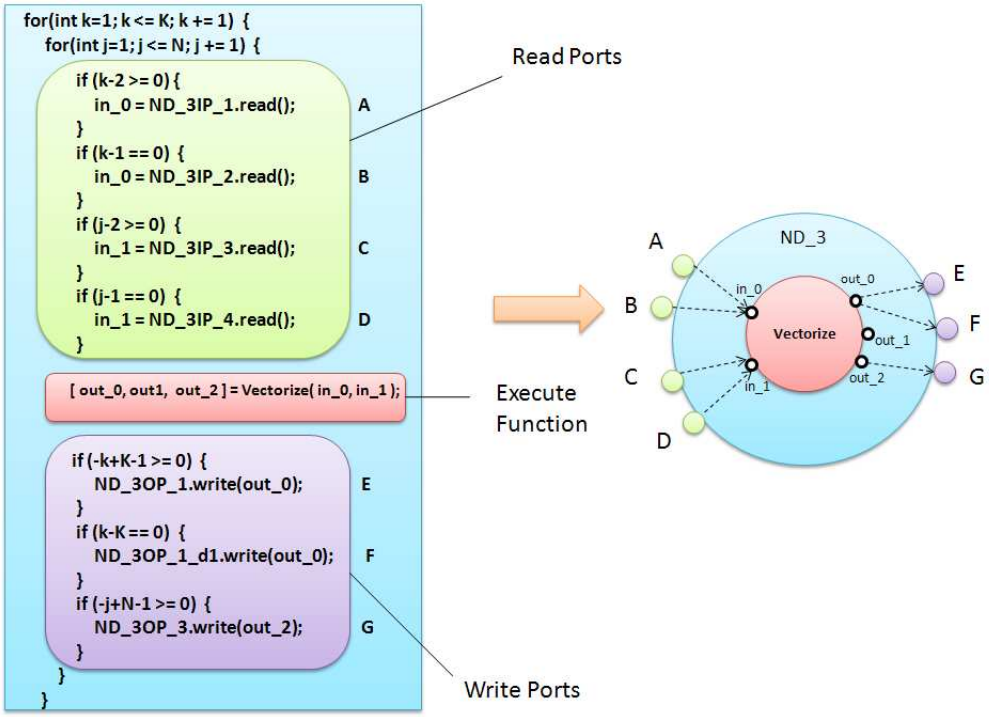


Figure 5.4: Process ND_3 iteration space with control statements for input and output ports, selected for the *Vectorize* function

