Stony Brook University

OFFICIAL COPY

The official electronic file of this thesis or dissertation is maintained by the University Libraries on behalf of The Graduate School at Stony Brook University.

© All Rights Reserved by Author.

Reconfigurable Architecture for Mixed Processing Elements with Buffer Based Representation

A Thesis Presented

by

Sumit Tiwari

to

The Graduate School

in partial fulfillment of the

requirements

for the Degree of

Master of Science

in

Electrical Engineering

Stony Brook University

December 2014

Stony Brook University
The Graduate School

Sumit Tiwari

We, the thesis committee for the above candidate for the
Master of Science degree,
hereby recommend acceptance of this thesis.

Dr. Sangjin Hong – Thesis Advisor
Professor, Department of Electrical and Computer Engineering

Dr. Alex Doboli – Second Reader
Professor, Department of Electrical and Computer Engineering

This thesis is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Thesis

Reconfigurable Architecture for Mixed Processing Elements with Buffer Based Representation

by

Sumit Tiwari

Master of Science

in

Electrical Engineering

Stony Brook University

2014

Modern reconfigurable logic devices are often multi-core and multi-processor architectures with complex intra-processor logic. To fully utilize the raw power of these devices is an arduous task; mapping design data-flows to such devices in a way that will maximize their performance involves careful consideration of a number of parameters, and is the subject of a good amount of research. This thesis presents a novel design, which uses the technique of buffer-based dataflow, a representation technique for realizing data-centric applications in reconfigurable platforms, to map complex logic systems with multiple processing elements to a reconfigurable target architecture having multi-core processors or multiple processors. The use of multi-core processors requires careful synchronization between the processing elements and we propose employing the buffer-based dataflow technique in conjunction with a controller to map the processing logic onto the reconfigurable platform and deal with the synchronization issues. The logic is implemented using a series of buffers and interconnections, and these are controlled by a

top-level global controller, responsible for their configuration and reconfiguration as well as path selection to enable dynamic switching between designs. The dynamic reconfigurability gained from our approach allows us to map multiple processing elements onto a single core and switch between them during run-time while maximizing performance. The proposed design is evaluated with SystemC and Xilinx ISE.

To The Four

Most Important People

In My Life

Table Of Contents

List of Figures viii

List of Tables x

Acknowledgements xi

1 Introduction 1

1.1 Buffer-Based Dataflow Design Approach Overview ………..………………….. 4

1.2 Constructing a Buffer-Based Dataflow ………………………………………….. 6

2 Mapping Characterization 9

2.1 Timing Information for Processing Elements …………………………………… 11

2.2 Multi-rate Support and Iteration Period Control ………………………………… 14

2.3 Mapping onto Target Platform …………………………………………………... 18

3 Controller Design 21

3.1 Overview ………………………………………………………………………… 21

3.2 Temporal Aspects ………………………………………………………………... 21

3.2.1 For Fixed Execution Times ……………………………………………… 23

3.2.2 For Variable Execution Times …………………………………………… 27

3.3 Structural Aspects ………………………………………………………………... 30

3.4 Overall Controller Design ……………………………………………………….. 37

4 Resource Utilization 39

4.1 Processor Utilization of Mapping ………………………………………………... 39

4.2 Power Consumption and Bus Utilization of Mapping ………………………….. 42

5 Evaluation **45**

5.1 Experimental Setup ………………………………………………………………. 45

5.2 Processing Elements Mapped as Hardware Logic ……………………………….. 48

5.3 Processing Elements Mapped to Processors ……………………………………… 50

6 Summary **52**

6.1 Future Scope ……………………………………………………………………… 52

6.2 Conclusions ………………………………………………………………………. 53

Bibliography **54**

List of Figures

1.1 A signal processing application represented as a dataflow................................ 5

1.2 Dataflow of Figure 1.1 obtained by inserting buffers...................................... 5

2.1 The dataflow for our proposed methodology... 10

2.2 Synchronization of data transfers through global and buffer controllers.................. 11

2.3 Example BBDF to obtain timing information.. 12

2.4 Design of Figure 2.3 mapped to a platform with interconnects and buffers with processing elements realized as hardware logic... 12

2.5 An example of BBDF and its buffer activity when different clock rates are specified for different processing elements.. 15

2.6 Illustration of iteration period adjustment by *start* signal manipulation.................. 16

2.7 Buffer activities at different speeds, illustrating that the storage requirements depend on signal timings.. 17

2.8 Dataflow of Figure 2.3 mapped to multi-processor system................................ 18

2.9 Timing of read and write signals when processing elements are mapped to processors as per Figure 2.8.. 20

3.1 The execution controller along with its connections and signals........................... 22

3.2 Timing diagram and program structure for a BBDF with different iteration periods and non-overlapping buffer execution.. 24

3.3 Timing diagram and corresponding program structure for iteration periods with overlapping buffer execution.. 26

3.4 The effects of delay: (a) A delay introduced in the design causes no issues as long as the iteration period is sufficiently large. (b) A faster design results in the buffers

operating prematurely and producing incorrect results………………………………... 28

3.5 Buffer controller adapted for variable execution times, and its signal timings for correct read/write operations……………………………………………………………... 29

3.6 Connection of the processor and the buffers using a bus……………………………… 30

3.7 Structural controller and its connection to the buffer controllers and interconnect switches………………………………………………………………………………… 31

3.8 The concept of buffer sharing: (a) Buffers 1 and 2 in different paths, and cannot be active at the same time. (b) They are replaced by a single buffer……………………... 32

3.9 Timing of buffer activities before and after buffer sharing……………………………. 33

3.10 Multiple paths between two processors………………………………………………... 34

3.11 Configuration of the paths, and their selection using global controller………………... 35

3.12 Partition structure and signals………………………………………………………….. 36

3.13 Processor deciding its own path through the global controller………………………… 37

3.14 Overall controller structure (the 'global' controller)…………………………………... 38

4.1 An example dataflow…………………………………………………………………... 39

4.2 Execution timing representation to map PEs to processors. (a) Execution timing for mapping PEs as per Figure 4.1. (b) Execution timing when mapped to processors…… 40

4.3 Considering power consumption for interconnected processors when elements of Figure 4.2(a) are mapped to them…………………………………………………….… 43

4.4 Estimated execution times when processing blocks of Figure 4.2(a) are mapped to four processors…………………………………………………………………………. 44

5.1 The first buffer-based dataflow used for evaluation…………………………………… 46

5.2 The second buffer-based dataflow used for evaluation………………………………... 46

5.3 Buffer timing flow for the dataflow in Figure 5.1……………………………………... 48

5.4 Buffer timing flow for the dataflow in Figure 5.2……………………………………... 49

5.5 Simulation results when processing elements are mapped to processors……………… 50

List of Tables

1.1 Operational Dependency derived from Figure 1.1…………………………………….. 7

2.1 Buffer Controller Parameters for platform in Figure 2.4……………………………… 13

2.2 New Buffer Controller Parameters for BBDF in Figure 2.4 (now mapped as in Figure 2.8)…………………………………………………………………………….. 19

5.1 Buffer controller configuration for dataflow in Figure 5.1……………………………. 47

5.2 Buffer controller configuration for dataflow in Figure 5.2……………………………. 47

ACKNOWLEDGEMENTS

While I will be credited as the sole author of this thesis, it is definitely a collaborative effort only made possible by the support of so many. I would thus like to begin by thanking my mentor and advisor, Professor Sangjin Hong, for agreeing to oversee my thesis, providing much needed guidance and for staying optimistic during the long droughts between insights. His awe-inspiring erudition, insistence on independent thinking, and amazing foresight has helped me in more ways than I can imagine. I would also like to thank Professors Alex Doboli and Peter Milder – the knowledge and wisdom that they have imparted to me through their coursework, and otherwise, will surely prove to be indispensable to me in my future.

Amongst the others that have guided and assisted me throughout my journey, I must first thank Tejal for being my iron pillar of support and the light of my life; Bhaskar for his constant companionship through good and bad; Michael and Julia for all their love and friendship; Abhinav, Abhishek and Lala for being brotherly figures and providing me with invaluable advice; and Anne and Jack for hosting my stay here in the States.

And last, but certainly not the least, I end by thanking my family, without whose love and support none of this would have been possible.

Chapter 1

Introduction

In many application-specific systems, one of the most important attributes is the rapid reconfigurability that allows a system to be adaptive to its changing environment. Typically, such capability is best supported by programmable processors such as digital signal processors (DSPs), amongst others. Most common DSP applications such as coding, filtering, and image processing require floating-point operations. In order to realize floating-point operations, processors are used to expedite the design cycle. In addition, processors are useful for realizing multiple applications in a time-shared manner. Thus, reconfigurable platforms such as Xilinx field-programmable gate arrays (FPGAs) include processors [1]. In the case where a complex system is represented as a dataflow graph having a large number of nodes (processing blocks), the processing blocks should be efficiently mapped to multi-core processor architecture to minimize hardware resources.

When a dataflow is synthesized in a target platform having multi-core processors and hardware logics, it becomes difficult to synchronize data transfers between processing blocks mapped to different processors (alternatively, one is mapped to a processor and the other is implemented as a hardware) because the execution time of processors varies due to the dynamic behavior of software such as interrupt handling and context switching. Thus, the execution times of processing blocks are estimated for mapping [2]–[5]. Methods for mapping on multi-core processors, which also use estimated execution times, include [6]–[8]. However, in the case where actual execution times are greater than the estimated times, the mapping based on the

estimated times may produce wrong results. To prevent this problem, Jung et al. [9] proposed a handshaking scheme between a centralized controller and sequential logics having variable execution time. However, the handshaking scheme has a limited capability to support the data transfers between processors (or between a processor and a hardware) because the data transfers on processors are also the programs having variable execution times [10].

Thus, while extensive research has been targeted towards sophisticated modeling techniques that enable us to map complex designs onto hardware, it often seems inadequate when it comes to mapping truly complicated programs onto real hardware: non trivial design issues such as flexibility of mapping methodology, complexity of controlling architecture and ease of dynamic reconfiguration remain unsolved. Moreover, such techniques focus primarily on functional modeling and place less emphasis on controller design, which often scales exponentially in complexity with a linear increase in the complexity of the design.

To deal with these issues, we propose a design methodology based on buffer-based dataflow (BBDF) [11], [12], and which is globally synchronized by a controller that handles the data transfers between multiple processing elements. BBDF is a transparent representation that bridges the gap between the algorithmic description of a design and its structural implementation with a buffer-centric perspective, as opposed to the conventional processing element-centric perspective. The proposed methodology is built around the principle of representing the algorithm as a BBDF and by using a powerful controller that can handle all the buffer parameters and interconnections, thereby making reconfiguration a straightforward process. The controller design of key importance here: because of the nature of logic architectures, the designer does not have a direct influence on the underlying platform, but rather, the architecture and the algorithmic characteristics of the controller define the achievable performance. Furthermore,

systems often contain a mix of both software and hardware-processing elements, not all of which may run on similar clock frequencies. Realizing certain elements as hardware while others as software and finally, some as processors, can lead to issues such as variable latencies, which will degrade performance. Our proposed controller design is robust and is deftly able to handle such issues, while still enabling dynamic design reconfiguration. Additionally, because the design complexity depends on the number of buffers and interconnections, as well as the temporal and structural characteristics of the design, we can make our design scalable by incorporating all of these parameters into our global controller. Thus, the main advantages of our proposed methodology are i) Feasible Dynamic Reconfiguration, ii) Design Scalability, and iii) Support for multi-frequency elements in the design.

BBDF inserts buffers between processing blocks in a given dataflow. A global controller globally synchronizes the buffer-based dataflow and every data transfer is done through the buffers between processing blocks. Due to the buffers, a pair of sending and receiving processors does not have to access the same bus simultaneously. Furthermore, the timing mismatch of data transfers due to the different bus speeds between two processors (or between a processor and a hardware) is solved with the buffer controller parameters in the level of a dataflow. By utilizing the data transfer characteristics of the buffer-based dataflow, we propose a mapping methodology for a target system having multi-core processors and programmable logics (or hardware). Since our mapping methodology does not include a hardware-software partitioning technique [13], each processing block is predetermined in such a way that it is either mapped to a processor or realized as hardware logic.

This thesis is organized as follows: Chapter 1 offers an overview on the basics of reconfigurable architecture, buffer-based dataflow, as well as the construction of such a

dataflow. Chapter 2 characterizes our design – it discusses the support for iteration period control and explains our mapping methodology. Chapter 3 illustrates the structure of the controller in our proposed design, its temporal and structural aspects as well. In Chapter 4 we take a look at the resource utilization of our methodology under different mapping conditions. In Chapter 5 we detail our experiments with the design, and present our results after mapping two complex dataflows to a multi-processor system. Finally, Chapter 6 presents our conclusions and the future scope and direction for our research.

1.1 Buffer-Based Dataflow Design Approach Overview

Application-specific systems are usually complex designs with multiple processing elements and a number of data paths and sequential elements. In general, all algorithms and designs can be represented as dataflows governed by the relation:

$$Y = P(X) \tag{1.1}$$

where an input X and an output Y are finite blocks of data, and P is the representation of the processing element (PE). X and Y are sequentially consumed and produced as data blocks, and their sizes may be different. Figure 1.1 shows an example of a signal processing application represented as such a dataflow with many processing elements. The arrows indicate the direction of the data transfers. Each processing elements includes both the functionality as well as the storage elements required for proper functioning. Processing elements in the dataflow execute the required program or algorithm on a finite set of data in every iteration period. Typical dataflows have multiple inputs, outputs, and even feedback elements.

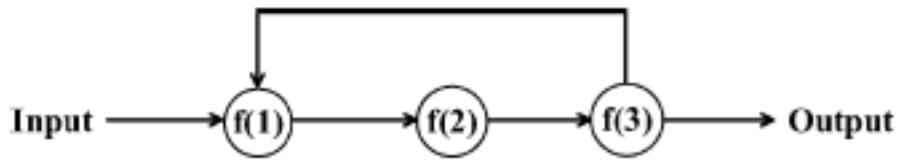


Figure 1.1: A signal processing application represented as a dataflow.

Figure 1.2 illustrates the buffer-based dataflow derived from Figure 1.1 by inserting buffers. Here, the relationship between processing blocks is isolated by inserting buffers to the edges of the dataflow in Figure 1.1. By separating the relationship between processing blocks, processing blocks are only able to represent the functionality. The isolation also enhances the reconfigurability of the overall system.

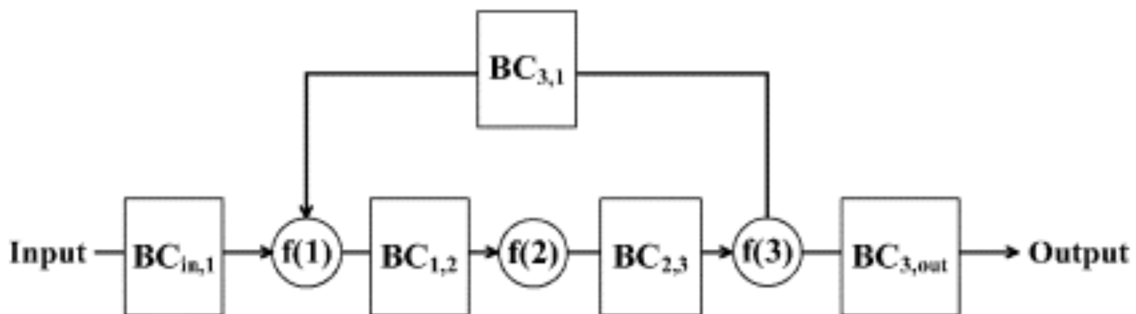


Figure 1.2: Dataflow of Figure 1.1 obtained by inserting buffers.

