# VLSI Design Methodology Based On A Parameterized Buffer Controller

A Dissertation Presented

by

Woohyung Chun

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Electrical Engineering

Stony Brook University

December 2009

**Stony Brook University**

The Graduate School

**Woohyung Chun**

We, the dissertation committee for the above candidate for the Doctor of Philosophy degree, hereby recommend acceptance of this dissertation.

Sangjin Hong, Dissertation Advisor
Professor, Department of Electrical & Computer Engineering

Ridha Kamoua, Chairperson of Defense
Professor, Department of Electrical & Computer Engineering

Alex Doboli,
Professor, Department of Electrical and Computer Engineering

Hongshik Ahn,
Professor, Department of Applied Mathematics and Statistics

This dissertation is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

## Abstract of the Dissertation

# VLSI Design Methodology Based On A Parameterized Buffer Controller

by

Woohyung Chun

Doctor of Philosophy

in

Electrical Engineering

Stony Brook University

2009

This thesis presents a VLSI design methodology utilizing a buffer-based data flow to reduce interconnect resources and to synchronize the data transfer between different processing elements (i.e. between processors / between a processor and a hardware logic). The buffer-based dataflow is a novel design representation suitable for implementing data-centric applications. Since the buffer-based dataflow isolates the functional execution and data transfer of each node by using parameterized buffer controllers, it is helpful for reducing overall design time and for increasing reconfigurability.

We first propose a sharing methodology which reduces the buffer memory and the number of buses used in the realization of a buffer-based dataflow. Buffer controllers in a buffer-based dataflow represent the interconnects for data transfers between nodes. In order to achieve interconnect resource reduction, we control data transfers by using buffer lifetimes and activity times parameterized in a buffer-based dataflow. In addition, the proposed methodology finds the sharing case that consumes the minimum energy within the search range determined by the costs of buffers and buses.

We also propose a mapping methodology for the case where nodes of a buffer-based dataflow are realized as programs running on processors. From the buffer-based dataflow and estimated execution times of functional blocks and data transfers, the proposed methodology creates a mapped partition and generates the template code which runs on the processors of a target platform. We also use a processor initiation scheme to prevent wrong operations from happening when actual execution takes longer than estimated.

Finally, we evaluate the proposed sharing methodology with dataflow graphs representing data-centric applications. Also, our proposed mapping methodology and the generated template code are evaluated with the SystemC model and Xilinx ISE. The proposed methodologies are applicable to the high-throughput implementation of VLSI systems, for which the simplification of control structure is critical, and to the design of reconfigurable system-on-chip (SoC).

**To Lyna and Jiseon**

# Contents

# List of Figures

# Chapter 1

# Introduction

For complex designs, the greatest impact on performance, costs and functionality can be made at the architectural level [1]. Top-down design methodologies proceeding from the architectural to lower levels, are thus becoming popular. This design trend relaxes redesign efforts at higher levels of system assembly [2], and most designers endeavor to simplify the complex system under design from an early stage.

One of the most popular simplifying methods is to represent a target application as a dataflow graph. Especially, modeling digital signal processing (DSP) applications through coarse-grained dataflow graphs is widespread and is adopted in many high-level design frameworks [3]. When a dataflow graph is synthesized in a fine-grained reconfigurable platform such as field-programmable gate arrays (FPGAs), the interconnects of the dataflow consume more resources than the functional units do [4]. In addition, the implementation of multiplexors for data routing is very expensive in typical FPGA design [5]. In [6], Cong *et al.* presented a method that can bind a dataflow graph to a parameterizable register-file microarchitecture in order to reduce the interconnect and multiplexor complexity. However, this approach is limited in terms of reducing the number of multiplexors because the microarchitecture still uses multiplexors for data routing.

In this dissertation, we use the techniques developed for buffer-based dataflow representations [7], in which buffers are inserted between the nodes of a dataflow graph. Due to these buffers, data transfers between nodes can be separated from the functional executions of nodes [8]. Thus, a buffer-based dataflow can increase the reusability and reconfigurability of the interconnects between nodes. Moreover, the data transfers between nodes can be *timely multiplexed with the buffer controller parameters* at the level of a dataflow. Therefore, there is no need for additional multiplexors to route data transfers. In a data-centric application, interconnects account for a dominant source of energy consumption because data transfers through the interconnects frequently occur. We also discuss the relation between the interconnect resource and energy consumption in a buffer-based dataflow.

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, recent reconfigurable platforms such as Xilinx FPGAs include processors [15]. 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 a 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 (or, 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 [11, 16, 17, 19]. However, in the case where actual execution times is greater than the estimated times, the mapping based on the estimated times may produce wrong results. To prevent this problem, Jung et al. [20] 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 [21].

In a buffer-based dataflow, the timing mismatch of data transfers 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 hardwares). The proposed methodology translates the data transfer activities of processing blocks into the primitive templates running on processors.

The remainder of this dissertation is organized as follows: Chapter 2 reviews the method to construct a buffer-based dataflow with buffer controller parameters. Chapter 3 describes interconnect resource reduction through parameterized buffer controllers. It provides a controlling method of data transfer, and an energy consumption model for a buffer-based dataflow. In Chapter 4, we propose a mapping methodology which maps processing blocks to processor(s). The proposed methodology achieves the synchronized data transfer between processors (or, a processor and a hardware) by utilizing a buffer-based dataflow. It also provides a processor initiation scheme to prevent wrong operation when actual execution times take longer than estimated execution times. Chapter 5 finally concludes the research works in this dissertation, and mention the future works.

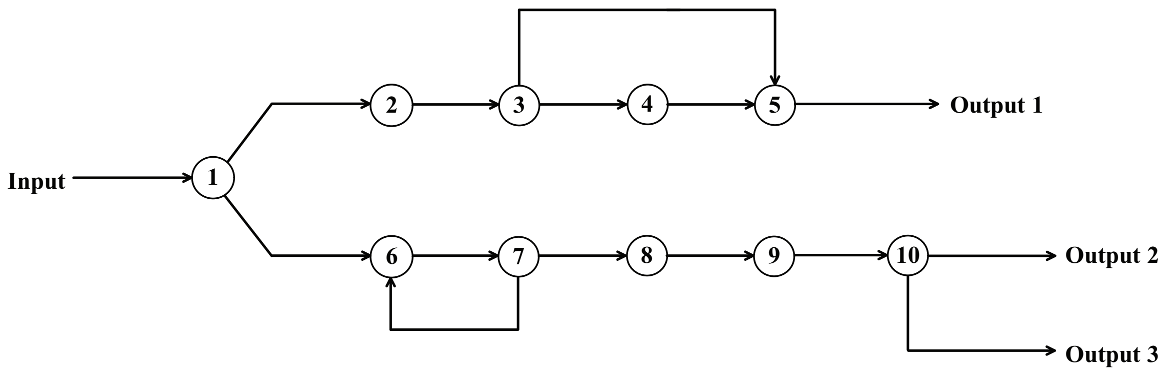
# Chapter 2

# Background

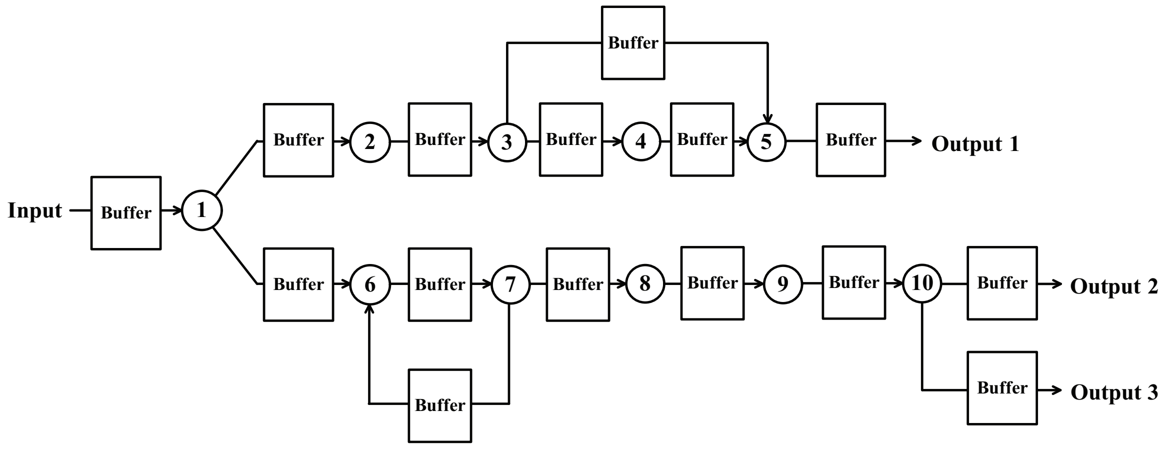
## 2.1 Buffer-Based Dataflow Representation

Most real-time signal processing algorithms consider sets of data as frames. Such systems possess two unique characteristics of execution. First, they can be represented as dataflow graphs which represent data dependency between processing blocks (nodes) [13]. Second, each node in the dataflow executes a set of data elements per frame (iteration). Fig. 2-1 shows a buffer-based dataflow that is constructed by inserting buffers between nodes.

In Fig. 2-1(a), a node can be regarded as the source and destination of data. For example, node 1 generates data as the source of the data fed to nodes 2 and 6. As shown in Fig. 2-1(b), this source-destination relationship can be isolated by inserting buffers between nodes. By separating the relationships between nodes, nodes only represent their functionality. The isolation also facilitates to reconfigure the overall system.



(a) A dataflow graph



(b) Buffer insertion

Figure 2-1: A buffer−based dataflow derived from a dataflow by buffer insertion.

In a buffer-based dataflow graph, inserting a buffer to an edge represents delivering a data frame from its source to destination. Thus, the size of data frames appearing at the input port of a buffer is identical to the size of data frames at the output port of the buffer. Furthermore, while a source node is writing data to a buffer, the corresponding destination node is able to read data from the buffer. Therefore, buffers in a buffer-based dataflow can be realized as the dual-port memory which allows simultaneous writing and reading access.

Let $BC_{i,j}$ denote the buffer between the producer node $i$ and the consumer node $j$. The primary parameters which determine the buffer controller structure and overall physical realization are the following [7]: logic latency ($L_i$), write offset ($nw_{i,j}$), read offset ($nr_{i,j}$), block size ($M_{i,j}$) and delay factor ($D_{i,j}$). The logic latency $L_i$ is the latency of node $i$. The write offset $nw_{i,j}$ represents the difference between reading data from the previous buffer and writing data to the current buffer without considering $L_i$. The read offset $nr_{i,j}$ 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 node $i$ and the reading speed of node $j$ are matched. However, if the writing speed of node $i$ is slower than the reading speed of node $j$, node $j$ does not read valid data from $BC_{i,j}$. The delay factor $D_{i,j}$ is to represent this rate mismatch between nodes $i$ and $j$. The block size $M_{i,j}$ characterizes the data size generated by node $i$. It also determines the maximum storage requirement of $BC_{i,j}$. The unit of all the parameters described above is the unit cycle of the target platform used.

Fig. 2-2 informally shows the write and read timing of $BC_{i,j}$. When the logic latency of node $i$ (i.e., $L_i$) and the write offset (i.e., $nw_{i,j}$) related to $BC_{i,j}$ elapse, node $i$ starts to write data to $BC_{i,j}$. If the writing speed of node $i$ is slower than the reading speed of node $j$, the reading of $BC_{i,j}$ may finish before the writing of $BC_{i,j}$ ends. In this case, node $j$ does not read the whole data generated from node $i$. In order to prevent wrong data transfers due to the mismatch of writing and reading speed, node $j$ starts to read data from $BC_{i,j}$ when $\max\{nr_{i,j}, D_{i,j}\}$ passes from the start of writing.

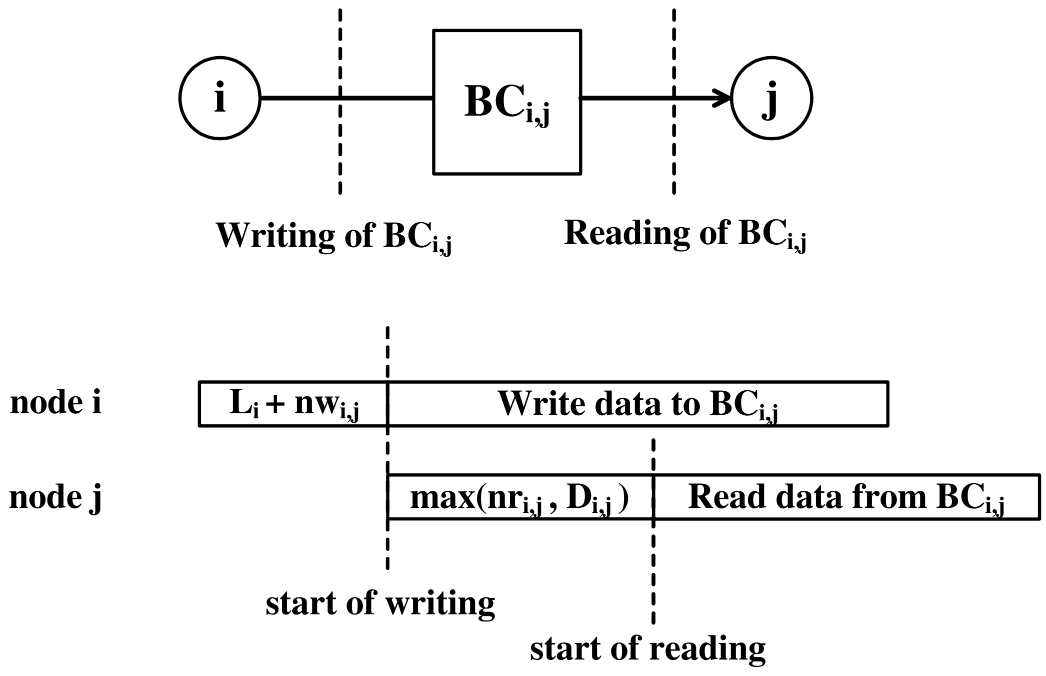


Figure 2-2: Illustration of read/write timing associated with $BC_{i,j}$.

## 2.2 Buffer Controller Parameter Characteristics

In order to construct a buffer-based dataflow, a table called the *buffer controller parameter table* should be extracted from the operational dependency of a dataflow and the offsets of fan-ins and fan-outs of processing elements, as shown in Fig. 2-3. Here, the fan-ins and fan-outs offsets represent the input and output characteristics of a node. When a node has multiple fan-ins, the timing differences between reading data from the multiple fan-ins determine the fan-ins offset. For example, suppose that node $i$ has two fan-ins and that the start time of reading data from one fan-in port is 10 cycles earlier than the start time of reading data from the other fan-out port. The fan-ins offset of node $i$ is then represented as [0, 10]. If a node has multiple fan-outs, the timing differences between writing data to the multiple fan-outs determine the fan-outs offset. The fan-ins and fan-outs offsets and the operational

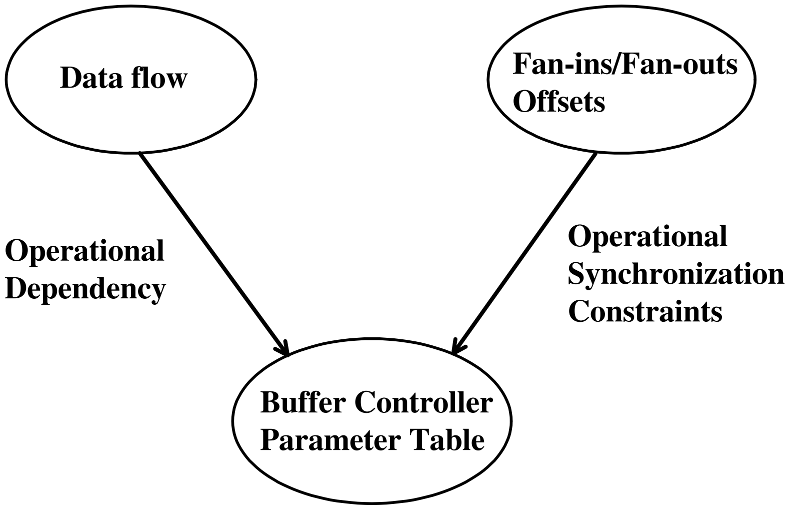

Data flow
Fan-ins/Fan-outs Offsets
Operational Dependency
Operational Synchronization Constraints
Buffer Controller Parameter Table

Figure 2-3: Generating the buffer-controller parameter table.

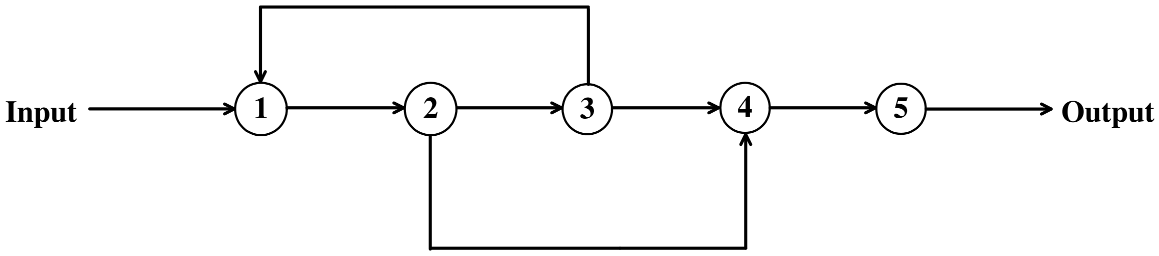


Figure 2-4: An example dataflow.

dependency determine the functional characteristics of a dataflow. Therefore, when a dataflow is converted to a buffer-based dataflow by inserting buffer controllers, the fan-ins and fan-outs offsets and the operational dependency *must* be preserved. Edge activity times represent the timing relation of data transfers through edges, and node execution times represent the time to process data from fan-in to fan-out edges.

For the dataflow shown in Fig. 2-4, Table 2.1 represents the edge activity times and node execution times. In Table 2.1, $e_{i,j}$ represents the edge from source node i to destination node j. $start\_write_{i,j}$ and $start\_read_{i,j}$ represent the start times of writing data and reading data through $e_{i,j}$, respectively. The operational dependency of each edge is described from $e_{input,1}$ to $e_{5,output}$. This dependency simply

Table 2.1: Operational Dependency Derived from Fig. 2-4

| | Operational dependency |
|---|---|
| $e_{input,1}$ | $start\_write_{input,1} < start\_read_{input,1}$ |
| $e_{1,2}$ | $start\_write_{1,2} < start\_read_{1,2}$ |
| $e_{2,3}$ | $start\_write_{2,3} < start\_read_{2,3}$ |
| $e_{2,4}$ | $start\_write_{2,4} < start\_read_{2,4}$ |
| $e_{3,1}$ | $start\_write_{3,1} < start\_read_{3,1}$ |
| $e_{3,4}$ | $start\_write_{3,4} < start\_read_{3,4}$ |
| $e_{4,5}$ | $start\_write_{4,5} < start\_{}_read_{4,5}$ |
| $e_{5,output}$ | $start\_write_{5,output} < start\_read_{5,output}$ |
| node 1 | $\max\{start\_read_{input,1}, start\_read_{3,1}\} + L_1$ $< start\_write_{1,2}$ |
| node 2 | $start\_read_{1,2} + L_2$ $< \min\{start\_write_{2,3},\ start\_write_{2,4}\}$ |
| node 3 | $start\_read_{2,3} + L_3 < start\_write_{3,4}$ |
| node 4 | $\max\{start\_read_{3,4},\ start\_read_{2,4}\} + L_4$ $< start\_write_{4,5}$ |
| node 5 | $start\_read_{4,5} + L_5 < start\_write_{5,output}$ |

represents $start\_write_{i,j} < start\_read_{i,j}$ because writing data through an edge always precedes reading data through the edge. In addition, since the edge is connected to the fan-in/fan-out port of a node, the functional execution of the node determines the operational dependency between fan-in and fan-out edges. The operational dependency between fan-in and fan-out edges are listed from node 1 to node 5. For example, node 1 has two fan-in edges, $e_{input,1}$ and $e_{3,1}$, and one fan-out edge, $e_{1,2}$. Before node 1 begins to write data through $e_{1,2}$, it reads data from $e_{input,1}$ and $e_{3,1}$. However, $e_{3,1}$ is the feedback loop. Therefore, the data read from node 3 have no dependency on the data transferred through $e_{2,3}$ in the current iteration period. If node 3 generates data by referring to the data read from node 2 in the current iteration period, the dataflow falls into the deadlock condition. That is, node 1 keeps waiting for the data from node 3 while node 2 waits for the data from node 1, and node 3

Table 2.2: Fan-ins and Fan-outs Offsets of Figure 2-4

| | Fan-ins Offset | Fan-outs Offset |
|---|---|---|
| node 1 | $[I_{input,1}\ I_{3,1}]$ | N/A |
| node 2 | N/A | $[O_{2,3}\ O_{2,4}]$ |
| node 3 | N/A | $[O_{3,1}\ O_{3,4}]$ |
| node 4 | $[I_{2,4}\ I_{3,4}]$ | N/A |
| node 5 | N/A | N/A |

also keeps waiting for the data from node 2. Thus, $start\_write_{3,1}$ is removed from the operational dependency of node 3 in Table 2.1. In order to construct the buffer based data flow preserving the intrinsic characterisitcs of a given data flow, we also need the information of fan-ins/fan-outs offsets as shown in Table 2.2.

In Table 2.2, $[I_{i,j}\ I_{k,j}]$ represents the fan-ins offset of node j having two fan-in edges, $e_{i,j}$ and $e_{k,j}$. $[O_{i,j}\ O_{i,k}]$ denotes the fan-outs offset of node i having two fan-out edges, $e_{i,j}$ and $e_{i,k}$. For example, $[I_{input,1}\ I_{3,1}]$ is the fan-ins offset of node 1 having $e_{input,1}$ and $e_{3,1}$ as fan-in edges. $[O_{2,3}\ O_{2,4}]$ is the fan-outs offset of node 2 having $e_{2,3}$ and $e_{2,4}$ as fan-out edges. In case that there is one fan-in/fan-out edge, "N/A" appears in Table 2.2. In terms of *start_write* and *start_read*, $[I_{i,j}\ I_{k,j}]$ and $[O_{i,j}\ O_{i,k}]$ are expressed as the following equations.

$$I_{i,j} = start\_read_{i,j} - min(start\_read_{i,j}, start\_read_{k,j}),$$

$$I_{k,j} = start\_read_{k,j} - min(start\_read_{i,j}, start\_read_{k,j}),$$

$$O_{i,j} = start\_write_{i,j} - min(start\_write_{i,j}, start\_write_{i,k}),$$

$$O_{i,k} = start\_write_{i,k} - min(start\_write_{i,j}, start\_write_{i,k}).$$

Fig. 2-5 shows the buffer-based dataflow converted from the dataflow of Fig. 2-4.

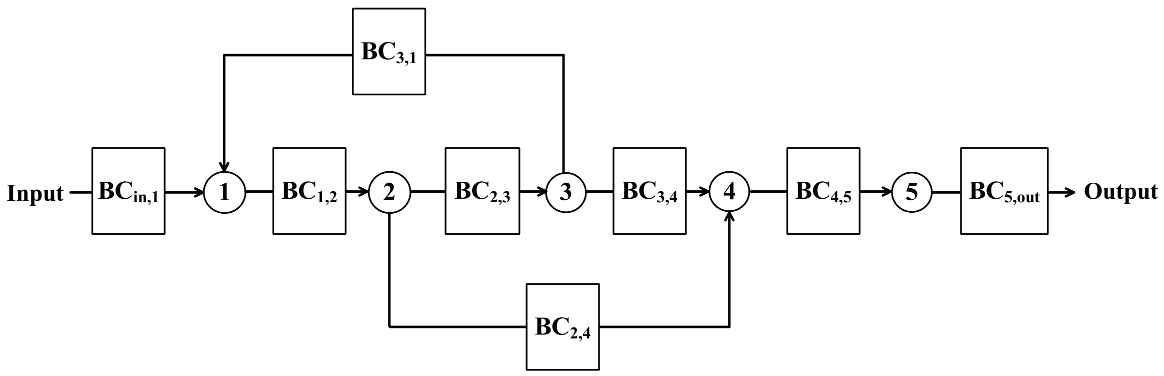


Figure 2-5: The buffer-based dataflow derived from Fig. 2-4.

Since $BC_{i,j}$ has the operational characteristics of $e_{i,j}$ in Table 2.1, $start\_write_{i,j}$ and $start\_read_{i,j}$ of Table 2.1 are used to represent the operational dependency of this buffer-based dataflow. In the buffer controller $BC_{i,j}$, the *start* signals are realized with the primary parameters introduced in Section 2.1 as follows:

$$start\_write_{i,j} = L_i + nw_{i,j} + start_i, \tag{2.1}$$

$$start\_read_{i,j} = start\_write_{i,j} + \max\{nr_{i,j}, D_{i,j}\}, \tag{2.2}$$

$$stop\_write_{i,j} = start\_write_{i,j} + M_{i,j}, \tag{2.3}$$

$$stop\_read_{i,j} = start\_read_{i,j} + M_{i,j}. \tag{2.4}$$

In (2.1), $start_i$ is the time value in which node $i$ begins reading data from the previous buffer controller through its fan-in port. Equation (2.2) reflects the rate mismatch between source node $i$ and destination node $j$. In one iteration period, the data transfer through each buffer controller is done once. Thus, once data have been written to (or read from) the buffer controller $BC_{i,j}$, the data are continuously being written to(or read from) $BC_{i,j}$ until the size of transferred data reaches $M_{i,j}$. Equations (2.3) and

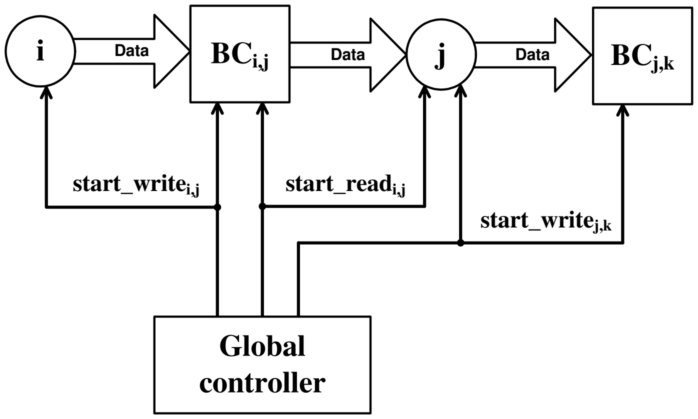


Figure 2-6: Buffer controllers are synchronized by a global controller.

(2.4) represent the end time of writing and reading data to/from $BC_{i,j}$, respectively.

To synchronize the data transfer between a node and a buffer controller, *start_write* and *start_read* are generated by a global controller, as shown in Fig. 2-6. Each element receives its *start_write* and *start_read* from a global controller. *start_write* enables the data transfer from a node to a buffer controller and *start_read* initiates the data transfer from a buffer controller to a node. *stop_write* (*stop_read*) corresponds to the signal transition which changes *start_write* (*start_read*) from 1 to 0.

# Chapter 3

# Energy-Aware Interconnect Resource Reduction Through Buffer Access Manipulation

## 3.1 Introduction

In this chapter, we propose the sharing methodology to reduce the numbers of buffers and buses in a buffer-based dataflow. Since the buffers in a buffer-based dataflow represent the interconnects between nodes, we propose a methodology for sharing buffers in a buffer-based dataflow in order to reduce the number of the interconnects required. Buffer sharing is a well-known technique to reduce system memory in register transfer level (RTL) designs. The register allocation problem occurring in compiler design is also related. Existing buffer-sharing techniques merge those buffers that have non-overlapping lifetimes into one [9–12]. These approaches are based on buffer lifetime analysis but possess limited control of buffer lifetimes.

In the proposed approach, buffer lifetimes are fully controlled in a given time constraint in order to increase the possibility of buffer sharing. Our buffer sharing method moves (or translates) buffer lifetimes within the time constraint so that the buffer lifetimes become non-overlapped with each other. Furthermore, for a finer-grained control of data transfers, a buffer lifetime is separated into two: reading-activity time and writing-activity time. In case activity times are non-overlapped, they are allocated to the same interconnect (i.e., bus) to reduce the number of buses. Thus, our bus-sharing method also translates activity times within the time constraint so that the translated activity times are non-overlapped with each other as much as possible. In addition, the proposed bus-sharing method further splits activity times in order to balance data transfers among buses.

Even though the techniques to share buffers and buses can reduce the synthesizable resources, they may increase the total energy consumption when the runtime operation such as data transfers has the dominant effect on the total energy consumption. When the buffers having different sizes of memory are merged to one, it is possible that a large buffer memory is activated by a small buffer-memory access. Therefore, the energy consumption by the buffer memory activation becomes larger after buffer sharing is applied. In addition, since the bus sharing increases the number of ports in a bus, bus sharing increases the energy consumption of data transfers due to the increased port-loading capacitance. Thus, the sharing case consuming the minimum energy may be different from the sharing case demanding the minimum resource, depending on the buffer and bus costs of the target system used. In order to determine the sharing case consuming the minimum energy, we construct an energy consumption model with the estimated buffer and bus costs.

This chapter is organized as follows. Section 3.2 provides the overview of the proposed approach to reduce interconnect resources by using a buffer-based dataflow. In Section 3.3, we propose a sharing methodology to reduce buffers and buses in a buffer-based dataflow. In order to explain the procedures to share buffers and buses, we define *slack* as the time to translate lifetimes and activity times so that they are non-overlapped. The method of splitting activity times is also proposed to balance data transfers through buses. In the end of Section 3.3, we establish our energy consumption model used to find the sharing case consuming the minimum energy. We also propose an algorithm to find the sharing case based on the energy consumption model. Section 3.4 evaluates the proposed method described in Section 3.3 and discusses the dataflow decomposition that is to reduce the complexity of evaluation. In the evaluation of energy consumption, we observe that the sharing case consuming the minimum energy does not correspond to the sharing case having the minimum resources.