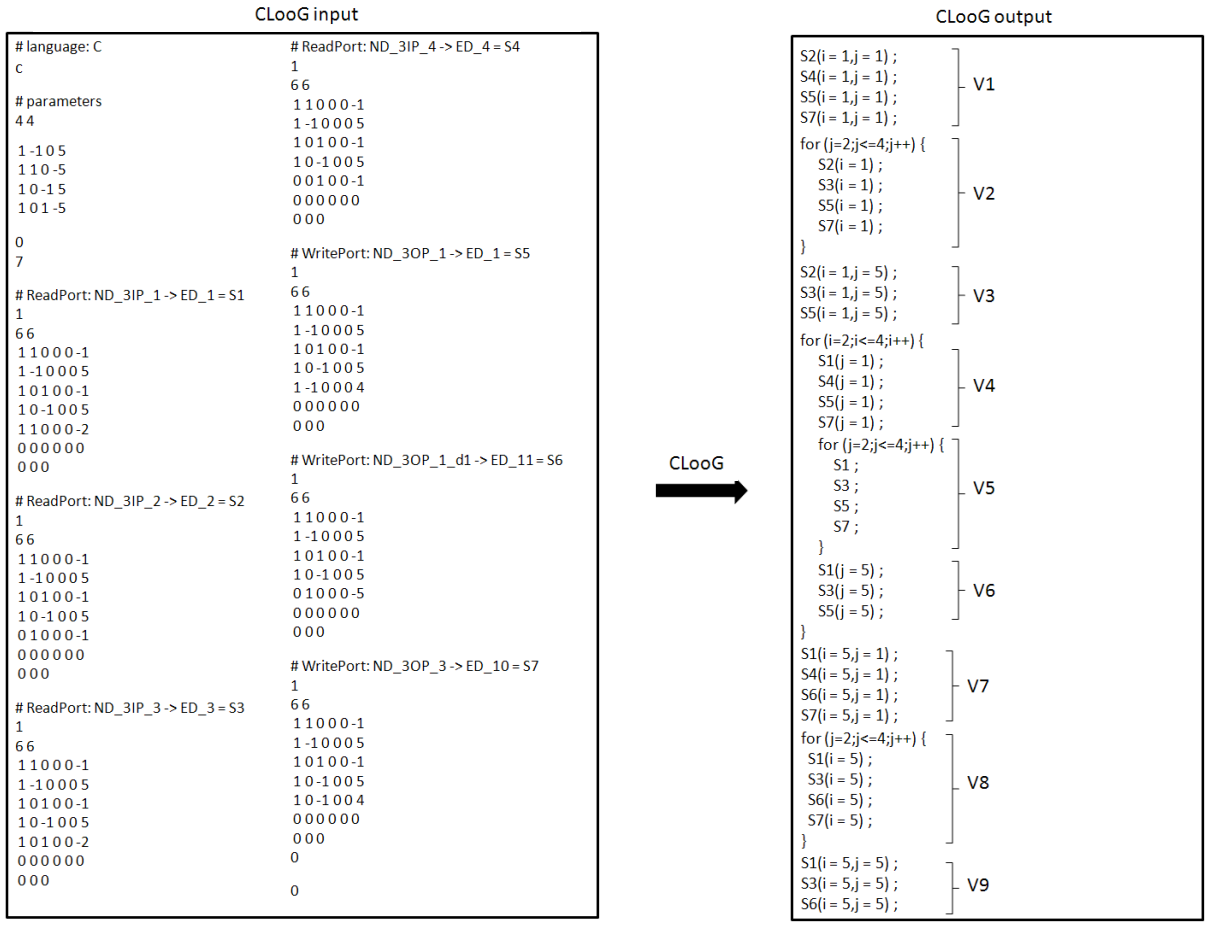


Figure 5.5: CLooG input and output for ND_3 of QRvr.

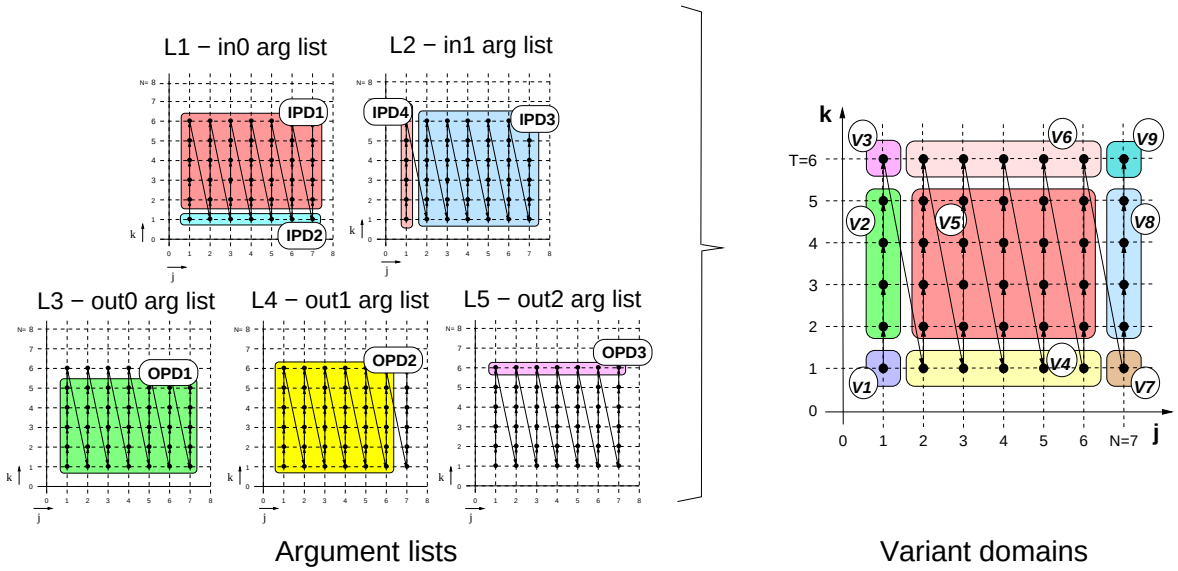


Figure 5.6: Variants domains of ND_3

V1 = (IPD2, IPD4, Vec, OPD1, OPD2, *)	V6 = (IPD1, IPD3, Vec, *, OPD2, OPD3)
V2 = (IPD1, IPD4, Vec, OPD1, OPD2, *)	V7 = (IPD2, IPD3, Vec, OPD1, *, *)
V3 = (IPD2, IPD4, Vec, *, OPD2, OPD3)	V8 = (IPD1, IPD3, Vec, OPD1, *, *)
V4 = (IPD2, IPD3, Vec, OPD1, OPD2, *)	V9 = (IPD1, IPD3, Vec, *, *, OPD3)
V5 = (IPD1, IPD3, Vec, OPD1, OPD2, *)	

Figure 5.7: Variants of ND_3 with active input and output port domains

Chapter 6

CellFlow

The CellFlow tool automatically maps streaming applications specified in Matlab onto the Cell platform in the SCO format. In this section we describe the design of CellFlow and briefly presents the components from which CellFlow is build. We also show some sample code generated by CellFlow.

6.1 The CellFlow Tool

We have develop the CellFlow tool to automatically translate streaming application written in Matlab into SCO model specification. The SCO model is captured in the C-code generated. This C-code is compiled using the GCC compiler chain available for the Cell. In Fig. 6.1 the design flow of CellFlow is shown.

The CellFlow tool takes as input:

- the application specification in KPN format, which is a XML file.
- the mapping specification of the KPN on the Cell platform, which is also an XML representation. This file contains mapping information of the processes that are mapped on the processors of the Cell platform, for example which process is mapped on which processor. The user can manually change this file if another mapping is desired.
- a communication library that provides the communication primitives for performing the communication between processes that are mapped on the processor cores of the Cell. This library implements all communication types possible in the mapping.
- a scheduler that evaluate and specify the execution order of a process.