In the buffer-based dataflow graph, inserting buffers represents the edges delivering data frames from the source to destination. Thus, the size of data frames appearing at the input port of a buffer is the same as the size of data frames at the output port of the buffer. Furthermore, while a source processing block is writing data to a buffer, the corresponding destination processing

block is able to read data from the buffer. The buffers are realized as the dual port memory that supports simultaneous writing and reading. As illustrated in Figure 1.2, the buffer between the producer processing block *f(i)* and the consumer processing block *f(j)* is denoted as $BC_{i,j}$. The primary parameters, which determine the buffer controller structure and overall physical realization are represented as logic latency (L_i), write offset ($nw_{i,j}$), read offset ($nr_{i,j}$), block size ($M_{i,j}$), and delay factor ($D_{i,j}$)[11]. The write offset represents the difference between reading data from the previous buffer and writing data to the current buffer without considering the logic latency. The read offset is the offset from the start of writing data to $BC_{i,j}$ to the start of reading data from $BC_{i,j}$ when the writing speed of processing block *f(i)* and the reading speed of processing block *f(j)* are matched. However, if the former is slower than the latter, processing block *f(j)* does not read valid data from $BC_{i,j}$. For this, the delay factor is used to represent the rate mismatch between processing blocks *f(i)* and *f(j)*.

1.2 Constructing a Buffer-Based Dataflow

A buffer-based dataflow is constructed by using the buffer controller parameter table, which is extracted from the operational dependency of a dataflow and the offsets of fan-ins and fan-outs of processing blocks. In Table I, $e_{i,j}$ represents the edge from the source *f(i)* to the destination *f(j)*. $start_write_{i,j}$ represents the start time of writing data through $e_{i,j}$ and $start_read_{i,j}$ is the start time of reading data through $e_{i,j}$. Since writing data to $e_{i,j}$ precedes reading data from $e_{i,j}$, $start_write_{i,j} < start_read_{i,j}$. In Table I, *f(1) – f(3)* represents the operational dependency between the fan-in and fan-out edges of processing blocks. In the operational dependency of *f(3)*, $start_write_{3,1}$ is removed because $e_{3,1}$ is a feedback loop. If *f(1)* reads the data generated by *f(3)* in

the current iteration period, the dataflow falls into the deadlock situation because *f(1)* and *f(3)* keep waiting for the data generated by each other.

Parameter	Operational Dependency
$e_{input,1}$	$start_write_{input,1} < start_read_{input,1}$
$e_{1,2}$	$start_write_{1,2} < start_read_{1,2}$
$e_{2,3}$	$start_write_{2,3} < start_read_{2,3}$
$e_{3,1}$	$start_write_{3,1} < start_read_{3,1}$
$e_{3,output}$	$start_write_{3,output} < start_read_{3,output}$
f(1)	$\max\{start_read_{input,1}, start_read_{3,1}\} + L_1 \leq start_write_{1,2}$
f(2)	$start_read_{1,2} + L_2 \leq start_write_{2,3}$
f(3)	$start_read_{2,3} + L_3 \leq start_write_{3,output}$

Table 1.1: Operational Dependency derived from Figure 1.1

In the buffer controller $BC_{i,j}$, the start signals are realized with the primary parameters introduced in Section 1.1 as follows:

$$start_write_{i,j} = L_i + nw_{i,j} + start_i \quad (1.2)$$

$$start_read_{i,j} = start_write_{i,j} + \max(nr_{i,j}, D_{i,j}) \quad (1.3)$$

$$stop_write_{i,j} = start_write_{i,j} + M_{i,j} \quad (1.4)$$

$$stop_read_{i,j} = start_read_{i,j} + M_{i,j} \quad (1.5)$$

In (1.2), $start_i$ is the time value in which *f(i)* begins reading data from the previous buffer controller through its fan-in port. Equation (1.3) reflects the rate mismatch between the source processing block and the destination processing block. In one iteration period, the data transfer

through each buffer controller is done once. Thus, once data has been written to (or read from) the buffer controller $BC_{i,j}$, the data is continuously being written to (or read from) $BC_{i,j}$ until the size of transferred data reaches $M_{i,j}$. Equations (1.4) and (1.5) represent the end time of writing and reading data to/from the buffer controller, respectively. The buffer memory size of $BC_{i,j}$, $MEM(BC_{i,j})$ is given by

$$MEM\left(BC_{i,j}\right) = \min\{M_{i,j}, \left(start_read_{i,j} - start_write_{i,j}\right)\} \tag{1.6}$$

In (1.6), when the reading of $BC_{i,j}$ starts before the end of writing data to $BC_{i,j}$, $MEM(BC_{i,j})$ is determined by the difference between $start_read_{i,j}$ and $start_write_{i,j}$. In this case, while *f(i)* is writing data to $BC_{i,j}$, *f(j)* can read data from $BC_{i,j}$. However, if the reading of $BC_{i,j}$ starts when the writing $BC_{i,j}$ is completed, $MEM(BC_{i,j})$ is $M_{i,j}$.

Chapter 2

Mapping Characterization

When a design dataflow is synthesized in a target platform having multi-core processors and programmable logics (such as the newer generation of FPGA devices), it is difficult to synchronize data transfers between processors, or between hardware and processors, because the programs running on a processor have variable execution times.

In order to synchronize the data transfers at the level of a dataflow graph, we use the BBDF approach for mapping processor blocks to processors. Our design creates a mapped 'partition' (defined completely in later sections) from the estimated times for functional executions and data transfers, and the resource constraints for the target platform. Due to the synchronization issues discussed above, our mapping algorithm tries to map consecutive processing blocks to the same processor. The data transfers of processing blocks mapped to processors are realized as target-dependent programs. Figure 2.1 shows the overall flow of our design methodology.

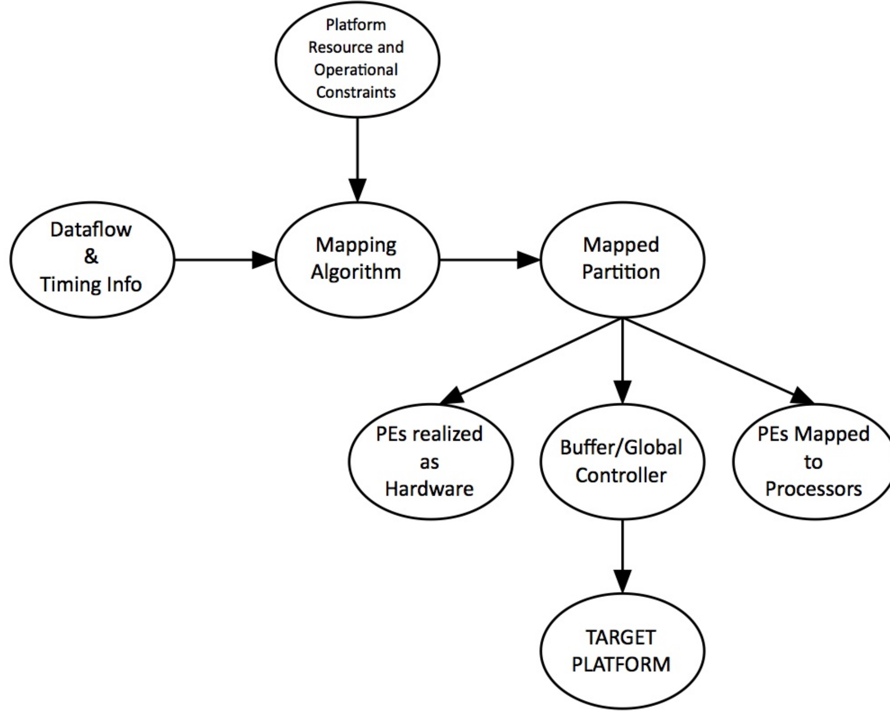


Figure 2.1: The data flow for our proposed methodology.

As shown in the above figure, the mapped partition uses the matching algorithm based on the timing information of individual processing elements. When it is finally mapped onto the target platform, synchronization of data transfers is realized through a global controller (and buffer controllers) as shown in Figure 2.2. It shows $\boldsymbol{f(i)}$ mapped as a processor, and $\boldsymbol{f(j)}$ implemented as hardware. The data transfer between the two elements is handled by $\mathbf{BC_{i,j}}$. In order to synchronize data transfers between them, the global controller generates $\mathbf{W_BC_{i,j}}$ and $\mathbf{R_BC_{i,j}}$; the former being the signal that enables $\boldsymbol{f(i)}$ to write data to $\mathbf{BC_{i,j}}$, whereas the latter initiates $\boldsymbol{f(j)}$ to read data from $\mathbf{BC_{i,j}}$.

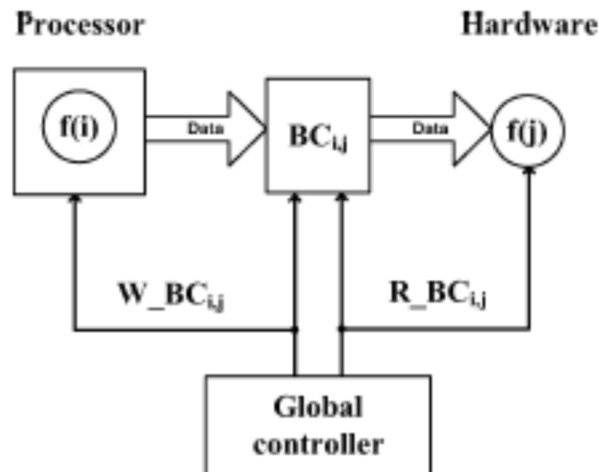


Figure 2.2: Synchronization of data transfers through global and buffer controllers.

2.1 Timing Information for Processing Elements

As can be seen in Figure 2.1, correct timing information is crucial to the proper functioning of the design. To obtain this timing information from any design, we map individual processing elements as hardware logic. The reason we choose to realize all elements as hardware logics is because we assume that no suitably efficient algorithm exists to map all elements to processors (which is indeed the focus of this design). Figure 2.3 shows an example BBDF and Figure 2.4 shows it mapped to a target platform connected by reconfigurable interconnects.

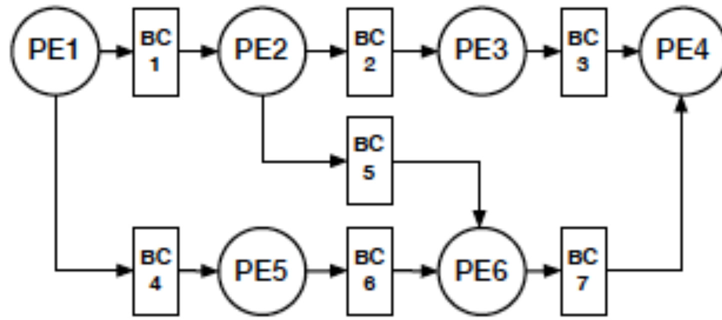


Figure 2.3: Example BBDF to obtain timing information

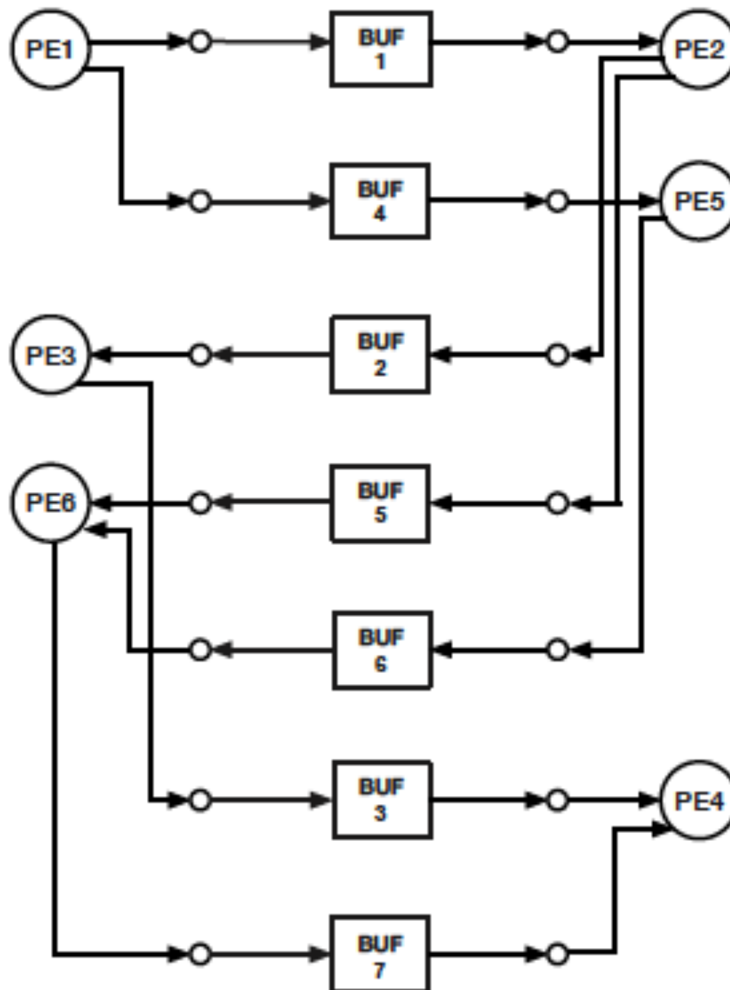


Figure 2.4: Design of Figure 2.3 mapped to a platform with interconnects and buffers with processing elements realized as hardware logic.

Using the previously derived relationships between the start signals (equations 1.2 to 1.5), we now try to approximate the read and write times for each buffer, and these are illustrated in Table 2.1. These times are based on the assumption that the execution time of each processing element is 50 cycles. The buffer controller requires this information to make sure that read and write operations are performed correctly. In the dataflow of Figure 2.3, data coming out of buffers 5 and 6 should arrive at processing element 6 at the same time as per the correct program flow. To make sure that processing element 6 will process the correct data frame, the parameters $\boldsymbol{nw}_{\mathbf{1,4,5,6}}$ and $\boldsymbol{nr}_{\mathbf{1,4,5,6}}$ are calculated to get the same read times for buffers 5 and 6. The same technique is used to calculate the other read and write times. The principle used is that when there are several fan-ins to a processing element, the effective data coming out of those fan-ins should arrive at the processing element at the same time.

BC	**L**	**nw**	**nr**	**D**	**M**	**start_time**	**start_write**	**start_read**
BC_1	10	1	1	0	32	10	21	22
BC_2	10	10	1	0	32	72	92	93
BC_3	15	12	1	0	32	143	170	171
BC_4	15	5	1	0	32	10	30	31
BC_5	30	6	1	0	32	72	108	109
BC_6	20	7	1	0	32	81	108	109
BC_7	10	1	1	0	32	159	170	171

Table 2.1: Buffer Controller Parameters for platform in Figure 2.4.