## 3.2 Approach and Objectives

Fig. 3-1 illustrates the realization of a dataflow in a reconfigurable platform. When a dataflow is synthesized in the reconfigurable target platform such as FPGA, the edges are realized through pre-assigned interconnects. If a dataflow is converted to a buffer-based dataflow, inserting buffer controllers doubles the number of interconnects

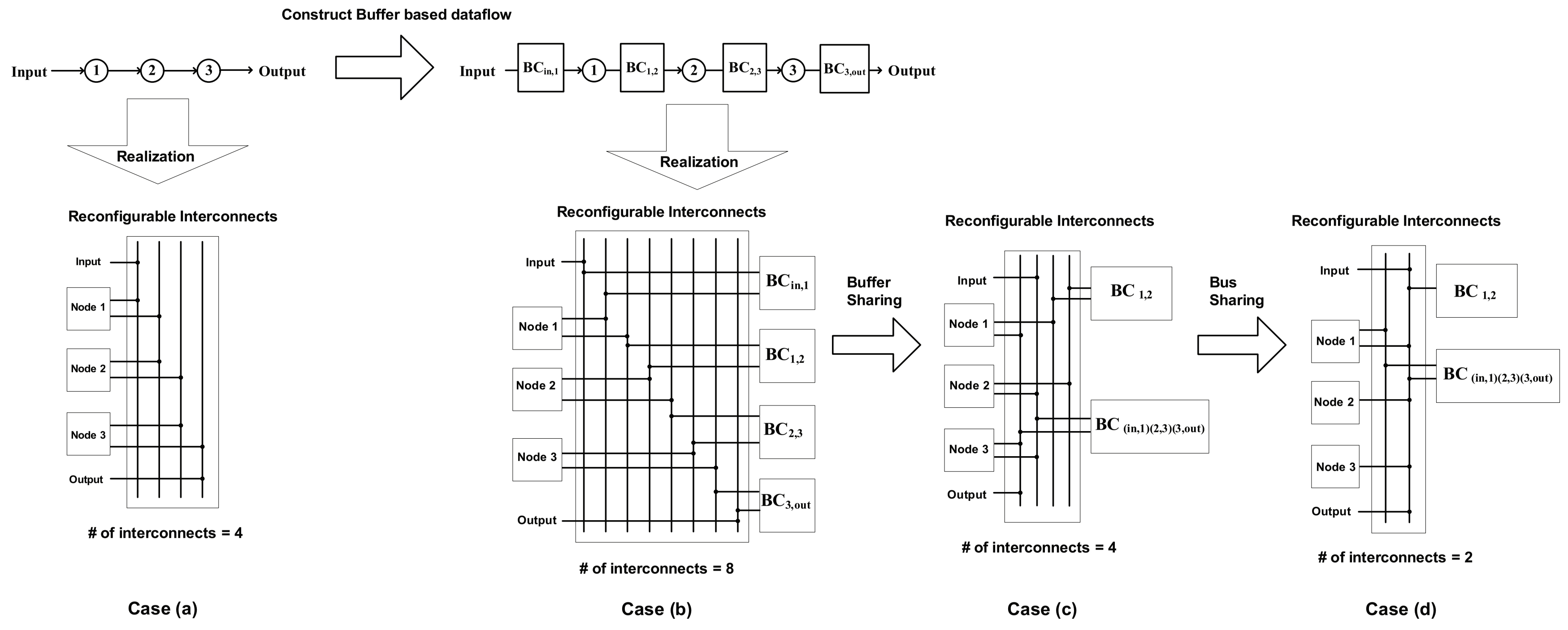


Figure 3-1: Realization of a dataflow in a reconfigurable platform.

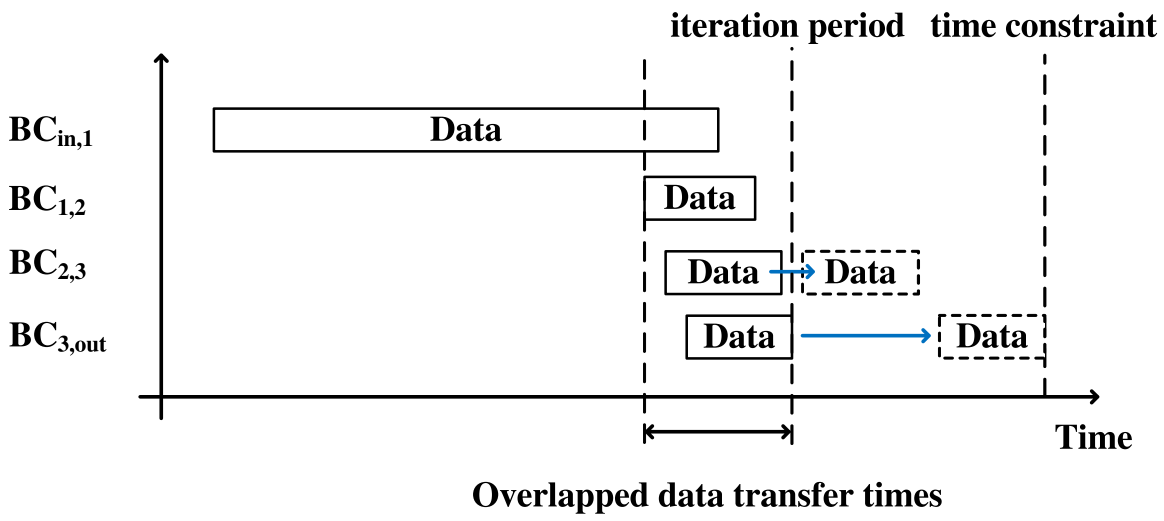


Figure 3-2: Control of data transfers for buffer sharing in Fig. 3-1.

as shown in Case (b). However, the buffer-based dataflow controls data transfers by using the buffer controller parameters introduced in Chapter 2. Therefore, the data transfer control does not require any change of the functional characteristics of nodes. *If data transfers are controlled to use the same interconnects*, the number of interconnects may be reduced.

In Fig. 3-1, data transfers are controlled by applying the buffer and bus sharing technique to the buffer-based dataflow. As the result of sharing, compared with Case (a), the number of interconnects is reduced by two in Case (d). In Fig. 3-1, $BC_{(in,1)(2,3)(3,out)}$ is the buffer controller which $BC_{in,1}$, $BC_{2,3}$ and $BC_{3,out}$ are merged into. Since the buffer controller corresponds to the edge of the original dataflow, buffer sharing reduces the number of interconnects (i.e., buses) in a target platform.

Fig. 3-2 illustrates the control of data transfers for the buffer sharing shown in Fig. 3-1. The rectangles enclosed with solid lines correspond to the data transfer times of the buffer controllers in the $y$ axis. The rectangles with dotted lines and

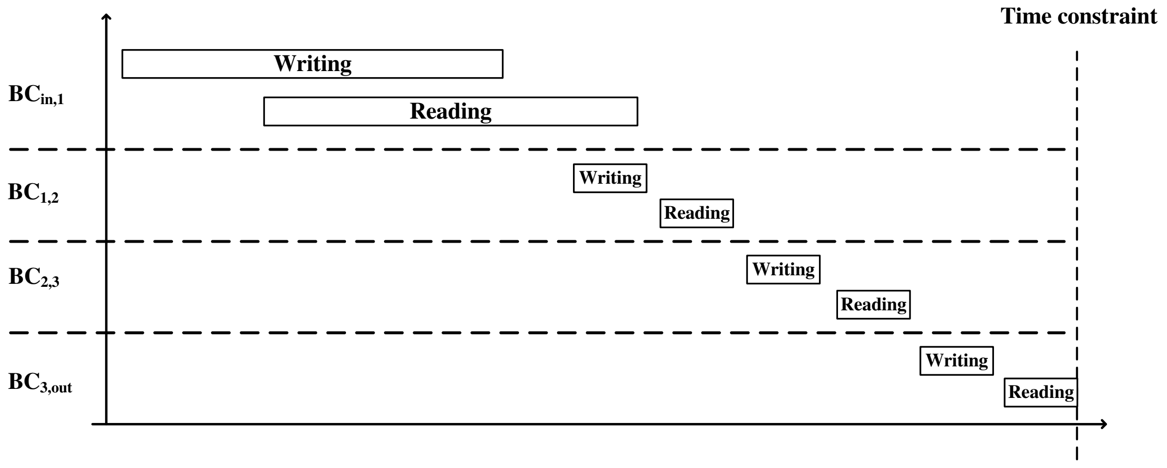


Figure 3-3: Finer-grained view of the data transfers shown in Fig. 3-2

blue arrows represent delayed data transfers. The data transfers corresponding to the solid rectangles require four buffer controllers because the data transfer times are overlapped in the iteration period. However, if a time constraint is larger than the iteration period such that the data transfers corresponding to the dotted rectangles are delayed to be non-overlapped, the number of buffer controllers required for data transfers is reduced into two. Therefore, in Fig. 3-2, the data transfers of $BC_{in,1}$, $BC_{2,3}$ and $BC_{3,out}$ are done through $BC_{(in,1)(2,3)(3,out)}$ of Fig. 3-1. For a finer-grained control of data transfers, we further divide the data transfer of a buffer controller into two parts, namely writing and reading. Fig. 3-3 illustrates the case where the writing and reading parts of the buffer controllers in Fig. 3-2 are arranged to be non-overlapped within a time constraint. Note that non-overlapped writings and readings can be assigned to the same bus. Thus, the bus sharing of Fig. 3-1 is achieved by assigning the reading of $BC_{in,1}$ to one bus and all others to the same bus. Compared with the case where the original dataflow is directly realized, the number of buses is reduced into two. However, the data transfers through buses are unbalanced because

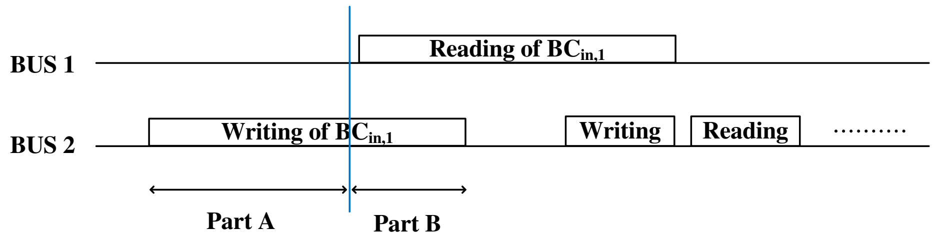


Figure 3-4: Splitting the writing of $BC_{in,1}$ in Fig. 3-3

only one reading (i.e. the reading of $BC_{in,1}$) occupies one bus. To resolve this, the writing of $BC_{in,1}$ can be split, as shown in Fig. 3-4. Here, the writing of $BC_{in,1}$ is split into two: parts A and B. During the time corresponding to part A, there is no data transfer in BUS1. Thus, part A is assigned to BUS1 to balance the data transfers between BUS1 and BUS2.

## 3.3 Proposed Methodology

The three techniques described in the previous section are called *buffer sharing* (Section 3.3.1), *bus sharing* (Section 3.3.2) and *data transfer splitting* (Section 3.3.4), respectively. In this section, we explain more details of these three techniques in turn. We also discuss in Section 3.3.3 the relationship between the interconnect resource sharing and energy consumption. Finally, Section 3.3.5 explains the energy consumption model to determine the sharing case which consumes the minimum energy.

### 3.3.1 Buffer Lifetime and Buffer Sharing

The buffer lifetime of $BC_{i,j}$ is defined as the time period from the start time when node $i$ writes the first data to $BC_{i,j}$ to the end time when node $j$ reads the last data from $BC_{i,j}$. Thus, the buffer lifetime of $BC_{i,j}$, $T(BC_{i,j})$ is given by

$$T(BC_{i,j}) = stop\_read_{i,j} - start\_write_{i,j}. \tag{3.1}$$

Fig. 3-5 illustrates the parameterized buffer-based dataflow corresponding to the dataflow in Fig. 2-1(a). The buffer-based dataflow shown in Fig. 3-5 has the minimum iteration period possible. Therefore, the buffer lifetimes have their minimal values, and they are maximally overlapped each other. If a dataflow has multiple outputs, the end of an iteration period is the maximum value among *stop_read*s of outputs. Thus, the iteration period of Fig. 3-5 is from $start\_write_{0,1}$ to $stop\_read_{10,out2}$. Since buffer controllers are shared when buffer lifetimes are non-overlapped, buffer lifetimes are translated in order to create non-overlapped areas between lifetimes. The range of translated lifetimes is defined as the *slack* of lifetime:

$$slack = given\_constraint - \min\{iteration\_period\} \tag{3.2}$$

where given_constraint comes from the application specification and $min\{\ iteration\_period\}$ is the minimum iteration period of the buffer-based dataflow corresponding to the application. Thus, when $slack\_of\_lifetime > 0$, buffer lifetimes can be translated.

Buffer sharing is done through merging buffer controllers. In order to merge $BC_{i,j}$

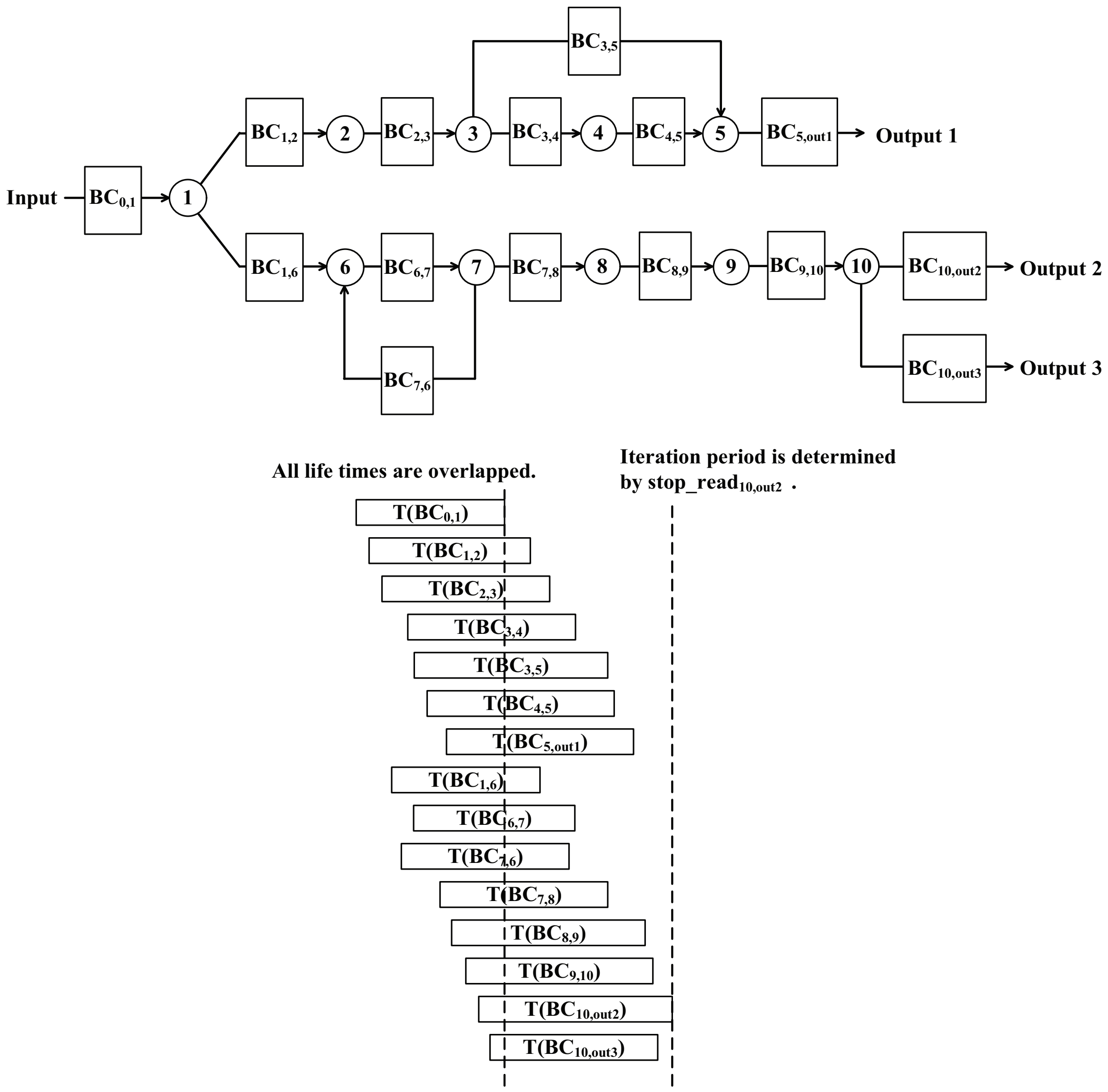


Figure 3-5: The parameterized buffer-based dataflow derived from Fig. 2-1(a).

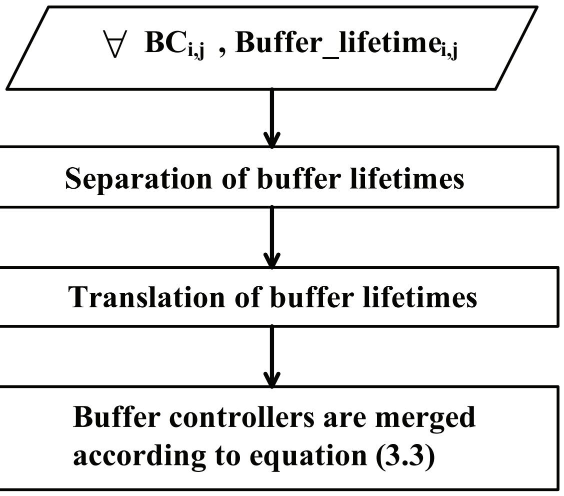

∀ BCi,j , Buffer_lifetimei,j
Separation of buffer lifetimes
Translation of buffer lifetimes
Buffer controllers are merged according to equation (3.3)

Figure 3-6: Proposed buffer sharing procedure.

and $BC_{k,l}$ into the merged buffer controller $BC_{(i,j)(k,l)}$, the lifetimes of $BC_{i,j}$ and $BC_{k,l}$ must satisfy the following condition:

$$\max\{stop\_read_{i,j}, stop\_read_{k,l}\} - \min\{start\_write_{i,j}, start\_write_{k,l}\} > T(BC_{i,j}) + T(BC_{k,l}) \quad (3.3)$$

where $i \neq k, j \neq l$. Equation (3.3) represents that two lifetimes are non-overlapped if the difference between the maximum *stop_read* and the minimum *start_write* of two lifetimes is larger than the summation of the two lifetimes.

Fig. 3-6 shows the overall procedure of the proposed buffer sharing. The first step is to separate buffer lifetimes in a coarse-grained manner. For the separation, buffer lifetimes are moved only to the right. The next step is to translate buffer lifetimes in a finer-grained manner. In order to decrease the number of overlapped lifetimes, buffer lifetimes are moved to either left or right. When all separation and translation steps are completed, buffer controllers are merged according to (3.3).

**Algorithm 1** Maximal Separation Scheme

```
1: MAX_SEPARATION( $T(BC_{i,j})$ )
2:
3: while (1) do
4:    if (node $j$ is an output) and (node $i$ has a single fan-out) then
5:       $nw_{i,j} \leftarrow nw_{i,j}$ + given_constraint $- stop\_read_{i,j}$;
6:    else
7:       Count the number of overlapped lifetimes;
8:       Record the counting number to $cnt\_bf\_tr$;
9:       $nw_{i,j} \leftarrow nw_{i,j} + slack\_of\_lifetime$;
10:      Count the number of overlapped lifetimes;
11:      Record the counting number to $cnt\_af\_tr$;
12:      if ($cnt\_bf\_tr \leq cnt\_af\_tr$) then
13:         //The translation of $T(BC_{i,j})$ increases the number of overlapped lifetimes
14:         //Restore the previous $nw_{i,j}$
15:         $nw_{i,j} \leftarrow nw_{i,j} - slack\_of\_lifetime$;
16:         break;
17:      end if
18:   end if
19: end while
```

For the separation of lifetimes in Fig. 3-6, we propose a procedure called the *maximal separation scheme*, which is detailed in Algorithm 1. When a constraint is given for $slack\_of\_lifetime > 0$, the scheme begins moving lifetimes toward the output (i.e., lifetimes are moved to the right) in order to decrease the number of overlapped lifetimes. If two lifetimes, $T(BC_{i,j})$ and $T(BC_{k,l})$ does not satisfy (3.3), then $T(BC_{i,j})$ and $T(BC_{k,l})$ are overlapped. The maximal separation scheme stops when the number of overlapped lifetimes is not decreased by the translation of lifetimes. When a buffer lifetime is translated, only *nw* of the corresponding buffer controller is changed because the amount of varying *nw* reflects to all *start* and *stop* signals as described in equations (2.1)–(2.4). If a buffer controller translates its lifetime to the right, its *nw* is increased. When a buffer controller translates its lifetime to the left, its *nw* is decreased.

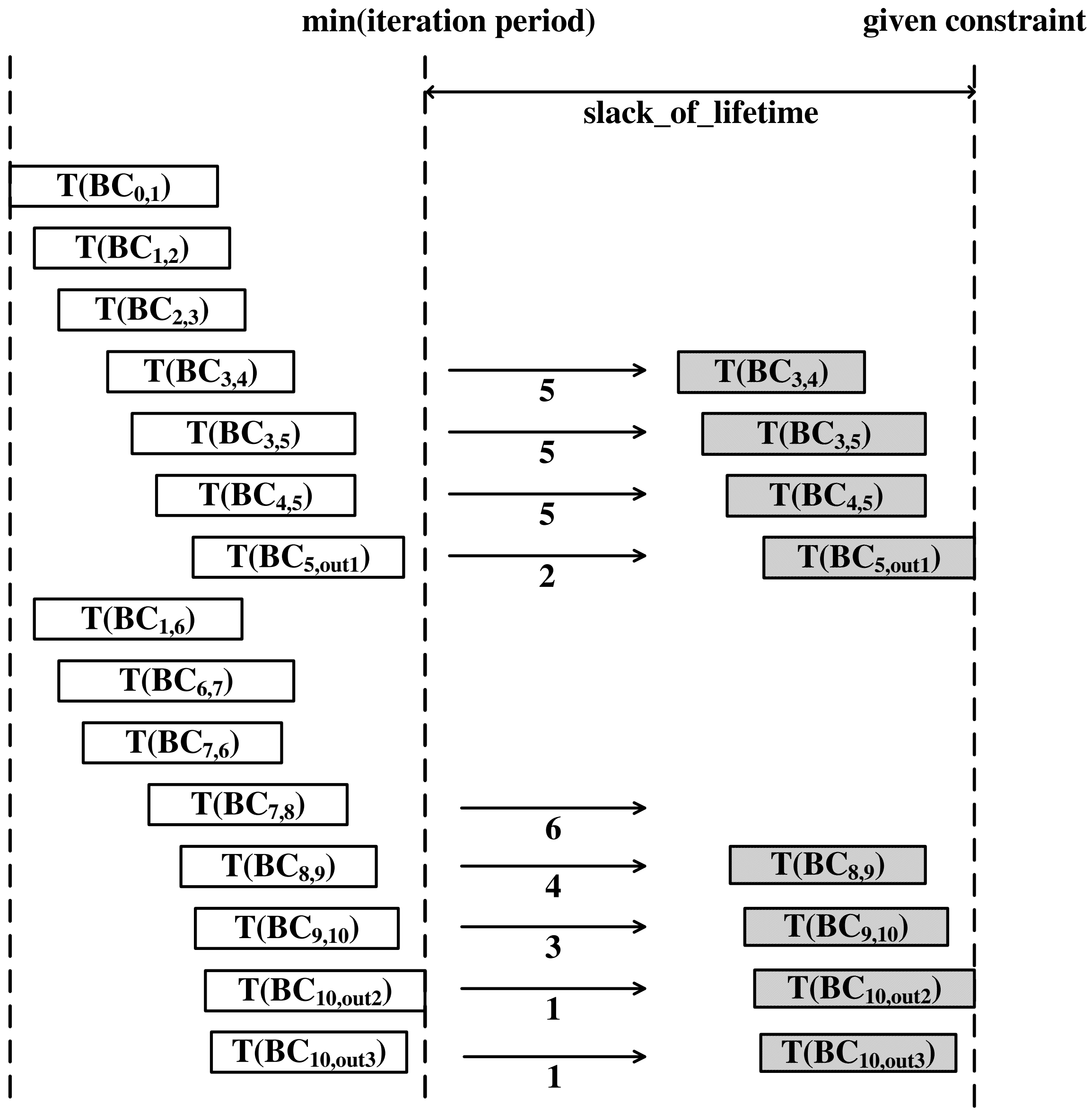


Figure 3-7: Maximal separation of the lifetimes shown in Fig. 3-5.

Fig. 3-7 shows how the maximal separation scheme is applied to the lifetimes of Fig. 3-5. The shaded boxes represent the translated lifetimes and the numbers below arrows indicate the order of translations. For the maximal separation between lifetimes of the input and output sides, the amount of each translation except $BC_{5,out1}$ is equal to $slack\_of\_lifetime$. In case of $BC_{5,out1}$, its amount of translation is larger than $slack\_of\_lifetime$. Even though original $stop\_read_{5,out1} <$ original $stop\_read_{10,out2}$, the translated $stop\_read_{5,out1}$ does not have to be less than the translated $stop\_read_{10,out2}$ because there is neither operational dependency nor the fan-ins/fan-outs offset between $BC_{5,out1}$ and $BC_{10,out2}$. Thus, the original $stop\_read_{5,out1}$ is moved up to the given constraint. This translation case corresponds to line 4 of Algorithm 1.

The translation starts from the buffer controllers connected to outputs. The next translation begins from the buffer controller having the largest *start_write* because the largest *start_write* represents that the buffer controller has no operational dependency on the other buffer controllers which do not yet translate their lifetimes. This rule determines the order of translations. In the fifth translation of Fig. 3-7, $BC_{3,4}$, $BC_{3,5}$ and $BC_{4,5}$ simultaneously translate their lifetimes because $BC_{4,5}$ has the fan-in offset with $BC_{3,5}$, and $BC_{3,5}$ has the fan-out offset with $BC_{3,4}$. The separation stops at the sixth translation because the lifetime translation of $BC_{7,8}$ increases the number of overlapped lifetimes.

When the separation process is completed, a new *slack* is created in the middle of the input and output sides. The range of *slack* is from the maximum *stop_read* of the input side to the minimum *start_write* of the output side. In order to further decrease the number of overlapped lifetimes, lifetimes are translated within the *slack* range. The buffer controllers in the input side translate their lifetimes to the right until (upper bound of $slack - 1$). The right translation updates the upper bound of *slack* as *start_write* of the translated lifetime. The buffer controllers in the output side translate their lifetimes to left until (lower bound of $slack + 1$). The left translation updates the lower bound of *slack* as *stop_read* of the translated lifetime. Algorithm 2 details the procedure described above and corresponds to the step labeled 'Translation of buffer lifetimes' in Fig. 3-6.

Fig. 3-8 illustrates the lifetime translations when the separation process depicted in Fig. 3-7 is completed. The shaded boxes represent the translated lifetimes and the numbers below arrows denote the ordering of translations. The upper bound of *slack* is $start\_write_{3,4}$ of $T(BC_{3,4})$ and the lower bound of *slack* is $stop\_read_{7,8}$ of $T(BC_{7,8})$. However, when $T(BC_{7,8})$ is translated, all lifetimes in the input side are also translated in the same amount as $T(BC_{7,8})$ to preserve fan-ins/fan-outs offsets. Thus, the translation does not reduce the number of overlapped lifetimes. In order to translate one lifetime at a time, the right translation starts from $T(BC_{2,3})$ and the left translation begins from $T(BC_{8,9})$.