Taking these input files CellFlow generates C-code in SCO format that can be mapped on the Cell. It creates a directory with Cells code. This directory contains two folders named ppu

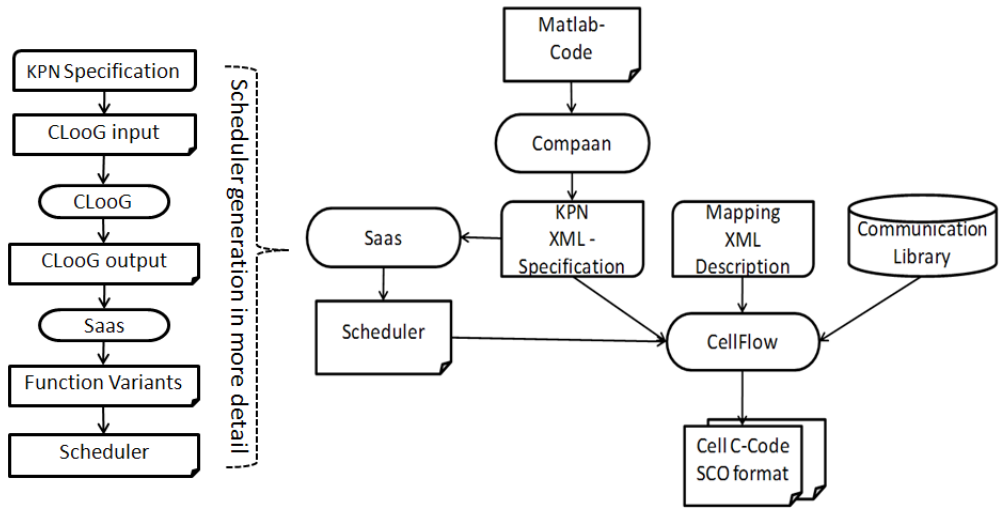


Figure 6.1: CellFlow design flow shows the process of translating Matlab code into SCO model specification

and spu and files needed for the communication for both PPU and SPE. The ppu folder contains code that is going to mapped on the PPU and the spu folder contains code for the SPEs. If more than one SPE is occupied then the code for all SPE are included in the spu folder. Since the PPU controls the program that is executed on the SPEs, the ppu folder contains the main.cc file and other files needed to perform the communication for PPU. The main.cc file contains the code for all processes that are mapped on the PPU. On the other hand the spu folder creates an SPE.cpp file for every SPE that is used. Beside these files, the spu folder contains the SPU communication files. The working of CellFlow in speudo code is given in Fig. 6.2. First Matlab code is translated

```
generate KPN from Matlab;
foreach process P in KPN do
  generate variants;
  generate schedule using variants;

  foreach link of P do
      generate communication method;
  end
end
```

Figure 6.2: Speudo algorithm used by CellFlow

into KPN specification using the compaan tool. The result is a network of processes connected

by links. For each process, we generate variants using the CLooG tool. Next, we specify a schedule for the process using these variants. And for each link that is connected to the process, a communication function is generated. Finally, the schedule and communication part of a process are combined in a while-loop to obtain the SCO model description of a process as shown in Fig. 6.3. The while-loop is then executed on the SPE or PPU, depending on the process type.

```
while(1) {
  Schedule;
  Communicate;
}
```

Figure 6.3: Code in SCO format

6.2 Code generated by CellFlow

The CellFlow tool generates C-code for the SPE and PPE processors of the Cell platform. In the previous section, we saw a detailed description of process ND_3 of the QRvr application. In this section, we will describe the schedule code generated for process ND_3 in more detail.

Process ND_3 has a two dimensional iteration space (i.e, *k* and *j*). In the *initLoopIndex* method the loop iterators that iterate over the iteration domain are initialized with the lowerbound values from the KPN specification. The KPN specification contains all necessary information of processes. To distinguish different processes when mapping on the same processor we append the process name to each method generated for an process.

```
void initLoopIndex_ND_3() {
    k = 1;
    j = 1;
}
```

Figure 6.4: init method

The *selectVariant* selects the proper variant from the CLooG's output at run-time and assign an ID to each variant. These ID will be used by the *fireVariant* method to identify the variant that needs to be fired. CLooG uses for-loop to indicate the iteration domain for an variant. We translate these for-loop into if-statements since we want to go in a function but always return. In fig 6.5 the different subdomains in which a variant is active is listed. For example, in iteration (1,1) of Fig. 5.6 variant *V1* is active according to the first if-statement of the *selectVariant* method. For each iteration in the iteartion space, the *selectVariant* method list the variants that is active

for that particular iteartion. The iteration space is partitioned in the number of variants that is found, as illustrated in figure 5.6.

In the *isDataPresent* function, the data availability for a variant is checked. If all input buffers of a variant have data and all output buffers of the variant have room, then the function returns true otherwise false. Each variant uses different input and output buffers to read and write data. This means that each variant has its own *isDataPresent* function. The *isDataPresent* is the actual firing rule the scheduler checks. If *isDataPresent* is true, the firing rule is satisfied and the function can be executed. As an example we show the *isDataPresent* function of variant *V1* of ND_3 as listed in Fig .6.6. The *isDataPresent* function for variant *V1* checks if data exists at input buffers ED_2 and ED_4 and if the output buffers ED_1 and ED_10 have room by using the *hasData* and *hasRoom* functions. The *hasData* function is used to check for data in input buffers and *hasRoom* to check for space in output buffers. Buffer ED_1 is a selfloop and therefore a different function is generated called, *hasRoomSelf*.

Each variant has an *execute* function. For example, process ND_3 has nine variants, which means that nine *execute* functions have to be generated. The *execute* function for variant *V1* is given in Fig. 6.7. Within the execute method, the data checked by the *isDataPresent* function, is read and pass to the *Vectorize* function. The result of the *Vectorize* function is written to the output buffers. The read and write function are non-blocking. This is not needed as the scheduler guarantees that data is present and that room is available to write out the result.