2.2 Multi-rate Support and Iteration Control

While most designs usually have processing elements operating at the same frequencies (in which case, the start and stop signals are governed by the relations given in equations 1.2 to 1.5), this is not always the case as required by the designer. Figure 2.5 shows a situation where the write operation to a buffer by PE_i and the read operation from the same buffer by PE_j occurs at different clock rates. In order to support such situations, the delay factor $D_{i,j}$ is used to synchronize the different elements within the buffer controller. We define f_i and f_j as the clock rates (in Hertz) of PE_i and PE_j respectively. When f_i is greater than f_j the slower process does not have to wait as long as there is at least one block of valid data in the buffer. In this case, there is no overflow, since the next data block is not generated before the current data block is completely used (read, in this case), and therefore the delay factor is not needed. However, if f_i is less than f_j the faster process has to wait in order to prevent data underflow, until enough data is written to the buffer. In this case, the following equation gives the minimum delay:

$$D_{i,j} = \left\lceil \left(\frac{M - nr_{i,j}}{f_i} - \frac{M - nr_{i,j} - 1}{f_j} \right) \times f_j \right\rceil \quad (2.1)$$

The control signals to the buffer, *start_write* and *start_read* should be synchronized according to the clock rates of individual processing elements, since the control signals are activated by the execution controllers using the global clock rate: we define the global clock rate $\boldsymbol{f_G}$ (in Hertz) as the rate of the fastest clock rate in the design that can be used to prevent missing any control signal. Then, *start_write* and *start_read* are obtained by multiplying $\boldsymbol{f_G/f_i}$ to the writing parameters ($\boldsymbol{L_i}$ and $\boldsymbol{nw_{i,j}}$) and $\boldsymbol{f_G/f_j}$ to the reading parameters ($\boldsymbol{D_j}$ and $\boldsymbol{nr_{i,j}}$) respectively.

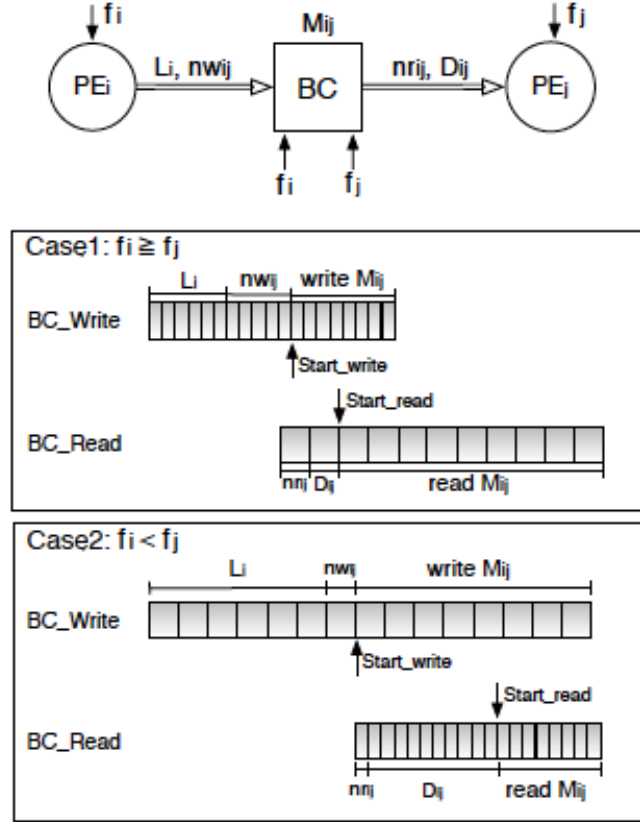


Figure 2.5: An example of a BBDF and its buffer activity when different clock rates are specified for different processing elements.

Thus, equations 1.2 and 1.3 are modified as:

$$start_write_{i,j} = \left\lceil (L_I + nw_{i,j}) \times \frac{f_G}{f_i} \right\rceil \tag{2.2}$$

$$start_read_{i,j} = start_write_{i,j} + \left\lceil (D_j + nr_{i,j}) \times \frac{f_G}{f_j} \right\rceil \tag{2.3}$$

The key benefits of enabling straightforward multi-rate support is that we can assign an arbitrary clock to any processing element while still satisfying the iteration period requirements of the design. When high-speed processing is necessary for a particular processing element, we

set it to operate at a higher frequency in a critical section of the design. Similarly, all elements are by default assigned the lowest clock-speed that satisfies the iteration period requirements of the design. This makes our architecture power aware and minimizes the power footprint of our methodology. Figure 2.6 illustrates two possible cases of iteration periods given a BBDF.

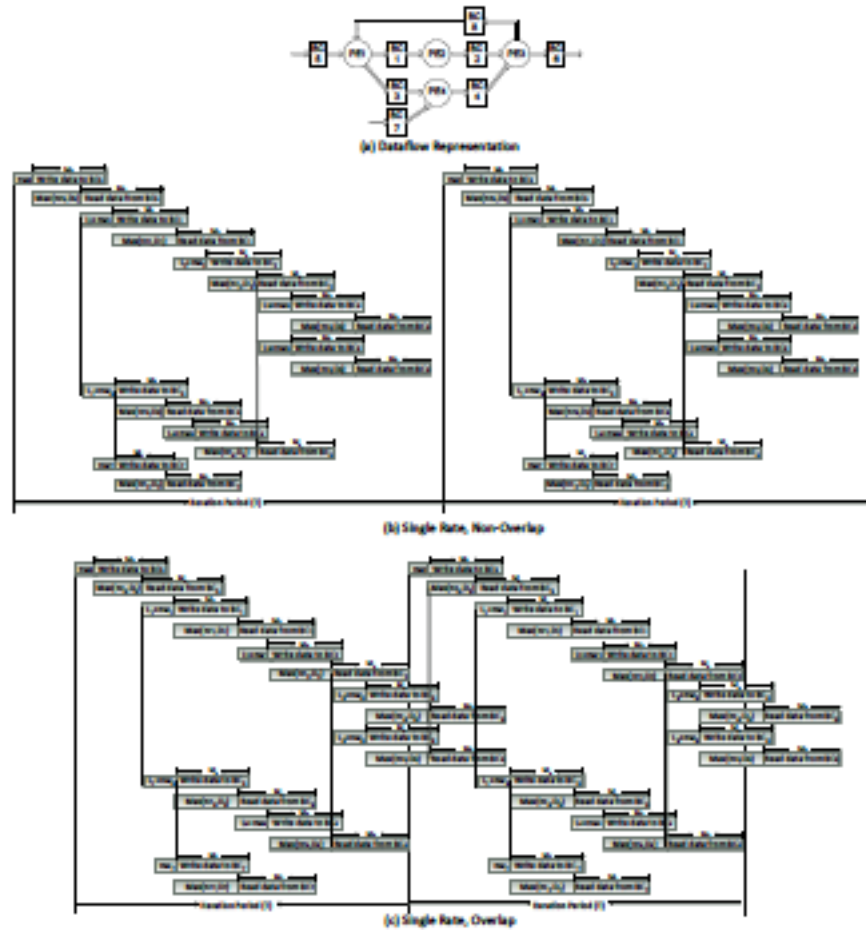


Figure 2.6: Illustration of iteration period adjustment by *start* signal manipulation.

As shown in the figure, many iteration periods are made possible by simply varying the timing of the *start* control signal, and the buffer controller handles the other control signals. Individual

buffer speeds may vary as long as the fan-in constraints, which are enforced such that the original execution characteristics are not modified, are satisfied.

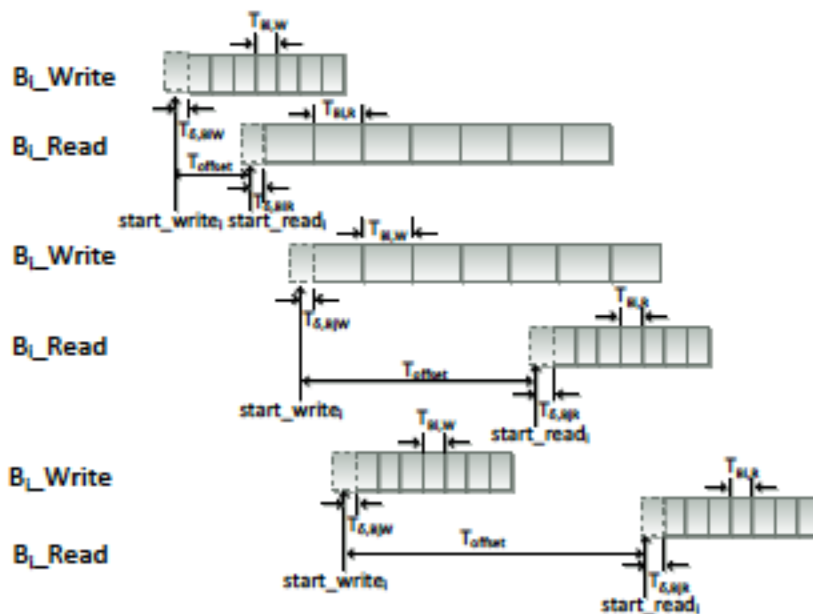


Figure 2.7: Buffer activities at different speeds, illustrating that the storage requirements depend on the signal timings.

Figure 2.7 shows read and write operations being performed at different speeds. Even though the data block size $M_{i,j}$ may be too large for $BC_{i,j}$, the actual storage required by the implementation is not. For each buffer, the *start_write* and *start_read* signals are separated by t_{offset}. The storage requirement for each buffer is thus given by:

$$storage_size_i = \left\lceil \frac{start_read_i - start_write_i}{T_{wi}} \right\rceil \tag{2.4}$$

where T_{wi} is the time required to successfully complete a write operation. Hence, the total amount of data storage required by the program depends on the control signals generated by the

buffer controller. In the event that t_{offset} is larger than the entire buffer activity duration, the storage requirement will be limited by the block size $M_{i,j}$.

2.3 Mapping onto Target Platform

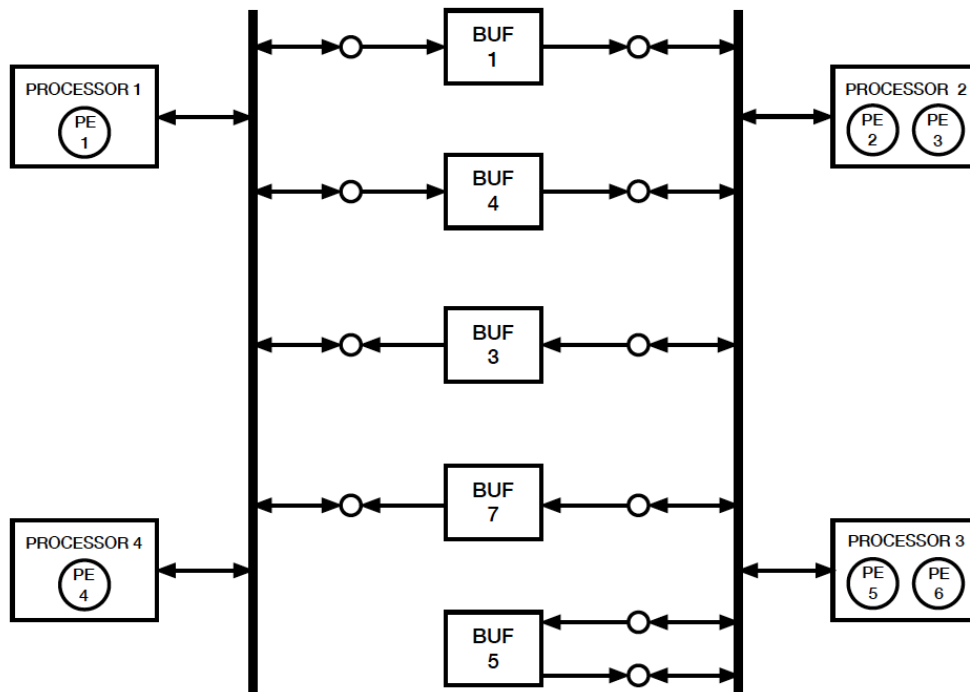


Figure 2.8: Dataflow of Figure 2.3 mapped to a multi-processor system (no elements realized as hardware logic)

After obtaining timing information from the BBDF, and satisfying the constraints of the given iteration period, our methodology will attempt to map the design to the target platform. Figure 2.8 shows such a mapping. The division of processing elements is based on the principles

previously derived. The processors communicate via a shared bidirectional bus, which in turn is connected to the interconnect structure which enables data transfer to and from the buffers. This mapping method can encounter several problems in real systems. In the case that processing elements in one processor are on several different sequential data paths and rely on the outputs of the previous processing elements, the overall execution time will increase since certain processing elements will have to wait longer.

BC	L	nw	nr	D	M	start_time	start_write	start_read
BC_1	10	1	1	0	32	10	21	22
BC_3	15	1	1	0	32	127	143	144
BC_4	15	1	1	0	32	53	69	70
BC_5	30	1	1	0	32	72	103	104
BC_7	10	1	1	0	32	175	186	187

Table 2.2: New Buffer Controller Parameters for BBDF in Figure 2.4 (now mapped as in Figure 2.8).

Based on the assumption that the execution time of each processing element is 50 cycles, the execution time of the mapped processor is the same or combination of two processing elements. The buffer controller parameters are then changed as shown in Table 2.2. The total iteration time increases as the read/write operations performed through the bus take longer. The following equations provide the new relationships between the different signals taking into account the longer time; as is evident, the data from buffers 3 and 7 arrives at different times and this would lead to wrong results:

$$start_write_{1,5} \geq start_write_{1,2} + M_{1,2} = stop_write_{1,2} \quad (2.5)$$

$$start_write_{3,4} \geq start_write_{2,6} + M_{2,6} = stop_write_{2,6} \quad (2.6)$$

$$start_read_{2,6} \geq start_read_{1,5} + M_{1,5} = stop_read_{1,5} \quad (2.7)$$

$$start_read_{6,4} \geq start_read_{3,4} + M_{3,4} = stop_read_{3,4} \quad (2.8)$$

Figure 2.9 demonstrates the signal timings of the dataflow mapped to the multi-processor platform. These signal timings depend upon the operations as well as the handshake signals required by the processors. Since all the processors are using a common bus, iteration time is longer as each processor must wait to get control over the bus. This iteration time can be made smaller by increasing the processor speed, as all the handshake signals will then arrive faster. The number of buffers, and thus the overall complexity, in the mapped platform with processors can be reduced by applying the concept of 'buffer sharing' (explained in later sections). This requires the use of additional signals, and is handled by the execution and structural controllers during run-time. Our platform uses this technique heavily and our final results are based on its use.

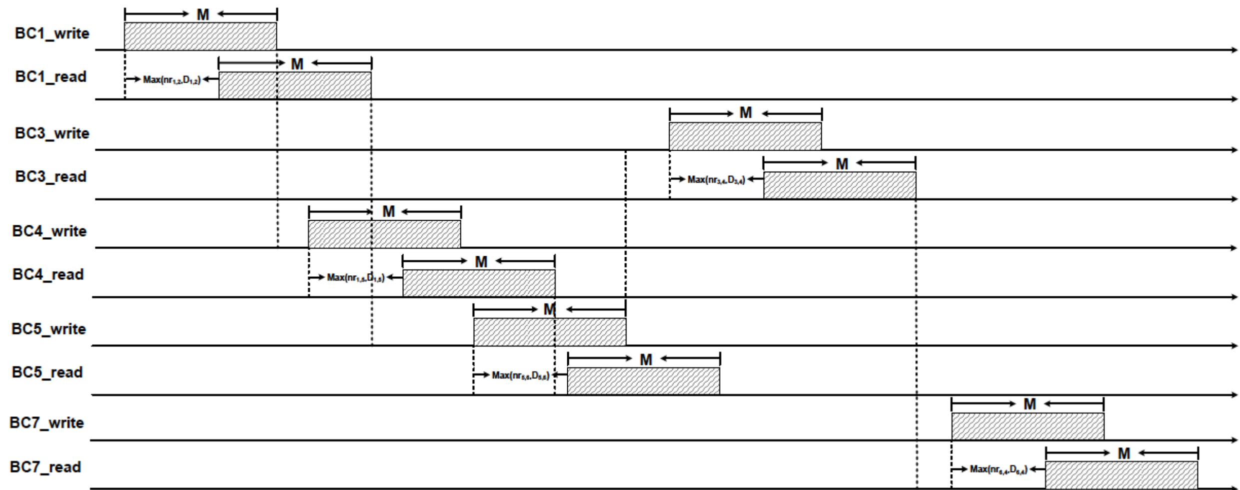


Figure 2.9: Timing of read and write signals when processing elements are mapped to processors as per Figure 2.8.