### 3.3.2 Activity time and Bus Sharing

A buffer controller has one writing and one reading ports, and one writing and one reading activity times are defined for each buffer controller. In $T(BC_{i,j})$, the writing activity time is defined as the time from $start\_write_{i,j}$ to $stop\_write_{i,j}$. The reading

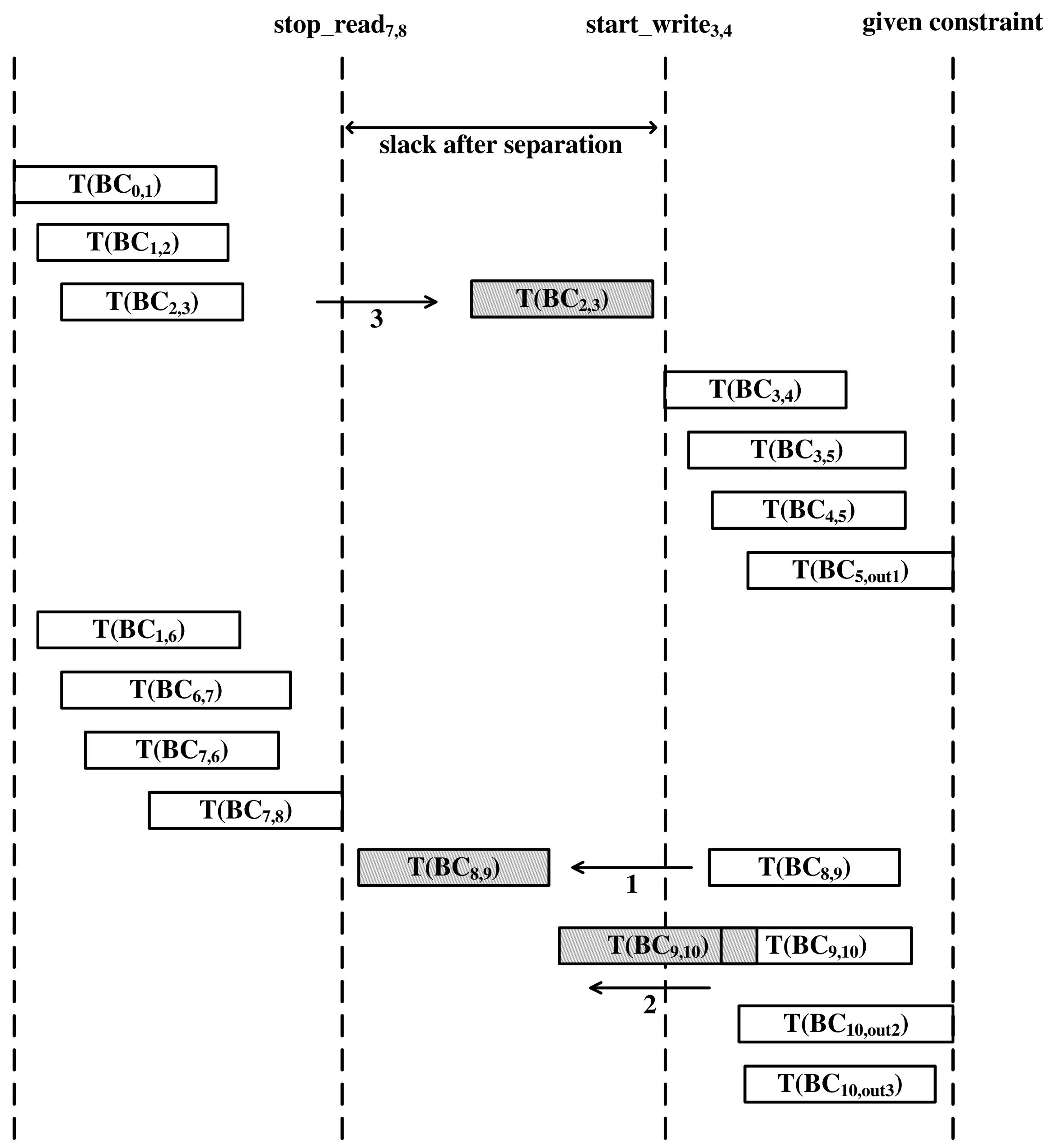


Figure 3-8: Lifetime translation when the separation in Fig. 3-7 is completed.

**Algorithm 2** Translation of lifetimes when the maximal separation scheme is completed

```
1: TRANSLATION_AFTER_SEPARATION( $T(BC_{i,j})$ )
2:
3: if ($T(BC_{i,j})$ in the input side) then
4:     Count the number of overlapped lifetimes;
5:     Record the counting number to $cnt\_bf\_tr$;
6:     //Right translation of $T(BC_{i,j})$
7:     $nw_{i,j} \leftarrow nw_{i,j}$ + (upper bound of $slack - stop\_read_{i,j} - 1$);
8:     Count the number of overlapped lifetimes;
9:     Record the counting number to $cnt\_af\_tr$;
10:    if ($cnt\_bf\_tr \leq cnt\_af\_tr$) then
11:        // Since the translation of $T(BC_{i,j})$ is invalid,
12:        // $nw_{i,j}$ is restored to its previous value.
13:        $nw_{i,j} \leftarrow nw_{i,j}$ − (upper bound of $slack - stop\_read_{i,j} - 1$);
14:    else
15:        // Since the translation of $T(BC_{i,j})$ is valid,
16:        // upper bound of $slack$ is updated.
17:        upper bound of $slack \leftarrow start\_write_{i,j}$;
18:    end if
19: else if ($T(BC_{i,j})$ in the output side) then
20:    Count the number of overlapped lifetimes;
21:    Record the counting number to $cnt\_bf\_tr$;
22:    //Left translation of $T(BC_{i,j})$
23:    $nw_{i,j} \leftarrow nw_{i,j} - (start\_write_{i,j}$ − lower bound of $slack) + 1$;
24:    Count the number of overlapped lifetimes;
25:    Record the counting number to $cnt\_af\_tr$;
26:    if ($cnt\_bf\_tr \leq cnt\_af\_tr$) then
27:        // Since the translation of $T(BC_{i,j})$ is invalid,
28:        // $nw_{i,j}$ is restored to its previous value.
29:        $nw_{i,j} \leftarrow nw_{i,j} + (start\_write_{i,j}$ − lower bound of $slack) - 1$;
30:    else
31:        // Since the translation of $T(BC_{i,j})$ is valid,
32:        // lower bound of $slack$ is updated.
33:        lower bound of $slack \leftarrow stop\_read_{i,j}$;
34:    end if
35: end if
```

activity time is defined as the time from $start\_read_{i,j}$ to $stop\_read_{i,j}$. Fig. 3-9 shows the writing and reading activity times in $T(BC_{i,j})$. $W_{i,j}$ and $R_{i,j}$ represent the writing and reading activities of $BC_{i,j}$, respectively, and $T(W_{i,j})$ and $T(R_{i,j})$ correspond to the writing and reading activity times, respectively. The duration of each activity time is determined by $M_{i,j}$, the size of transferred data. The direction of writing activity is from a node to a buffer controller; the direction of reading activity is the

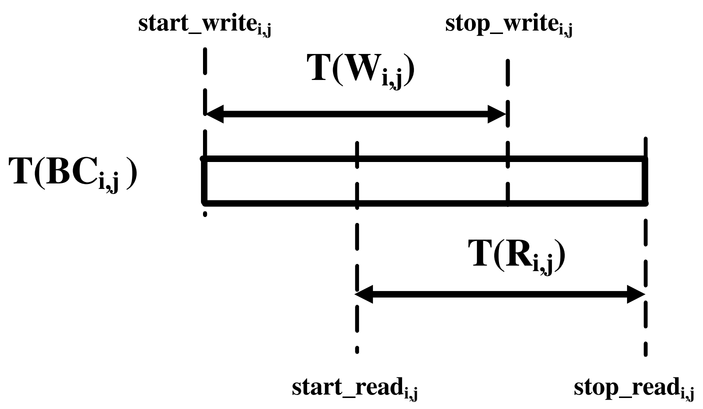


Figure 3-9: Writing and reading activity times in $T(BC_{i,j})$

opposite. Each writing or reading activity uses a bus for the data transfer between a node and a buffer controller. Bus sharing is done by assigning non-overlapped activity times to the same bus. In order to assign activity times of $BC_{i,j}$ and $BC_{k,l}$ to the same bus, the activity times must satisfy the following condition:

$$\begin{aligned} & max\{stop(Act_{i,j}), stop(Act_{k,l})\} - min\{start(Act_{i,j}), start(Act_{k,l})\} \\ & \qquad > M_{i,j} + M_{k,l}, \quad Act_{i,j} \in \{W_{i,j}, R_{i,j}\}, \quad Act_{k,l} \in \{W_{k,l}, R_{k,l}\} \quad (3.4) \end{aligned}$$

where $i \neq k, j \neq l$, $stop(Act)$ represents *stop* of activity time and $start(Act)$ denotes *start* of activity time. For example, if $Act_{i,j} = W_{i,j}$, $stop(Act_{i,j})$ is $stop\_write_{i,j}$ and $start(Act_{i,j})$ is $start\_write_{i,j}$. Equation (3.4) represents that two activity times are non-overlapped if the difference between the maximum *stop* and the minimum *start* of two activity times is larger than the summation of transferred data sizes of the two activities.

Fig. 3-10 shows the overall procedure of the proposed bus sharing technique. In the first step, all activity times are separated without considering a time constraint.

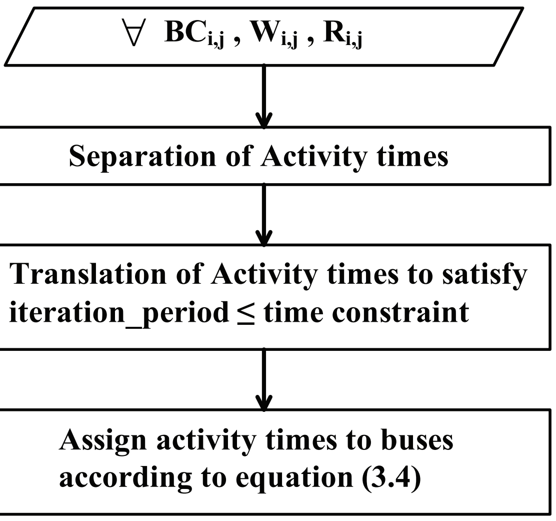

$\forall$ $BC_{i,j}$ , $W_{i,j}$ , $R_{i,j}$
Separation of Activity times
Translation of Activity times to satisfy iteration_period ≤ time constraint
Assign activity times to buses according to equation (3.4)

Figure 3-10: Proposed bus sharing procedure.

Then, if the iteration period exceeds a given time constraint, activity times are translated to satisfy the condition that the iteration period should be less or equal to the time constraint. When all separation and translation steps end, according to (3.4), non-overlapped activity times are assigned to the same bus.

We consider two strategies to use buses: one is to separate writing and reading activities, and the other is not to distinguish writing and reading activities. Based on these bus strategies, we propose the method of separating activity times for bus sharing. Basically, the method separates activity times to be non-overlapped with each other. The separation has two steps: one is separating activity times within a single buffer controller, and the other is separating activity times among buffer controllers.

Algorithm 3 provides the details of the procedure labeled 'Separation of activity times' in Fig. 3-10. Line 11 represents that two activity times are overlapped if the difference between the maximum *stop* and the minimum *start* of two activity times is

**Algorithm 3** Separation of activity times

```
1: //$Act_{i,j} \in \{ W_{i,j}, R_{i,j} \}$, $Act_{k,l} \in \{ W_{k,l}, R_{k,l} \}$
2: **SEPARATION**($Act_{i,j}$, $Act_{k,l}$)
3:
4: if ($i == k$) and ($j == l$) then
5:     //Separation of activity times within a single buffer controller.
6:     if ($Act_{i,j} \neq Act_{k,l}$) then
7:         $nr_{i,j} \leftarrow nr_{i,j} + (stop\_write_{i,j} - start\_read_{i,j}) + 1;$
8:     end if
9: else
10:    //Separation of activity times between buffer controllers.
11:    if $(\max\{stop(Act_{i,j}) , stop(Act_{k,l})\} - \min\{start(Act_{i,j}), start(Act_{k,l})\}) \leq (M_{i,j} + M_{k,l})$ then
12:        if $\max\{stop(Act_{i,j}) , stop(Act_{k,l})\} == stop(Act_{i,j})$ then
13:            if $Act_{i,j} == W_{i,j}$ then
14:                $nw_{i,j} \leftarrow nw_{i,j} + (stop(Act_{k,l}) - start\_write_{i,j}) + 1;$
15:            else if $Act_{i,j} == R_{i,j}$ then
16:                $nr_{i,j} \leftarrow nr_{i,j} + (stop(Act_{k,l}) - start\_read_{i,j}) + 1;$
17:            end if
18:        else if $\max\{stop(Act_{i,j}) , stop(Act_{k,l})\} == stop(Act_{k,l})$ then
19:            if $Act_{k,l} == W_{k,l}$ then
20:                $nw_{k,l} \leftarrow nw_{k,l} + (stop(Act_{i,j}) - start\_write_{k,l}) + 1;$
21:            else if $Act_{k,l} == R_{k,l}$ then
22:                $nr_{k,l} \leftarrow nr_{k,l} + (stop(Act_{i,j}) - start\_read_{k,l}) + 1;$
23:            end if
24:        end if
25:    end if
26: end if
```

smaller than the summation of transferred data sizes of the two activities. According to the sequence of two separation steps, the following two separation approaches are possible: (1) the separation within a single buffer controller precedes the separation among buffer controllers; (2) the activity times among buffer controllers are separated prior to the separation of activity times within a buffer controller.

Fig. 3-11 illustrates these two separation approaches applied to the buffer controllers depicted in Fig. 3-5 under the assumption that the target platform uses separate buses for writing and reading activities. The shaded boxes represent the overlapped region of writing and reading activity times within buffer lifetimes. The

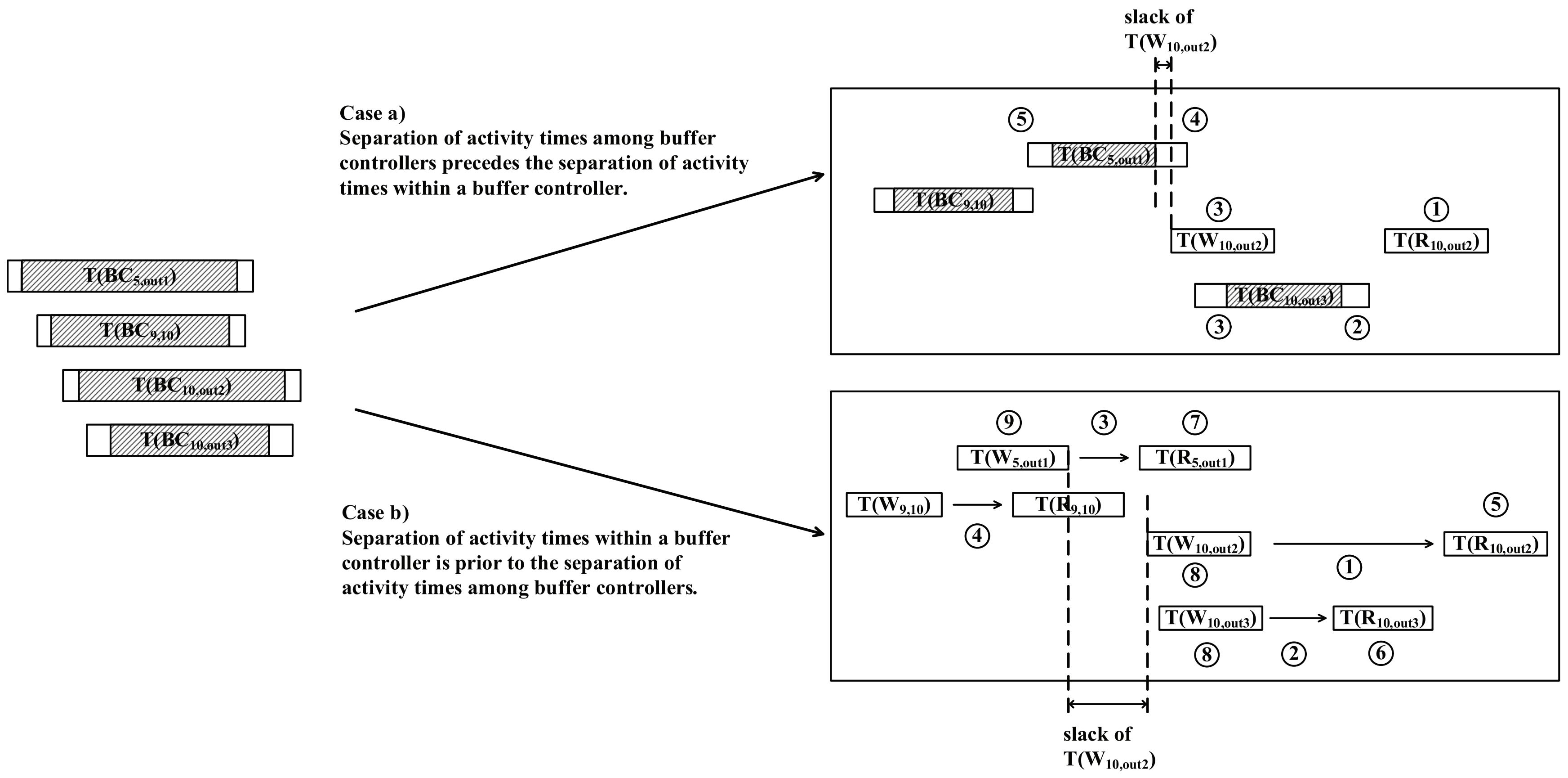

T(BC$_{5,out1}$)
T(BC$_{9,10}$)
T(BC$_{10,out2}$)
T(BC$_{10,out3}$)
Case a)
Separation of activity times among buffer controllers precedes the separation of activity times within a buffer controller.
Case b)
Separation of activity times within a buffer controller is prior to the separation of activity times among buffer controllers.
slack of T(W$_{10,out2}$)
T(W$_{10,out2}$)
T(R$_{10,out2}$)
T(W$_{9,10}$)
T(R$_{9,10}$)
T(W$_{5,out1}$)
T(R$_{5,out1}$)
T(W$_{10,out3}$)
T(R$_{10,out3}$)

Figure 3-11: Two separation approaches applied to the buffer controllers in Fig. 3-5 when buses are separated for writing and reading activities.

circled numbers denote the order of separation. The arrows represent the internal separation of writing and reading activity times within a buffer lifetime. Case a) represents the case that the separation of activity times among buffer controllers precedes the separation of activity times within a buffer controller. Case b) corresponds to the case that the separation of activity times within a buffer controller is prior to the separation of activity times among buffer controllers.

In case a) of Fig. 3-11, the separation process starts from the reading activity of a buffer controller connected to an output because the reading activity does not have the operational dependency on other activities. If there are multiple outputs, the output reading activity having the largest *stop_read* value is selected for the first separation. In the figure, for the first separation, $T(R_{10,out2})$ is thus translated to the right in order to be non-overlapped with $T(R_{10,out3})$. The translation of $T(R_{10,out2})$ leads to the change of $nr$ as follows:

$$\Delta nr_{10,out2} = stop\_read_{10,out3} - start\_read_{10,out2} + 1$$

where $\Delta nr_{10,out2}$ is the amount of increased $nr_{10,out2}$ due to the translation of $T(R_{10,out2})$. The next separation step is determined by the *stop* signals of activity times. Since $stop\_read_{10,out3} > stop\_write_{10,out2}$, the sequence of separation is $T(R_{10,out3}) \rightarrow T(W_{10,out2})$. When $T(R_{10,out3})$ is translated, $T(R_{10,out2})$ is also translated by the same amount to preserve the previous separation between $T(R_{10,out2})$ and $T(R_{10,out3})$.

When buses separate writing and reading activities, the separation of activity times within a buffer controller does not reduce the number of buses because writing and reading activities within a buffer controller are not assigned to the same bus. Therefore, in Fig. 3-11, case b) requires more separation steps than case a).

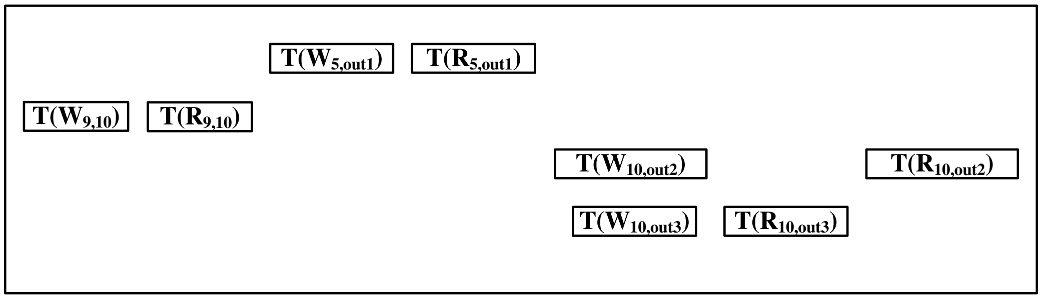
According to the separation approaches, the *slack* size of an activity time varies. The *slack* of an activity time represents the range within which translating an activity time does not increase the number of overlapped activity times. In case a) of Fig. 3-11, the *slack* size of an activity time is 1 because the separation distance between non-overlapped, adjacent writing (or reading) activity times is 1. For example, when the separation of $T(W_{10,out2})$ is completed, $start\_write_{10,out2} = stop\_write_{5,out1} + 1$. However, in case b) of Fig. 3-11, the *slack* size is affected by the internal overlap between writing and reading activity times within a lifetime. Thus, $start\_write_{10,out2} = stop\_write_{5,out1} + stop\_write_{9,10} - start\_read_{9,10} + 1$ (i.e., the translation amount to separate $T(R_{9,10})$ from $T(W_{9,10})$ is reflected). As the result, case b) has larger *slack* of $T(W_{10,out2})$ than case a) has as illustrated in Fig. 3-11.

If iteration_period > given_constraint when the separation of activity times is completed, a *coming-back* scheme begins to satisfy iteration_period $\leq$ given_constraint from the activity time having the largest *slack* size in a descending order. By reducing each *slack* sizes to 1, iteration_period approaches to given_constraint. If iteration_period is still larger than given_constraint even after all *slack* sizes are minimized, the left translation of activity times starts from the output reading activity. Each left translation overlaps activity times as much as possible to reduce iteration_period. When iteration_period $\leq$ given_constraint, the left translation stops. Algorithm 4 describes the coming-back scheme corresponding to the translation of activity times in Fig. 3-10.

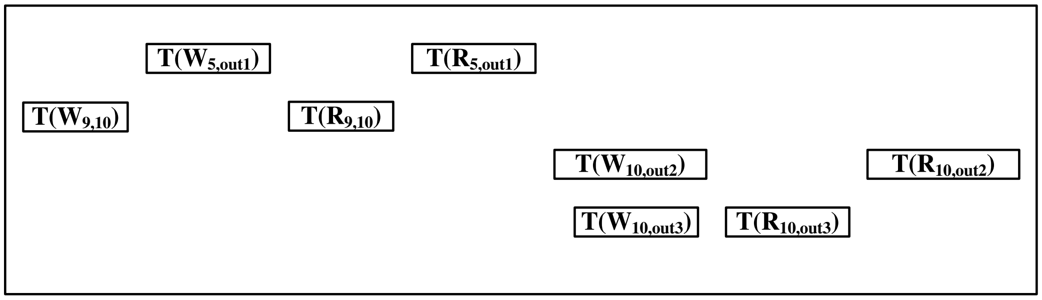
**Algorithm 4** Coming-back scheme

```
1: COMING_BACK(given_constraint)
2:
3: while (iteration_period > given_constraint) do
4:     Minimize slack of activity times;
5:     if (iteration_period ≤ given_constraint) then
6:         Break;
7:     end if
8:     if (All slacks of activity times are minimized) then
9:         left translation of activity times;
10:    end if
11: end while
```

In case buses do not distinguish writing and reading activities, the two separation approaches shown in Fig. 3-11 further isolate overlapped regions between writing and reading activities. Fig. 3-12 illustrates such cases. Fig. 3-12(a) and 3-12(b) correspond to case a) and case b) of Fig. 3-11, respectively. In both cases, activity times are translated to be non-overlapped with other activity times except the case where activity times have fan-out offsets. Thus, the numbers of non-overlapped activity times are the same in both cases. However, the numbers of *non-overlapped lifetimes* of these two approaches are different due to the different separation sequences used. Compared to Fig. 3-12(a), lifetimes of Fig. 3-12(b) are more stretched such that the number of overlapped lifetimes of Fig. 3-12(b) is larger than that of Fig. 3-12(a). In Fig. 3-12(a), the number of overlapped lifetimes is 1 (i.e., $BC_{10,out2}$ and $BC_{10,out3}$). On the other hand, in Fig. 3-12(b), the number of overlapped lifetimes is 2 (i.e., $BC_{10,out2}$ and $BC_{10,out3}$, $BC_{5,out1}$ and $BC_{9,10}$). To this end, our proposed bus sharing separates activity times among buffer controllers prior to the separation of activity times within a buffer controller. Note that the bus sharing technique allocates as many activity times to the same bus as possible in order to reduce the number of buses. It is thus



(a)



(b)

Figure 3-12: Two separation approaches when buses are not distinguished for writing and reading activity. (a) Separating activity times among buffer controllers is prior to the separation of activity times within a buffer controller. (b) Separating activity times within a buffer controller precedes the separation of activity times among buffer controllers.

possible that the data transfers through buses are unbalanced, as shown in Fig. 3-13. Here, we assume that $M_{8,9} = M_{10,out2} = 100$ and $M_{5,out1} = M_{10,out3} = 50$. In one iteration period, BUS1 is activated for 250 cycles and BUS2 is activated for 150 cycles to transfer data. For balanced data transfers through BUS1 and BUS2, BUS2 may be activated for extra 50 cycles to transfer the data of $R_{8,9}$ instead of BUS1. This issue will be further discussed in Section 3.3.4.

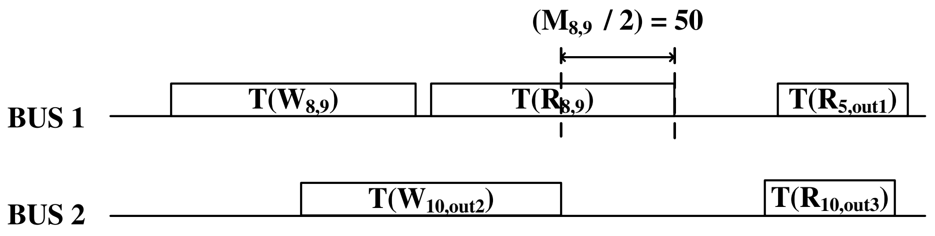


Figure 3-13: Unbalanced data transfers.

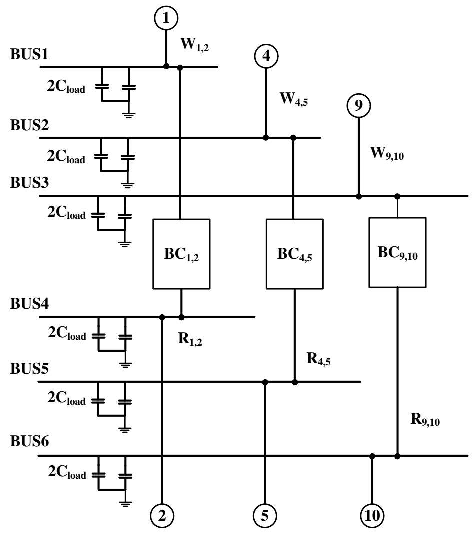
### 3.3.3 Relation between Activity time and Lifetime

We have so far characterized the lifetimes and activity times of buffer controllers for buffer and bus sharing. For buffer sharing, lifetimes are translated for maximal separation. For bus sharing, activity times are translated for separation within a buffer controller or among buffer controllers. Since the buffer and bus sharing starts from the minimal lifetimes, lifetimes are stretched only when activity times are separated. When the lifetime of $BC_{i,j}$ is stretched, $MEM(BC_{i,j})$, (i.e., the memory size of $BC_{i,j}$) is increased because of the following:

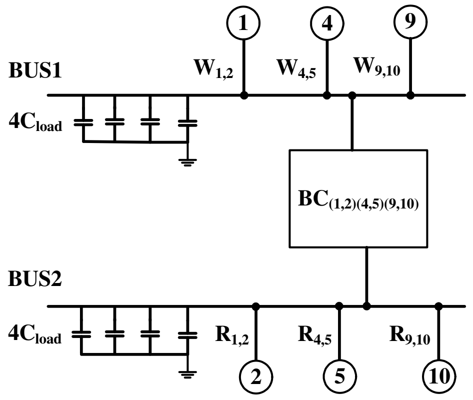
$$MEM(BC_{i,j}) = \min\{M_{i,j}, (start\_read_{i,j} - start\_write_{i,j})\}. \tag{3.5}$$

In (3.5), when writing and reading activity times are overlapped within the lifetime of $BC_{i,j}$, the buffer memory, $MEM(BC_{i,j})$ is determined by the difference between $start\_read_{i,j}$ and $start\_write_{i,j}$. Otherwise, $BC_{i,j}$ needs the buffer memory to store $M_{i,j}$, the total amount of transferred data. In order to minimize the buffer memory size, we translate lifetimes prior to activity times.