After an execution is performed, the loop iterators needs to be updated to indicate the next iteration. This is done by the *update* method listed in Fig. 6.8. The loop-iterators *j* and *k* are used to iterate over the iteration domain of process ND_3. If the inner most loop iterator *j* is smaller than or equal to the upperbound then it is incremented. Otherwise, the outer most loop iterator *k* is incremented if it is smaller than or equal to the upperbound defined for this iterator. In this example both loop-iterators has the same upperbound, namely 5.

The *fireVariant* method is used to fire variants. It test which variant needs to be fired and calls the associated methods needed for the firing. It structures the generated *isDataPresent*, *execute* and *update* methods in such a way that if the *isDataPresent* returns true then a variant can be fired.

Combining the *selectVariant* and *fireVariant* method together, the schedule can be obtained.

```
void selectVariant_ND_3() {
    if (k-1 && j-1) {
         v = 1;
    }
    else if (2 <= j && j <= 4) {
        if (k-1) {
             v = 2;
        }
    }
    else if (k-1 && j-5) {
         v = 3;
    }
    else if (2 <= k && k <= 4) {
        if (j-1) {
             v = 4;
        }
        else if (2 <= j && j <= 4) {
             v = 5;
        }
        else if (j-5) {
             v = 6;
        }
    }
    else if (k-5 && j-1) {
         v = 7;
    }
    else if (2 <= j && j <= 4) {
        if (k-5) {
             v = 8;
        }
    }
    else if (k-5 && j-5) {
         v = 9;
    }
}
```

Figure 6.5: Algorithm to select variants

```
bool isDataPresentV1_ND_3() {
    if (hasData(ED_2) && hasData(ED_4) && hasRoomSelf(ED_1) &&
          hasRoom(ED_10)) {
        return true;
    }
    else
        return false;
}
```

Figure 6.6: isDataPresent method

```
void executeV1_ND_3() {
    unsigned int x1, x2 = 0;
    unsigned int y1, y2 = 0;

    read(ED_2, &x1);
    read(ED_4, &x2);
    Vectorize(x1, x2, &y1, &y2);
    writeSelf(y1, ED_1);
    write(y2, ED_10);
}
```

Figure 6.7: execute method

```
void update_ND_3() {
    if (j <= 5) {
        j++;
    }
    else if (j > 5) {
        if (k <= 5) {
            k++;
        }
    }
}
```

Figure 6.8: update method

```
void fireVariant_ND_3() {
    if (v3 == 1) {
        if (isDataPresentV1_ND_3()) {
            executeV1_ND_3();
            update_ND_3();
        }
    }
    else if (v3 == 2) {
        if (isDataPresentV2_ND_3()) {
            executeV2_ND_3();
            update_ND_3();
        }
    }

    .
    .
    .
}
```

Figure 6.9: fireVariant method

```
void schedule_ND_3() {
    selectVariant_ND_3();
    fireVariant_ND_3();
}
```

Figure 6.10: schedule method

Chapter 7

Experiments & Results

In this section, we present the results of two implemented applications in the SCO model specification using CellFlow. The first application is a test application used to illustrate that we can model an application in SCO model and map on the Cell. The second application implemented in SCO format is the M-JPEG application. The M-JPEG application is a real functional application. All experiments are performed using the Cell architecture implemented in Playstation3 console.

7.1 Applications

Self-created application. The first application we evaluated is an application we have constructed to expose particular cases in the mapping step. The process network of this application consists of 5 processes and two channels between processes to connect them with each other. The application and its mapping on the Cell are depicted in Fig. 7.1. Process P1 and P5 are source/producer and sink/consumer processes and are mapped on the PPE. In our implementation, we have created only one thread in the PPE and all processes mapped on the PPE are executed in this single thread. The rest of the processes are simple transformers and are mapped one-to-one on SPE's. For the SPE, we have implemented three mappings. In the first mapping indicated in Fig. 7.1 (A), we map process P2, P3 and P4 on different SPEs. For the second mapping depicted in Fig. 7.1 (B), we map process P2 and P3 on the same SPE and process P4 on a different SPE. And in the thrird mapping process P2, P3, and P4 are mapped on the same SPE, as shown by Fig. 7.1 (C). These mappings shows that merging multiple processes on a SPE is possible with the CellFlow tool. The code that is executed on SPE0, where the number of tasks vary for each mapping, is showed in Fig. 7.2. The left part shows the SPE code for the first mapping where only process P2 is executed on SPE0. In the middel part we see that process P2 and P3 are executed simultaneously on SPE0. In the while loop, we see the Schedule of process P2 and P3. From this construction we can conclude that either process P2 or P3 can fire or both

can fire. Adding more tasks on a SPE affects the while loop by introducing more schedules for the additional processes that are mapped on the SPE.

Each process has its own schedule and mapping more than one process on an SPE introduce name overlapping. We need to distinguish different schudules by giving them a unique name. We solve this problem by appending the process name to the functions name that are generated for each process. For example, if process P1 and P2 are mapped on one SPE, then the functions generated for P1 are: initP1(), selectVariantP1(), isDataPresentP1(), executeP1(), updateP1(), fireVariantP1() and scheduleP1(). And the functions generated for P2 are: initP2(), selectVariantP2(), isDataPresentP2(), executeP2(), updateP2(), fireVariantP2() and scheduleP2().