Chapter 3

Controller Design

3.1 Overview

As explained previously, the controller is one of the most crucial parts of our design. In order to ensure that the program produces correct outputs, it is necessary to use correctly timed control signals as well as synchronizing data transfers between elements. The controller handles all of these issues working in conjunction with the local buffer controllers for each buffer.

The global controller that we have defined in the previous sections is actually a combination of two separate controllers – the structural controller and the execution controller. The reasons for splitting functionality between two controllers will be made clear in the coming text. In this section, we first analyze the temporal aspects of our controller as we analyze the execution controller. We then look at the structural controller, responsible for handling the structural aspects of the program.

3.2 Temporal Aspects

Figure 3.1 shows the structure of the execution controller that handles the execution signals for the design, incorporating within itself the functionality to handle different iteration periods. The execution controller generates bit patterns for controlling the buffer controllers. Every two bits of information describe the activation signals for the write and read operations respectively. For example, the pattern "10" would start the write operation but not the read

operation for a particular buffer controller. Similarly, a sequence such as "01010000" would only start the read operations for the first two buffers, while the other two buffers are left inactivated. The bit patterns are thus stored as programs in the program memory as a large sequence of 1s and 0s. Thus, modifying the program memory contents easily controls the execution timing of the data flow. As an example, if we want to delay the iteration period of the algorithm by a hundred cycles, we simply insert one hundred 0s into the program content.

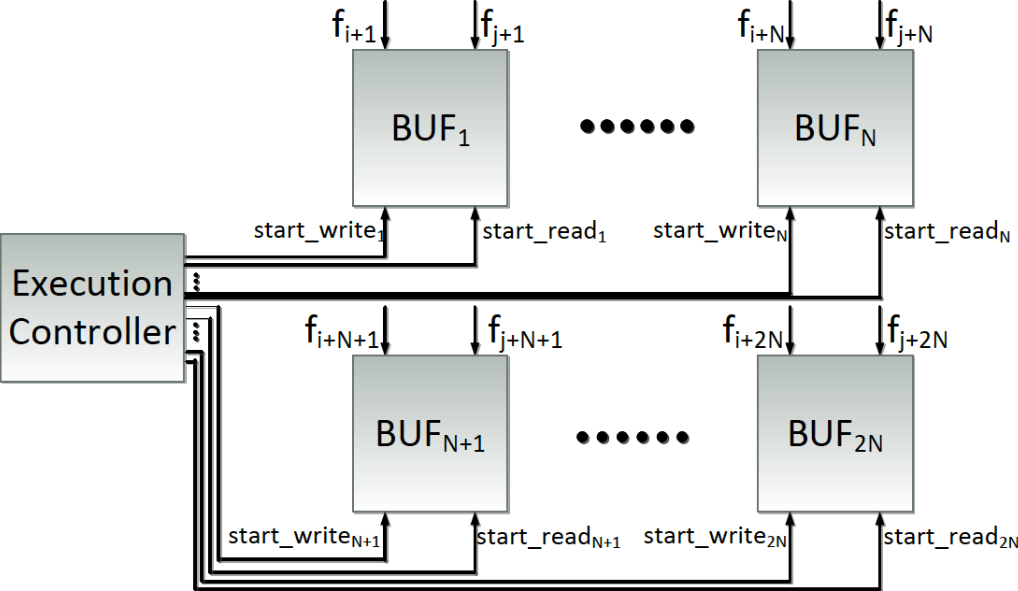


Figure 3.1: The execution controller along with its connections and signals.

The execution controller relies on accurate timing information from the mapping algorithm. This information is directly dependent on the execution times of individual processing elements. Thus, depending on whether the execution times are fixed or not (which in turn depends on whether the elements are realized as hardware logics or mapped onto processors), we

will obtain different timing information. Here, we consider each case separately and observe its impact on controller design.

3.2.1 For Fixed Execution Times

Since our design is to allow different iteration periods, we must consider the effect of this iteration period of the temporal aspects of the controller. Figure 3.2 shows the timing diagram of an example BBDF with different iteration period requirements and non-overlapping buffer activity. For these timing diagrams, their corresponding program content structure is shown below. Extracting any portion of the timing can generate the execution program, as long as the length of the interval is equal to the iteration period. Since this example has only two buffers, the width of the program content is four. In addition, the number of non-zero rows is at most four, since there are four instances where the *start* signals are generated. If two or more buffer start times are identical, the two rows may contain multiple 1s.

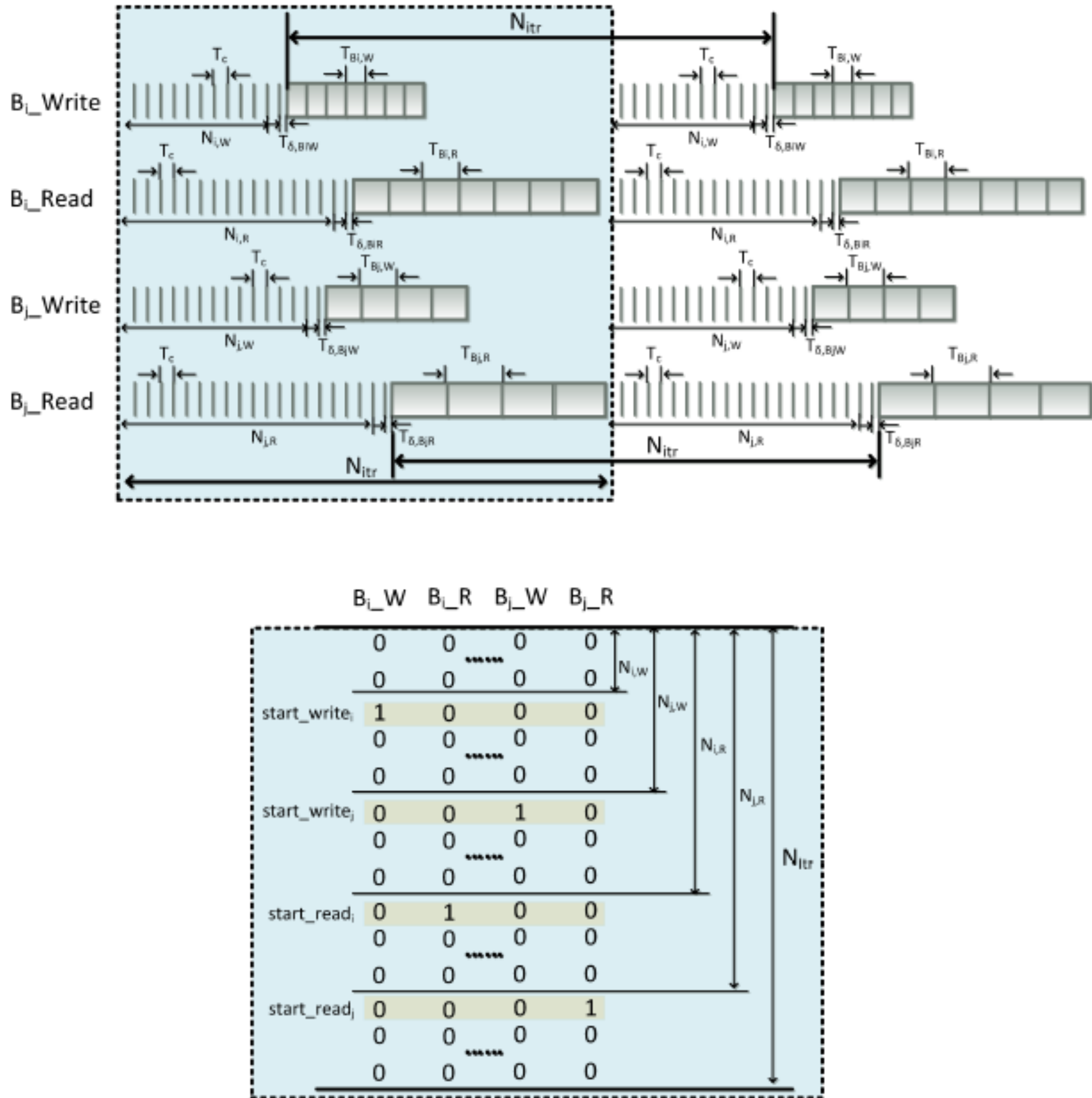


Figure 3.2: Timing diagram and program structure for a BBDF with different iteration periods and non-overlapping buffer execution.

Assuming the controller frequency as shown in the figure, the total length of the program is:

$$N_{itr} = T_{itr} \times f_{controller} \tag{3.1}$$

where T_{itr} is the iteration period in the absolute time scale and $f_{controller}$ is the frequency of the controller that generates the control signals (the *start* and *stop* signals in the figure). If the frequency is high, the program size will proportionately increase. Note that the *start* signal generated by the controller is before the start of the actual buffer activity.

Figure 3.2 illustrates the buffer activity timing and its corresponding program structure where buffer activities overlap across iterations. In this case, the program is generated by selecting the interval where the buffer activities are non-overlapping (this period is indicated with a shaded box).

As discussed previously, if the controller clock frequency is high and the iteration period is long, the control program size can be significantly large. In order to reduce this program size (and our design does aim for minimizing it), we propose to select the controller frequency $f_{controller}$, which satisfies the following conditions for all *start_read* and *start_write* times with respect to the beginning of the iteration period, as

$$start_read_i - T_{ri} < \frac{k_{ri}}{f_{controller}} < start_read_i \tag{3.2}$$

where T_{ri} is the time required for the buffer read operation, and k_{ri} is some integer constant. Similarly, for the *start_write* times, we have

$$start_write_i - T_{wi} < \frac{k_{wi}}{f_{controller}} < start_write_i \tag{3.2}$$

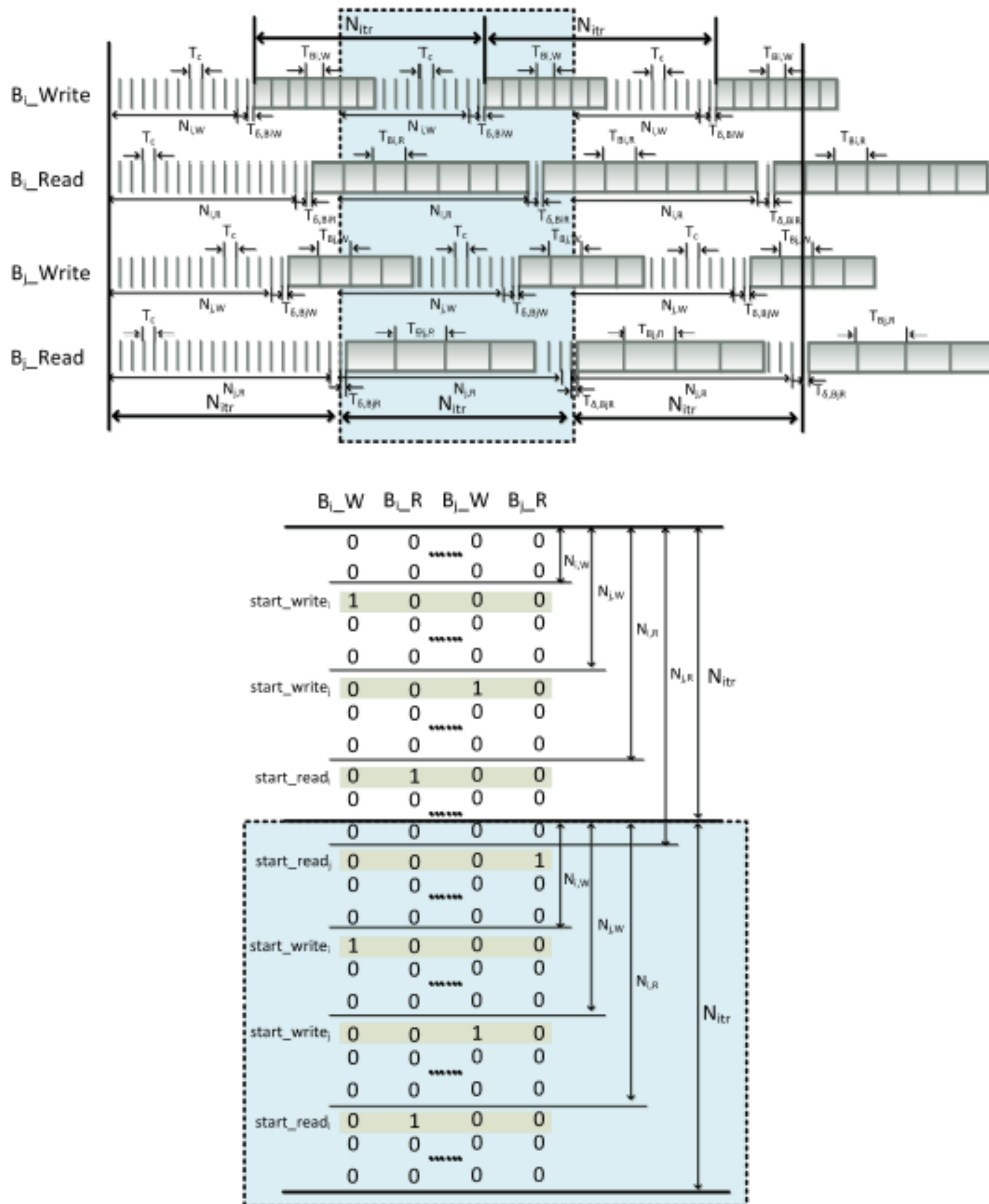


Figure 3.3: Timing diagram and corresponding program structure for iteration period with overlapping buffer execution.

where T_{wi} is the time required for the buffer write operation, and k_{wi} is again an integer constant. Extending these conditions to all the buffer controllers present in the design, we can see that for *N* buffer controllers, we will have *2N* such conditions that must be satisfied.

3.2.2 For Variable Execution Times

In all real designs mapped to any processor platform, variable execution times are bound to arise due to platform constraints and architecture, or due to some other limitations. Regardless of the reason, dealing with this latency is crucial for correct operation of the design. This, however, is heavily dependent on the iteration period requirements of the design. This is illustrated in Figure 3.4, which shows two scenarios with slow and fast iteration periods respectively. When a fixed latency Δ is introduced in any design, the buffer activity starts later, but a large iteration period (a slow design) is easily able to compensate for this as shown in Figure 3.4(a). But if the iteration period is smaller (a fast design), then the buffer activities will start at the wrong times, and this results in incorrect operation, as shown by Figure 3.4(b).

One way to meet the timing requirements and avoid incorrect operation due to the delay is to increase the speed, i.e., the frequency of the processing elements. But as frequency and power share a direct relationship, increasing the frequency of the processing elements (or more appropriately, the processors onto which they are mapped) results in a proportionate increase in the power consumed. Thus, if the delay present in the design is quite large (and we consider a fixed delay in the design), our methodology tries to adapt to this delay by proportionately increasing the speed of the processors in the platform. The power consumed by the platform therefore grows as this delay increases.