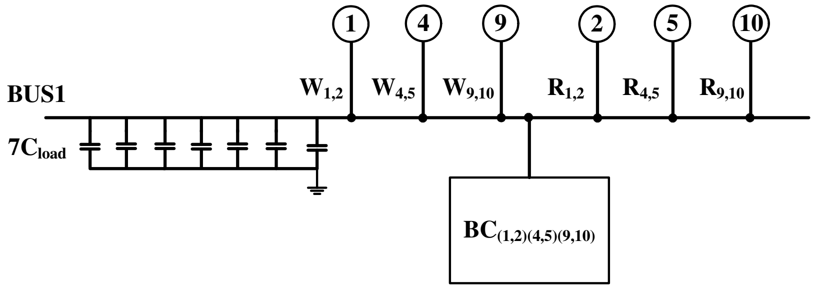
Fig. 3-14 illustrates the relationship between buffer sharing and bus sharing when buses are not separated for writing and reading activities. $BC_{(1,2)(4,5)(9,10)}$ represents



(a) Neither buffer nor bus sharing is applied



(b) Only buffer sharing is applied



(c) Both buffer and bus sharing are applied

Figure 3-14: Buffer and bus sharing when buses are not separated for writing and reading activities.

the shared buffer controller merging $BC_{1,2}$, $BC_{4,5}$ and $BC_{9,10}$. BUS1–BUS6 are pre-assigned buses in the target platform used. The lifetimes of $BC_{1,2}$, $BC_{4,5}$ and $BC_{9,10}$ are non-overlapped, and there are enough *slack*s for the translation of all activity times to be non-overlapped. In Fig. 3-14(a), neither buffer sharing nor bus sharing is applied. In this case, each bus has only one activity. When only buffer sharing is applied as shown in Fig. 3-14(b), the number of buses is two for the writing and reading activities of $BC_{(1,2)(4,5)(9,10)}$. In addition, since only one activity transfers data through a bus at a time, there is *no additional hardware required* in order to multiplex accessing a bus/a shared buffer controller. On the other hand, due to the increased number of activities assigned to each bus, the load capacitance per bus increases. Furthermore, the activated buffer memory size for each activity is determined by the maximum buffer memory size among $BC_{1,2}$, $BC_{4,5}$ and $BC_{9,10}$. In Fig. 3-14(c), the activated buffer memory size is the same as in Fig. 3-14(b), but the load capacitance per bus has increased more than that in Fig. 3-14(b) because all activities are assigned to one bus. The dynamic energy−consumption amounts of Fig. 3-14(a), 3-14(b) and 3-14(c) are

$$\begin{aligned}
&2 \times \{2C_{load}V^2 fM_{1,2} + 2C_{load}V^2 fM_{4,5} + 2C_{load}V^2 fM_{9,10}\} \\
&\qquad + 2 \times \{MEM(BC_{1,2}) \times T(BC_{1,2}) + MEM(BC_{4,5}) \times T(BC_{4,5}) \\
&\qquad\qquad + MEM(BC_{9,10}) \times T(BC_{9,10})\} \times mem\_cost, \quad (3.6)
\end{aligned}$$

$$2 \times \{4C_{load}V^2fM_{1,2} + 4C_{load}V^2fM_{4,5} + 4C_{load}V^2fM_{9,10}\}$$
$$+ 2 \times \{MEM(BC_{(1,2)(4,5)(9,10)}) \times T(BC_{1,2}) + MEM(BC_{(1,2)(4,5)(9,10)}) \times T(BC_{4,5})$$
$$+ MEM(BC_{(1,2)(4,5)(9,10)}) \times T(BC_{9,10})\} \times mem\_cost, \quad (3.7)$$

$$2 \times \{7C_{load}V^2fM_{1,2} + 7C_{load}V^2fM_{4,5} + 7C_{load}V^2fM_{9,10}\}$$
$$+ 2 \times \{MEM(BC_{(1,2)(4,5)(9,10)}) \times T(BC_{1,2}) + MEM(BC_{(1,2)(4,5)(9,10)}) \times T(BC_{4,5})$$
$$+ MEM(BC_{(1,2)(4,5)(9,10)}) \times T(BC_{9,10})\} \times mem\_cost, \quad (3.8)$$

respectively, where $C_{load}$ is the load capacitance of the buses in Fig. 3-14, $f$ is the operating frequency of the buses, $V$ is the supply voltage and *mem_cost* is the power consumption per unit memory access. Compared with (3.6), the load capacitance for each activity is doubled in (3.7) and is 3.5 times higher in (3.8). In addition, the maximum buffer memory size, $MEM(BC_{(1,2)(4,5)(9,10)})$ (= max$\{MEM(BC_{1,2})$, $MEM(BC_{4,5})$, $MEM(BC_{9,10})\}$) is activated for accessing each buffer memory in (3.7) and (3.8). Therefore, the energy consumption in (3.6) $<$ the energy consumption in (3.7) $<$ the energy consumption in (3.8).

### 3.3.4 Split Activity Times

Even after both buffer sharing and bus sharing are completed, there still exist spare times between activity times in a bus. Thus, the translation of activity times within the ranges of spare times does not reduce the number of buses. However, the spare

**Algorithm 5** Split of activity times

```
1: //$Act_{i,j} \in \{ W_{i,j}, R_{i,j} \}$
2: SPLIT($Act_{i,j}$, spare_time, split_amount)
3:
4: if (spare_time > 1) then
5:     //Split creates new $start(Act'_{i,j})$
6:     $start(Act'_{i,j}) \leftarrow stop(Act_{i,j})$ − split_amount;
7:     Translate $Act'_{i,j}$ to be assigned to the different BUS;
8: end if
```

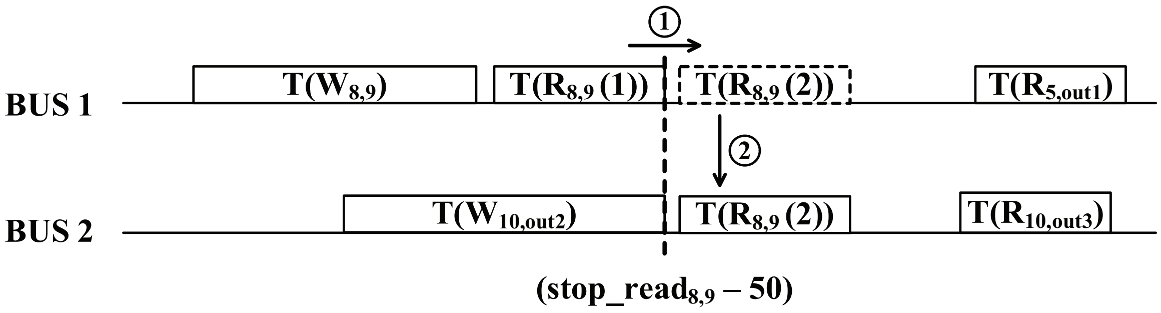


Figure 3-15: Splitting $T(R_{8,9})$

times are used to split activity times for balanced data transfers through buses. Algorithm 5 describes the split of activity times.

Since the split of an activity time creates a new activity time, line 6 of Algorithm 5 generates $start(Act'_{i,j})$ (i.e., either $start\_write'_{i,j}$ or $start\_read'_{i,j}$) for the new activity time. Fig. 3-15 illustrates the case where Algorithm 5 is applied when $M_{8,9} = M_{10,out2}$ $= 100$ and $M_{5,out1} = M_{10,out3} = 50$. In one iteration period, BUS1 is activated for 250 cycles and BUS2 is activated for 150 cycles to transfer data. For balanced data transfers through BUS1 and BUS2, the half of $T(R_{8,9})$ is assigned to BUS2.

The numbers within circles represent the sequence of splitting $T(R_{8,9})$. In the first step, $T(R_{8,9})$ is split into $T(R_{8,9}(1))$ and $T(R_{8,9}(2))$. According to line 6 of Algorithm 5, $start\_read'_{8,9}$ for $T(R_{8,9}(2))$ is $(stop\_read_{8,9} - 50)$. Then, $T(R_{8,9}(2))$ is translated by the amount of one cycle in order to be assigned to BUS2. Finally, $start\_read'_{8,9}$ becomes $(stop\_read_{8,9} - 49)$. However, since $T(R_{8,9}(2))$ adds one port to BUS2, the load capacitance of BUS2 also gets increased. The dynamic energy consumption for the case without splitting activity times and the case where splitting activity times is applied gets changed as follows:

$$3C_{load}V^2f \times 250 + 2C_{load}V^2f \times 150 = 1050C_{load}V^2f \tag{3.9}$$

and

$$3C_{load}V^2f \times 200 + 3C_{load}V^2f \times 200 = 1200C_{load}V^2f, \tag{3.10}$$

respectively, where $C_{load}$ is the load capacitance, $f$ is the operating frequency and $V$ is the supply voltage. Equation (3.9) is the dynamic energy consumption when $T(R_{8,9})$ is not split, and (3.10) is the dynamic energy consumption when $T(R_{8,9}(2))$ is split and assigned to BUS2. Due to the increased load capacitance in BUS2, the amount in (3.9) is less than the amount in (3.10).

When an activity is split, other activities having the operational dependency on the split activity must be split in the same ratio to preserve the functional characteristic of a node. The ratio of input and output data size in a node is predetermined before the original dataflow is converted to a buffer-based dataflow. Therefore, the ratio *must* be maintained even though the split activity is applied.

Fig. 3-16 illustrates the split activities in the inputs/outputs of a node. Here, *cont* represents no split activities. $n$-split denotes that one activity time is split into

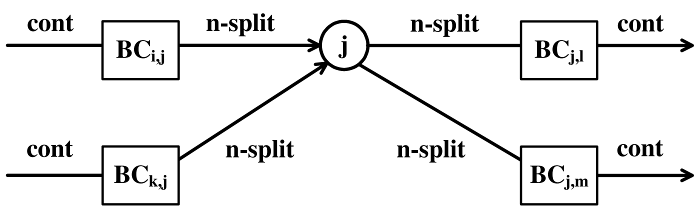


Figure 3-16: Split activities in the inputs/outputs of a node.

$n$ activity times. The ratio of input and output data size in node $j$ is given by

$$\frac{M^n_{i,j}}{M_{i,j}} = \frac{M^n_{j,l}}{M_{j,l}} = \frac{M^n_{k,j}}{M_{k,j}} = \frac{M^n_{j,m}}{M_{j,m}}, \tag{3.11}$$

where $M^n_{i,j}$ is the $n$-th part of the $n$-split reading activity in $BC_{i,j}$ and $M^n_{j,l}$ is the $n$-th part of $n$-split writing activity in $BC_{j,l}$. Note that the ratio of each part is identical.

### 3.3.5 Energy Consumption

According to the buffer and bus costs for energy consumption, the sharing case having the minimum resources may not be the case consuming the minimum energy. If the buffer cost is larger than the bus cost, the buffer sharing among the buffers having different sizes of buffer memory contributes the energy consumption more than the bus sharing that increases the number of ports in buses does. Otherwise, bus sharing affects the energy consumption more than buffer sharing does.

Therefore, in order to select the sharing case consuming the minimum energy, all sharing cases must be examined with the energy consumption model that incorporates all the factors such as buffer costs, bus costs, the number of buffers, the number of buses, the number of ports in buses, activity times, buffer lifetime and the size of activated buffer memory. However, the exhaustive search is a very time-consuming process if there are a great amount of sharing cases. For example, when the time constraint is given as 1900 cycles in the buffer-based dataflow in Fig. 3-5, more than 370,000 sharing cases are possible. In order to reduce the computational complexity, we need to limit the search range by reducing the number of sharing cases examined. For the buffer-based dataflow having five buffers and ten buses, Fig. 3-17 illustrates

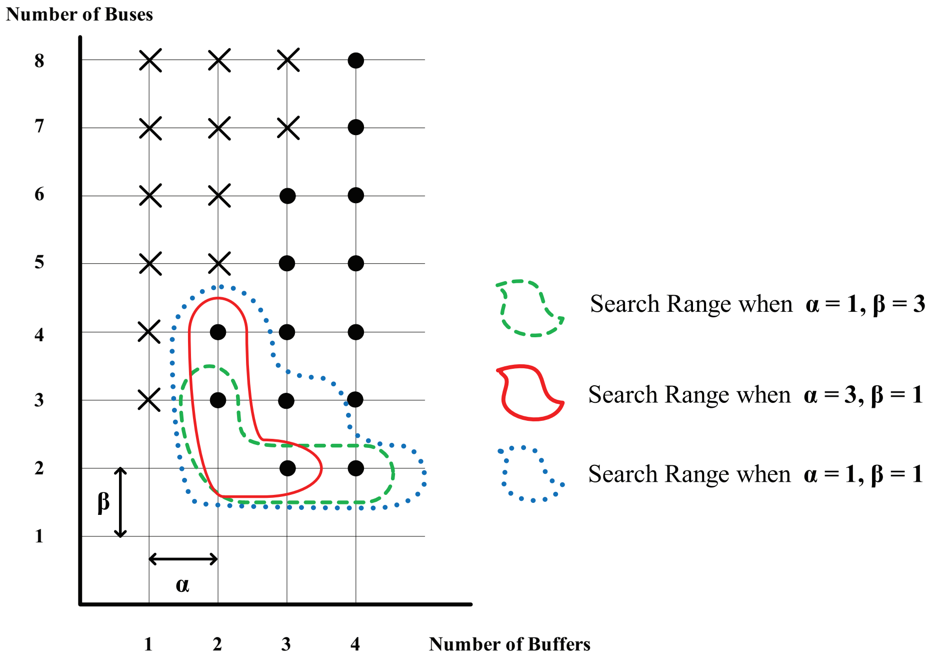

Number of Buses
8
7
6
5
4
3
2
1
β
α
1
2
3
4
Number of Buffers
Search Range when α = 1, β = 3
Search Range when α = 3, β = 1
Search Range when α = 1, β = 1

Figure 3-17: All possible sharing cases for the buffer-based dataflow that has five buffers and ten buses.

all possible sharing cases. Each point of the grid represents a possible sharing case with some buffers and buses. A point does not distinguish different sharing cases that have an identical number of buffers and buses. A cross mark in the grid represents that there is no possible sharing case in the given numbers of buffers and buses. For example, when the number of buffers is two, there is no case having five buses because the maximum number of buses is twice the number of buffers. The cases having neither a solid circle nor a cross mark in the grid are not considered by buffer and bus sharing. $\alpha$ and $\beta$ represent the cost of buffer energy consumption and the cost of bus energy consumption, respectively, and are dependent on the target platform used. The red (solid), green (dashed) and blue (dotted) lines represent the search ranges of finding the sharing case consuming the minimum energy according to $\alpha$ and $\beta$ (i.e., buffer and bus costs). The boundary condition determining the search range is found by the following equations:

$$\begin{aligned} &E_{est}(\text{number of buffers}, \text{number of buses}) \\ &\qquad = \{\alpha(\text{number of buffers}) + \beta(\text{number of buses})\} \times itr\_period, \end{aligned} \tag{3.12}$$

$$E_{bound} = \begin{cases} \{\alpha(\text{minimum number of buffers} + 1) + \beta(\text{minimum number of buses})\} \\ \times itr\_period, \text{ when } \alpha > \beta, \\ \{\alpha(\text{minimum number of buffers}) + \beta(\text{minimum number of buses} + 1)\} \\ \times itr\_period, \text{ when } \alpha < \beta, \\ \{\alpha(\text{minimum number of buffers} + 1) \\ +\beta(\text{minimum number of buses} + 1)\} \times itr\_period, \text{ when } \alpha = \beta, \end{cases} \tag{3.13}$$

where $E_{est}$(number of buffers, number of buses) is the estimated energy consumption value corresponding to the point (number of buffers, number of buses) in the grid of Fig. 3-17, *itr_period* is the iteration period of a given buffer-based dataflow and

$E_{bound}$ is the boundary condition to determine the search range for the minimum energy consumption.

If one resource (i.e., either buffer or bus) consumes more energy than the other, $E_{bound}$ is set to minimize the number of the resource consuming more energy. Thus, the search range is determined between the minimum number and (the minimum number + 1) of the resource, whichever is more expensive (i.e., the resource consuming more energy). In Fig. 3-17, when the cost of one resource is more expensive than the other, the search range is biased to the resource having the lower cost. If buffer and bus costs are the same, both buffers and buses have the same impact on determining the search range. In this case, the search range is evenly spread toward both buffer and bus axes in Fig. 3-17.

However, if the costs are inaccurately given, the sharing case having the minimum energy consumption may appear outside the search range. For example, in Fig. 3-17, when $\alpha = 1$ and $\beta = 3$, the sharing case consuming the minimum energy may appear at (3,3) if actual $\alpha$ is 3. However, (3,3) resides outside the search range. When the search range is determined, the sharing case for the minimum energy consumption is found within the search range based on the following energy consumption model:

$$\begin{aligned} E_{total} &= \alpha \times E_{buffer} + \beta \times E_{BUS} \\ &= \alpha \times \{(\text{ \# of BCs } \times \text{ itr\_period }) + \sum_{n=1}^{\text{\# of BCs}} (T(BC_n) \times MEM(BC_n))\} \\ &\quad + \beta \times \{ \sum_{n=1}^{\text{\# of buses}} \{(\text{\# of ports in } BUS_n) \times \sum_{m=1}^{\text{\# of act in } BUS_n} T(Act_m)\}\}, \end{aligned} \tag{3.14}$$

where $E_{total}$ is the total energy consumption, $E_{buffer}$ is the energy consumption by buffers, $E_{BUS}$ is the energy consumption by buses, (# of act in $BUS_n$) denotes the number of activities in $BUS_n$ and $T(Act_m)$ is the activity time assigned to $BUS_n$. (# of BCs $\times$ itr_period) represents the leakage energy consumption of buffer controllers. The other terms indicate the dynamic energy consumption by buffer memory access and data transfers through buses. Since the iteration period, the number of BCs and the size of activity time come from the buffer-based dataflow to which buffer and bus sharing is not applied, they are not determined by the sharing methodology.

Compared with $E_{est}$ in (3.13), $E_{total}$ incorporates not only the number of resources but also the variables related to the dynamic energy consumption such as the activated buffer memory sizes, lifetimes, activity times and the numbers of ports in buses. Therefore, all sharing cases within a search range are evaluated with $E_{total}$ to find the sharing case consuming the minimum energy. Algorithm 6 summarizes the procedure of finding the sharing case which consumes the minimum energy.

## 3.4 Evaluation of The Proposed Methodology

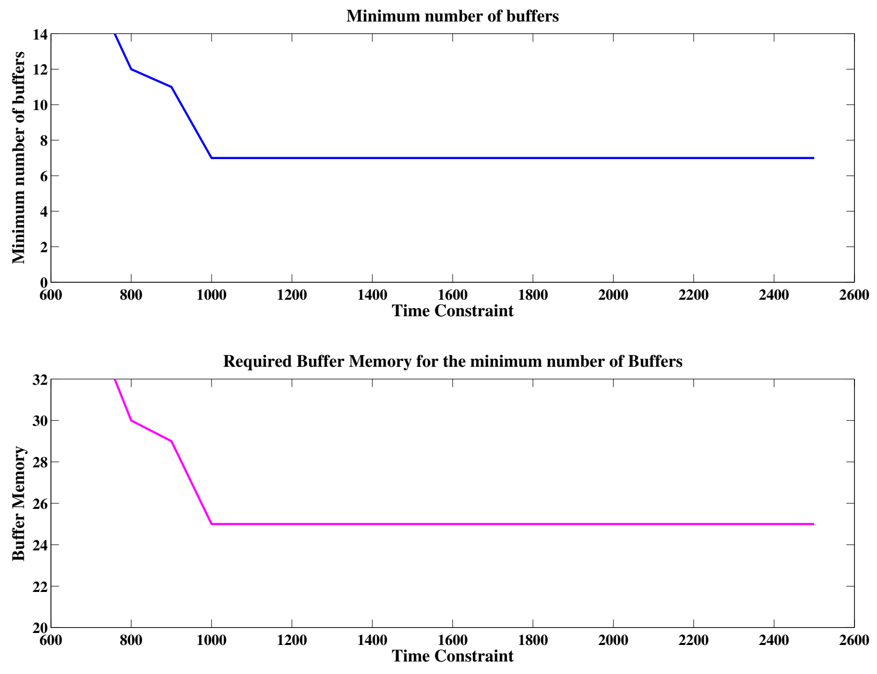
In this section, we evaluate the methodology proposed in Section 3.3 using the buffer-based dataflow shown in Fig. 3-5. This example should work as a representative of many practical cases; this dataflow consists of a feed-forward path, multiple fan-ins, multiple fan-outs and a feedback loop. The proposed methodology was implemented in C.

**Algorithm 6** Find the sharing case consuming the minimum energy

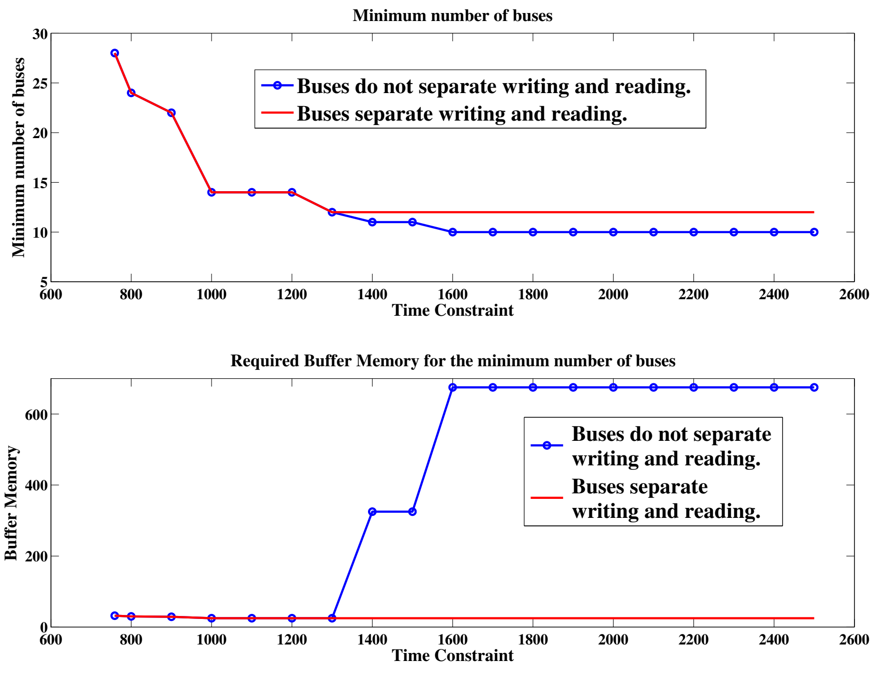
```
1: // $X$ = set of buffer sharing cases
2: // $Y$ = set of bus sharing cases
3: // $S$ = set of the search range to find the minimum energy consumption
4: // $n(x)$ = number of buffers in $x \in X$
5: // $n(y)$ = number of buses in $y \in Y$
6: // $\alpha$ = buffer cost for energy consumption
7: // $\beta$ = bus cost for energy consumption
8: // $E_{est}$ $(n(x), n(y))$ = estimated energy consumption for the sharing cases $(n(x), n(y))$
9: // $E_{bound}$ = boundary condition to determine the search range for the minimum energy con-
   sumption
10: // $E_{total}$ = total energy consumption in equation (3.14)
11:
12: **MINIMUM_ENERGY($\alpha$, $\beta$)**
13:
14: // Buffer sharing is applied.
15: $X \leftarrow$ Buffer_sharing of Fig. 3-6;
16:
17: // Bus sharing is applied.
18: $(X, Y) \leftarrow$ Bus_sharing of Fig. 3-10;
19:
20: // Set up boundary condition to determine the search range
21: **if** $(\alpha > \beta)$ **then**
22:     $E_{bound} = \alpha$ (minimum $n(x)$ + 1) + $\beta$ (minimum $n(y)$);
23: **else if** $(\alpha < \beta)$ **then**
24:     $E_{bound} = \alpha$ (minimum $n(x)$) + $\beta$ (minimum $n(y)$ + 1);
25: **else if** $(\alpha = \beta)$ **then**
26:     $E_{bound} = \alpha$ (minimum $n(x)$ + 1) + $\beta$ (minimum $n(y)$ + 1);
27: **end if**
28:
29: // Search range is constructed with $E_{bound}$
30: $S = \{ (n(x), n(y)) \mid E_{est}\ (n(x), n(y)) \leq E_{bound} \}$;
31:
32: //Find $(x, y)$ having the minimum $E_{total}$
33: min$\{ E_{total}(x, y, \alpha, \beta) \mid (x, y) \in \{ (x,y) \mid (n(x), n(y)) \in S \} \}$;
34:
```

## 3.4.1 Evaluation of Buffer Sharing and Bus Sharing

The buffer lifetimes of Fig. 3-5 are listed in Table 3.1. Fig. 3-18 shows the results of buffer and bus sharing. In all plots of Fig. 3-18, the x-axis represents the given time constraint. The top plots of Fig. 3-18(a) and Fig. 3-18(b) show the minimum number of buffers and buses in each time constraint, respectively. In Fig. 3-18(a) and Fig. 3-18(b), the bottom plots show the amount of buffer memory when the minimum



(a) Minimum number of buffers and required buffer memory



(b) Minimum number of buses and required buffer memory

Figure 3-18: Results of buffer and bus sharing.

Table 3.1: Buffer Controller Parameters for the Minimum Iteration Period of the Dataflow in Fig. 3-5

| | L | M | *start_write* | *start_read* |
|---|---|---|---|---|
| $BC_{0,1}$ | 0 | 500 | 0 | 1 |
| $BC_{1,2}$ | 50 | 450 | 52 | 53 |
| $BC_{2,3}$ | 70 | 400 | 124 | 125 |
| $BC_{3,4}$ | 100 | 300 | 226 | 227 |
| $BC_{3,5}$ | 100 | 250 | 246 | 247 |
| $BC_{4,5}$ | 50 | 250 | 278 | 297 |
| $BC_{5,out1}$ | 110 | 150 | 408 | 409 |
| $BC_{1,6}$ | 50 | 400 | 102 | 103 |
| $BC_{6,7}$ | 50 | 350 | 154 | 155 |
| $BC_{7,6}$ | 0 | 500 | 2 | 3 |
| $BC_{7,8}$ | 100 | 200 | 302 | 303 |
| $BC_{8,9}$ | 50 | 150 | 354 | 355 |
| $BC_{9,10}$ | 50 | 100 | 406 | 407 |
| $BC_{10,out2}$ | 50 | 300 | 458 | 459 |
| $BC_{10,out3}$ | 50 | 100 | 478 | 479 |

number of buffers/buses is used in each time constraint.

In Fig. 3-18(a), for the time constraint greater than 1000, the minimum number of buffers becomes seven because at least seven buffer controllers are required to preserve the fan-ins/fan-outs offsets and the feedback loop.

In Fig. 3-18(b), when a target platform does not separate buses for writing and reading, the buffer memory size increases for the time constraint greater than 1300 because the separation of activity times within a buffer controller starts there. However, when writing and reading buses are separated in the target platform used, the buffer memory size does not increase because there is no separation of activity times within a buffer controller.

Fig. 2-1(a) in Section 2 is the original dataflow (without buffer controllers) of Fig. 3-5. In order to realize the dataflow of Fig. 2-1(a), the target platform used utilizes

Table 3.2: The number of buses when the proposed sharing methodology is applied to SIRF [8] and IPv4 forwarding [14]

| | # of buses (original) | # of buses (sharing) | Buffer memory | Constraint | Reduction ratio |
|---|---|---|---|---|---|
| SIRF [8] | 9 | 5 | 308 bytes | 4Mbps | 44.4 % |
| IPv4 [14] | 13 | 8 | 136 bytes | 33M Packets/s | 38.5 % |

15 buses. When our proposed sharing methodology is applied to the buffer-based dataflow of Fig. 3-5, which is converted from the dataflow of Fig. 2-1(a), the total number of buses becomes smaller than 15 if the time constraint is greater than or equal to 1000.

We also apply the proposed sharing methodology to data-centric applications such as sample importance resample filter (SIRF) [8] and IPv4 forwarding [14]. Table 3.2 summarizes the results we obtained.

In Table 3.2, the second column shows the number of buses in the original dataflow. When each node in the original dataflow is implemented as an individual hardware, the number of edges is the same as the number of buses in the original dataflow. The third column is the number of buses when the proposed sharing methodology is applied to the original dataflow. Since our sharing methodology uses the buffer-based dataflow, which is constructed from its original dataflow, additional buffer memory is required as shown in the fourth column of the table. The fifth column is the constraint which comes from the application specification, and the sixth column indicates the bus reduction ratio. As shown in Table 3.2, our proposed sharing methodology can reduce the number of buses with additional buffer memory for controlling data transfers.

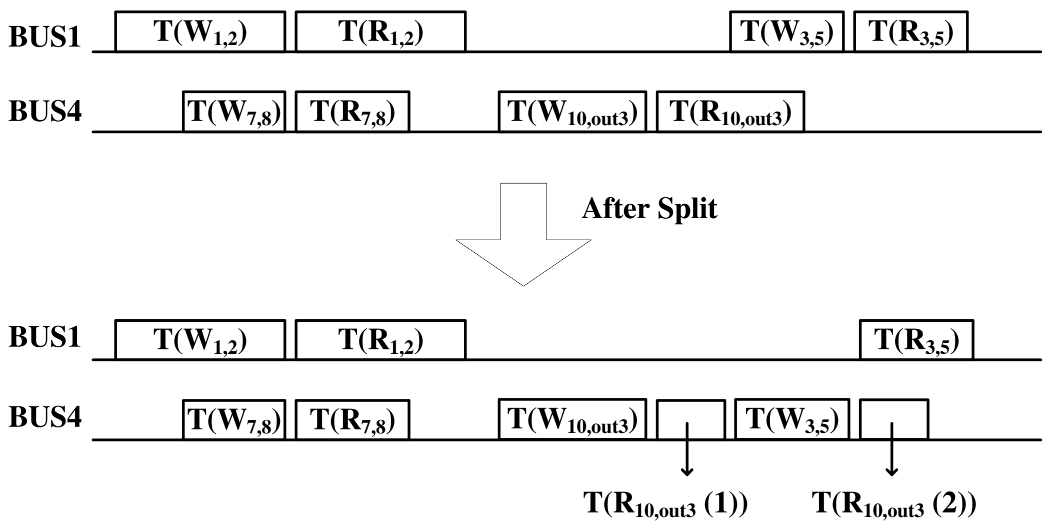


Figure 3-19: Evaluation of split activity times.

## 3.4.2 Evaluation of Split Activity Times

In order to evaluate the technique to split buffer access activity, we derive from Fig. 3-5 one sharing case where time constraint is 1900. In this case, the amount of data transfers through $BUS1$ is 1400 and the amount of data transfers through $BUS4$ is 600. By applying the proposed splitting technique, the amounts of data transfers through $BUS1$ and $BUS4$ become 1150 and 850 respectively.

Fig. 3-19 shows the process. Here, $M_{1,2} = 450$, $M_{3,5} = 250$, $M_{7,8} = 200$ and $M_{10,out3} = 100$. In order to assign $T(W_{3,5})$ to $BUS4$, $T(R_{10,out3})$ is split into two parts, namely $T(R_{10,out3}(1))$ and $T(R_{10,out3}(2))$. For the space of $T(W_{3,5})$ in $BUS4$, $T(R_{10,out3}(2))$ is translated to the right. The translation effect of $T(R_{10,out3}(2))$ is reflected to $D$ in the buffer controller parameter table as shown in Table 3.3.