The mapping specification in XML format for the three mappings is shown in Fig. 7.3. The mapping specification shows which KPN process is mapped on which Cell processor and can be modified manually. The difference between the three mapping specifications is that the left mapping uses three SPE processor, the middel one two SPE and the right mapping uses only one SPE.

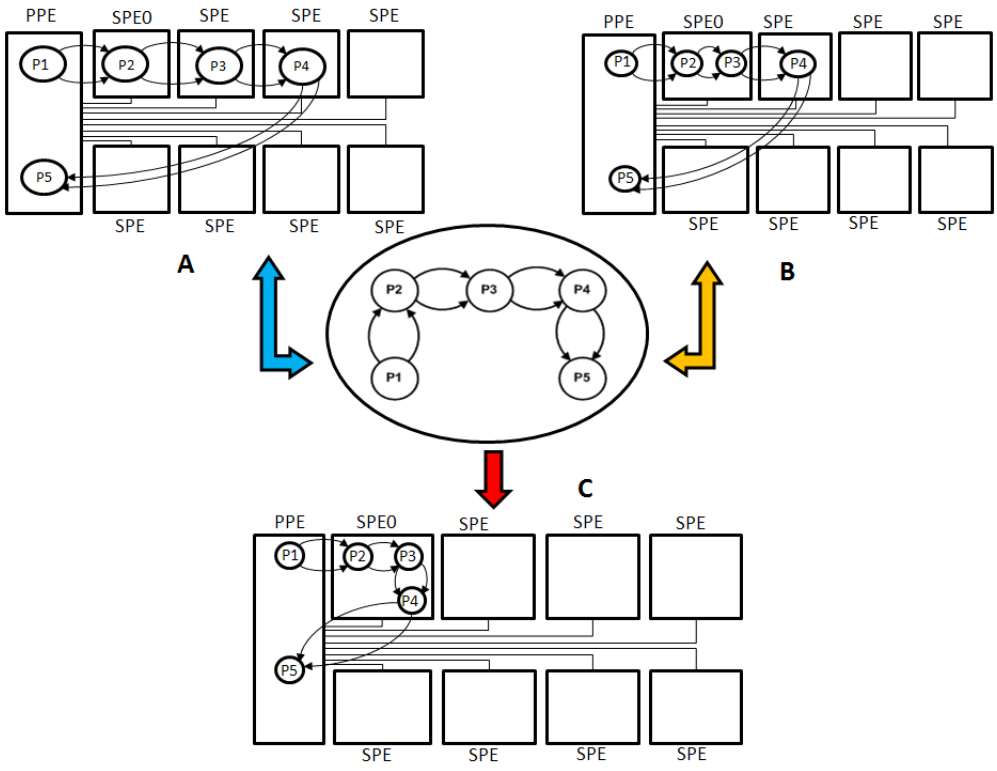


Figure 7.1: Self created application with different SPE mappings.

Motion JPEG application. The second application we have evaluated is the Motion-JPEG decoder (M-JPEG). We applied our approach to the M-JPEG application that is depicted in Fig.