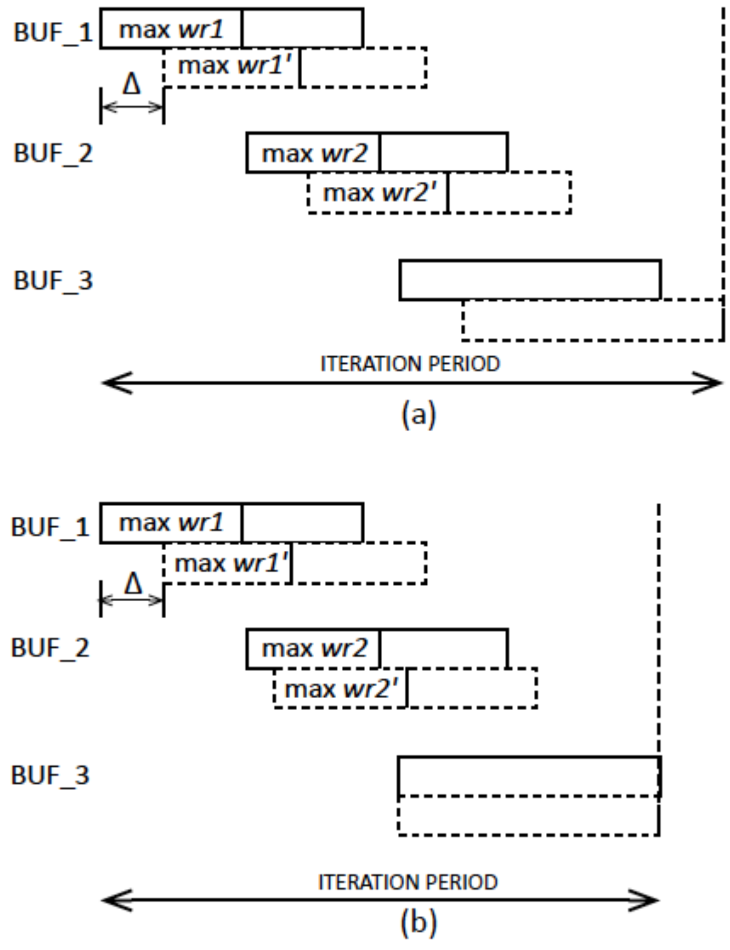


Figure 3.4: The effects of a delay. (a) A delay introduced in the design causes no issues as long as the iteration period is sufficiently large. (b) A faster design results in the buffers operating prematurely and producing incorrect results.

Since it is difficult to estimate the execution time of processors precisely, the buffer controller is changed to adapt to the multi-processor mapping as illustrated in Figure 3.5. The processor passes a ready signal to the buffer, indicating the actual start of the write time without reading the earlier data. Figure 3.5(b) shows the timing when the ready signal comes later than the start write time calculated from parameters D_i and nr_i. The write operation will wait until the

arrival of the ready signal. In the case that the processor finishes executing the previous dataset before the calculated *start_write* time, the buffer sends the data immediately after the calculated *start_write* time. So the ready signal from the processor acts as a handshake signal to eliminate the difficulty of estimating the execution time of the processor. For the read process of the buffer, the buffer counts $N + nr_i$ cycles and then waits for the ready signal from the following processor.

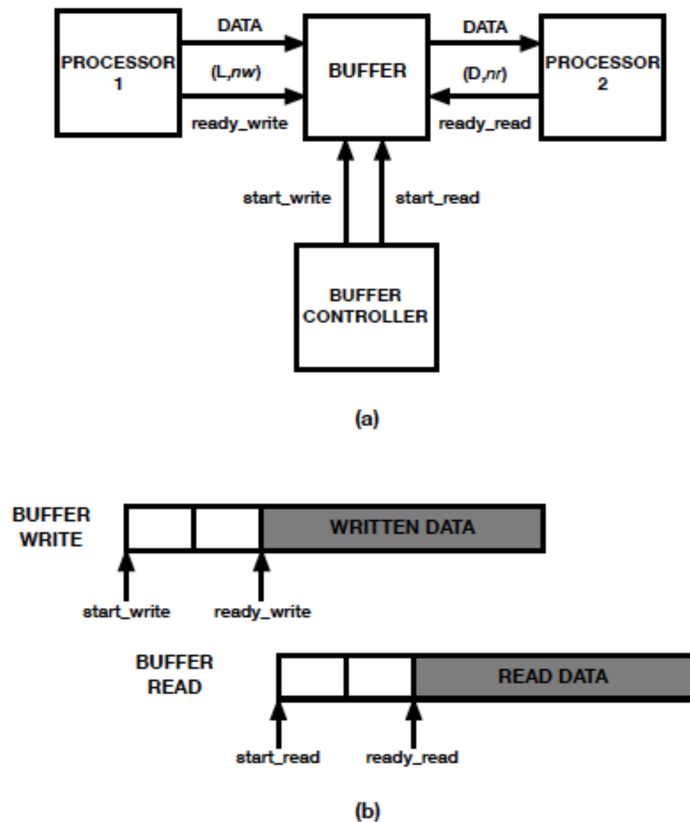


Figure 3.5: Buffer controller adapted for variable execution times, and its signal timings for correct read/write operations.

For the actual connections on the platform, the processor is connected to its *N* buffers through a bus. The bus is *M*-bits wide, where *M-1* bits are for the data and the other 1 bit is used for the control signal between the processors and the buffer in both directions. The connections

between the processors and the buffers are shown in Figure 3.6. The data bus carries the data transmitted between the processors and the buffers. The control bus sends the ready signals to the processor and the buffer to determine the actual start read and write signals of the buffer.

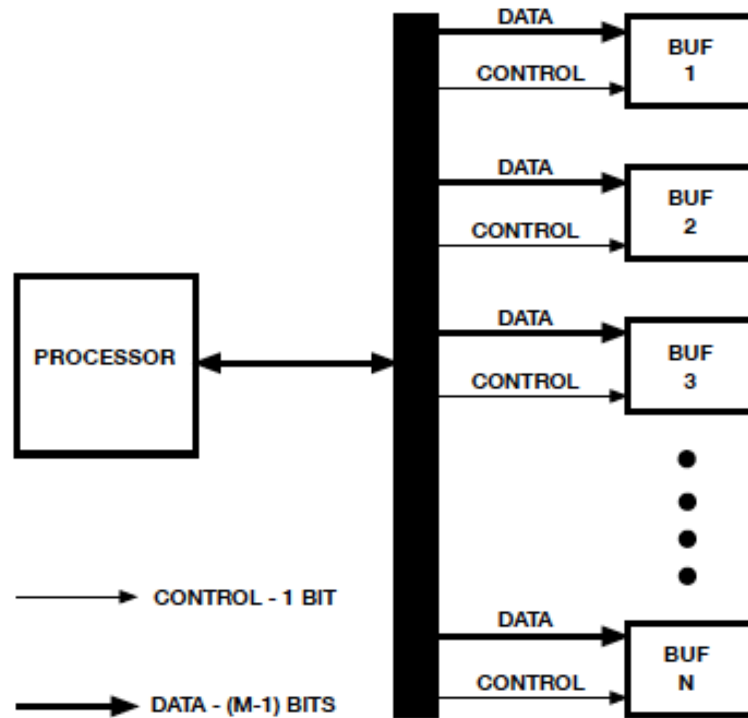


Figure 3.6: Connection of the processor and the buffers using a bus.

3.3 Structural Aspects

The structural controller, shown in use in Figure 3.7, manages dynamic reconfiguration. The purpose of this controller is to configure the data flow by managing interconnections as well as the buffer controllers. All registers in the buffer controllers and interconnection multiplexers are mapped into memory, and therefore reconfiguring the data flow is simply a matter of writing the new configuration content to the registers. The size of an address depends on the number of buffer controllers and the interconnection complexity. The program memory of the structural

controller stores multiple configurations, which allow for different data flows to be constructed by writing the corresponding control program content to the registers.

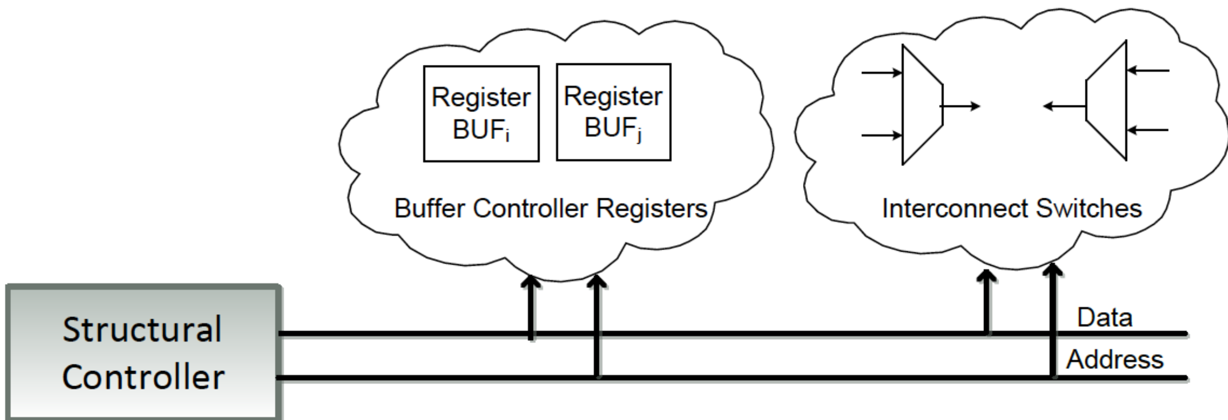


Figure 3.7: Structural controller and its connection to the buffer controllers and interconnect switches. The registers within the buffer controllers and the switches are address mapped.

The structural controller in our design employs a number of techniques to make sure that only the minimum number of buffers is being used during program execution. One such concept is buffer sharing. If read/write activities amongst some buffers do not overlap with each other, these buffers can be replaced by one buffer in certain special cases. There are a number of scenarios where this concept can be used to reduce the total number of buffers. Consider the case of Figure 3.8(a), where processor 1 is connected to processors 2 and 3 through buffers 1 and 2. In this case, processor 1 decides the path of the dataflow. The data will be written to buffer 1 or 2 depending on the computational decision made by processor 1. In any case, both the buffers can never be active at the same time. Once the processor has decided its dataflow path, the buffers lying on the other path are effectively never used in the same cycle.

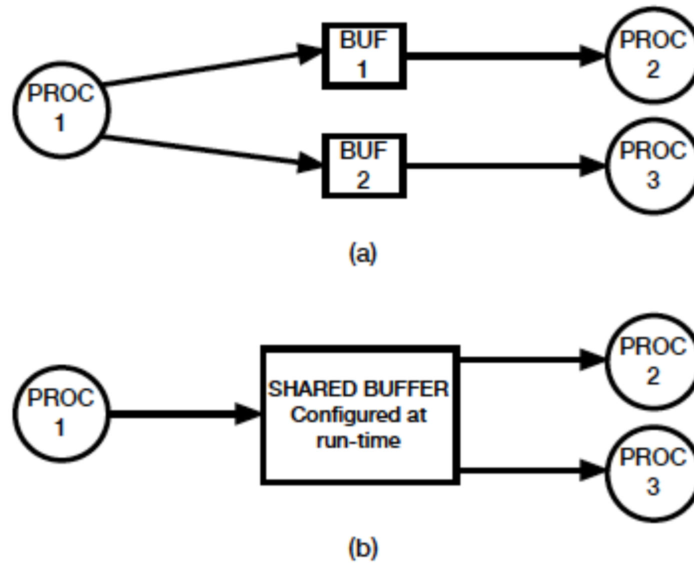


Figure 3.8: The concept of buffer sharing. (a) Buffers 1 and 2 in different paths, and cannot be active at the same time. (b) They are replaced by a single buffer.

Taking advantage of this mutual exclusiveness, we replace buffers 1 and 2 with a single buffer labeled buffer 1, as shown in Figure 3.8(b). As soon as the dataflow path is decided upon, the processor will send this information to the global controller, which will then configure this shared buffer using the technique of dynamic path selection and handle the structural reconfiguration. Figure 3.9 shows the timing before and after buffer 1 and buffer 2 are shared by using a single buffer labeled buffer 1. Buffer sharing can also be implemented for select periods of time.

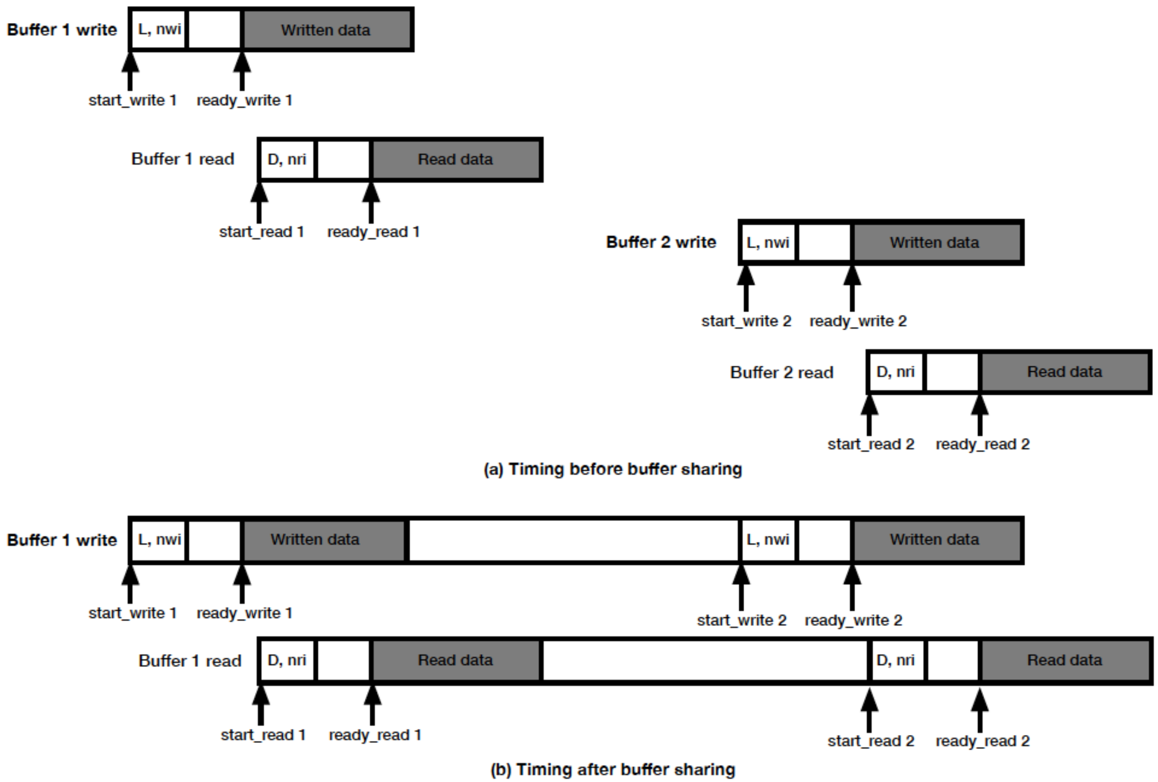


Figure 3.9: Timing of buffer activities before and after buffer sharing.

The *start_read* and *start_write* signals from the execution controller to buffer 2 are sent to buffer 1 instead. Similarly, the processors connected with buffer 2 are connected to buffer 1 instead during the buffer sharing period. These connections and control signals have to be handled by the structural controller. Thus, in the buffer sharing scheme, a single one that can then be connected to any of the data paths replaces some buffers lying on different data paths. The other case is that the processor itself decides its data path. When the buffer sharing scheme is used, the host processor calculates which buffers are to be replaced and sends this information to the controllers on the execution platform. If the path is to be decided by the processors, each processor only decides which buffer should be connected with it. In this scheme, the path is formed by the collective decisions of all the processors regarding their own path.