From (3.5), the buffer memory for $BC_{10,out3}$ is $\min\{M_{10,out3}, \max\{nr_{10,out3}, D_{10,out3}\}\}$. This is because $start\_read_{10,out3} - start\_write_{10,out3} = \max\{nr_{10,out3}, D_{10,out3}\}$. In both cases (i.e., split and non-split cases), $M_{10,out3}$ determines the buffer memory size be-

Table 3.3: Buffer Controller Parameters of $BC_{10,out3}$ when split buffer access activity is applied in Fig. 3-19.

| | L | nr | nw | D | M | *start_read* | *stop_read* |
|---|---|---|---|---|---|---|---|
| $BC_{10,out3}$ Without Split | 50 | 101 | 624 | 0 | 100 | 1203 | 1303 |
| $BC_{10,out3}$ With Split | 50 | 101 | 624 | 0 | 50 | 1203 | 1253 |
| | | | | 252 | 50 | 1505 | 1555 |

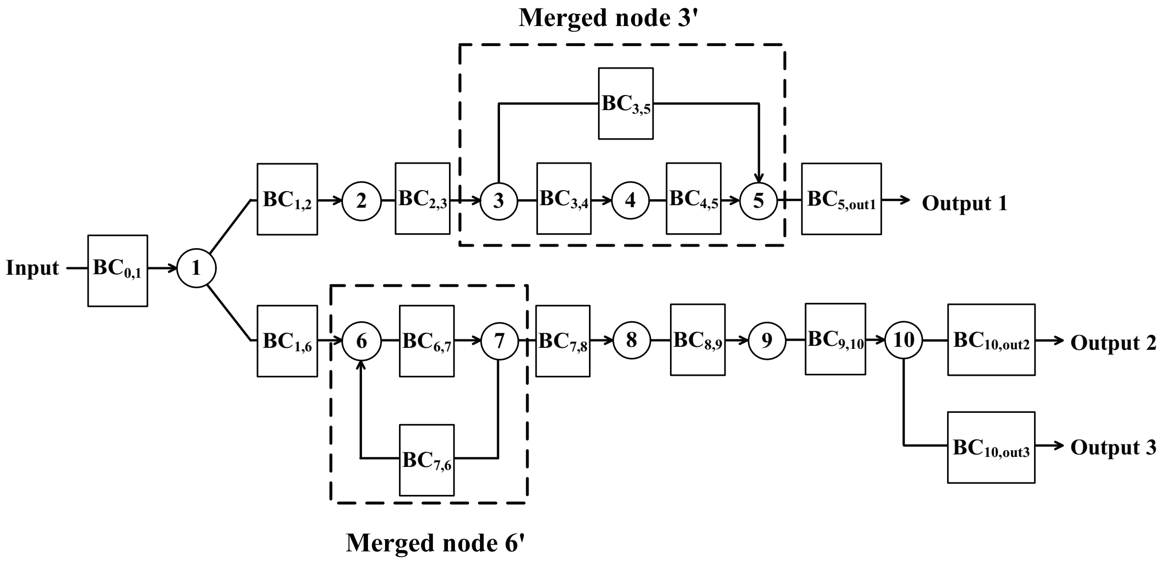


Figure 3-20: Decomposition of the buffer-based dataflow in Fig. 3-5.

cause $M_{10,out3} < \max\{nr_{10,out3}, D_{10,out3}\}$. Therefore, the buffer memory is unchanged even though $T(R_{10,out3}(2))$ is translated.

### 3.4.3 Decomposing A Buffer-Based Data Flow to Reduce the Complexity of Evaluation

If a buffer-based dataflow has many buffers, intensive computation is required to find the sharing cases satisfying the given cost such as the buffer memory size and the number of buses. In order to reduce the computational complexity, a given buffer-based dataflow is hierarchically decomposed as shown in Fig. 3-20.

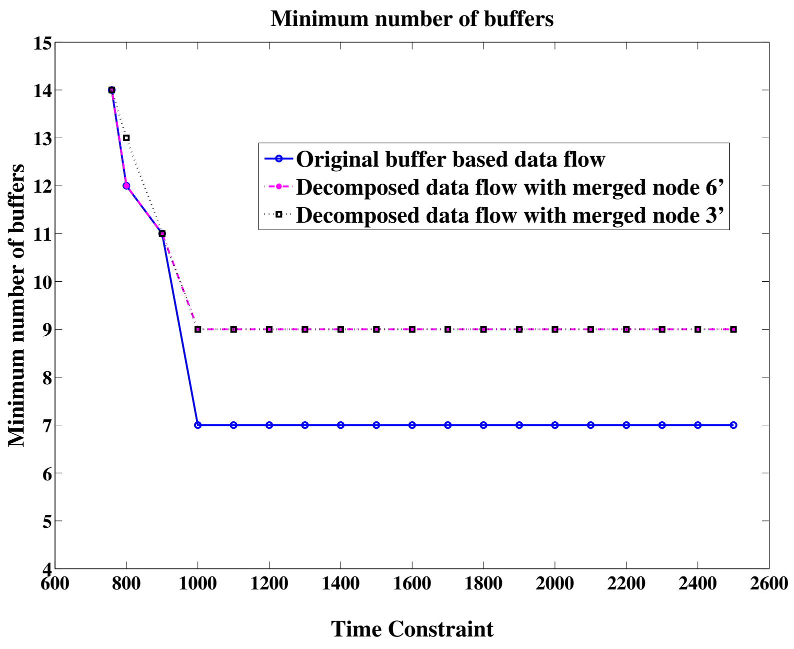
In Fig. 3-20, since $BC_{6,7}$ and $BC_{7,6}$ form a feedback loop, these buffer controllers are not merged into one, whereas nodes 6 and 7, $BC_{6,7}$ and $BC_{7,6}$ are merged to node $6'$. The latency $L_{6'}$ of the merged node $6'$ is found by using the operational dependency from Table 3.1. After the nodes and the buffer controllers are merged, $BC_{1,6}$ and $BC_{7,8}$ are changed to $BC_{1,6'}$ and $BC_{6',8}$, respectively. This leads to the following:

$$\begin{aligned} start\_read_{1,6'} &= start\_read_{1,6}, \\ start\_write_{6',8} &= start\_write_{7,8}, \\ start\_read_{1,6'} + L_{6'} &< start\_write_{6',8}. \end{aligned} \tag{3.15}$$

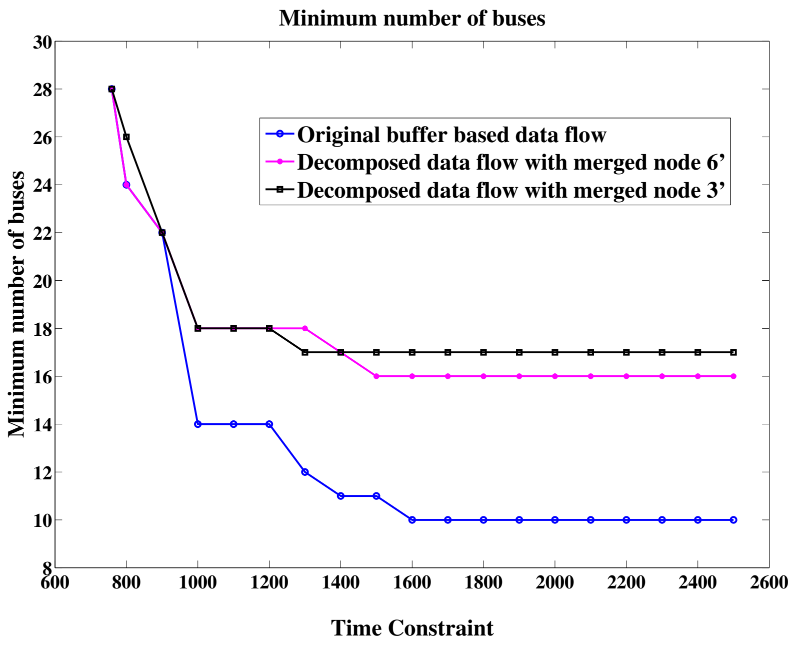
From (3.15) and Table 3.1, $L_{6'} < 199$. Since the lifetimes listed in Table 3.1 have their minimal values, $L_{6'} = 198$. In the same fashion, the latency $L_{3'}$ of the merged node $3'$ becomes 282. Fig. 3-21 shows the evaluation results when either node $6'$ or node $3'$ is used.

In Fig. 3-21, as the time constraint increases, the gap between the original buffer-based dataflow and the decomposed dataflow increases because the buffer controllers within a merged node do not involve any sharing. Thus, the minimum number of buffers/buses found in the decomposed buffer-based dataflow does not reach the minimum number found in the original buffer-based dataflow. However, by the decomposition, the computational complexity of finding the sharing cases satisfying a given cost is reduced.

To demonstrate the reduction of the computational complexity by decomposing a



(a) Minimum number of buffers in the decomposed dataflows



(b) Minimum number of buses in the decomposed dataflows

Figure 3-21: Evaluation results with the decomposed dataflow.

buffer-based dataflow, the sharing cases, in which the number of buffers is 9 and the buffer memory size is 27 with the time constraint of 1900, are found in the original buffer-based dataflow and the decomposed dataflow having the merged node $3'$ in Fig. 3-20. In the first step, the sharing cases satisfying the number of buffers = 9 with time constraint = 1900 are found in both the original buffer-based dataflow and the decomposed buffer-based dataflow. In our evaluation, there are 180792 cases found in the original buffer-based dataflow and 9720 cases found in the decomposed dataflow. From these cases, the sharing cases satisfying the buffer memory = 27 are found.

As the results of our simulation, 56128 buffer sharing cases are found in the original buffer-based dataflow, and 7920 buffer sharing cases are found in the decomposed buffer-based dataflow. The sharing cases satisfying the given cost are found in the decomposed buffer-based dataflow with the higher probability: i.e., $\frac{7920}{9720} = 0.81$ in the decomposed buffer-based dataflow, which is greater than $\frac{56128}{180792} = 0.31$ in the original buffer-based dataflow. Thus, decomposing the buffer-based dataflow indeed reduces the computational complexity of finding the sharing cases satisfying the given cost.

### 3.4.4 Energy Consumption

The sharing case consuming the minimum energy can be found by Algorithm 6 proposed in Section 3.3.5. In this section, we simulate the algorithm with the buffer controller parameters in Table 3.1 and the time constraint of 1900 cycles (1 cycle = 10 ns).

Fig. 3-22 shows the search ranges according to buffer and bus costs. In the figure,

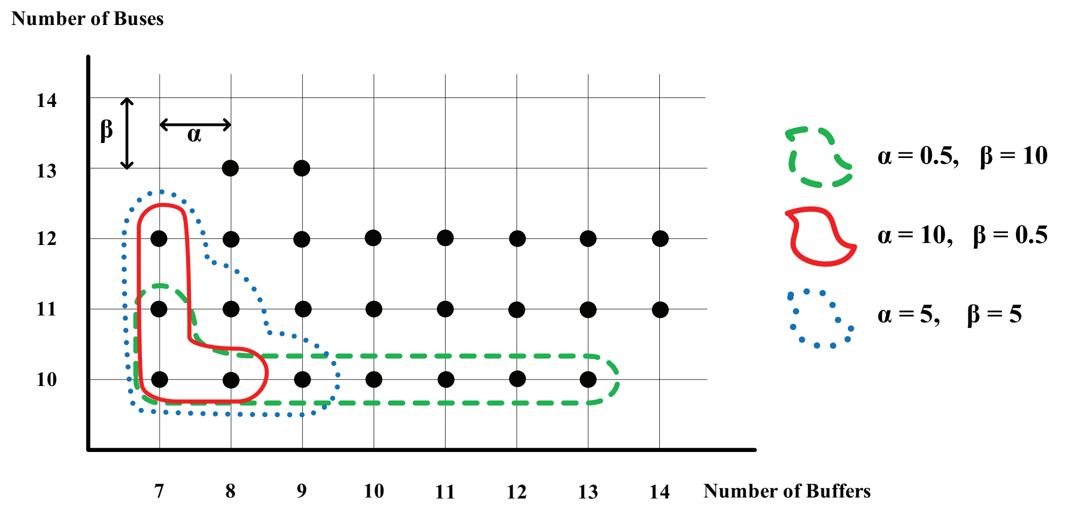
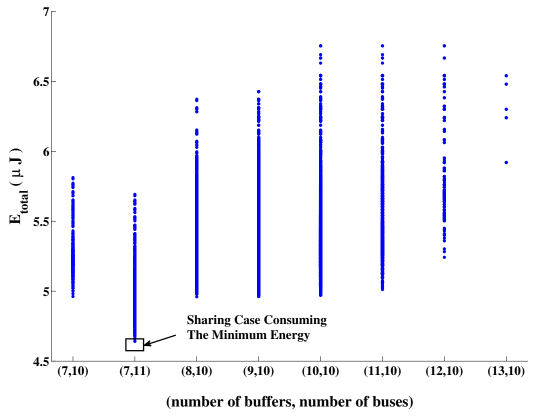


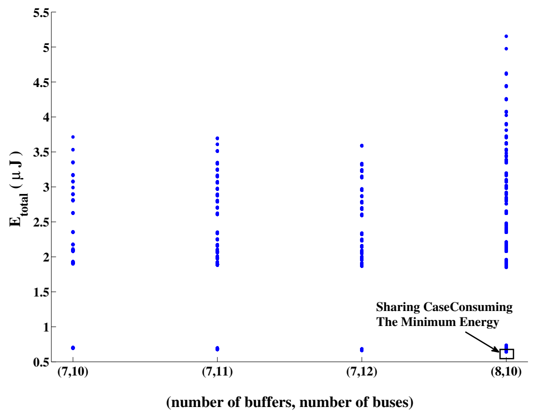
Figure 3-22: Search ranges for the minimum energy consumption.

$\alpha$, the unit of buffer cost, is mW/buffer and $\beta$, the unit of bus cost, is mW/bus. The regions enclosed with colored lines represent the search ranges according to $\alpha$ and $\beta$. The green (dashed) line denotes the search range when $\alpha = 0.5$ and $\beta = 10$. The red (solid) line represents the search range when $\alpha = 10$ and $\beta = 0.5$. The region within blue (dotted) line corresponds to the search range when $\alpha = 5$ and $\beta = 5$. Each point in the grid represents the sharing cases having the same numbers of buffers and buses. Even though only 25 points appear in the grid, the total number of sharing cases is 371,856. After the search range is found by $E_{est}$ and $E_{bound}$ in (3.13), the number of the sharing cases examined to find the minimum energy consumption is greatly reduced. For example, when $\alpha = 5$ and $\beta = 5$, the number of sharing cases within the search range becomes the largest value (i.e., 40,016) among the search ranges in Fig. 3-22. However, this value is only 10.8% of the total number of sharing cases.

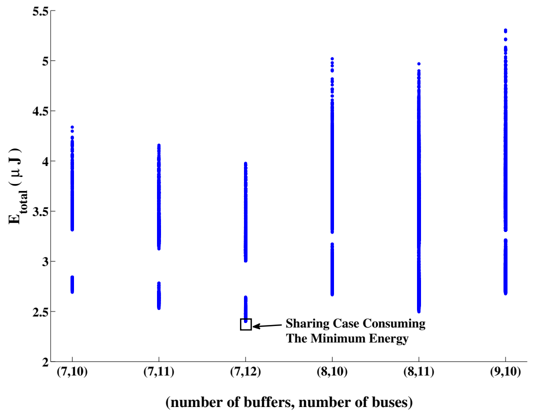
Fig. 3-23 shows the simulation results of finding the sharing case consuming the minimum energy within each search range of Fig. 3-22. In Fig. 3-23, the pair values



(a) $E_{total}$ when $\alpha = 0.5$ mW/buffer and $\beta = 10$ mW/bus



(b) $E_{total}$ when $\alpha = 10$ mW/buffer and $\beta = 0.5$ mW/bus



(c) $E_{total}$ when $\alpha = 5$ mW/buffer and $\beta = 5$ mW/bus

Figure 3-23: Simulation results of finding the sharing case consuming the minimum energy

(number of buffers, number of buses) in the x-axis correspond to the points within each search range in Fig. 3-22. The pair values of Fig. 3-23(a), Fig. 3-23(b) and Fig. 3-23(c) correspond to the points within the green, red and blue lines of Fig. 3-22, respectively. Since one point in Fig. 3-22 represents the sharing cases having the same numbers of buffers and buses, the $E_{total}$ values in one point vary because of the different sharing combination with the same numbers of buffers and buses. As shown in Fig. 3-23, depending on the values of $\alpha$ and $\beta$, the sharing case having the minimum $E_{total}$ (i.e., the sharing case consuming the minimum energy) may appear at different points in Fig. 3-22. Furthermore, no minimum $E_{total}$ appears at the sharing case having the minimum resources (i.e., the point (7,10)). Our proposed methodology thus configures the search range as shown in Fig. 3-22. However, the misestimated values of $\alpha$ and $\beta$ mislead to finding the sharing case consuming the minimum energy. For example, when $\alpha$ and $\beta$ are estimated as 0.5 mW/buffer and 10 mW/bus in the target platform used, respectively, the sharing case having the minimum $E_{total}$ appears at (7,11) in the grid of Fig. 3-22. However, if the actual $\alpha$ is 5 mW/buffer and actual $\beta$ is 5 mW/bus in the target platform, the actual sharing case having minimum $E_{total}$ appears at (7,12) in the grid of Fig. 3-22. Furthermore, the point (7,12) is not within the search range in the case where $\alpha = 0.5$ mW/buffer and $\beta = 10$ mW/bus.

# Chapter 4

# Buffer Controller Based Multiple Processing Element Utilization for Dataflow Synthesis

## 4.1 Introduction

Since programs running on processors have variable execution times, it is difficult to synchronize the data transfer between processing blocks mapped to different processors (alternatively, one is mapped to a processor and the other is implemented as a hardware logic). In order to synchronize the data transfers, we use a buffer-based dataflow which inserts buffers between processing blocks in a given dataflow. The buffer-based dataflow is globally synchronized by a global controller 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, by isolating the data transfer from the functional execution of a processing block, 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 hardwares). The proposed methodology translates the data transfer activities of processing blocks into the primitive templates running on processors. With the primitive templates and estimated execution times, the methodology creates a mapped partition. From the mapped partition and the library for target-specific bus operations, template code written in C language is automatically generated. In order to prevent wrong operations due to the variable execution times, we also devise a processor initiation scheme which notifies the ends of functional executions and data transfers to the global controller. Thus, even if execution times change with large variations, the execution ordering is preserved.

This chapter is organized as follows: Section 4.2 describes the approach and objectives of this research. In Section 4.3, we characterize the mapping technique that utilizes the buffer-based dataflow. Section 4.4 discusses the resource utilization of mapping and the synchronization of estimated execution times. This section also proposes the mapping algorithm and template code generation. In Section 4.5, we evaluate the proposed mapping algorithm with the SystemC model and demonstrate that the generated template code works on Xilinx ISE 10.1.

## 4.2 Our Approach and Objectives

When a dataflow is synthesized in a target platform having multi-core processors and programmable logics, it is critical to synchronize data transfers between processors/between a processor and a hardware logic. Even though all processing elements are globally synchronized, synchronizing data transfers is difficult because the programs running on processors have variable execution times.

In order to synchronize the data transfers at the level of a dataflow graph, we use the buffer-based dataflow for mapping processing blocks to processors. Our methodology creates a mapped partition from the buffer-based dataflow representing an application, the resource constraint of a target platform and estimated times for functional executions and data transfers. The data transfers of processing blocks mapped to processors are realized as target-dependent primitive templates. The primitive templates are the programs parameterized for data transfers in a buffer-based dataflow. Fig. 4-1 shows the overall flow of the proposed mapping methodology. In Section 4.3, the mapping of processing blocks is characterized with primitive templates. In Fig. 4-1, the mapped partition has the global timing information to synchronize data transfers between processors (or between a processor and a hardware). When the mapped partition is synthesized in a target platform, the synchronization of data transfers is realized through a global controller and buffer controllers as illustrated in Fig. 4-2.

In Fig. 4-2, $f(i)$ is mapped to a processor, and $f(j)$ is implemented as a hardware. The data transfer from $f(i)$ to $f(j)$ is done through $BC_{i,j}$. In order to synchronize data transfers between each processing block (i.e., $f(i)$ and $f(j)$) and $BC_{i,j}$, the global

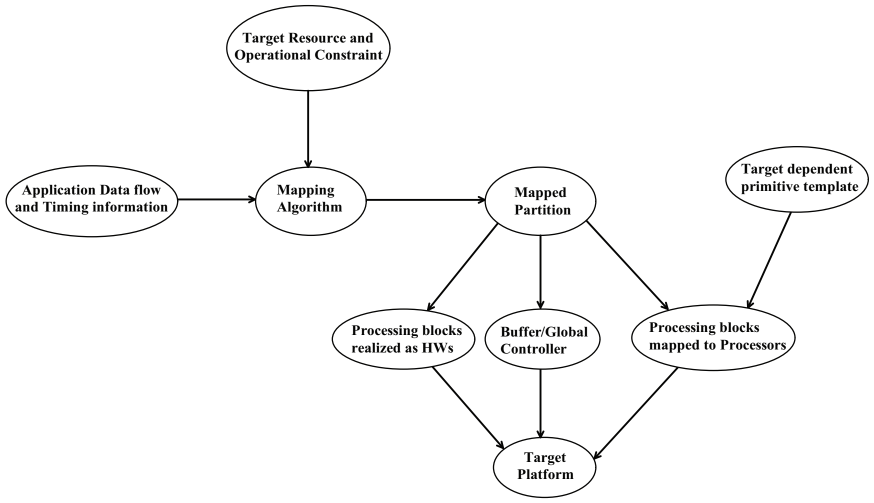

Target Resource and Operational Constraint
Application Data flow and Timing information
Mapping Algorithm
Mapped Partition
Target dependent primitive template
Processing blocks realized as HWs
Buffer/Global Controller
Processing blocks mapped to Processors
Target Platform

Figure 4-1: Overall flow of proposed mapping methodology.

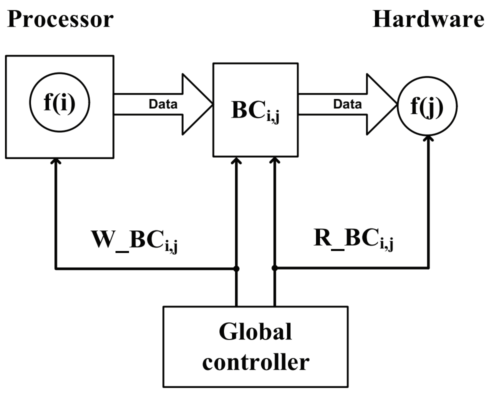


Figure 4-2: Synchronization of data transfers through a global controller and a buffer controller.

controller generates $W\_BC_{i,j}$ and $R\_BC_{i,j}$. $W\_BC_{i,j}$ is the signal which enables $f(i)$ to write data to $BC_{i,j}$, whereas $R\_BC_{i,j}$ is the signal which initiates $f(j)$ to read data from $BC_{i,j}$.

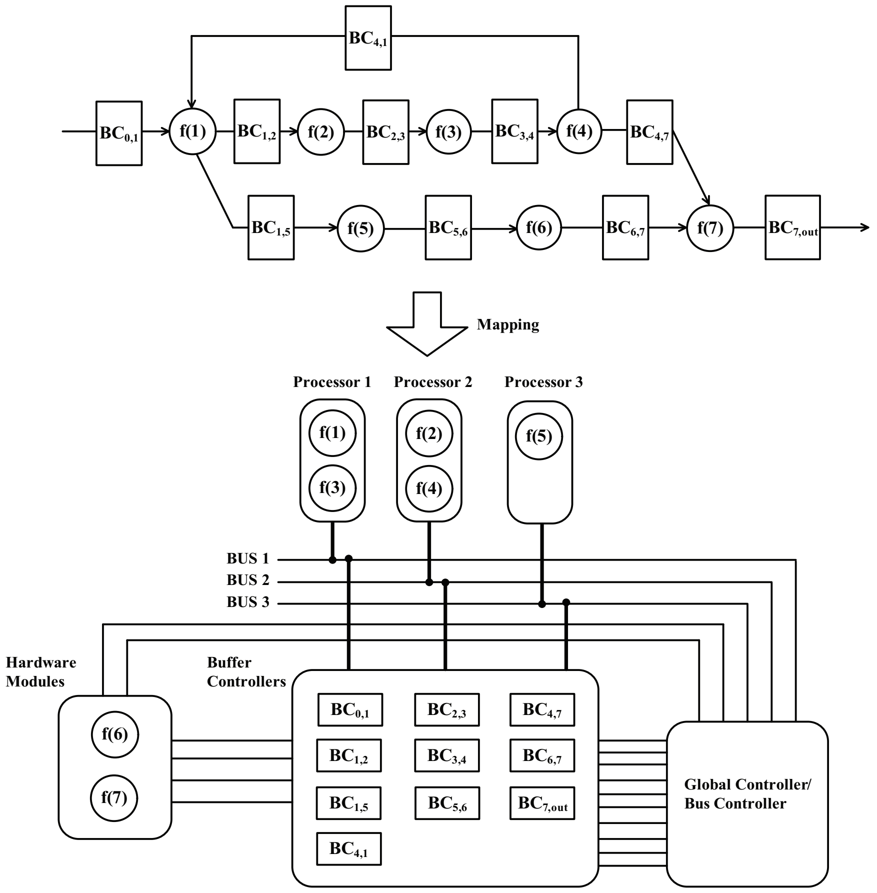


Figure 4-3: Mapping a buffer-based dataflow to a target platform.

## 4.3 Mapping Characterization

This section characterizes the mapping of processing blocks in a buffer-based dataflow onto a processor architecture. Fig. 4-3 illustrates that a buffer-based dataflow is mapped to a target platform having multi-core processors and hardware logics. In Fig. 4-3, the processing blocks mapped to processors are realized as the programs running on processors. The other processing blocks are implemented as hardware logics. Buffer controllers are located outside the processors. The interconnects between processors and buffer controllers are the buses provided by the target platform used. The global controller synchronizes the data transfers between the processing blocks which are either mapped to processors or realized as hardware logics and buffer controllers. Therefore, the mapped processing blocks need the entities for the data transfers with buffer controllers outside the processors. Section 4.3.1 defines primitive templates as the entities for the data transfers with buffer controllers. In Section 4.3.2, some design parameters are introduced to generate the primitive templates. Section 4.3.3 investigates how the design parameters affect mapping.

### 4.3.1 Entities for Data Transfers of Processing Blocks Mapped to Processors

When a processing block is realized as a hardware logic, its functional execution and data transfer may work at the same time. However, if a processing block is mapped to a processor architecture, not only its functional execution is the program running on a processor, but also its data transfers sequentially run on the target processor as a program. The primitive templates, $RCV$ and $SND$ are devised to realize the data transfers of the processing blocks mapped to a processor. Since buffer controllers are outside a processor, $RCV$ and $SND$ use the bus provided by a target processor to transfer data between the processing block mapped to a processor and a buffer controller. $RCV$ reads data from a buffer controller, whereas $SND$ writes data to a buffer controller. In order to notate the direction of data transfer, $RCV_{i,j}$ and $SND_{i,j}$ are used. The subscript $i$ represents the source of data, and $j$ indicates the destination of data. The operations of $RCV_{i,j}$ and $SND_{i,j}$ are described as the pseudo-code in Fig. 4-4.

```
function $RCV_{i,j}$ ( $size\_of\_M_{i,j}$ ) {

    while( $R\_BC_{i,j}$ == 0)
        wait();

    // Read data from $BC_{i,j}$ through a bus interface.
     $Input\_of\_p[f(i)] =$ read (bus, $size\_of\_M_{i,j}$);

}
end function
```

**(a) Functional Description of $RCV_{i,j}$**

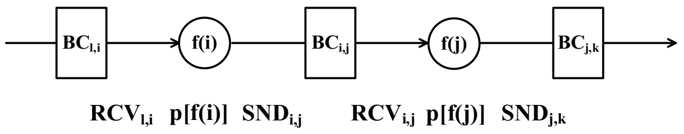
```
function $SND_{i,j}$ ($Output\_of\_p[f(i)]$, $size\_of\_M_{i,j}$ ) {

    while( $W\_BC_{i,j}$ == 0)
        wait();

    // Write data to $BC_{i,j}$ through a bus interface.
     write(bus, $Output\_of\_p[f(i)]$, $size\_of\_M_{i,j}$);

}
end function
```

**(b) Functional Description of $SND_{i,j}$**

Figure 4-4: Pseudo codes of $SND_{i,j}$ and $RCV_{i,j}$.

In Fig. 4-4, $RCV_{i,j}$ corresponds to the fan-in operation of $f(j)$, and $SND_{i,j}$ represents the fan-out operation of $f(i)$. $R\_BC_{i,j}$ is the signal initiating $RCV_{i,j}$ to read data from $BC_{i,j}$. $W\_BC_{i,j}$ is also the signal initiating $SND_{i,j}$ to write data to $BC_{i,j}$. Both $R\_BC_{i,j}$ and $W\_BC_{i,j}$ are generated by a global controller. The "read" and "write" functions indicate target-dependent bus operations. The "bus" in "read" and "write" functions represents the descriptor for reading/writing data from/to the corresponding buffer controller through a bus interface. The transferred data through $RCV$ and $SND$ are bound to the arguments of the function which corresponds to the mapped processing block. The template codes running on a target processor are generated from the argument passing (binding) between primitive templates (i.e.,

$SND$, $RCV$) and functions, as shown in Fig. 4-5.



BC$_{l,i}$
f(i)
BC$_{i,j}$
f(j)
BC$_{j,k}$
RCV$_{l,i}$ p[f(i)] SND$_{i,j}$
RCV$_{i,j}$ p[f(j)] SND$_{j,k}$

```
$Input\_of\_p[f(i)] = RCV_{l,i}$ ( $size\_of\_M_{l,i}$ );

function p[f(i)] ($Input\_of\_p[f(i)]$ ) {
    while( $END\_R\_BC_{l,i}$ == 0)
          wait();

  Do the functional execution of p[f(i)];
  Store $Output\_of\_p[f(i)]$;
}
end function

$SND_{i,j}$ ($Output\_of\_p[f(i)]$, $size\_of\_M_{i,j}$ );

$Input\_of\_p[f(j)] = RCV_{i,j}$ ( $size\_of\_M_{i,j}$ );

function p[f(j)] ($Input\_of\_p[f(j)]$){
  while( $END\_R\_BC_{i,j}$ == 0)
          wait();

  Do the functional execution of p[f(j)];
  Store $Output\_of\_p[f(j)]$;
}
end function

$SND_{j,k}$ ($Output\_of\_p[f(j)]$, $size\_of\_M_{j,k}$);
```

Figure 4-5: Pseudo template codes of a given buffer-based dataflow.