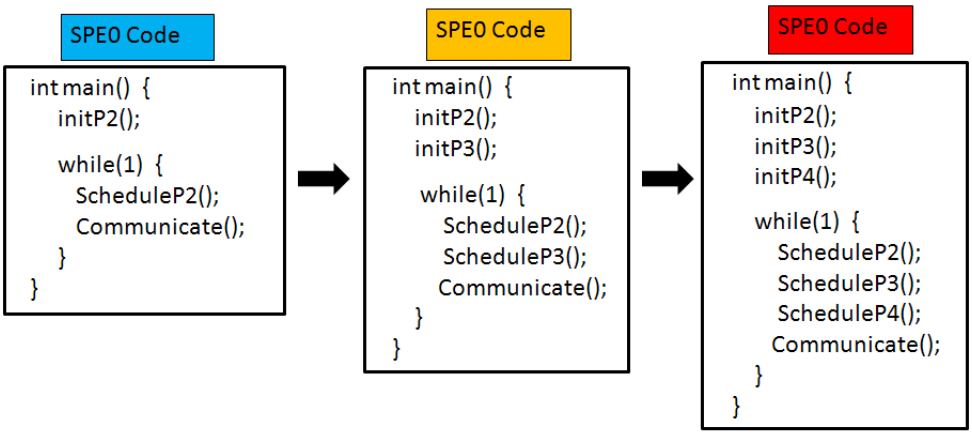


Figure 7.2: Code that is executed on SPE0 for the mappings shown in Fig. 7.1

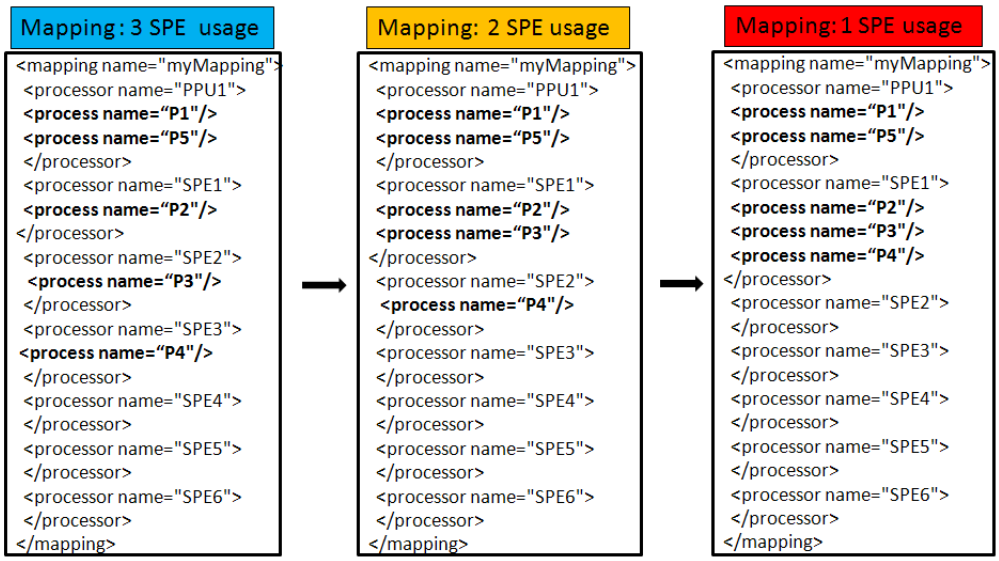


Figure 7.3: XML-Mapping specification for the mappings shown in Fig. 7.1.

7.4 and measure the throughput of different mappings for the Cell. Fig. 7.5 depicts the different mappings we implemented for the M-JPEG application on the Cell platform. The results derived from the implemented mappings are given in Tabel 7.1. What we observe is that mapping 1 and 3 show the same throughput but different number of Cells resource usage. Mapping 1 uses 3 SPEs and each SPE execute a single task. This would be the mapping if only one task could be mapped to a SPE (this would resemble the threaded mapping of M-JPEG). In mapping 3 we see that 2 SPEs are occupied, SPE1 with a single task and SPE2 with two tasks. The results of these

two mappings shows that we can obtain the same throughput while using less SPEs.

From the obtained results we can conclude that expressing streaming application in SCO model, allows us to use Cells compute intensive resources in a many-to-one fashion. Furthermore, the two applications described above are also good examples of the usability of our CellFlow tool. Without CellFlow, it would have taken quite some effort to build equivalent applications, as all communication and scheduling have to be programmed manually.

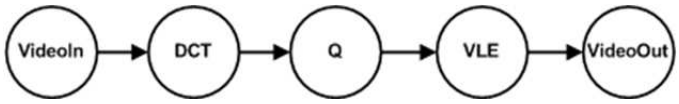
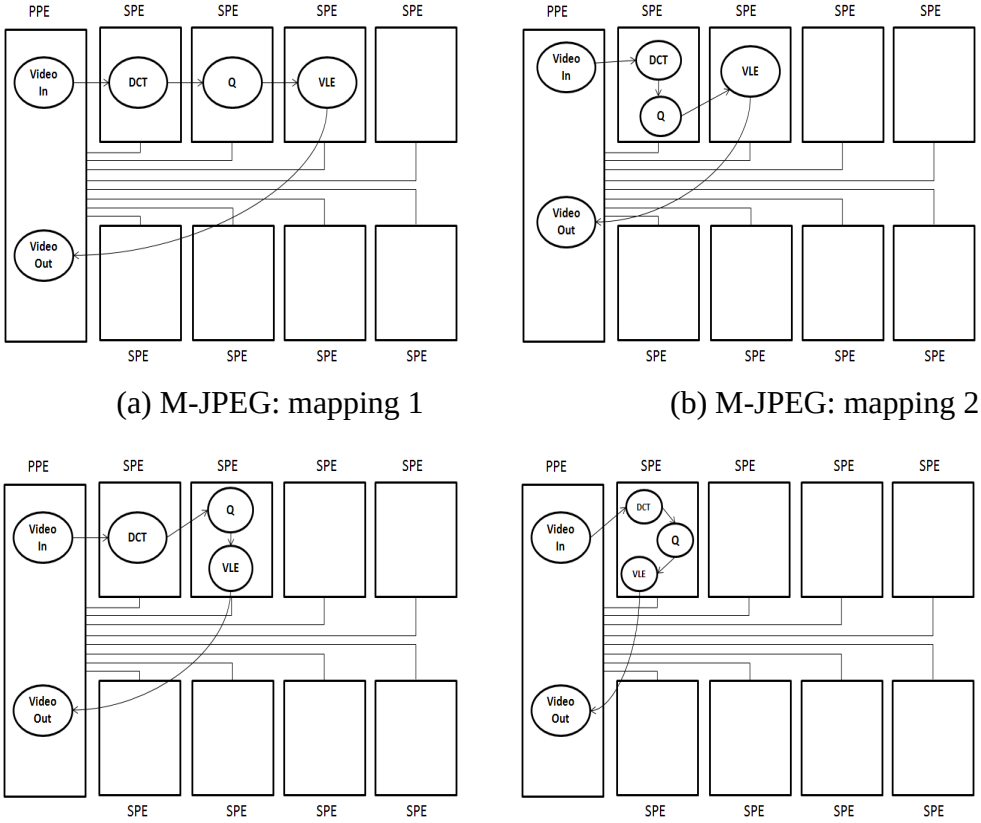


Figure 7.4: M-JPEG Process Network

Mapping	PPU	SPE1	SPE2	SPE3	Throughput
1	VideoIn and VideoOut	DCT	Q	VLE	19257625
2	VideoIn and VideoOut	DCT and Q	VLE	-	23922134
3	VideoIn and VideoOut	DCT	Q and VLE	-	19114574
4	VideoIn and VideoOut	DCT, Q and VLE	-	-	32936404

Table 7.1: Throughput measurement using different mapping possibilities for JPEG application.