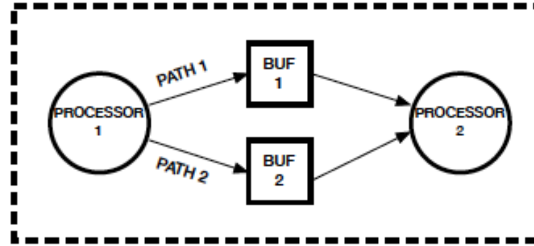


Figure 3.10: Multiple paths between two processors.

Consider the buffer-sharing scheme, where the processor decides to change the buffer connected to it. As shown in Figure 3.10, at time *t*, processor 1 is connected to processor 2 through buffer 1. But at a later instance, processor 1 is connected to processor 2 through buffer 2. The buffer sharing command is sent from the host processor. The interface controller accepts the command from the host processor and sends the buffer information from the memory to the global controller. Within the global controller, the structural controller is in charge of the modifications of the interconnection information of the processor and the buffers. These two paths, consisting of different buffers, are not formed at the same time. They are configured separately and a selector decides which path to choose as shown in Figure 3.11. This selector resides in the execution controller.

When the buffer sharing scheme is being used, the structural controller in the global controller configures all the possible paths (path 1 and path 2 in this example). When the platform is executing, the selector in the execution controller will decide which path should be active at different times. Once a path is selected, the structural controller sends the configuration information to the appropriate buffers in the partition.

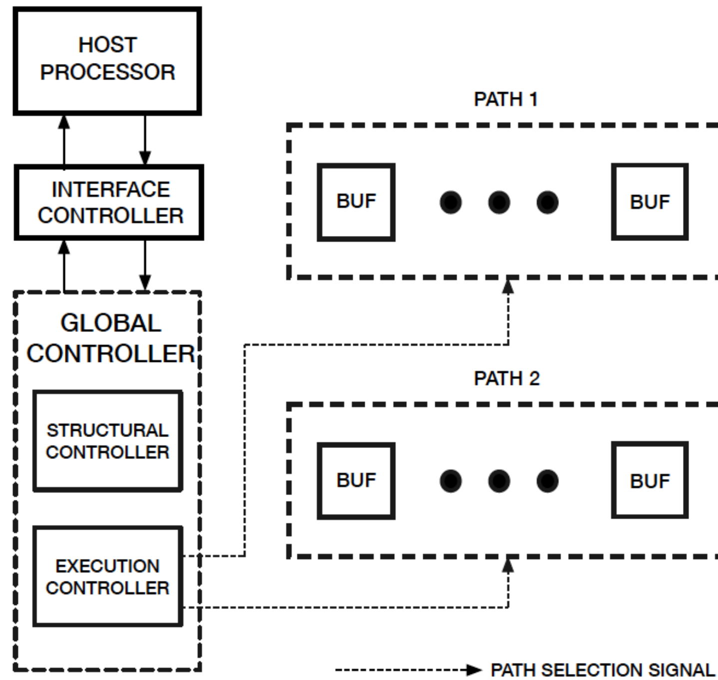


Figure 3.11: Configuration of the paths, and their selection using the global controller.

Now we formally define the 'partition' to be the set of the appropriate structural and execution controllers for a set of buffers. Figure 3.12 shows this configuration. The execution controller in the partition generates the *start*_read and *start_write* signals for the buffers. The execution monitor serves as the coordinator between the structural controller and the execution controller and is responsible for sending signals to both the controllers. Once the execution monitor detects that the read period of buffer 2 is over, it triggers a signal to the structural controller in order to configure the connection of buffer 1 in place of buffer 2 and update the necessary parameters. The execution monitor then asks the execution controller to activate buffer 1 instead of buffer 2. Finally, the *start_read* and *start_write* signals are sent to buffer 1 by the execution controller.

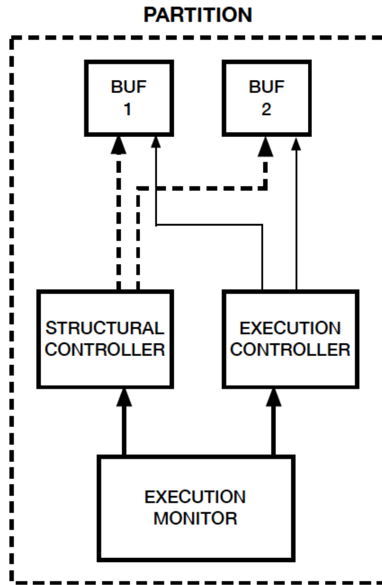


Figure 3.12: Partition structure and signals

In the case that a processor decides its data path, the processor controls its following buffer by communicating with the global controllers. Figure 3.13 shows this communication. The processor issues a request to use path 1 and sends information about the path to the global controller, which includes the buffer ID connected to the processor. Once the global controller accepts the request to reconfigure the path, it asks the structural controller to reconfigure the interconnections, and asks the execution controller to activate the buffer. Once both the controllers save the information of path 1 and correctly configure it, they relay this information to the global controller, which then gives the ready signal to the processor. The processor is then ready to use this path as well as to configure the next path.

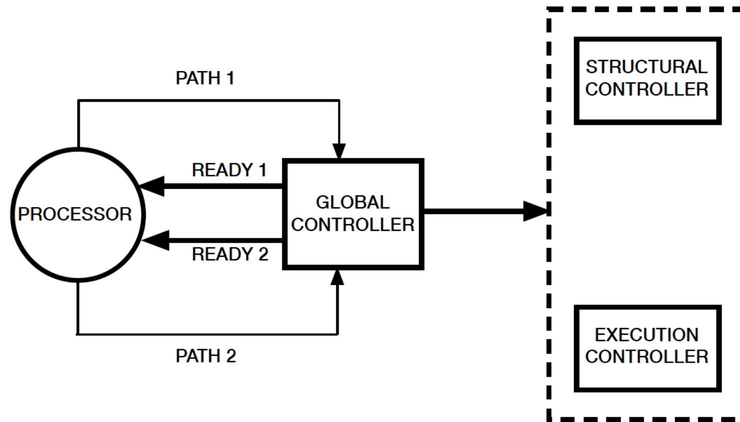


Figure 3.13: Processor deciding its own path through the global controller.

3.4 Overall Controller Design

The overall controller for the architecture is shown in Figure 3.14. The controller consists of two components as discussed. To make dynamic reconfiguration feasible, the execution controller and the structural controller coordinate with each other during execution of the program. The interaction between these two controllers is initiated under a branch condition. The branch condition can be set by the program memory or an external signal. When the branch condition is met, the finite state machine (FSM) in the execution controller accesses the address table (indicated by the three bits next to JB). These three bits are the table size of the FSM, and a total of eight (2^3) possible jumps or branches can be supported. For a jump, the FSM accesses the table using the three-bit address. The address in the execution controller program memory table is thus accessed; it is retrieved and the content of the FSM table is unavailable. However, for a branch, the address in the FSM table indicates the table address in the structural controller. The structural controller then accesses the program memory from the start address to the end address indicated in its own address table (since the size of the program depends on the number of buffer

controllers and interconnection complexity, the program memory for the reconfiguration can have a different data size, and therefore the address table contains two addresses - one for the beginning and one for the end). The target configuration is then loaded into the BBDF architecture, and the structural controller communicates with the execution controller to load the corresponding control program.

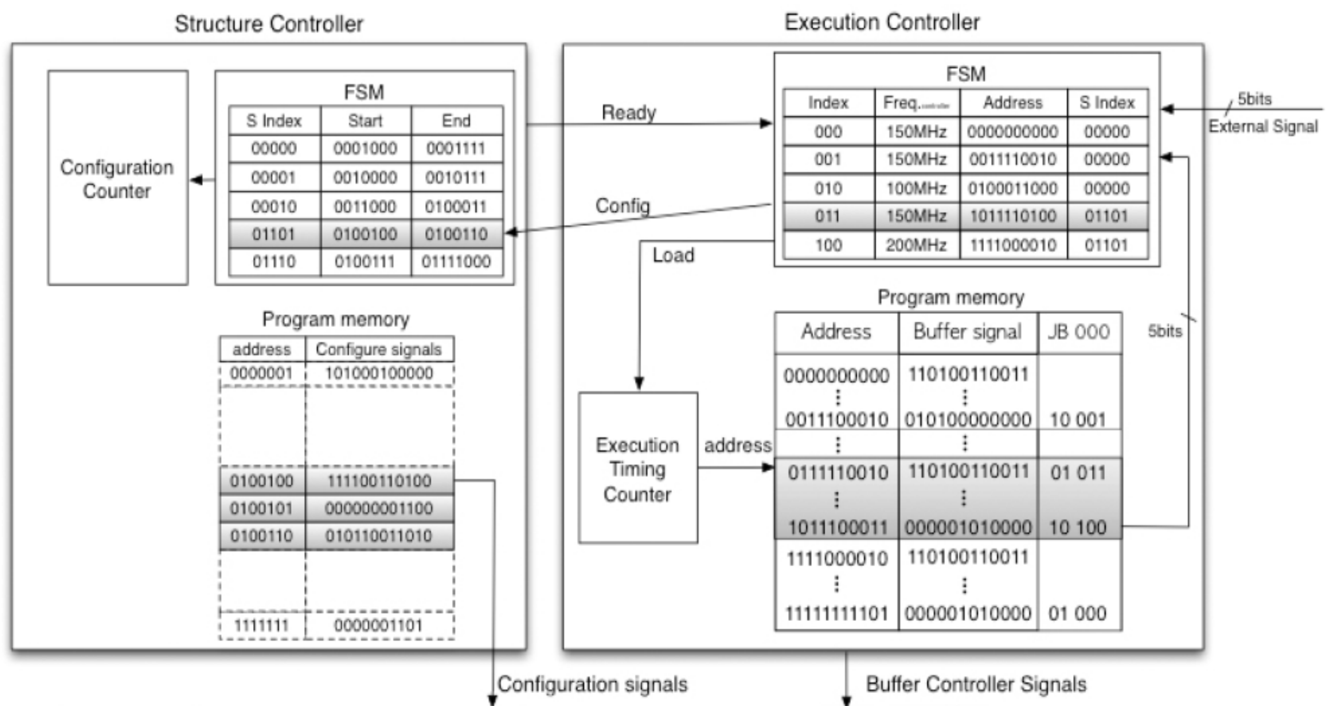


Figure 3.14: Overall controller structure (the ‘global’ controller)

Chapter 4

Resource Utilization

4.1 Processor Utilization of Mapping

One of the primary goals of our methodology from its start has been to make sure that multi-processor platforms (or equivalently, platforms with multi-core processors) are utilized efficiently. To achieve this, we tried to formulate methods that allow us to map processing elements to only the minimum number of processors.

In order to map processing elements to processors, the execution timing of such blocks are represented by primitive templates such as SND and RCV. Since buffer controllers are outside a processor, SND and RCV use the bus provided by a target processor to transfer data between the processing element mapped to a processor and a buffer controller. RCV reads data from a buffer controller, whereas SND writes data to it. Subscripts *i* and *j* represent the source and destination of data transfer respectively.

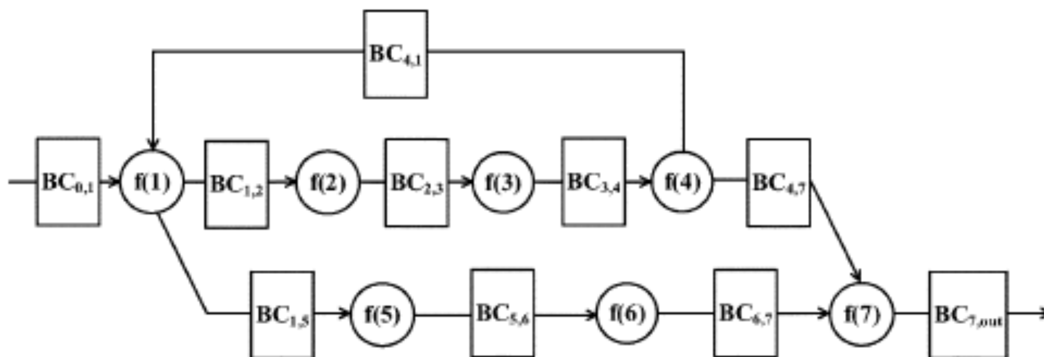


Figure 4.1: An example dataflow

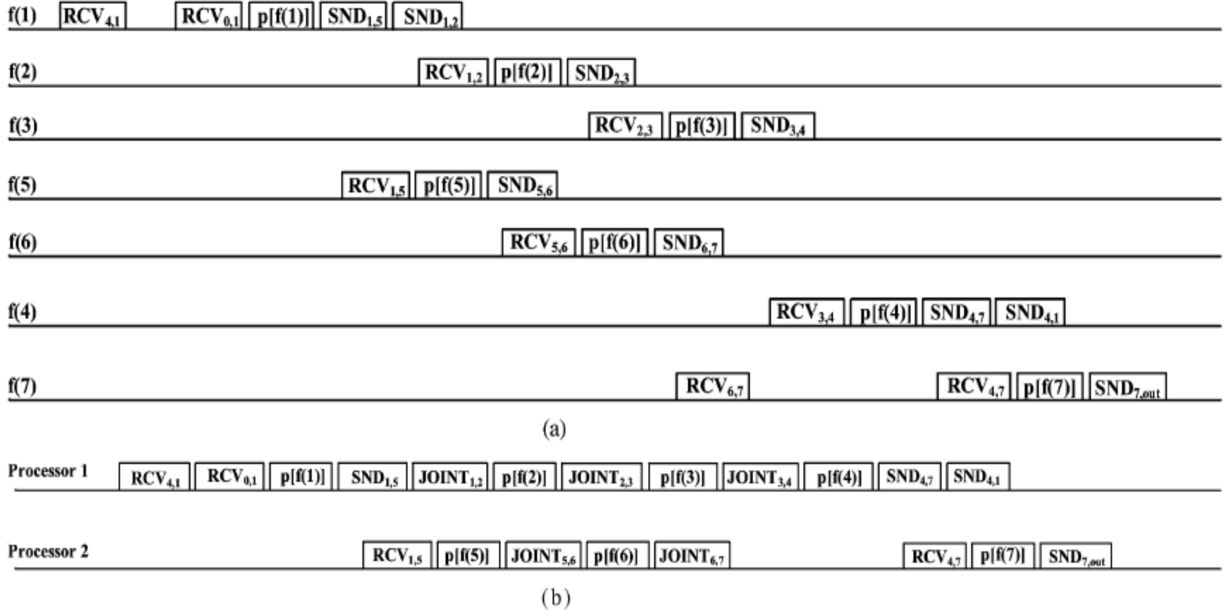


Figure 4.2: Execution timing representation to map PEs to processors. (a) Execution timing for mapping PEs as per Figure 4.1. (b) Execution timing when PEs of 4.2(a) are mapped to processors.

p[f(i)] represents the functional execution of the processing element realized as a program. Thus, given a dataflow as in Figure 4.1, Figure 4.2 represents the execution timing representation.