In Fig. 4-5, $p[f(i)]$ and $p[f(j)]$ are the programs corresponding to $f(i)$ and $f(j)$, respectively. $END\_R\_BC_{l,i}$ ($END\_R\_BC_{i,j}$) is the signal to notify $p[f(i)]$ ($p[f(j)]$) that $RCV_{l,i}$ ($RCV_{i,j}$) is completed. Both $END\_R\_BC_{l,i}$ and $END\_R\_BC_{i,j}$ are generated by a global controller. Since the relation of functional arguments represents the data dependency between programs, it determines the sequence (ordering) of programs. Therefore, data transfers and functional executions are fully sequentialized. For example, function $p[f(i)]$ starts when $RCV_{i,j}$ ends and returns its value to $Input\_of\_p[f(i)]$. However, in case $f(i)$ and $f(j)$ are mapped to different processors, $SND_{i,j}$ and $RCV_{i,j}$ simultaneously access $BC_{i,j}$ via different buses under the following condition:

$$T(W\_BC_{i,j}) < T(R\_BC_{i,j}) < T(END\_W\_BC_{i,j})$$

where $T(W\_BC_{i,j})$ is the start time of writing data to $BC_{i,j}$, $T(R\_BC_{i,j})$ is the start time of reading data from $BC_{i,j}$ and $T(END\_W\_BC_{i,j})$ is the end time of writing data to $BC_{i,j}$. 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, the data transfer is realized as $JOINT$ which merges $SND$ and $RCV$. Fig. 4-6 shows the pseudo-code of $JOINT_{i,j}$.

```
function $\boldsymbol{JOINT_{i,j}}$ ($Output\_of\_p[f(i)]$, $size\_of\_M_{i,j}$ )
{
    // Data transfer from the output of f(i) to the input of f(j).
    memcpy ( $Input\_of\_p[f(j)]$ , $Output\_of\_p[f(i)]$, $size\_of\_M_{i,j}$);
}
end function
```

Figure 4-6: Pseudo code of $JOINT_{i,j}$.

In Fig. 4-6, compared with $RCV$ and $SND$ operations, there is no signal from a global controller because $JOINT$ does not use any bus for data transfer. Therefore, when pairs of $SND$ and $RCV$ are replaced with $JOINT$s, the total number of bus accesses is reduced by (2 $\times$ number of $JOINT$s). In addition, since $JOINT$s replace buffer controllers, the total number of buffer controllers is reduced by the number of $JOINT$s.

### 4.3.2 Design Parameters For Primitive Template Generation

In Section 4.3.1, the signals, $R\_BC$, $W\_BC$ and $END\_R\_BC$, are used to represent the data transfers between the processing block mapped to a processor and a buffer controller. However, the signals do not represent the data transfers of the processing block implemented as a hardware logic because hardware logics may be able to transfer data while they are performing their functions. In order to represent the data transfers of hardware logics, the primary parameters introduced in Section 2.1 are used to define *start* and *stop* signals as follows:

$$\begin{aligned} start\_write_{i,j} &= L_i + nw_{ij} + start_i, & (4.1)\\ stop\_write_{i,j} &= start\_write_{i,j} + M_{ij}, & (4.2)\\ start\_read_{i,j} &= start\_write_{i,j} + max(nr_{ij}, D_{ij}), & (4.3)\\ stop\_read_{i,j} &= start\_read_{i,j} + M_{ij}, & (4.4)\end{aligned}$$

where $start\_write_{i,j}$ represents the start time of writing data to $BC_{i,j}$, $start\_read_{i,j}$ denotes the start time of reading data from $BC_{i,j}$, $stop\_write_{i,j}$ corresponds to the end time of writing data to $BC_{i,j}$ and $stop\_read_{i,j}$ is the end time of reading data from $BC_{i,j}$. In (4.1), $start_i$ is the absolute time that $f(i)$ begins reading data from the previous buffer controller through its fan-in port.

In the buffer-based dataflow where all processing blocks are realized as hardware logics, $R\_BC$, $W\_BC$ and $END\_R\_BC$ correspond to $start\_read$, $start\_write$ and $stop\_read$, respectively. In case $f(i)$ is mapped to a processor, $L_i$ is replaced with the

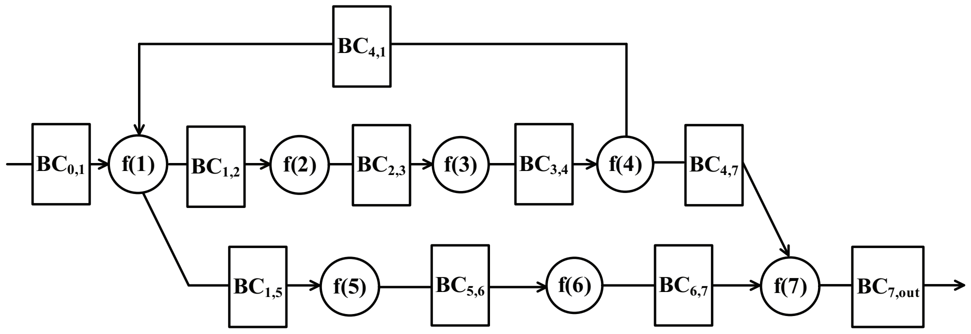


Figure 4-7: A buffer-based dataflow

execution time of the program $p[f(i)]$. In addition, $start\_write_{i,j}$ is delayed to (end of program $p[f(i)] + 1$). Therefore, (4.1) changes as follows:

$$start\_write_{i,j} = T_i + start_i + 1 \tag{4.5}$$

where $T_i$ is the execution time of the program $p[f(i)]$ and $start_i$ is the start time of $p[f(i)]$. Since fan-in operations are also sequentialized, $start_i$ corresponds to (the end time of the last $RCV$ preceding $p[f(i)] + 1$). With the design parameters in (4.1)–(4.5), Section 4.3.3 compares the case in which processing blocks are implemented as hardwares with the case where processing blocks are mapped to a processor architecture.

## 4.3.3 Mapping Effects

In this section, we investigate mapping effects by comparing the case that all processing blocks are hardwares. Fig. 4-7 is a buffer-based dataflow with parameterized buffer controllers. When $f(1)$ and $f(2)$ are implemented as hardware logics, the operational dependency from the input of $f(1)$ to the output of $f(2)$ is represented as

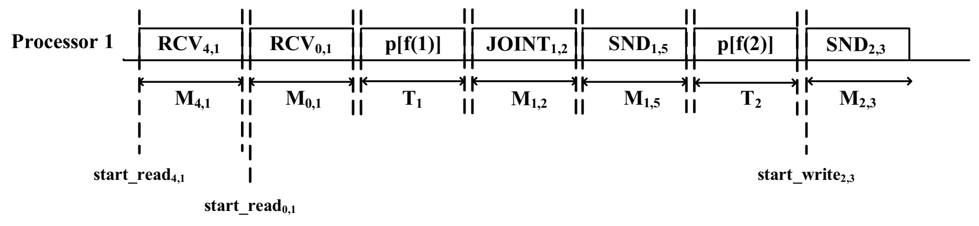


Figure 4-8: Execution timing when $f(1)$ and $f(2)$ of Fig. 4-7 are mapped to processor 1.

follows:

$$\begin{aligned} start\_write_{2,3} &> start\_read_{1,2} + L_2, \\ start\_read_{1,2} &> start\_write_{1,2}, \\ start\_write_{1,2} &> max(start\_read_{4,1}, start\_read_{0,1}) + L_1. \end{aligned} \tag{4.6}$$

From (4.6), the minimum $start\_write_{2,3}$ for the hardware realization is given by

$$\begin{aligned} &min(start\_write_{2,3}) \text{ for hardware} = L_2 + min(start\_read_{1,2}) + 1 \\ &= max(start\_read_{4,1}, start\_read_{0,1}) + L_1 + L_2 + 3. \end{aligned} \tag{4.7}$$

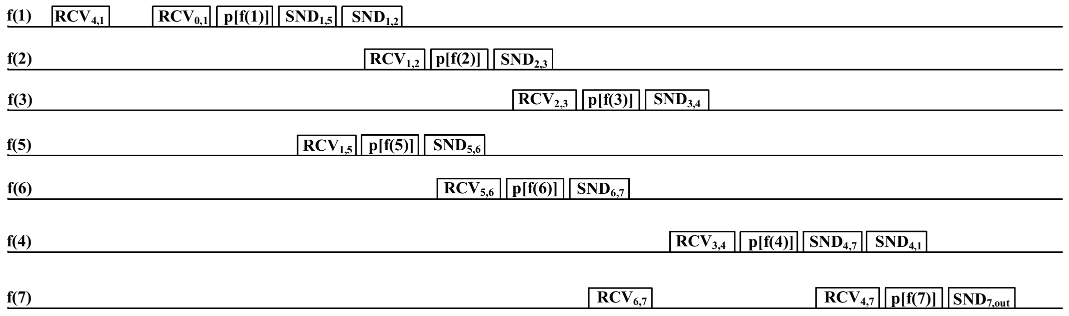
Fig. 4-8 shows the execution timing from the input of $f(1)$ to the output of $f(2)$ when $f(1)$ and $f(2)$ of Fig. 4-7 are mapped to processor 1. In Fig. 4-8, since two successive processing blocks, $f(1)$ and $f(2)$, are mapped to the same processor, $JOINT_{1,2}$ is used for the data transfer between $f(1)$ and $f(2)$. The execution time from $RCV_{4,1}$ to $SND_{2,3}$ is the minimum value because all data transfers and functional executions start as soon as their previous programs are completed. From the execution timing, the minimum $start\_write_{2,3}$ for processor 1 is given by:

$$\begin{aligned} min(start\_write_{2,3}) \text{ for processor } &= start\_read_{4,1} + T_1 + T_2 \\ &\quad + M_{4,1} + M_{0,1} + M_{1,5} + M_{1,2} + 6. \end{aligned} \quad (4.8)$$

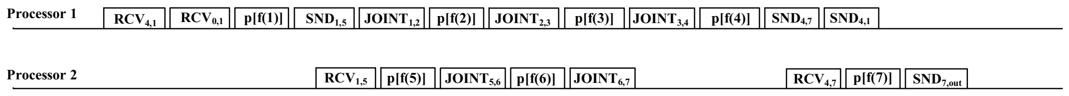
Compared with (4.7), all previous execution times (including data transfers) reflect to $start\_write_{2,3}$. Therefore, for the processing blocks mapped to processors, *start* signals *must be timely ordered* to maintain the correct execution of programs running on processors. If the execution time of $RCV_{4,1}$ takes longer than $M_{4,1}$, processor 1 receives $start\_read_{0,1}$ before $RCV_{4,1}$ is completed. The timing mismatch may lead to the wrong operations of the programs running on processor 1. In Section 4.4, we discuss the synchronization issue and propose a solution.

## 4.4 Resource Utilization and Synchronization

This section discusses the resource utilization of a target platform to which processing blocks in a buffer-based dataflow are mapped. Since the target platform consists of processors and hardware logics such as Xilinx FPGAs, we investigate the synchronization of data transfers between processors and between a processor and a hardware logic. Due to the variation of execution times in a processor, Section 4.4.1 provides the mapping based on *maximally estimated execution times.* Section 4.4.2 discusses the condition to reduce the number of processor buses. In Section 4.4.3, we propose a processor initiation scheme for the correct operation when actual execution times



(a) Execution timing for the mapping of processing blocks in Fig. 4-7



(b) Execution timing when processing blocks of Fig. 4-9(a) are mapped to processors

Figure 4-9: Execution timing representation to map processing blocks to processors.

take longer than maximally estimated execution times. Section 4.4.4 discusses the timing mismatch problem in the case of processor-hardware coexistence and provides its solution. Based on the discussion from Section 4.4.1 to Section 4.4.4, we propose a mapping algorithm and a template code generation method in Sections 4.4.5 and 4.4.6.

### 4.4.1 Processor Utilization of Mapping

In order to map processing blocks to processors, the execution timing of processing blocks is represented by using primitive templates such as $SND$ and $RCV$. When a buffer-based dataflow is given as Fig. 4-7, Fig. 4-9 shows the execution timing representation of processing blocks.

In Fig. 4-9(a), a straightforward way of mapping may be to map each processing block to an individual processor. In this case, the number of processors is the same as the number of processing blocks. Thus, seven processors are required for mapping processing blocks. However, if the execution times of processing blocks are *non-overlapped*, the processing blocks 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 execution times satisfy the following non-overlapped condition [$EXE_i$: execution times of f(i), $EXE_j$: execution times of f(j)]:

$$\begin{aligned} &max(end(EXE_i), end(EXE_j)) - min(start(EXE_i), start(EXE_j)) \\ &\qquad > (end(EXE_i) - start(EXE_i)) + (end(EXE_j) - start(EXE_j)) \end{aligned} \tag{4.9}$$

where $start(EXE)$ and $end(EXE)$ correspond to the start and end times of $SND$, $RCV$ and functional execution, respectively. Equation (4.9) represents that two execution times are non-overlapped if the difference between the maximum $end(EXE)$ and the minimum $start(EXE)$ of two execution times is larger than the summation of two execution times. According to the execution type of processing block $f(i)$ (i.e., $SND$, $RCV$ and $p[f(i)]$), $start(EXE)$ and $end(EXE)$ of equation (4.9) are translated to

$$end(EXE_i) = \begin{cases} stop\_write, & \text{if } EXE_i = \text{SND of f(i)}, \\ stop\_read, & \text{if } EXE_i = \text{RCV of f(i)}, \\ \text{end of p[f(i)]}, & \text{if } EXE_i = \text{p[f(i)]}, \end{cases}$$

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

The number of processors is further reduced by using $JOINT$. In Fig. 4-9(a), the execution times of $f(1)$ and $f(2)$ are only overlapped in $SND_{1,2}$ and $RCV_{1,2}$. By replacing $SND_{1,2}$ and $RCV_{1,2}$ with $JOINT_{1,2}$, $f(1)$ and $f(2)$ are mapped to the same processor. In the same way, $f(1)$–$f(4)$ are mapped to processor 1 and $f(5)$–$f(7)$ are mapped to processor 2 as shown in Fig. 4-9(b). Therefore, the number of processors required for mapping is the same as the number of the feed-forward paths because the buffer controllers in a single feed-forward path are replaced with $JOINT$ operations when consecutive processing blocks are mapped to processors.

Even though $JOINT$ is applied to reduce the number of processors, the mapping result with $JOINT$s may not satisfy the number of processors provided by the target platform used. If the number of processors in the target platform is 1, the execution timing of Fig. 4-9(b) is not directly mapped to a 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 $f(4)$.

When $f(i)$ is mapped to a processor, its functional execution is realized as program $p[f(i)]$. If $p[f(i)]$ is preempted by the task having the higher priority, the execution time of $p[f(i)]$ is delayed. The delayed execution of $p[f(i)]$ leads to the wrong $SND$ operation as illustrated in Fig. 4-10.

In Fig. 4-10, processor 1 receives $start\_write_{i,j}$ while $p[f(i)]$ is running. Processor

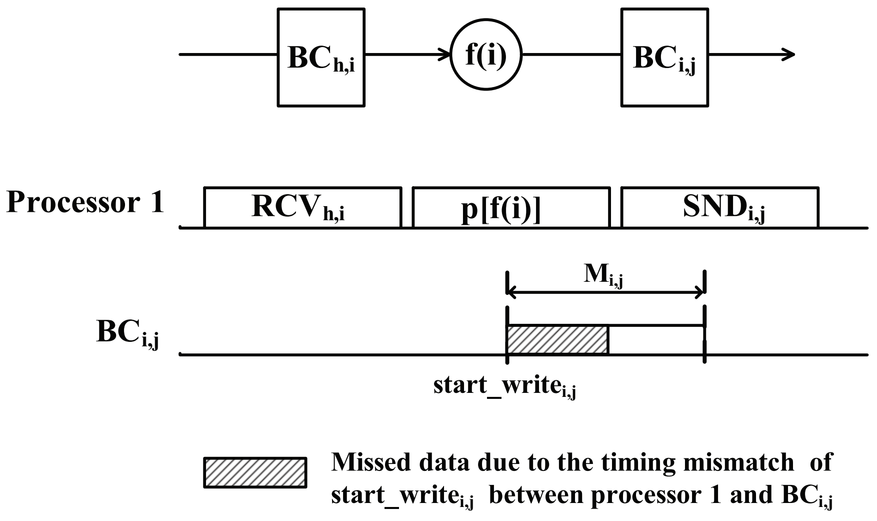


Figure 4-10: Processor 1 receives $start\_write_{i,j}$ during the execution of $p[f(i)]$.

1 starts running $SND_{i,j}$ when $p[f(i)]$ is finished. However, $BC_{i,j}$ starts receiving data from processor 1 as soon as it gets $start\_write_{i,j}$ from a global controller. Therefore, $BC_{i,j}$ misses data due to the timing mismatch of $start\_write_{i,j}$ between processor 1 and $BC_{i,j}$.

In order to determine the execution time of $p[f(i)]$, $T_i(max)$ is introduced. $T_i(max)$ is estimated as the longest time to complete the execution of $p[f(i)]$. If $T_i(max)$ is mispredicted, it affects the mapping result. Table 4.1 lists $T_i(max)$ and $T_i(actual)$ when $f(1) - f(7)$ of Fig. 4-9(a) are mapped to processors. Fig. 4-11 shows the mapping results of Fig. 4-9(a) when $T_i(max)$ and $T_i(actual)$ in Table 4.1 are applied.

In Fig. 4-11, in case $T_i(max)$ of Table 4.1 is applied, processor 3 is used to map $f(7)$. Compared with the case where $T_i(actual)$ is applied, one more processor is used for mapping. Thus, the wrong estimation may waste the resource of a target platform. In addition, since $RCV$, $SND$ and $JOINT$ are also programs running on processors, the execution times of $RCV$, $SND$ and $JOINT$ must be estimated. In

Table 4.1: Estimated $T_i(max)$ and $T_i(actual)$ of the processing blocks in Fig. 4-9(a)

| Processing Block $f(i)$ | $T_i(max)$ [cycles] | $T_i(actual)$ [cycles] |
|---|---|---|
| $f(1)$ | 100 | 300 |
| $f(2)$ | 250 | 200 |
| $f(3)$ | 280 | 200 |
| $f(4)$ | 800 | 150 |
| $f(5)$ | 1250 | 200 |
| $f(6)$ | 300 | 200 |
| $f(7)$ | 700 | 400 |

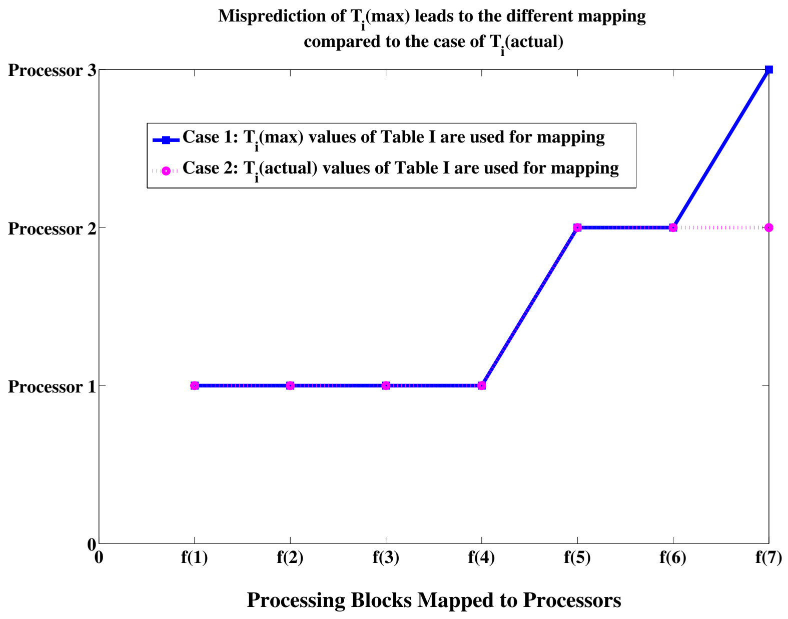
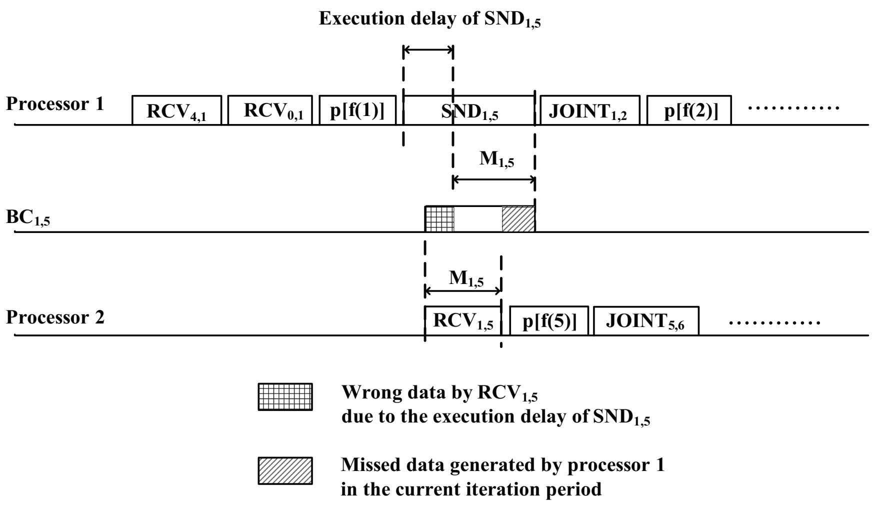


Figure 4-11: Mapping results of Fig. 4-9(a) when $T_i(max)$ and $T_i(actual)$ in Table 4.1 is applied.

Fig. 4-9(b), if the execution of $SND_{1,5}$ in processor 1 takes longer than $M_{1,5}$, $RCV_{1,5}$ of processor 2 *partially* receives the data from $BC_{1,5}$. Therefore, processor 2 does not execute its functions (i.e., $p[f(5)]$, $p[f(6)]$ and $p[f(7)]$) correctly. Fig. 4-12 illustrates the case.

In Fig. 4-12, due to the execution delay of $SND_{1,5}$, even though $SND_{1,5}$ of processor 1 starts before $RCV_{1,5}$ of processor 2 begins, $SND_{1,5}$ is finished after $RCV_{1,5}$



Execution delay of SND1,5
Processor 1
RCV4,1
RCV0,1
p[f(1)]
SND1,5
JOINT1,2
p[f(2)]
M1,5
BC1,5
M1,5
Processor 2
RCV1,5
p[f(5)]
JOINT5,6
Wrong data by RCV1,5 due to the execution delay of SND1,5
Missed data generated by processor 1 in the current iteration period

Figure 4-12: When the execution time of $SND_{1,5}$ takes longer than $M_{1,5}$ in Fig. 4-9(b).

is completed. Therefore, processor 2 receives wrong data from $BC_{1,5}$ and misses the data which processor 1 generates in the current iteration period. For the correct operation, $RCV_{1,5}$ starts running when $SND_{1,5}$ finishes writing all data to $BC_{1,5}$. In order to determine the end of $SND_{1,5}$, the execution time of $SND_{1,5}$ is estimated as its maximum value, $T_{max}(SND_{1,5})$. Thus, $start\_read_{1,5}$ for $RCV_{1,5}$ is changed as follows:

$$start\_read_{1,5} = stop\_write_{1,5} + 1 = start\_write_{1,5} + T_{max}(SND_{1,5}) + 1,$$
$$T_{max}(SND_{1,5}) > M_{1,5}.$$

Fig. 4-13 illustrates the case where $start\_read_{1,5}$ with the estimated $T_{max}(SND_{1,5})$ is applied to Fig. 4-12. In Fig. 4-13, the actual data transfer by $SND_{1,5}$ is done from $start\_write_{1,5}$ to $(start\_write_{1,5} + T_{max}(SND_{1,5}))$. Since $RCV_{1,5}$ starts running after $SND_{1,5}$ is completed (i.e., $start\_read_{1,5} = stop\_write_{1,5} + 1$), processor 2 always

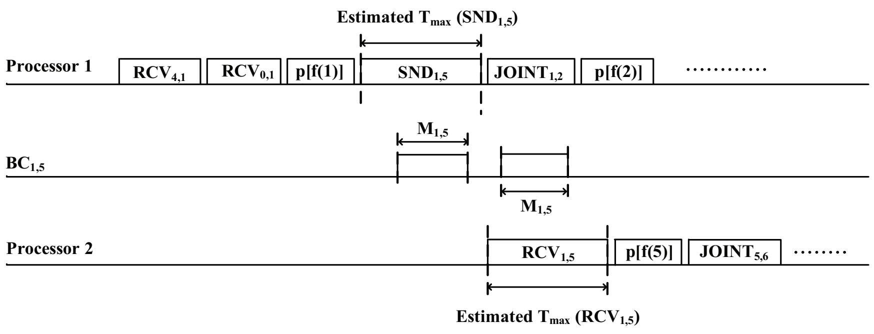


Figure 4-13: When the estimated $T_{max}(SND_{1,5})$ is applied to Fig. 4-12.

receives the valid data which processor 1 generates in the current iteration period. In the same way, the execution time of $RCV_{1,5}$ is also estimated as $T_{max}(RCV_{1,5})$ to receive the valid data from $BC_{1,5}$. For the synchronization of data transfers, the execution times of $SND$s, $RCV$s and $JOINT$s are estimated as their maximum values. In addition, the estimated execution times are reflected to *start* and *stop* signals as follows:

$$\begin{aligned}
&stop\_write_{i,j} = start\_write_{i,j} + T_{max}(SND_{i,j}),\\
&start\_read_{i,j} = stop\_write_{i,j} + 1,\\
&stop\_read_{i,j} = start\_read_{i,j} + T_{max}(RCV_{i,j}),\\
&stop\_joint_{i,j} = start\_joint_{i,j} + T_{max}(JOINT_{i,j}),\\
&T_{max}(SND_{i,j}),\ T_{max}(RCV_{i,j}),\ T_{max}(JOINT_{i,j}) > M_{i,j},
\end{aligned} \tag{4.10}$$

where $start\_write_{i,j}$ and $start\_read_{i,j}$ are generated by a global controller and other signals are not generated but estimated to determine $start\_write$ and $start\_read$.

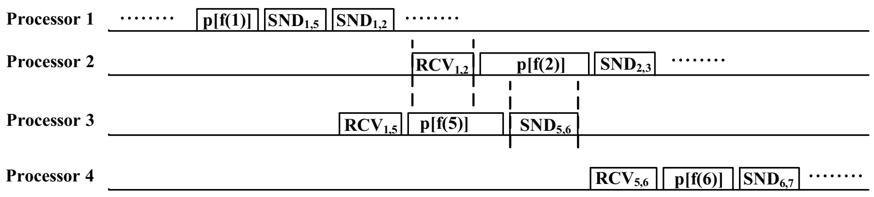


Figure 4-14: Estimated execution times when processing blocks of Fig. 4-7 are mapped to 4 processors.

If $SND_{i,k}$ starts when $JOINT_{i,j}$ is completed, $start\_write_{i,k}$ is $(stop\_joint_{i,j} + 1)$. However, in case actual execution take longer than estimated, the signals of (4.10) fail to synchronize data transfers. In Section 4.4.3, we propose a scheme to synchronize data transfers by revising the signals of (4.10).

### 4.4.2 Bus Utilization of Mapping

Our basic assumption 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. Fig. 4-14 shows the execution timing when processing blocks of Fig. 4-7 are mapped to 4 processors.

In Fig. 4-14, 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. When processors $i$ and $j$ share the same bus, their bus operations satisfy the following ($BUS_i$: execution times of a bus operation in processor $i$, $BUS_j$: execution times of a bus operation in processor $j$):

$$\begin{aligned} & max(end(BUS_i), end(BUS_j)) - min(start(BUS_i), start(BUS_j)) \\ & \qquad > (end(BUS_i) - start(BUS_i)) + (end(BUS_j) - start(BUS_j)) \end{aligned} \tag{4.11}$$

where $start(BUS)$ and $end(BUS)$ correspond to the start and end times of $SND$ and $RCV$, respectively. Equation (4.11) represents that the execution times of two bus operations in processors $i$ and $j$ are non-overlapped if the difference between the maximum $end(BUS)$ and the minimum $start(BUS)$ of two bus operations is larger than the summation of the execution time of the two bus operations. According to the data transfer types of processor $i$ (i.e., $SND$ and $RCV$), $start(BUS)$ and $end(BUS)$ in (4.11) are translated to

$$end(BUS_i) = \begin{cases} stop\_write, & \text{if } BUS_i \text{ is a } SND \text{ in processor } i, \\ stop\_read, & \text{if } BUS_i \text{ is a } RCV \text{ in processor } i, \end{cases}$$

$$start(BUS_i) = \begin{cases} start\_write, & \text{if } BUS_i \text{ is a } SND \text{ in processor } i, \\ start\_read, & \text{if } BUS_i \text{ is a } RCV \text{ in processor } i. \end{cases}$$

However, the execution times of Fig. 4-14 are estimated values for mapping. If actual execution times are not within the range of estimated values, the bus sharing among processors leads to wrong results. In Fig. 4-14, when the actual execution of $RCV_{1,5}$

takes longer than estimated, $RCV_{1,5}$ is overlapped with $RCV_{1,2}$. In this case, $RCV_{1,5}$ does not receive the correct data from $BC_{1,5}$ because $RCV_{1,2}$ occupies the shared bus when processor 2 receives $start\_read_{1,2}$ from the global controller.