(a) M-JPEG: mapping 1

(b) M-JPEG: mapping 2

(c) M-JPEG: mapping 3

(d) M-JPEG: mapping 4

Figure 7.5: Different mapping strategy for M-JPEG Application on Cell archicture

Chapter 8

Conclusions and Future Work

We have presented SCO, a new programming model for streaming applications for the Cell BE platform. One of the main contributions of this work is the possibility to map multiple processes on the Synergistic Processor Element of the Cell BE platform. The new model shares the computing power of SPEs by allowing many-to-one mappings on the Cell BE platform. A further step was the CellFlow tool itself, build upon the CELLCC tool that automatically map streaming applications on the Cell BE platform using the KPN model of computation. The CellFlow tool is a fully automated tool that can convert streaming applications written in Matlab into SCO format by using the SBF dataflow model as an intermediate model. CellFlow provides some flexibility when assigning processes to an SPE. We can map an arbitary number of processes on a SPE using the CellFlow tool. The only limitation will be the memory (256 KB) of an SPE.

To prove the usability of CellFlow, we have implemented the M-JPEG application in SCO format on the Cell. Since the M-JPEG application only has single channels between its processes, we also implement an self created application with multiple channels between its processes in SCO model specification. We have executed these applications on the target Cell platform and presented the results of these experiments, which proved that the SCO model approach provides facilities to map multiple processes onto one SPE with good throughput numbers.

For future work, the presented approach and results can be validated further by implementing more applications in SCO format such as the QRvr application. Implementing other application with the CellFlow tool will confirm the stability of the CellFlow tool. Another very interesting work could be to specify multiple different applications in SCO format and map these multiple independent applications on the Cell platform simultaneously. Further, the communication can be optimized. Currently for each communication that needs to be happen, we walk in a round robin manner through all channels and perform the communication. This can be optimized by only checking the neccessary channels needed for the concerned communication type. Since the observe part of the SCO model was not investigated in this thesis, it can be implemented to observe the stability of applications.