In Figure 4.2(a), a straightforward way of mapping would be to map each PE to a different processor (or core). In this case, the number of processors is the same as the number of PEs (7 for the example). However, if the execution times of the processing blocks are non-overlapping, they are mapped to the same processor in order to reduce the number of processors. When *f(i)* and *f(j)* are mapped to the same processor, their executions times satisfy the following non-overlapped condition [EXE_i is the execution time of *f(i)* and EXE_j is the execution time of *f(j)*]:

$$\max\left(end(EXE_i), end(EXE_j)\right) - \min\left(start(EXE_i), start(EXE_j)\right) \tag{4.1}$$
$$> \left(\mathrm{end}(\mathrm{EXE_i}) - \mathrm{start}(\mathrm{EXE_i})\right) + \left(\mathrm{end}(\mathrm{EXE_j}) - \mathrm{start}(\mathrm{EXE_j})\right)$$

where start(EXE) and end(EXE) correspond to the start and end times of SND, RCV and functional execution, respectively. Equation 4.1 represents that two execution times are non-overlapped if the difference between the maximum end(EXE) and minimum start(EXE) of two execution times is larger than the summation of two execution times. According to the execution type of PE *f(i)* (i.e., SND, RCV and *p[f(i)]*), start(EXE) and end(EXE) of equation 4.1 are translated to

$$\mathrm{end}(\mathrm{EXE_i}) = \begin{cases} \mathrm{stop_write}, & \text{if } \mathrm{EXE_i} = \mathrm{SND\ of\ f(i)} \\ \mathrm{stop_read}, & \text{if } \mathrm{EXE_i} = \mathrm{RCV\ of\ f(i)} \\ \mathrm{end\ of\ p[f(i)]}, & \text{if } \mathrm{EXE_i} = \mathrm{p[f(i)]} \end{cases}$$

$$\mathrm{start}(\mathrm{EXE_i}) = \begin{cases} \mathrm{start_write}, & \text{if } \mathrm{EXE_i} = \mathrm{SND\ of\ f(i)} \\ \mathrm{start_read}, & \text{if } \mathrm{EXE_i} = \mathrm{RCV\ of\ f(i)} \\ \mathrm{start\ of\ p[f(i)]}, & \text{if } \mathrm{EXE_i} = \mathrm{p[f(i)]} \end{cases}$$

The number of processors is further reduced by using JOINT. When two consecutive processing blocks are mapped to the same processor, it is unnecessary to transfer data through the buffer controller because their arguments can be internally bound. In this case, data transfer is realized as JOINT, which mergers RCV and SND. In Figure 4.2(a), the execution times of *f(1)* and *f(2)* are overlapping in only $\mathrm{SND}_{1,2}$ and $\mathrm{RCV}_{1,2}$. By replacing $\mathrm{SND}_{1,2}$ and $\mathrm{RCV}_{1,2}$ with $\mathrm{JOINT}_{1,2}$, both processing elements are mapped to the same processor. In the same way *f(1) – f(4)* are mapped to processor 2 as shown in Figure 4.2(b). If the number of processors in the target platform is 1, the execution timing of Figure 4.2(b) is not directly mapped to single processor architecture. In this

case, for mapping all processing blocks to a single processor, the execution times from *f(5)* to *f(7)* shift right to the end of the execution times from *f(1)* to *f(4)*.

4.2 Power Consumption and Bus Utilization of Mapping

Figure 4.3 illustrates the interconnection between processors when the processing elements of Figure 4.2(a) are mapped to two processors. C_{load} represents the port loading capacitance of a bus. In Figure 4.3(a), buffer controllers are not used for mapping. In Figure 4.3(b), two buffer controllers $\text{BC}_{1,5}$ and $\text{BC}_{4,7}$ are attached to the buses for synchronizing data transfers between *f(1)* and *f(5)* and between *f(4)* and *f(7)*, respectively. However, these buffer controllers increase the port loading capacitance of the buses and can lead to additional dynamic power dissipation.

For example, when *f(1)* (mapped to processor 1) sends data to *f(5)* (mapped to processor 2), the dynamic energy consumptions of the mapping shown in Figure 4.3(a) and (b) are $2C_{\text{load}}V^2fM_{1,5}$ and $6C_{\text{load}}V^2fM_{1,5}$, respectively, where C_{load} is the load capacitance of the buses, f is the operating frequency of the buses and V is the supply voltage. In Figure 4.3(b), since the data transfer from *f(1)* and *f(5)* is realized as both $\text{SND}_{1,5}$ and $\text{RCV}_{1,5}$, the port loading capacitance, $3C_{\text{load}}$, doubly contributes to the energy consumption.

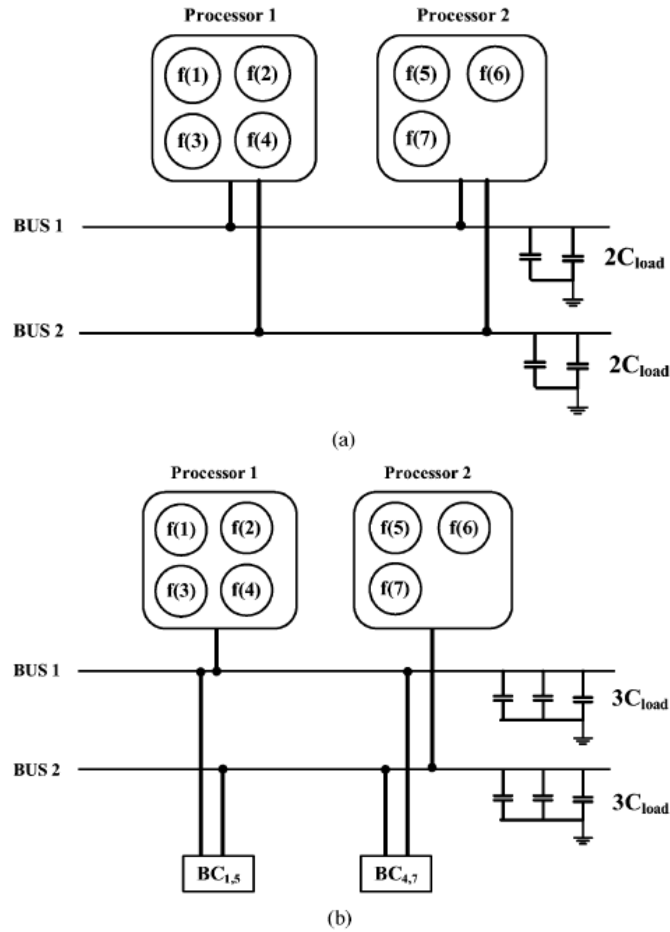


Figure 4.3: Considering power consumption for interconnected processors when elements of Figure 4.2(a) are mapped to them. (a) Mapping processing elements without buffer controllers. (b) Mapping processing elements with buffer controllers.

Our basic consumption is that a single processor has its own bus. However, if the execution times of bus operations (i.e., SND and RCV) are not overlapped among processors, the processors share the same bus in order to reduce the number of buses used. This directly affects the power consumption of the platform as we have seen above. Figure 4.4 shows the execution timing when processing elements of Figure 4.2(a) are mapped to four processors instead of two.

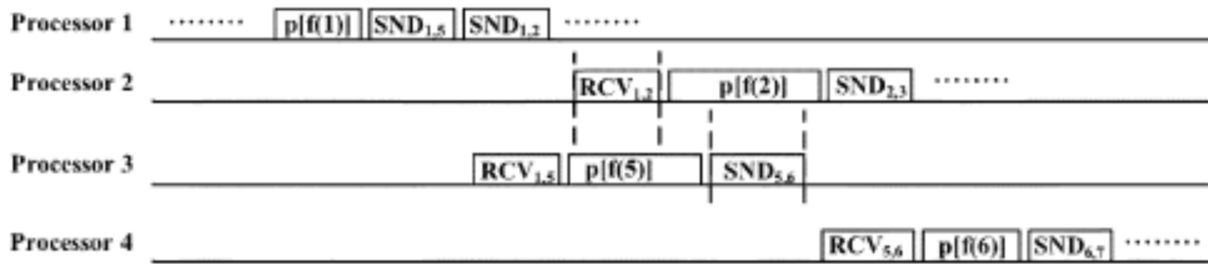


Figure 4.4: Estimated execution times when processing blocks of Figure 4.2(a) are mapped to four processors.

In Figure 4.4, the execution time of $RCV_{1,2}$ is overlapped with the execution time of *p[f(5)]*. Thus, *f(2)* and *f(5)* are not mapped to the same processor. However, *p[f(5)]* does not access the bus which $RCV_{1,2}$ uses for the data transfer with $BC_{1,2}$. In addition, *p[f(2)]* of processor 2 does not use the bus for $SND_{5,6}$ of processor 3. Therefore, processors 2 and 3 share the same bus because the execution times of SND and RCV between them are non-overlapped.

However, the execution times of Figure 4.4 are estimated values for mapping only. If actual execution times are not within the range of estimated values, the bus sharing among processors leads to wrong results.

Chapter 5

Evaluation

In this section we evaluate our design methodology and verify its operation for two buffer-based dataflows. We use SystemC to model a target platform having multi-core processors and reconfigurable logics. Our testing methodology consists of mapping our test dataflows to the processors on the target platform as well as implementing them as hardware logics.

5.1 Experimental Setup

Our evaluation uses the buffer-based dataflows of Figures 5.1 and 5.2, and targets a platform consisting of multiple processors at 400MHz and buses at 100MHz. We chose the aforementioned examples for evaluation of our methodology since they have adequate complexity (that is usually found in real-world applications) to test all of our proposed solutions – they comprise of feedback and feed-forward loops that can have an impact on the final program generated by our design. Tables 5.1 and 5.2 list the buffer controller parameters for the dataflows in Figures 5.1 and 5.2 respectively. The clock frequency for each element in each dataflow has been fixed, as is the block size M and the iteration period (at the start of the execution). It is possible to increase or decrease the iteration period as long as all the constraints are met.

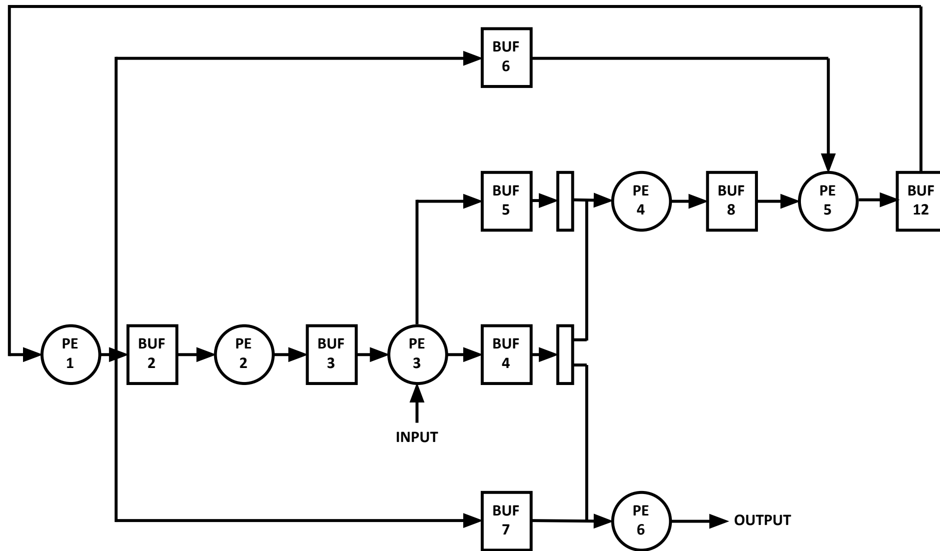


Figure 5.1: The first buffer-based dataflow used for evaluation.

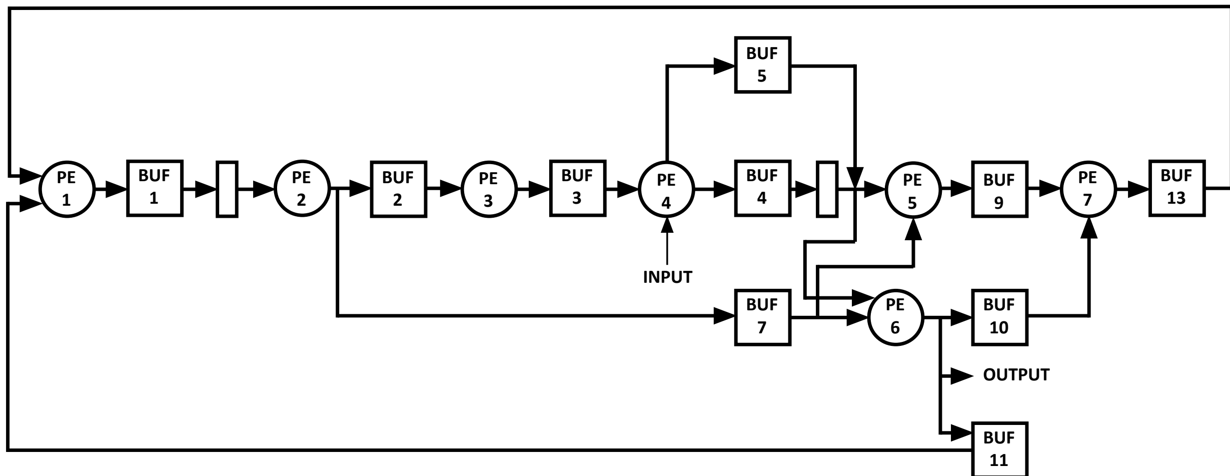


Figure 5.2: The second buffer-based dataflow used for evaluation.

BUF	M	Write		Read		Size
		Time	Frequency	Time	Frequency	
2	64	13	100MHz	77	300MHz	43
3	64	89	300MHz	90	300MHz	2
4	64	101	300MHz	102	100MHz	64
5	64	132	300MHz	133	200MHz	32
6	64	13	100MHz	157	300MHz	64
7	64	13	100MHz	102	100MHz	64
8	64	141	200MHz	157	300MHz	32
12	64	183	300MHz	184	100MHz	64

Table 5.1: Buffer controller configuration for dataflow in Figure 5.1

BUF	M	Write		Read		Size
		Time	Frequency	Time	Frequency	
1	64	4	400MHz	5	100MHz	64
2	64	18	100MHz	82	300MHz	43
3	64	94	300MHz	95	300MHz	2
4	64	106	300MHz	107	100MHz	64
5	64	137	300MHz	138	100MHz	64
7	64	18	100MHz	107	100MHz	64
9	10	246	100MHz	256	400MHz	10
10	4	246	100MHz	250	400MHz	4
11	4	246	100MHz	279	400MHz	4
13	10	284	400MHz	285	400MHz	2

Table 5.2: Buffer controller configuration for dataflow in Figure 5.2

5.2 Processing Elements Mapped as Hardware Logic

The mapping of individual processing elements as hardware logics demonstrates our design's capability for structural reconfiguration. Since all the processing elements have fixed execution times, there are no unaccounted variations in the execution time of individual elements in the program. Figure 5.3 illustrates the timing flow for the dataflow in Figure 5.1. The gray bars represent buffer activity, as per the characteristics described in Table 5.1. The different reconfigurations represent different iteration periods for the dataflow. During the first run of the design, the execution flow was interrupted at the end with an external signal that changed the iteration period and began the second run. This iteration is allowed to complete, and an internal signal changes the iteration period for yet another time, thus beginning the third run.