### 4.4.3 Processor Initiation Scheme and Global Controller

When actual execution times take longer than estimated times, data transfers are not synchronized so that entire dataflow operations may produce wrong results. In order to prevent the misprediction of execution times, we propose a *processor initiation scheme* notifying the end of a program to a global controller. The processor initiation scheme is applied to the design of the mapped partition, which is created from the estimated execution times.

For the correct execution of $SND_{i,j}$ following $p[f(i)]$, at the end of the functional execution $p[f(i)]$, the processor which runs $p[f(i)]$ generates $initiate\_start\_write_{i,j}$ signal to inform a global controller that $p[f(i)]$ is completed. As soon as the global controller receives $initiate\_start\_write_{i,j}$, it sends $start\_write_{i,j}$ to both the processor and the buffer controller $BC_{i,j}$. Thus, the global controller needs the input port for receiving $initiate\_start\_write_{i,j}$ from the processor. When the processor initiation scheme is applied, $start\_write_{i,j}$ is changed to

$$start\_write_{i,j} = initiate\_start\_write_{i,j} + 1 = start\_of\_p[f(i)] + T_i(actual) + 1$$

where $start\_of\_p[f(i)]$ is the start time of the program $p[f(i)]$, and it is determined by the last fan-in operation (i.e., $RCV$ or $JOINT$) prior to $p[f(i)]$. In the proposed processor initiation scheme, *stop* signals are used to determine *start* signals for data transfers. The *stop* signals for $SND$, $RCV$ and $JOINT$ are generated by processors as follows:

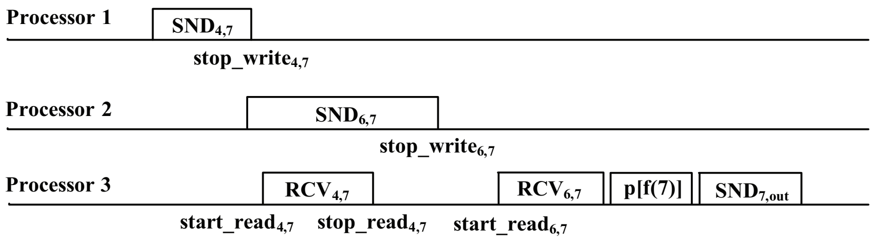
$$\begin{aligned} stop\_write_{i,j} &= start\_write_{i,j} + T_{actual}(SND_{i,j}), \\ stop\_read_{i,j} &= start\_read_{i,j} + T_{actual}(RCV_{i,j}), \\ stop\_joint_{i,j} &= start\_joint_{i,j} + T_{actual}(JOINT_{i,j}), \end{aligned}$$

where $T_{actual}(\cdot)$s represent the actual execution times of data transfers. However, in the case of $RCV_{i,j}$ in multiple fan-ins, if a global controller generates $start\_read_{i,j}$ only by referring to $stop\_write_{i,j}$ (i.e., $start\_read_{i,j} = stop\_write_{i,j} + 1$), the *deadlock* condition may occur. Fig. 4-15 illustrates the deadlock problem when $f(4)$, $f(6)$ and $f(7)$ of Fig. 4-7 are mapped to 3 different processors.

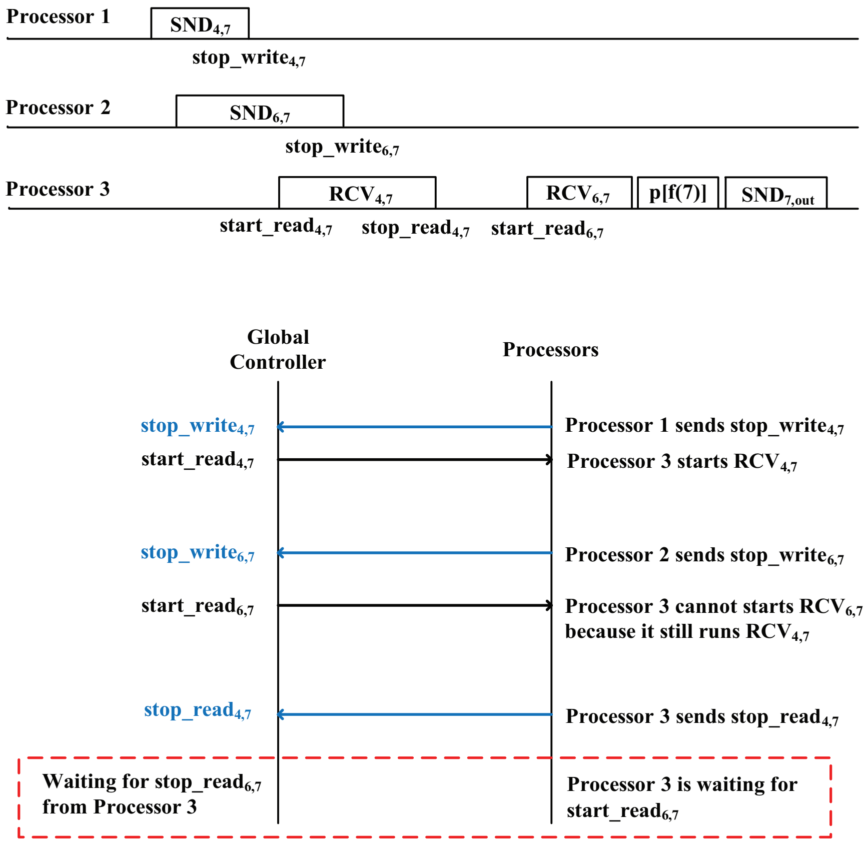
As shown in Fig. 4-15(a), when $stop\_read_{4,7} \leq stop\_write_{6,7}$, $start\_read_{6,7}$ ($= stop\_write_{6,7} + 1$) does not lead to the operational problem. However, if $stop\_read_{4,7} > stop\_write_{6,7}$, $start\_read_{6,7}$ creates the deadlock problem indicated as the dotted line of Fig. 4-15(b). In order to prevent the deadlock problem, the global controller generates $start\_read_{6,7}$ when it receives both $stop\_write_{6,7}$ and $stop\_read_{4,7}$. Therefore, *start_read* in multiple fan-ins is expressed as

$$start\_read_{i,j} = max(stop\_write_{i,j}, stop\_read_{k,j}) + 1$$

for $i \neq j \neq k$, where $RCV_{k,j}$ is prior to $RCV_{i,j}$.



(a) Correct operation when $start\_read_{6,7} = stop\_write_{6,7} + 1$



Processor 1
SND4,7
stop_write4,7
Processor 2
SND6,7
stop_write6,7
Processor 3
RCV4,7
RCV6,7
p[f(7)]
SND7,out
start_read4,7
stop_read4,7
start_read6,7
Global Controller
Processors
stop_write4,7
Processor 1 sends stop_write4,7
start_read4,7
Processor 3 starts RCV4,7
stop_write6,7
Processor 2 sends stop_write6,7
start_read6,7
Processor 3 cannot starts RCV6,7 because it still runs RCV4,7
stop_read4,7
Processor 3 sends stop_read4,7
Waiting for stop_read6,7 from Processor 3
Processor 3 is waiting for start_read6,7

(b) Deadlock condition when $start\_read_{6,7} = stop\_write_{6,7} + 1$

Figure 4-15: Deadlock problem when *start_read* of multiple fan-ins is determined by *stop_write*.

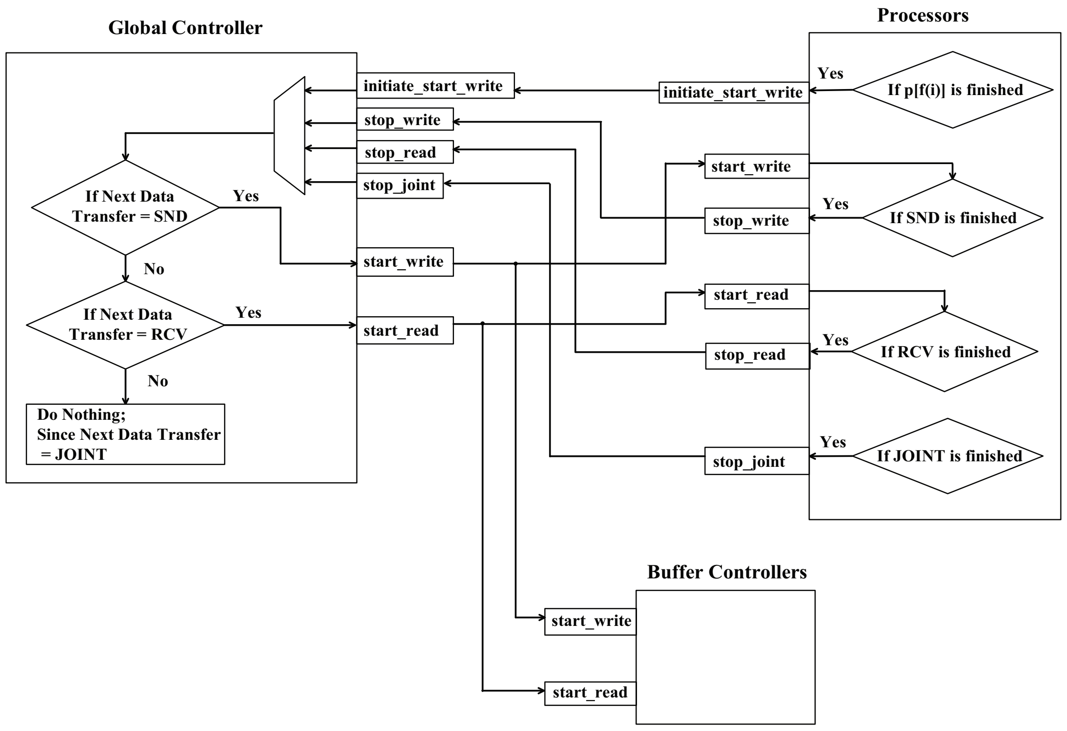


Figure 4-16: Illustration of the signalling for processor initiation scheme.

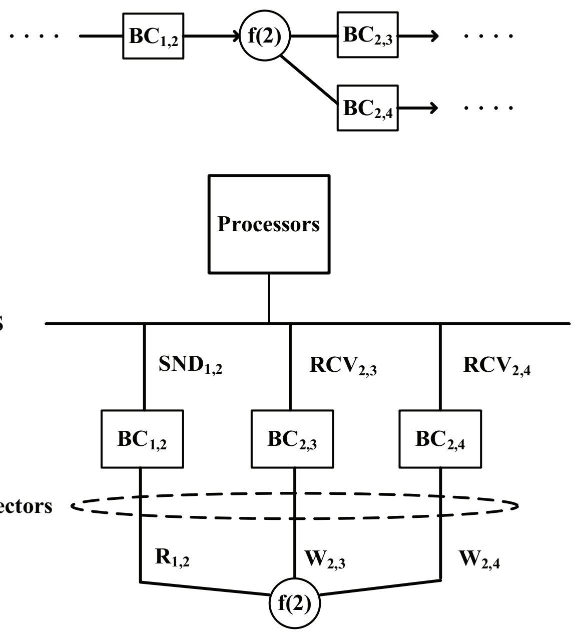
Fig. 4-16 illustrates the relation of *start* and *stop* signals when processor initiation scheme is applied. Here, the global controller has four input signals from processors and two output signals to processors and buffer controllers. Even when the execution times of the programs have a large variation, the processor initiation scheme guarantees the correct operation by the signal handshaking between a global controller and processors. However, as shown in Fig. 4-16, it increases the interconnect resources and the number of ports in a global controller and processors.

### 4.4.4 Processor - Hardware Coexistence of Mapping

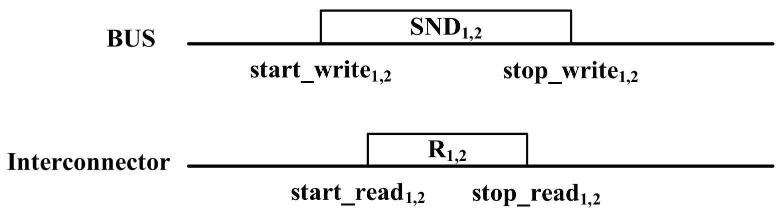
So far, we have considered the case where all processing blocks of a buffer-based dataflow are mapped to processors. In this section, we investigate the synchronization of data transfers in the case where some of processing blocks in a buffer-based dataflow are mapped to processors and others are realized as hardware logics.

When one processing block is implemented as a hardware logic and other processing blocks are mapped to processors, the buffer controllers, which are connected to the fan-in(s) and fan-out(s) of the processing block realized as a hardware, are accessed by both a processor through a bus and a hardware logic via an interconnect. In this case, the timing mismatch problem exists between writing and reading of the buffer controllers as illustrated in Fig. 4-17.

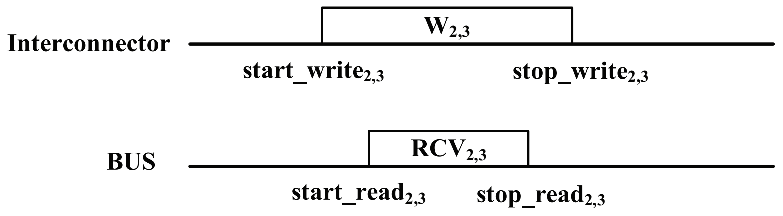
Fig. 4-17 illustrates the case where only $f(2)$ is realized as a hardware logic in a buffer-based dataflow. $R_{1,2}$, $W_{2,3}$ and $W_{2,4}$ represent the reading and writing activities of the processing block $f(2)$. For example, $R_{1,2}$ corresponds to the activity that $f(2)$ reads the data from $BC_{1,2}$. Thus, the execution time of $R_{1,2}$ takes from $start\_read_{1,2}$ to $stop\_read_{1,2}$. In Fig. 4-17(a), if the interconnect speed is faster than $BUS$ speed, $R_{1,2}$ reads the same data more than once. Furthermore, as shown in Fig. 4-17(b), $R_{1,2}$ *partially* receives the data which $SND_{1,2}$ writes to $BC_{1,2}$ in the current iteration period because $R_{1,2}$ finishes before $SND_{1,2}$ ends (i.e., $stop\_read_{1,2} < stop\_write_{1,2}$). If $BUS$ speed is faster than the interconnect speed, $RCV_{2,3}$ reads the same data more than once and fails to read all the data written by $W_{2,3}$ in the current iteration period because $stop\_read_{2,3} < stop\_write_{2,3}$ as shown in Fig. 4-17(c). Even if the interconnect speed and $BUS$ speed are the same, it is not guaranteed that the data transfers are correctly done because the execution times of $SND$ and $RCV$ are non-deterministic. For the correct data transfers through the buffer controllers accessed by both a processor and a hardware logic, the reading of the buffer controllers *must*



(a) Only $f(2)$ is implemented as a hardware.



(b) When the interconnect speed is faster than $BUS$ speed



(c) When $BUS$ speed is faster than the interconnect speed

Figure 4-17: Timing mismatch problem in the processor-hardware coexistence of mapping.

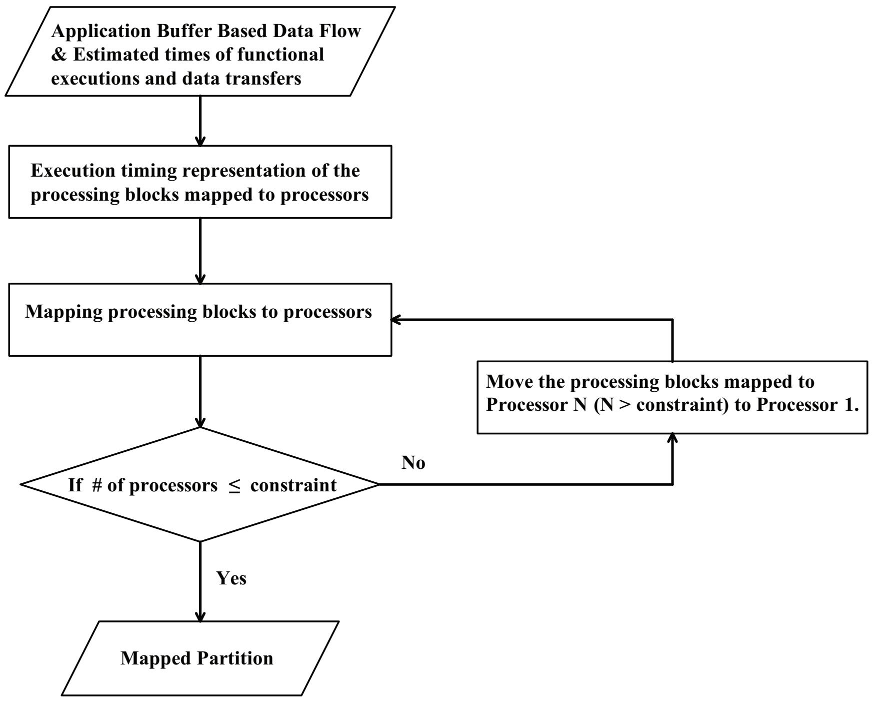

Application Buffer Based Data Flow & Estimated times of functional executions and data transfers
Execution timing representation of the processing blocks mapped to processors
Mapping processing blocks to processors
Move the processing blocks mapped to Processor N (N > constraint) to Processor 1.
No
If # of processors ≤ constraint
Yes
Mapped Partition

Figure 4-18: Proposed mapping algorithm.

start when the writing of the buffer controllers are completely done. Therefore, even though $f(2)$ is implemented as a hardware logic, $start\_read_{1,2}$ is determined not by (4.3) but by (4.10) (i.e., $start\_read_{1,2} = stop\_write_{1,2} + 1$).

## 4.4.5 Mapping Algorithm

We propose a mapping algorithm which creates the mapped partition satisfying the given resource constraint. In the first step, as shown in Fig. 4-9(a), we represent the execution timing of a buffer-based dataflow with the estimated times of functional executions and data transfers. From the execution timing representation, our proposed mapping algorithm creates the mapped partition. Fig. 4-18 shows the overall procedure of the proposed mapping algorithm.

**Algorithm 7** Mapping processing blocks to processors

```
//$G = (V, E)$: a buffer-based dataflow,
//$V$ = the set of processing blocks, $E$ = the set of buffer controllers.
//$V = V_p \bigcup V_h$,
//$V_p$ = the set of processing blocks mapped to processors,
//$V_h$ = the set of processing blocks realized as hardware logics.
//$E_{i,j}$ = the buffer controller between $V_i$ and $V_j$.
//np = number of processors
//processor[np] = array of the processors which $V_i$ and $V_j$ are mapped to.
//non_overlap(a,b) = check if execution times of a and b are non-overlapped,
//non_overlap(a,b) corresponds to equation (4.9).
//non_overlap_BUS(a,b) = check if bus operations of a and b are non-overlapped.
//non_overlap_BUS(a,b) corresponds to equation (4.11).
//$V_i$, $V_j \notin V_h$
//$V_i$, $V_j \in V_p$
//np = 0;

//Mapping $V_i$ to processors
**MAPPING($V_i$)**

**for all** ($V_i$) **do**
    **for all** ($V_j \in$ processor[np]) **do**
        **if** (non_overlap($V_i$, $V_j$)) == TRUE **then**
            add $V_i$ to processor[np];
        **else**
            //If there is $BC_{i,j}$,
            //Check whether $V_i$ is mapped to processor[np] by using $JOINT_{i,j}$
            **if** ($E_{i,j} \in E$) **then**
                remove $SND_{i,j}$ from $V_i$;
                remove $RCV_{i,j}$ from $V_j$;
                //If only $SND_{i,j}$ and $RCV_{i,j}$ is overlapped,
                //$V_i$ is mapped to processor[np] by using $JOINT_{i,j}$
                **if** (non_overlap($V_i$, $V_j$) == TRUE) **then**
                    add $V_i$ to processor[np];
                    add $JOINT_{i,j}$ to processor[np];
                **else**
                    // Restore $V_i$ and $V_j$
                    add $SND_{i,j}$ to $V_i$
                    add $RCV_{i,j}$ to $V_j$
                    //$V_i$ is assigned to new processor.
                    add $V_i$ to processor[np++];
                **end if**
            **else**
                //Since $E_{i,j}$ does not exist,
                //$V_i$ is assigned to new processor.
                add $V_i$ to processor[np++];
            **end if**
        **end if**
    **end for**
**end for**
//Bus sharing among processors
np = 0;
**for all** ($V_i \in$ processor[np], $V_j \in$ processor[np+1]) **do**
    //If $SND$s and $RCV$s are non-overlapped between two processors,
    //the processors share the same bus.
    **if** (non_overlap_BUS($V_i$, $V_j$) == TRUE) **then**
        assign BUS of processor[np] to processor[np+1];
        remove BUS of processor[np+1];
    **end if**
    np++;
**end for**
```

Algorithm 7 corresponds to the "Mapping processing blocks to processors" box of Fig. 4-18. In Fig. 4-18, when the number of processors provided by a target platform is smaller than that of the mapping found by Algorithm 7, the processing blocks mapped to processor $N$ ($N >$ the number of processors in a target platform) are moved to processor 1. The execution times of the moved processing blocks are sequentialized to be non-overlapped with the execution times of the existing processing blocks in processor 1. In addition, all processing blocks (including the moved processing blocks) mapped to processor 1 are ordered to maintain the operational dependency of a buffer-based dataflow. Thus, the iteration period of the buffer-based dataflow is delayed. Algorithm 7 iterates until it finds the mapping to satisfy the number of processors provided by a target platform. It also finds the bus sharing among processors to reduce the number of buses in a target platform. The final outcome of the mapping algorithm is the mapped partition.

### 4.4.6 Template Code Generation

Our proposed mapping algorithm finds the mapped partition from a buffer-based dataflow, estimated times of functional executions and data transfers, and a resource constraint (i.e., the number of processors). The mapped partition has the information on which processing blocks are mapped to which processors. It also has the timing information of processors and hardware logics.

When the functional execution of a mapped processing block is realized as a subroutine, it is necessary to bind the input and output arguments of the subroutine

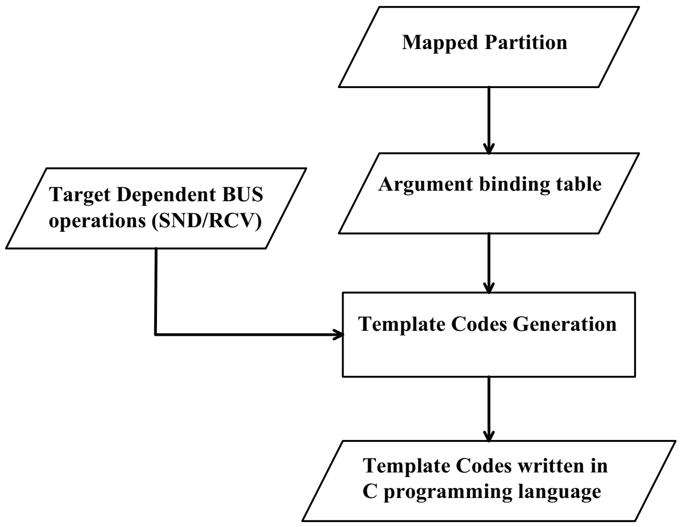

Mapped Partition
Target Dependent BUS operations (SND/RCV)
Argument binding table
Template Codes Generation
Template Codes written in C programming language

Figure 4-19: Procedure of template codes generation.

to the primitive templates for data transfers such as $SND$, $RCV$ and $JOINT$. An input/output argument binding table is extracted from the mapped partition by using the operational dependency represented as the subscripts of programs. If $p[f(i)]$ , $JOINT_{i,j}$ and $p[f(j)]$ run on processor 1, the output argument of $p[f(i)]$ is passed to the input argument of $JOINT_{i,j}$ because the subscript $i$ of $JOINT_{i,j}$ represents $p[f(i)]$. In the same way, the output argument of $JOINT_{i,j}$ is passed to the input argument of $p[f(j)]$ because the subscript $j$ of $JOINT_{i,j}$ corresponds to $p[f(j)]$. Since bus operations are target dependent, $SND$ and $RCV$ are implemented by referencing the external library for target processors. From the argument binding table and the library for the target dependent bus operations, the template codes for target processors are generated. Fig. 4-19 shows the procedure to generate the template code.

## 4.5 Evaluation of Proposed Methodology

In this section, we evaluate the proposed mapping methodology and template code generation. We use SystemC to model a target platform having multi-core processors and reconfigurable logics. The proposed mapping algorithm in Fig. 4-18 is realized through the C programming language. The template code generation in Fig. 4-19 is validated in the Xilinx ISE environment.

### 4.5.1 Evaluation Setup

Our evaluation uses the buffer-based dataflow of Fig. 4-7 and targets the platform consisting of multi-core processors cores at 400MHz and buses at 100MHz. Table 4.2 lists the buffer controller parameters of Fig. 4-7.

Table 4.2: Buffer Controller Parameters of Fig. 4-7

| In | Out | nr | D | M | *start_write* | *start_read* |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 16 | 8 | 9 |
| 1 | 5 | 1 | 0 | 16 | 37 | 38 |
| 5 | 6 | 1 | 0 | 32 | 49 | 50 |
| 1 | 2 | 1 | 0 | 24 | 51 | 52 |
| 6 | 7 | 1 | 0 | 32 | 59 | 60 |
| 2 | 3 | 1 | 0 | 24 | 67 | 68 |
| 3 | 4 | 1 | 0 | 16 | 83 | 84 |
| 4 | 7 | 1 | 0 | 16 | 99 | 100 |
| 4 | 1 | 1 | 0 | 8 | 102 | 103 |
| 7 | out | 1 | 0 | 30 | 115 | 116 |

In Table 4.2, "In" and "Out" represent the input and output of the corresponding buffer controller, respectively. As explained in Section 2.1, $nr$, $D$ and $M$ correspond to the read offset, delay factor and block size, respectively. The estimated execution times of processing blocks are shown in Table 4.3. In addition, the execution times

Table 4.3: Estimated $T_i(max)$ of the processing blocks in Fig. 4-7

| Processing block $f(i)$ | Estimated $T_i(max)$ |
|---|---|
| $f(1)$ | 6000 ns |
| $f(2)$ | 6000 ns |
| $f(3)$ | 16000 ns |
| $f(4)$ | 4000 ns |
| $f(5)$ | 4000 ns |
| $f(6)$ | 8000 ns |
| $f(7)$ | 8000 ns |

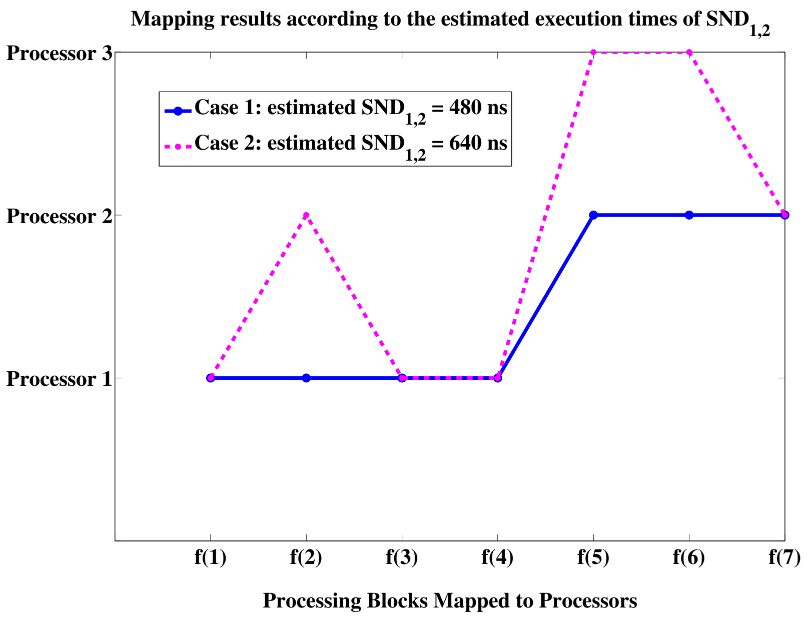


Figure 4-20: Mapping cases according to the estimation of $SND_{1,2}$.

of $SND$s and $RCV$s are also estimated as twice as the corresponding $M$ values in Table 4.2. Fig. 4-20 shows that mapping is changed according to the estimation of $SND_{1,2}$.

In Fig. 4-20, if the actual execution time of $SND_{1,2}$ is always within 480 ns, the mapping result corresponds to case 1. In this case, $BC_{1,2}$, $BC_{2,3}$, $BC_{3,4}$, $BC_{5,6}$ and $BC_{6,7}$ are mapped to $JOINT$s. Thus, the total number of buffer controllers to be realized as hardwares is 5 (= 10 $BC$s in Table 4.2 − 5 $JOINT$s). When the actual

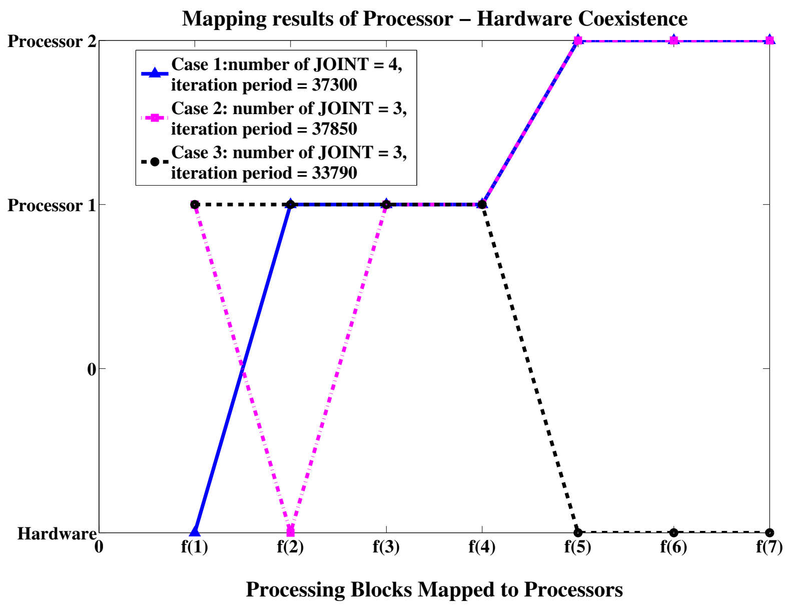


Figure 4-21: Mapping results of processor-hardware coexistence.

execution time of $SND_{1,2}$ vary from 480 ns to 640 ns, the mapping produces wrong outputs in the dataflow. In this case, the execution time of $SND_{1,2}$ is re-estimated as 640 ns. The estimation changes the mapping from case 1 to case 2. Case 2 uses one more processor to have the same iteration period with case 1. In addition, since the mapping of case 2 has 2 $JOINT$s, the total number of buffer controllers implemented as hardwares is 8. If some processing blocks are realized as hardware logics, the iteration period is further reduced. Fig. 4-21 shows the mapping results when some of processing blocks in case 1 of Fig. 4-20 are realized as hardware logics.