Appendix A

Code generated by saas for the schedule of ND_3

```
void initLoopIndex_ND_3() {
    k3 = 1;
    j3 = 1;
}

bool isDataPresentV1_ND_3() {
    if (hasData(ED_2) && hasData(ED_4) && hasRoomSelf(ED_1) && hasRoom(ED_10)) {
        return true;
    }
    else
        return false;
}

bool isDataPresentV2_ND_3() {
    if (hasData(ED_2) && hasData(ED_3) && hasRoomSelf(ED_1) && hasRoom(ED_10)) {
        return true;
    }
    else
        return false;
}

bool isDataPresentV3_ND_3() {
    if (hasData(ED_2) && hasData(ED_3) && hasRoomSelf(ED_1)) {
        return true;
    }
    else
        return false;
}

bool isDataPresentV4_ND_3() {
    if (hasDataSelf(ED_1) && hasData(ED_4) && hasRoomSelf(ED_1) && hasRoom(ED_10)) {
        return true;
    }
    else
        return false;
}
```

```
bool isDataPresentV5_ND_3() {
    if (hasDataSelf(ED_1) && hasData(ED_3) && hasRoomSelf(ED_1) && hasRoom(ED_10)) {
        return true;
    }
    else
        return false;
}

bool isDataPresentV6_ND_3() {
    if (hasDataSelf(ED_1) && hasData(ED_3) && hasRoomSelf(ED_1)) {
        return true;
    }
    else
        return false;
}

bool isDataPresentV7_ND_3() {
    if (hasDataSelf(ED_1) && hasData(ED_4) && hasRoom(ED_11) && hasRoom(ED_10)) {
        return true;
    }
    else
        return false;
}

bool isDataPresentV8_ND_3() {
    if (hasDataSelf(ED_1) && hasData(ED_3) && hasRoom(ED_11) && hasRoom(ED_10)) {
        return true;
    }
    else
        return false;
}

bool isDataPresentV9_ND_3() {
    if (hasDataSelf(ED_1) && hasData(ED_3) && hasRoom(ED_11)) {
        return true;
    }
    else
        return false;
}

void executeV1_ND_3() {
    unsigned int x1, x2 = 0;
    unsigned int y1, y2 = 0;

    read(ED_2, &x1);
    read(ED_4, &x2);
    y1 = x1;
    y2 = x2;
    writeSelf(y1, ED_1);
    write(y2, ED_10);
}

void executeV2_ND_3() {
    unsigned int x1, x2 = 0;
    unsigned int y1, y2 = 0;

    read(ED_2, &x1);
    read(ED_3, &x2);
```

```
    y1 = x1;
    y2 = x2;
    writeSelf(y1, ED_1);
    write(y2, ED_10);
}

void executeV3_ND_3() {
    unsigned int x1, x2 = 0;
    unsigned int y1 = 0;

    read(ED_2, &x1);
    read(ED_3, &x2);
    y1 = x1;
    writeSelf(y1, ED_1);
}

void executeV4_ND_3() {
    unsigned int x1, x2 = 0;
    unsigned int y1, y2 = 0;

    readSelf(ED_1, x1);
    read(ED_4, &x2);
    y1 = x1;
    y2 = x2;
    writeSelf(y1, ED_1);
    write(y2, ED_10);
}

void executeV5_ND_3() {
    unsigned int x1, x2 = 0;
    unsigned int y1, y2 = 0;

    readSelf(ED_1, x1);
    read(ED_3, &x2);
    y1 = x1;
    y2 = x2;
    writeSelf(y1, ED_1);
    write(y2, ED_10);
}

void executeV6_ND_3() {
    unsigned int x1, x2 = 0;
    unsigned int y1 = 0;

    readSelf(ED_1, x1);
    read(ED_3, &x2);
    y1 = x1;
    writeSelf(y1, ED_1);
}

void executeV7_ND_3() {
    unsigned int x1, x2 = 0;
    unsigned int y1, y2 = 0;

    readSelf(ED_1, x1);
    read(ED_4, &x2);
    y1 = x1;
    y2 = x2;
    write(y1, ED_11);
```

```
    write(y2, ED_10);
}

void executeV8_ND_3() {
    unsigned int x1, x2 = 0;
    unsigned int y1, y2 = 0;

    readSelf(ED_1, x1);
    read(ED_3, &x2);
    y1 = x1;
    y2 = x2;
    write(y1, ED_11);
    write(y2, ED_10);
}

void executeV9_ND_3() {
    unsigned int x1, x2 = 0;
    unsigned int y1 = 0;

    readSelf(ED_1, x1);
    read(ED_3, &x2);
    y1 = x1;
    write(y1, ED_11);
}

void update_ND_3() {
    if (j3 <= 5) {
        j3++;
    }
    else if (j3 > 5) {
        if (k3 <= 5) {
            k3++;
        }
    }
}



void selectVariant_ND_3() {
    if (k3-1 && j3-1) {
         v3 = 1;
    }
    else if (2 <= j3 && j3 <= 4) {
        if (k3-1) {
             v3 = 2;
        }
    } // end if j3
    else if (k3-1 && j3-5) {
         v3 = 3;
    }
    else if (2 <= k3 && k3 <= 4) {
        if (j3-1) {
             v3 = 4;
        }
        else if (2 <= j3 && j3 <= 4) {
             v3 = 5;
        } // end if j3
        else if (j3-5) {
             v3 = 6;
```

```
        }
    } // end if k3
    else if (k3-5 && j3-1) {
         v3 = 7;
    }
    else if (2 <= j3 && j3 <= 4) {
        if (k3-5) {
             v3 = 8;
        }
    } // end if j3
    else if (k3-5 && j3-5) {
         v3 = 9;
    }
    else
        v3 = 0;
}


void fireVariant_ND_3() {
    if (v3 == 1) {
        if (isDataPresentV1_ND_3()) {
            executeV1_ND_3();
            update_ND_3();
        }
    }
    else if (v3 == 2) {
        if (isDataPresentV2_ND_3()) {
            executeV2_ND_3();
            update_ND_3();
        }
    }
    else if (v3 == 3) {
        if (isDataPresentV3_ND_3()) {
            executeV3_ND_3();
            update_ND_3();
        }
    }
    else if (v3 == 4) {
        if (isDataPresentV4_ND_3()) {
            executeV4_ND_3();
            update_ND_3();
        }
    }
    else if (v3 == 5) {
        if (isDataPresentV5_ND_3()) {
            executeV5_ND_3();
            update_ND_3();
        }
    }
    else if (v3 == 6) {
        if (isDataPresentV6_ND_3()) {
            executeV6_ND_3();
            update_ND_3();
        }
    }
    else if (v3 == 7) {
        if (isDataPresentV7_ND_3()) {
            executeV7_ND_3();
            update_ND_3();
```

```
        }
    }
    else if (v3 == 8) {
        if (isDataPresentV8_ND_3()) {
            executeV8_ND_3();
            update_ND_3();
        }
    }
    else if (v3 == 9) {
        if (isDataPresentV9_ND_3()) {
            executeV9_ND_3();
            update_ND_3();
        }
    }
}

void schedule_ND_3() {
    selectVariant_ND_3();
    fireVariant_ND_3();
}
```

Bibliography

[1] International Business Machines Corporation (IBM). “The Cell project at IBM Research”: http://www.research.ibm.com/cell/.

[2] Sander van der Maar. “Tomography mapped onto the Cell Broadband Processor”, 2007.

[3] Xin David Zhang. “A streaming Computation Framework for the Cell Processor”, 2007.

[4] Michael Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Christopher Leger, Andrew A. Lamb, Jeremy Wong, Henry Homan, David Z. Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In ASPLOS, 2002.

[5] Edward Ashford Lee and David G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. IEEE Transactions on Computers, 1987.

[6] Emanuele Cannella. “Performance Evaluation of Multi-threading Operating Systems in MPSoCs Generated by ESPAM”, 2007-2008.

[7] Leiden Institute for Advanced Computer Science (LIACS). “Compilation of Matlab to Process Networks (Compaan)”: http://www.liacs.nl/ cserc/compaan/.

[8] Bart Kienhuis and Ed F. Deprettere. “Modeling Stream-Based Applications using the SBF model of computation”.

[9] Cedric Bastoul. “CLooG, A Loop Generator For Scanning Polyhedra”.

[10] Steven Derrien, Alexandru Trujan, Claudiu Zissulescu, Bart Kienhuis and ED F. Deprettere. “Deriving efficient control in Process Networks with Compaan/Laura”.

[11] R.L. Walke, R.W.M. Smith, and G. Lightbody. 20GFLOPS QR processor on a Xilinx Virtex-e FPGA. In proceedings of SPIE advanced signal, 1999.