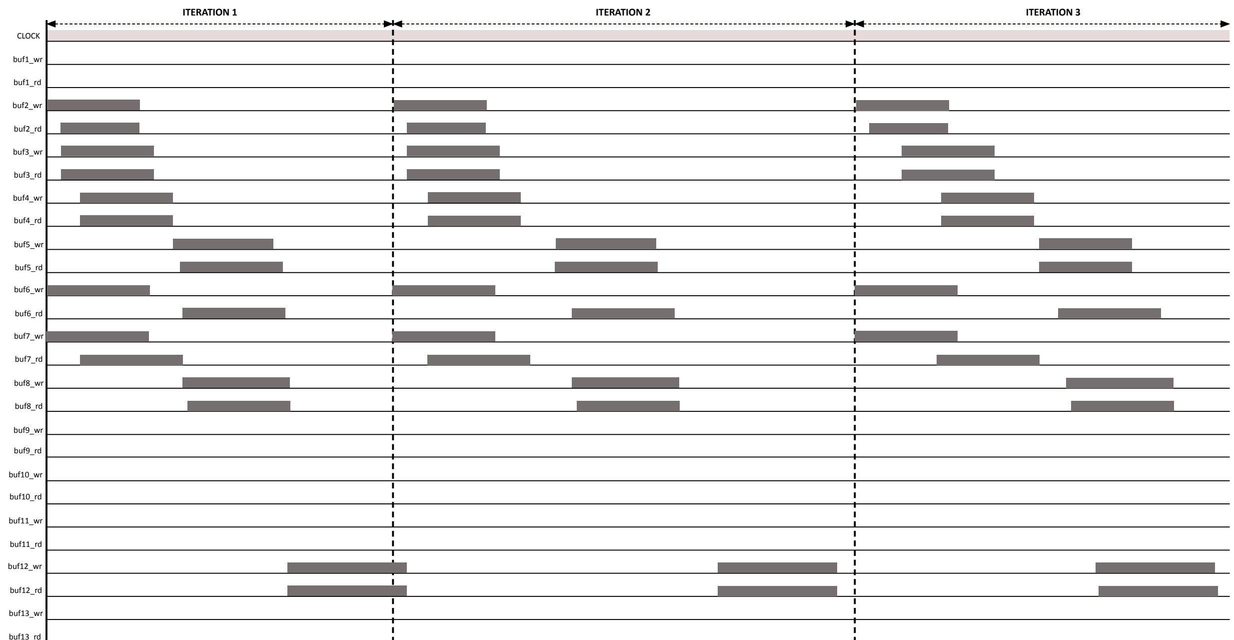


Figure 5.3: Buffer timing flow for the dataflow in Figure 5.1.

With the same test methodology, similar results are obtained for the dataflow in Figure 5.2. The results show that as long as the iteration period (and other) constraints are met, the overall execution flow is correctly maintained even if the iteration periods are changed while execution.

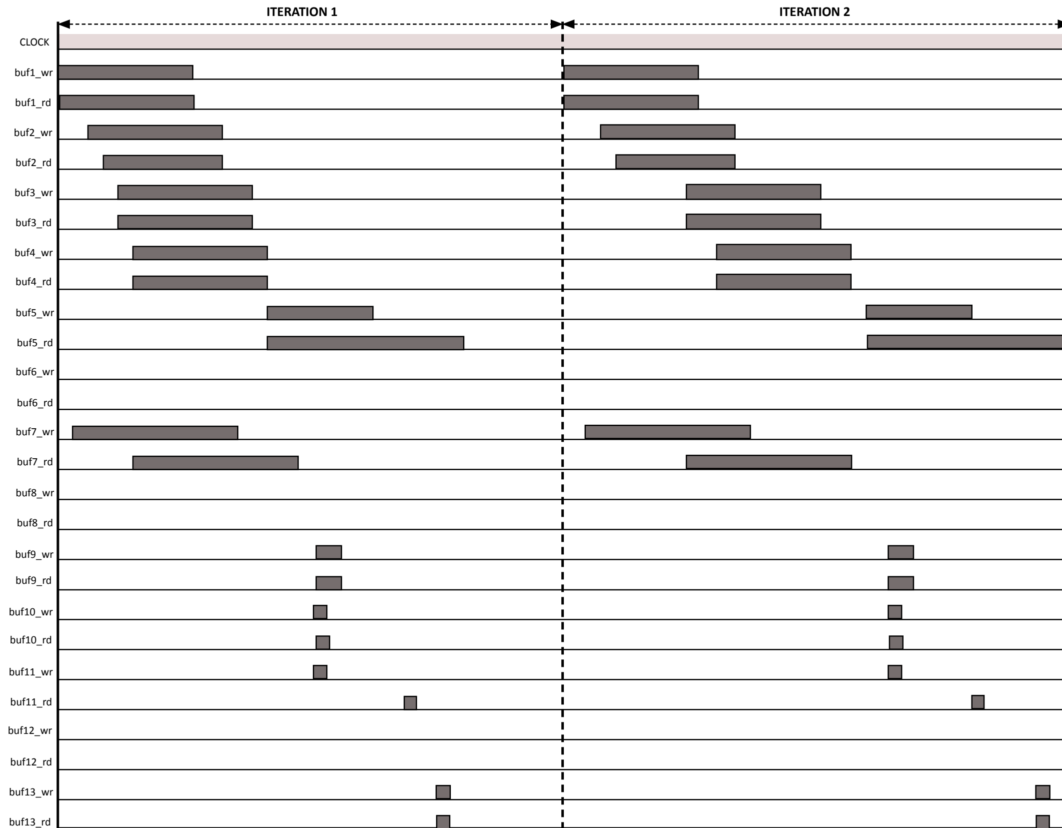
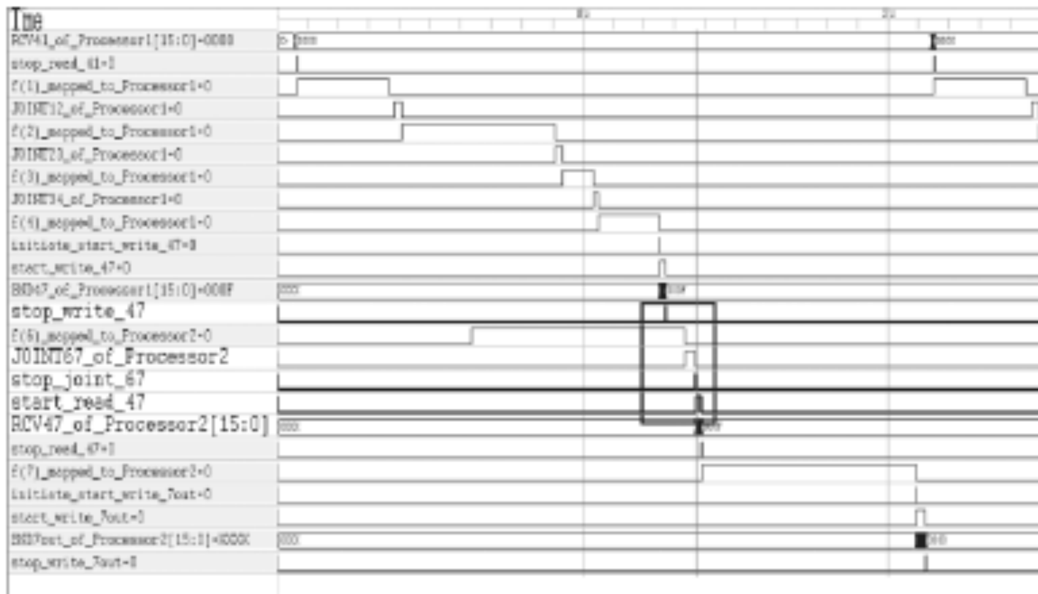
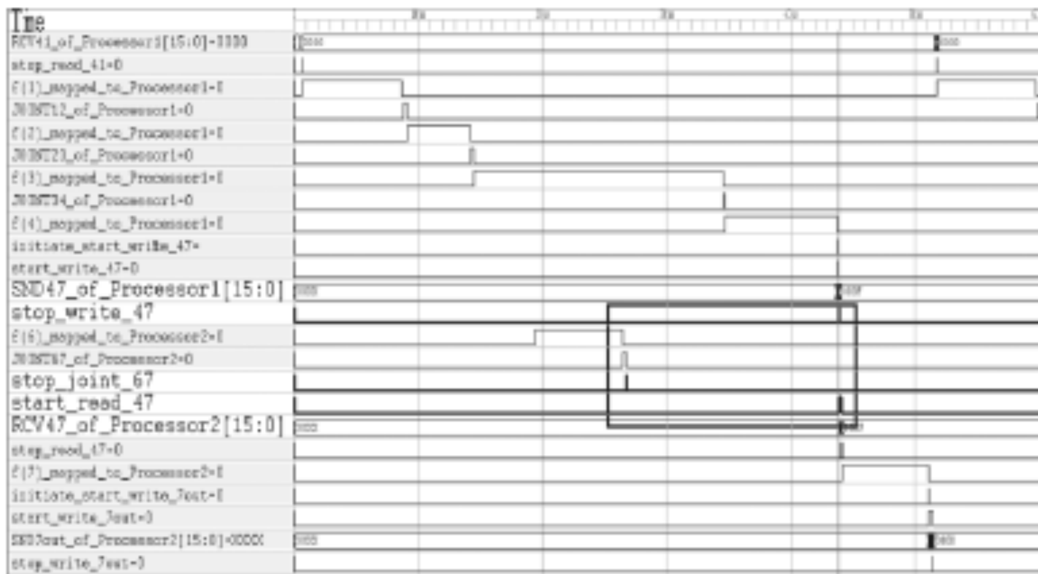


Figure 5.4: Buffer timing flow for the dataflow in Figure 5.2.

5.3 Processing Elements Mapped to Processors



(a)



(b)

Figure 5.5: Simulation results when processing elements are mapped to processors. (a) $stop_joint_{6,7}$ determines $start_read_{4,7}$. (b) $stop_write_{4,7}$ determines $start_read_{4,7}$.

When we map multiple processing elements on a single processor (or a processor core), the temporal aspects of the design change significantly. Now, in addition to dynamic reconfiguration, the design must handle the introduction of variable execution times as discussed in previous chapters. We map the test dataflows to our platform with 2 processors, and the results obtained are shown in Figure 5.5. We can see that the global controller sends a *start* signals, and processors send *stop* signals to the global controller. Due to the variation in execution times, $\text{start_read}_{4,7}$ is enabled when either $\text{stop_joint}_{6,7}$ in Figure 5.5(a) or $\text{stop_write}_{4,7}$ in Figure 5.5(b) are made high by the processors. For the correct operation in both cases, the global controller generates $\text{start_read}_{4,7}$ when it receives $\text{stop_write}_{4,7}$ and $\text{stop_joint}_{6,7}$ from processor 1 and processor 2. As a result, Figure 5.5(b) shows that processor 2 starts running $\text{RCV}_{4,7}$ when it finishes $\text{JOINT}_{6,7}$. Similarly, processor 2 begins running $\text{RCV}_{4,7}$ when processor 1 ends $\text{SND}_{4,7}$.

Chapter 6

Summary

6.1 Future Scope

There has been extensive research in designing mapping algorithms that can make efficient use of available resources. From the past few years, substantial efforts have been put into trying to reduce the complexity of controllers that enable operation in such designs, which is the direct result of trying to leverage the improvements in multi-processor platforms. Our work has tried to present a simple (yet full-fledged) methodology that can map complex real-world dataflows to multi-processor platforms. But as is the case with technology, there is further room for improvement in our proposed design.

- Our design relies on accurate timing information for all processing blocks in the dataflow. The accuracy of this information can have significant effects on the results obtained from the dataflow. However, such information is not always reliable. We aspire to incorporate some functionality in our design that can handle variations in such timing information.
- One of the latest trends in the field of reconfigurable architecture is having processors co-exist with hardware logic. This presents additional problems of synchronization in the design – we must now deal with fixed execution times on one side and variable ones on the other. We aim to extend this functionality into our design.

6.2 Conclusions

In this thesis, we have proposed a mapping methodology to synthesize processing elements of a dataflow in a target platform having multiple processors and programmable logics, and enable on-the-fly selection between dataflows. In order to achieve synchronized data transfers between processors (or a processor and hardware), we used the buffer-based dataflow representation. From the buffer-based dataflow and estimated execution times of the processing elements and data transfers, our proposed methodology creates a mapped partition that satisfies a set of given resource constraints. After the mapped partition is created, the code for mapping processing elements to processors is generated by the design. Our methodology also includes the design of a 'global' controller that provides support for dynamic reconfiguration. The global controller also handles variations in execution times that are introduced in the design due to the mapping of elements on processors. The proposed methodology was evaluated with SystemC modeling. Through the evaluation, we demonstrated that the mapping by our proposed methodology is successfully working, has the minimum possible program size, and has support for multiple iteration period requirements.

Bibliography

[1] Xilinx Inc., San Jose, CA, "Virtex-5 and Virtex-6 field programmable gate arrays," Dec. 20, 2009. [Online]. Available: http://www.xilinx.com

[2] J. Liu, M. Lajolo, and A. Sangiovanni-Vincentelli, "Software timing analysis using HW/SW cosimulation and instruction set simulator," in *Proc. 6th Int. Workshop Hardw./Softw. Codes., IEEE Comput. Soc.*, Mar. 1998, pp. 65–69.

[3] A. Bouchhima, S. Yoo, and A. Jeraya, "Fast and accurate timed execution of high level embedded software using HW/SW interface simulation model," in *Proc. Conf. Asia South Pacific Des. Autom.*, Jan. 2004, pp. 469–474.

[4] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y. Joo, "PeaCE: A hardware- software codesign environment for multimedia embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 3, pp. 1–25, Aug. 2007.

[5] F. Fummi, M. Loghi, M. Poncino, and G. Pravadelli, "A cosimulation methodology for HW/SW validation and performance estimation," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 2, pp. 1–32, Mar. 2009.

[6] Y. Yi, W. Han, X. Zhao, A. T. Erdogan, and T. Arslan, "An ILP formulation for task mapping and scheduling on multi-core architecture," *Proc. DATE*, pp. 33–38, Apr. 2009.

[7] D. Lee, S. S. Bhattacharyya, and W. Wolf, "High-performance buffer mapping to exploit DRAM concurrency in multiprocessor DSP systems," in *Proc. IEEE/IFIP Int. Symp. Rapid Syst. Prototyping*, Jun. 2009, pp. 137–144.

[8] C. Lee, S. Kim, and S. Ha, "A systematic design space exploration of MPSoC based on synchronous data flow specification," *J. Signal Process. Syst.*, vol. 58, no. 2, pp. 193–213, Feb. 2010.

[9] H. Jung, H. Yang, and S. Ha, "Optimized RTL code generation from coarse-grain dataflow specification for fast HW/SW cosynthesis," *J. Signal Process. Syst.*, vol. 52, no. 1, pp. 13–34, Jul. 2008.

[10] T. Yen and W. Wolf, "Communication synthesis for distributed embedded systems," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Des., IEEE Computer Soc.*, Nov. 1995, pp. 288–294.

[11] J. Mun, S. H. Cho, and S. Hong, "Flexible controller design and its application for concurrent execution of buffer centric dataflows," *J. VLSI Signal Process. Syst.*, vol. 47, no. 3, pp. 233–257, Jun. 2007.

[12] S. Hong, J. Lee, A. Athalye, P. M. Djuric, and W. Cho, “Design methodology for domain specific parameterizable particle filter realizations,” *IEEE Trans. Circuits Syst. I, Reg. Papers,* vol. 54, no. 9, pp.1987–2000, Sep. 2007.

[13] A. Kalavade and E. A. Lee, “A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem,” in *Proc. 3rd Int. Workshop Hardw./Softw. Co-Des.*, 1994, pp. 42–48.

[14] W. Chun, S. Yoon, and S. Hong, “Buffer Controller-Based Multiple Processing Element Utilization for Dataflow Synthesis”, *IEEE Trans. VLSI Syst.*, pp. 124-137, April 2010.