In Fig. 4-21, case 1 corresponds to the case where the processing block having multiple fan-ins and fan-outs (i.e., $f(1)$) is realized as a hardware. Case 2 represents that the processing block having a single fan-in and fan-out (i.e., $f(2)$) is implemented as a hardware. Case 3 indicates that the successive processing blocks constructing one feed-forward path ($f(5) \rightarrow f(6) \rightarrow f(7)$) are realized as hardwares. The latency of hardware logics is configured as $f(1) = 270$ ns, $f(2) = 140$ ns, $f(5) = 100$ ns, $f(6) = 80$ ns and $f(7) = 140$ ns. In the results of mapping, even though the hardware latency of $f(2)$ is smaller than that of $f(1)$, the iteration period of case 1 is shorter than that of case 2 because more $SND$s and $RCV$s become writing and reading activities of the hardware logic. When there are multiple feed-forward paths in a buffer-based dataflow, the number of processors for mapping is equal to/smaller than the number of paths (the number of processors is smaller than the number of paths when all execution times among paths are non-overlapped) because our mapping algorithm maps consecutive processing blocks to the same processor as much as possible by using $JOINT$. In case 3, since all successive processing blocks of one feed-forward path are realized as hardwares, the number of processors for mapping is reduced by 1.

### 4.5.2 Processors Only

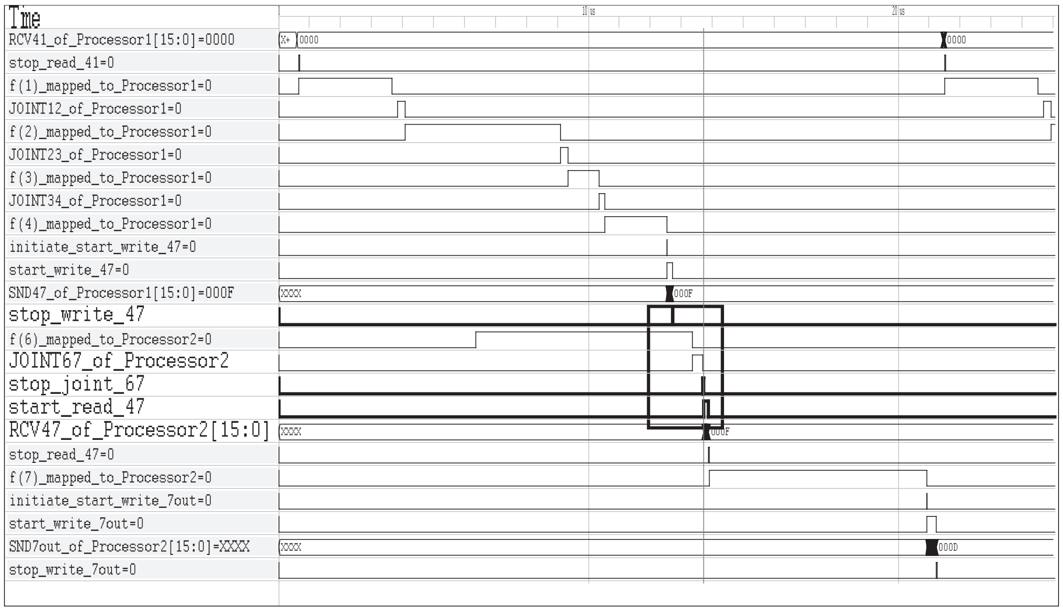
For the correct operation of a buffer-based dataflow, execution times are estimated as their maximum values. However, in case maximum values rarely happen, the estimation may waste the resource in a target system. In "case 2" of Fig. 4-20, if the execution time of $SND_{1,2}$ is observed as 640 ns only in one particular iteration period, processor 3 runs only for one exceptional iteration period. In order to maximally utilize resources in a target system, we use the execution times with a high probability as the estimated values for mapping. When actual execution times take longer than estimated, we apply the processor initiation scheme. Fig. 4-22 shows the simulation results of the processor initiation scheme applied to the mapped partition of "case 1" in Fig. 4-20.

In Fig. 4-22, a global controller generates *start* signals, and processors sends *stop* signals to the global controller. Due to the variation of execution times, $start\_read_{4,7}$ is enabled when either ($stop\_joint_{6,7}$ + 1) in Fig. 4-22(a) or ($stop\_write_{4,7}$ + 1) in Fig. 4-22(b). For the correct operation in both cases, the global controller generates $start\_read_{4,7}$ when it receives $stop\_write_{4,7}$ and $stop\_joint_{6,7}$ from processor 1 and processor 2. As a result, Fig. 4-22(a) shows that processor 2 starts running $RCV_{4,7}$ when it finishes $JOINT_{6,7}$. In Fig. 4-22(b), processor 2 begins running $RCV_{4,7}$ when processor 1 ends $SND_{4,7}$.

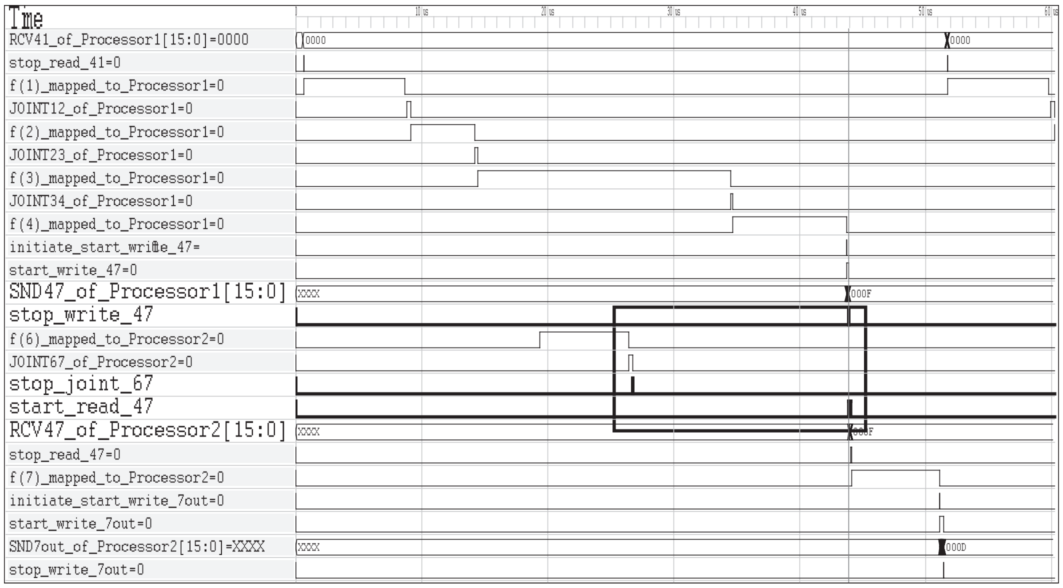
### 4.5.3 Processor-Hardware Coexistence

In the case of processor-hardware coexistence, the writing and reading of the buffer controller between a processor and a hardware are fully sequentialized to solve the timing mismatch between a processor bus and a hardware interconnect. Fig. 4-23 shows the simulation results when $f(1)$ to $f(4)$ are mapped to processor 1 and $f(5)$ to $f(7)$ are realized as hardwares in Fig. 4-7.

In Fig. 4-23, $start\_write_{5,6}$, $start\_read_{5,6}$, $start\_write_{6,7}$ and $start\_read_{6,7}$ are overlapped because $f(5)$, $f(6)$ and $f(7)$ are realized as the hardwares which are able to generate data while they are reading data. On the other hand, the timings of $start\_write_{1,5}$ and $start\_read_{1,5}$ ($start\_write_{4,7}$ and $start\_read_{4,7}$) are non-overlapped because the reading port of $BC_{1,5}$ ($BC_{4,7}$) is connected to a processor bus and its

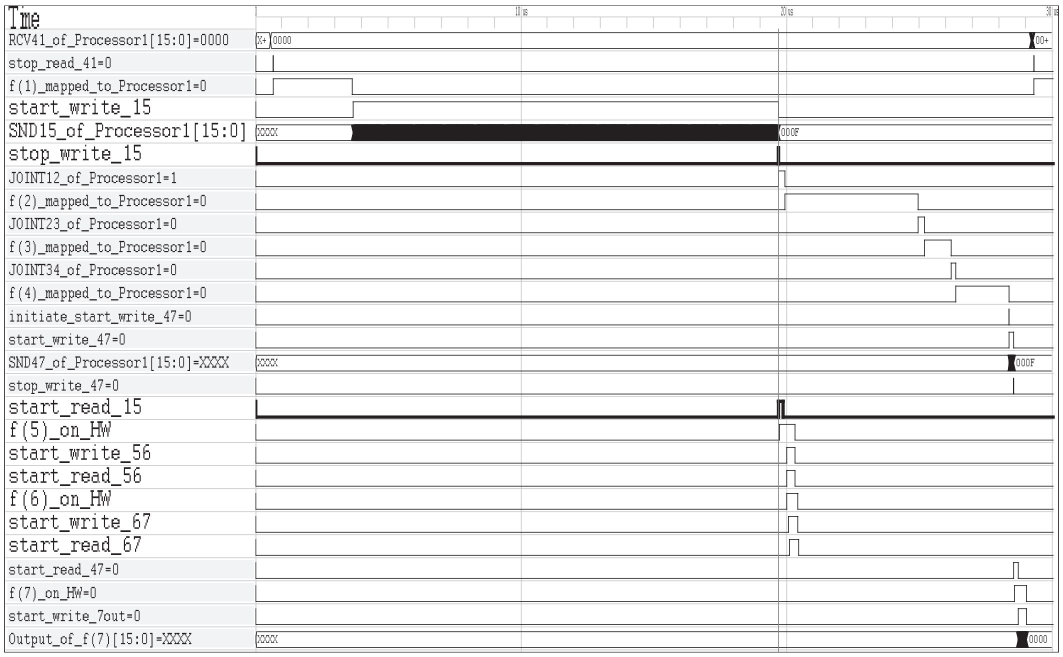


(a) $stop\_joint_{6,7}$ determines $start\_read_{4,7}$

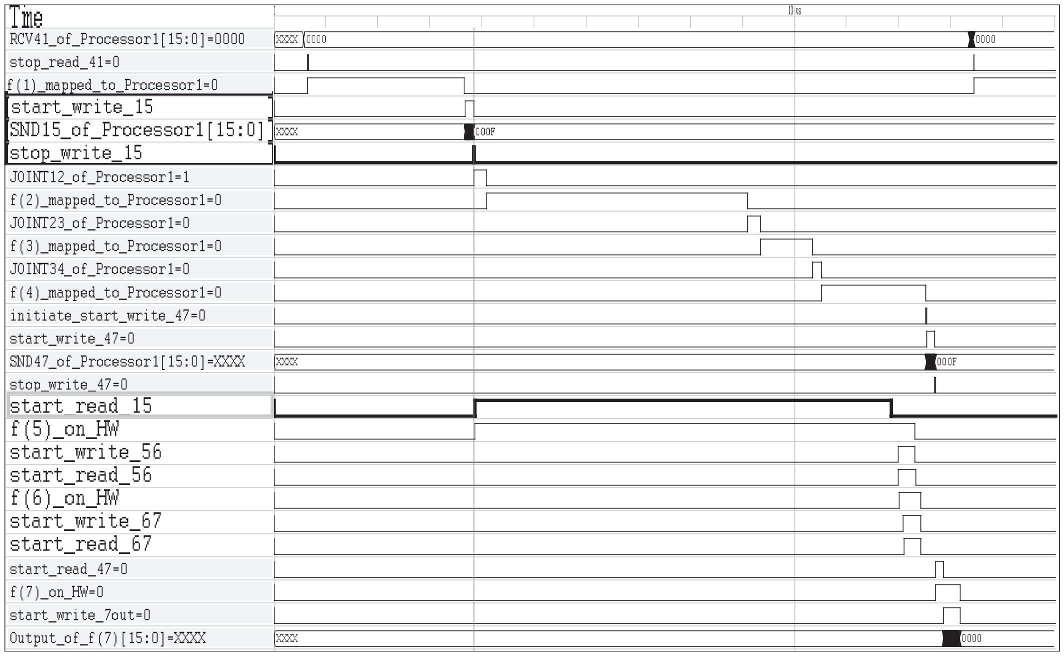


(b) $stop\_write_{4,7}$ determines $start\_read_{4,7}$

Figure 4-22: Simulation results of processor initiation scheme.



(a) When $SND_{1,5}$ takes longer than $R_{1,5}$



(b) When $R_{1,5}$ takes longer than $SND_{1,5}$

Figure 4-23: Simulation results of processor-hardware coexistence.

Table 4.4: Argument binding table for processor 1 of case 3 in Fig. 4-21

| Processing block | Argument | Data Transfer | dependency | size of M | order |
|---|---|---|---|---|---|
| $f(1)$ | $f1\_out\_1$ | N/A | $M\_1\_5$ | 16 | 1 |
| | $f1\_out\_2$ | N/A | $M\_1\_2$ | 24 | 2 |
| | $f1\_in\_1$ | RCV | $M\_0\_1$ | 16 | 1 |
| | $f1\_in\_2$ | RCV | $M\_4\_1$ | 8 | 2 |
| | $f1\_bus\_out\_1$ | SND | $M\_1\_5$ | 16 | 1 |
| | $f1\_bus\_out\_2$ | JOINT | $M\_1\_2$ | 24 | 2 |
| $f(2)$ | $f2\_out\_1$ | N/A | $M\_2\_3$ | 24 | 1 |
| | $f2\_in\_1$ | JOINT | $M\_1\_2$ | 24 | 1 |
| | $f2\_bus\_out\_1$ | JOINT | $M\_2\_3$ | 24 | 1 |
| $f(3)$ | $f3\_out\_1$ | N/A | $M\_3\_4$ | 16 | 1 |
| | $f3\_in\_1$ | JOINT | $M\_2\_3$ | 24 | 1 |
| | $f3\_bus\_out\_1$ | JOINT | $M\_3\_4$ | 16 | 1 |
| $f(4)$ | $f4\_out\_1$ | N/A | $M\_4\_7$ | 16 | 1 |
| | $f4\_out\_2$ | N/A | $M\_4\_1$ | 8 | 2 |
| | $f4\_in\_1$ | JOINT | $M\_3\_4$ | 16 | 1 |
| | $f4\_bus\_out\_1$ | SND | $M\_4\_7$ | 16 | 1 |
| | $f4\_bus\_out\_2$ | SND | $M\_4\_1$ | 8 | 2 |

writing port is connected to a hardware interconnect. Fig. 4-23(a) shows the result when $SND_{1,5}$ takes longer than $R_{1,5}$. This case represents that a processor bus speed is slower than a hardware interconnect speed. In contrast, Fig. 4-23(b) shows the case when $R_{1,5}$ takes longer than $SND_{1,5}$. In this case, a hardware interconnect speed is slower than a processor bus speed. In both cases, a global controller generates $start\_read_{1,5} = (stop\_write_{1,5} + 1)$ for the correct data transfer in the timing mismatch between a processor bus and a hardware interconnect.

## 4.5.4 Template Code Generation

From the mapped partition created by the mapping algorithm of Fig. 4-18, our methodology establishes the argument binding table to generate template codes. Table 4.4 lists the argument binding table for processor 1 of "case 3" in Fig. 4-21.

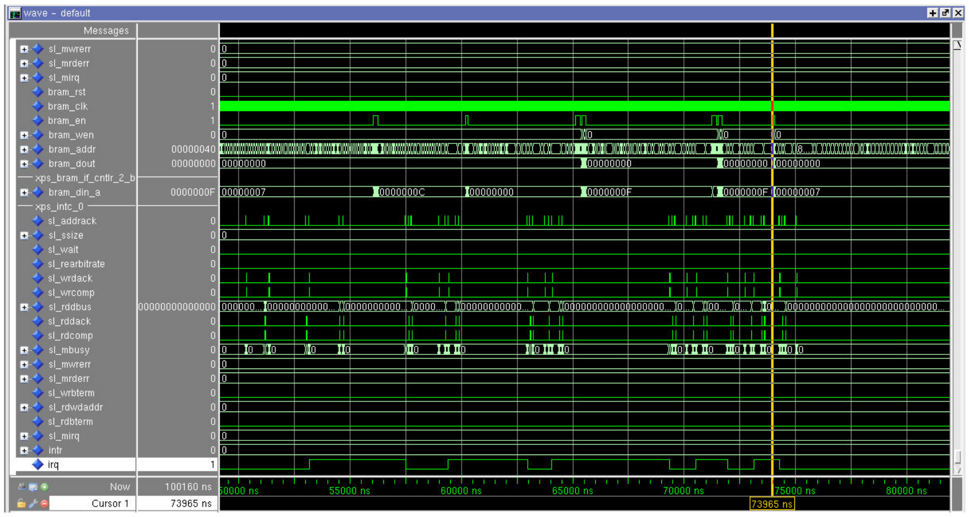
In Table 4.4, "dependency" represents the direction of argument passing between

```
int* RCV(unsigned int size_of_M, u32 address_offset)
{
    XCache_InvalidateDCacheRange_revised(XPAR_XPS_BRAM_IF_CNTLR_2_BASEADDR\
            + address_offset, size_of_M*sizeof(int));
    memcpy(return_RCV, (int*)(XPAR_XPS_BRAM_IF_CNTLR_2_BASEADDR\
            + address_offset), size_of_M*sizeof(int));
    return return_RCV;
}

void SND(int* output_of_fn, unsigned int size_of_M, u32 address_offset)
{
    memcpy((int*)(XPAR_XPS_BRAM_IF_CNTLR_2_BASEADDR + address_offset),\
            output_of_fn,    size_of_M*sizeof(int));
    XCache_FlushDCacheRange_revised(XPAR_XPS_BRAM_IF_CNTLR_2_BASEADDR \
            + address_offset, size_of_M*sizeof(int));
}
```

(a) Target dependent $RCV$ and $SND$ subroutines

```
void template_code()
{
    f1_in_1 = RCV(M_0_1);
    f1_in_2 = RCV(M_4_1);
    f1_out_1 = f1(f1_in_1, f1_in_2);
    f1_out_2 = f1(f1_in_1, f1_in_2);
    SND(f1_out_1,M_1_5);
    f1_bus_out_2 = JOINT(f1_out_2,M_1_2);
    f2_in_1 = JOINT(f1_bus_out_2,M_1_2);
    f2_out_1 = f2(f2_in_1);
    f2_bus_out_1 = JOINT(f2_out_1,M_2_3);
    f3_in_1 = JOINT(f2_bus_out_1,M_2_3);
    f3_out_1 = f3(f3_in_1);
    f3_bus_out_1 = JOINT(f3_out_1,M_3_4);
    f4_in_1 = JOINT(f3_bus_out_1,M_3_4);
    f4_out_1 = f4(f4_in_1);
    f4_out_2 = f4(f4_in_1);
    SND(f4_out_1,M_4_7);
    SND(f4_out_2,M_4_1);
}
```

(b) Argument binding subroutine



(c) Simulation result in ModelSim 6.3c of Xilinx ISE 10.1

Figure 4-24: Generated template codes for Xilinx Virtex-5 FXT.

the subroutine of a functional execution and the primitive template such as $RCV$, $SND$ and $JOINT$; "order" indicates the order of arguments. For example, $f1\_out\_2$ is the second output argument of $f1$. Since $f1\_bus\_out\_2$ has the same "dependency" (i.e., $M\_1\_2$) with $f1\_out\_2$, the output stored in $f1\_out\_2$ is passed to the input argument of $JOINT$. Based on the argument binding table, the template code are generated as shown in Fig. 4-24.

Fig. 4-24(a) shows $RCV$ and $SND$ primitive templates which are generated for the Xilinx Virtex-5 FXT target platform. Fig. 4-24(b) is the automatically generated template code based on Table 4.4. Compared with the subroutines of Fig. 4-24(a), the argument binding subroutine does not contain the target specific library because it is fully realized with the standard C library. In Fig. 4-24(c), the generated template code is working in ModelSim 6.3c on Xilinx ISE 10.1. In the result, when "irq" is set to 1, two $RCV$s and three $SND$s run because $start\_read$ and $start\_write$ are realized as the interrupt of PPC440 in Xilinx Virtex-5 FXT.

# Chapter 5

# Conclusion and Future Research

## 5.1 Conclusion

In this dissertation, we have presented a sharing methodology to reduce the interconnect resource in a data-centric applications represented as a buffer-based dataflow. The proposed sharing methodology rearranges the lifetimes and activity times of buffers to increase the possibility of buffer and bus sharing. Our approach also splits activity times in order to balance data transfers through buses. However, since buffer and bus sharing increases the dynamic energy consumption, we do not guarantee that the sharing case with the minimum resource consumes the minimum energy. In order to find the sharing case consuming the minimum energy, we thus establish an energy consumption model with the estimated buffer and bus costs. The proposed sharing methodology was evaluated with several data-centric applications. Through the evaluation, we confirmed that the sharing case having the minimum resource may not correspond to the sharing case consuming the minimum energy.

We also proposed a mapping methodology to synthesize processing elements of a dataflow in a target platform having multi-core processors and programmable logics. In order to achieve synchronized data transfers between processors (or between a processor and a hardware), we used the buffer-based dataflow. From the buffer-based dataflow and estimated execution times for functions and data transfers, the proposed methodology creates a mapped partition that satisfies a given resource constraint. After the mapped partition is created, the template code for the processing blocks mapped to processors is generated. We also devised a processor initiation scheme to prevent wrong operations when the actual execution times take longer than estimated. The proposed methodology was evaluated with the SystemC model and Xilinx ISE 10.1. Through the evaluation, we demonstrated that the mapping by our proposed methodology is successfully working even with misestimated execution times.

## 5.2 Future research

The proposed methodologies are applied to the synthesis of an application dataflow graph in order to reduce the interconnect resources and to synchronize data transfers between different processsing elements (i.e., between processors/between a processor and a hardware). However, they do not support the reconfiguration for synthesizing multiple application dataflow graphs to the same platform. In order for two application dataflow graphs which share a large number of processing blocks (e.g. MPEG4 and H.264) to be synthesized onto the same platform, we need a reconfigurable architecture having reconfigurable interconnects.

Reconfigurable architectures can be configured to any of different operations during post−fabrication, which allows flexibility traditionally provided only by programmable processors. Another important feature of reconfigurable architecture is the ability to have application- dependent structure (dataflow and control flow) that can achieve performance close to ASIC implementations. Due to these features, reconfigurable architectures have become platform for embedded processing systems. Also, the algorithms are becoming more and more adaptive in nature with respect to space and time. These algorithms have a unique execution characteristics where the real−time constraint varies dynamically in time. For example, computation requirements can be changed depending on performance level such as SNR as well as sampling rate of the incoming data. These characteristics provide an opportunity to investigate reconfigurable hardware design.

FPGAs are fine grained and boasting the ability to map any kind of algorithm. The FPGAs expose the limitations of handling the high throughput and low power applications created by the memory hungry systems. These commercial FPGAs are embedding some coarse grained blocks (like multipliers) to increase the performance. Nevertheless, they have not been able to achieve the performance desired by the systems today. The other approach to improve the performance is use of coarse grained reconfigurable architecture. Such an architecture has large number of processing elements and buffers designed in ASIC fashion and they interact with each other through reconfigurable interconnect. The idea is to group these individual logic blocks into a *Heterogenous Processing Element* (abbreviated to **HPE** ) to interact within themselves and with memory. Such a system has the potential to further reduce area and energy for embedded DSP applications. We will present an architecture consisting of HPE and BUF interacting through a scalable and reconfigurable interconnect. The reconfigurable architecture will realize multiple applications represented as buffer-based dataflows in the same platform.

# Bibliography

[1] Ken Kundert, “Design of mixed-signal Systems on a Chip,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, issue 12, pp. 1561-1571, Dec. 2000.

[2] McCorquodale, M.S. Gebara, F.H. Kraver, K.L. Marsman, E.D. Senger and R.M. Brown, “A Top-Down Microsystems Design Methodology and Associated Challenges,” *Design, Automation and Test in Europe Conference and Exhibition*, pp. 292-296, 2003.

[3] Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya, “Parameterized Dataflow Modeling for DSP Systems,” *IEEE Transactions On Siganl Processing*, vol. 49, no. 10, Oct. 2001.

[4] A. Singh, G. Parthasarathy and M. Marek-Sadowska, ”Efficient circuit clustering for area and power reduction in FPGAs,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, no. 4, pp. 643-663, Oct. 2002.

[5] D. Chen, J. Cong and Y. Fan, ”Low-power high-level synthesis for FPGA architectures,” *In Proceedings of the 2003 international Symposium on Low Power Electronics and Design*, pp. 134-139, Aug. 2003.

[6] J. Cong, Y. Fan and J. Xu, ”Simultaneous resource binding and interconnection optimization based on a distributed register-file microarchitecture,” *ACM Transaction on Design Automation of Electronic Systems*, vol. 14, no. 3, pp. 1-31, May. 2009.

[7] J. Mun, S. H. Cho, and S. Hong, ”Flexible Controller Design and Its Application for Concurrent Execution of Buffer Centric Dataflows,” *Journal of VLSI Signal Processing*, vol. 47, no. 3, pp. 233-257, Jun. 2007.

[8] S. Hong, J. Lee, A. Athalye, P. M. Djuric, W. Cho, ”Design Methodology for Domain Specific Parameterizable Particle Filter Realizations,” *IEEE Transactions on Circuits and Systems I:Regular Papers*, vol. 54, no. 9, pp. 1987-2000, Sep. 2007.

[9] P. Briggs, K. Cooper and L. Torczon, ”Improvements to graph coloring register allocation,” *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 428-455, May. 1994.

[10] P. K. Murthy and S. S. Bhattacharyya, "Buffer merging – a powerful technique for reducing memory requirements of synchronous dataflow specifications," *ACM Transactions on Design Automation of Electronic Systems*, vol. 9, no. 2, pp. 212-237, Apr. 2004.

[11] H. Jung, H. Yang and S. Ha, "Optimized RTL Code Generation from Coarse-Grain Dataflow Specification for Fast HW/SW Cosynthesis," *Journal of Signal Processing Systems*, vol. 52, no. 1, pp. 13-34, Jul. 2008.

[12] F. Berthelot, F. Nouvel and D. Houzet, "A Flexible system level design methodology targeting run-time reconfigurable FPGAs," *EURASIP Journal on Embedded Systems*, vol. 2008, pp. 1-18, Jan. 2008.

[13] R. Lauwereins, et al., "Grape-II: A system-level prototyping environment for DSP applications," *IEEE Computers*, pp. 35-43, February 1995

[14] N.R. Satish, K.Ravindran, K.Keutzer, "Scheduling Task Dependence Graphs with Vairable Task Execution Times onto Heterogeneous Multiprocessors", *Presentation slides on Embedded Systems Week*, Oct. 2008.

[15] Xilinx Inc. Virtex-5 and Virtex-6 Field Programmable Gate Arrays, June 25, 2009. http://www.xilinx.com.

[16] J. Liu, M. Lajolo, and A. Sangiovanni-Vincentelli, "Software timing analysis using HW/SW cosimulation and instruction set simulator," *In Proceedings of the 6th international Workshop on Hardware/Software Codesign, IEEE Computer Society*, pp. 65-69, Mar. 1998.

[17] A. Bouchhima, S. Yoo, and A. Jeraya, "Fast and accurate timed execution of high level embedded software using HW/SW interface simulation model," *In Proceedings of the 2004 Conference on Asia South Pacific Design Automation*, pp. 469-474, Jan. 2004.

[18] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon and Y. Joo, "PeaCE: A hardware-software codesign environment for multimedia embedded systems," *ACM Transaction on Design Automation of Electronic Systems*, vol. 12, no. 3, pp. 1-25, Aug. 2007.

[19] F. Fummi, M. Loghi, M. Poncino and G. Pravadelli, "A cosimulation methodology for HW/SW validation and performance estimation," *ACM Transaction on Design Automation of Electronic Systems*, vol. 14, no. 2, pp. 1-32, Mar. 2009.

[20] H. Jung, H. Yang, and S. Ha, "Optimized RTL Code Generation from Coarse-Grain Dataflow Specification for Fast HW/SW Cosynthesis," *Journal of Signal Processing Systems*, vol. 52, no. 1, pp.13-34, Jul. 2008.

[21] T. Yen and W. Wolf, "Communication synthesis for distributed embedded systems," *In Proceedings of the 1995 IEEE/ACM international Conference on Computer-Aided Design, IEEE Computer Society*, pp. 288-294, Nov. 1995.