Buffer Controller-Based Multiple Processing Element Utilization for Dataflow Synthesis

Woohyung Chun, *Member, IEEE*, Sungroh Yoon, *Member, IEEE*, and Sangjin Hong, *Senior Member, IEEE*

***Abstract*—This paper presents an effective design methodology which maps a complex system represented as a dataflow graph to a reconfigurable target architecture having multi-core processors and programmable logics. In order to synchronize data transfers between two processing blocks mapped to different processors (alternatively, one block is mapped to a processor and the other is realized as a hardware), we propose a mapping methodology that exploits the buffer-based dataflow, a new representation technique for realizing data-centric applications in reconfigurable platforms. 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 the target platform. We also use a processor initiation scheme to prevent wrong operations from happening when actual execution takes longer than estimated. Our proposed mapping methodology and the generated template code are evaluated with the SystemC model and Xilinx ISE.**

***Index Terms*—Buffer-based dataflow, data-centric applications, field-programmable gate array (FPGA), multi-core architecture, system mapping.**

I. Introduction

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

When a dataflow is synthesized in a target platform having multi-core processors and hardware logics, it becomes difficult to synchronize data transfers between processing blocks mapped to different processors (alternatively, one is mapped to a processor and the other is implemented as a hardware) because the execution time of processors varies due to the dynamic behavior of software such as interrupt handling and context switching. Thus, the execution times of processing blocks are estimated for mapping [2]–[5]. Methods for mapping on multi-core processors, which also use estimated execution times, include [6]–[8]. However, in the case where actual execution times are greater than the estimated times, the mapping based on the estimated times may produce wrong results. To prevent this problem, Jung *et al.* [9] proposed a handshaking scheme between a centralized controller and sequential logics having variable execution time. However, the handshaking scheme has a limited capability to support the data transfers between processors (or between a processor and a hardware) because the data transfers on processors are also the programs having variable execution times [10].

In order to synchronize data transfers, we use the buffer-based dataflow [11], [12], 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, 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). Since our mapping methodology does not include a hardware-software partitioning technique [13], each processing block is predetermined in such a way that it is either mapped to a processor or realized as a hardware logic.

For the processing blocks mapped to processors, the proposed methodology translates the data transfer activities 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 end of executions

Manuscript received September 21, 2009; revised December 28, 2009; accepted April 05, 2010. First published May 24, 2010; current version published June 24, 2011. This work was supported by the National Research Foundation of Korea (NRF) Grant funded by the Korea government (MEST) (2010-0000407 and 2010-0000631).

W. Chun is with System LSI Division, Samsung Electronics, Gyeonggi-Do 446-711, Korea (e-mail: wh.chun@samsung.com).

S. Yoon is with the School of Electrical Engineering, Korea University, Seoul 136-713, Korea (e-mail: sryoon@korea.ac.kr).

S. Hong is with the Department of Electrical and Computer Engineering, Stony Brook University—SUNY, Stony Brook, NY 11794-2350 USA (e-mail: snjhong@ece.sunysb.edu).

Digital Object Identifier 10.1109/TVLSI.2010.2049388

1063-8210/$26.00 © 2010 IEEE

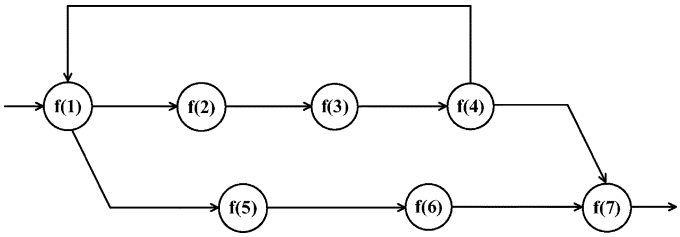


Fig. 1. Illustration of the coarse grain signal processing application represented as a dataflow.

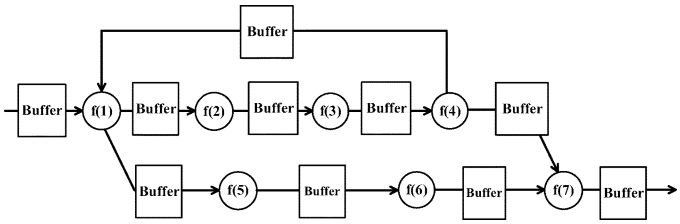


Fig. 2. Illustration of established dataflow by inserting buffers.

of all processors (including data transfers) to a global controller. Upon receiving the notification, the global controller generates signals to initiate the next execution of processors. Thus, even if execution times change with large variations, the execution ordering is preserved.

The remainder of this paper is organized as follows. Section II describes the background of the buffer-based dataflow and the motivation of this research. In Section III, we characterize the mapping technique that utilizes the buffer-based dataflow. Section IV 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 V, we evaluate the proposed mapping algorithm with the SystemC model and demonstrate that the generated template code works on Xilinx ISE. Finally, Section VI concludes with the summary of our contribution.

II. Background and Motivation

A. Buffer-Based Dataflow Design Approach Overview

Fig. 1 shows a coarse-grain signal processing application represented as a dataflow. Each processing block represents its functionality, and an edge between processing blocks denotes their operational dependency. The arrow in an edge indicates the direction of the corresponding data transfer.

Fig. 2 illustrates the buffer-based dataflow derived from Fig. 1 by inserting buffers. Here, the relationship between processing blocks is isolated by inserting buffers to the edges of the dataflow in Fig. 1. By separating the relationship between processing blocks, processing blocks are only able to represent the functionality. The isolation also enhances the reconfigurability of the overall system.

In the buffer-based dataflow graph, inserting buffers represents the edges delivering data frames from the source to destination. Thus, the size of data frames appearing at the input port of a buffer is the same as the size of data frames at the output port of the buffer. Furthermore, while a source processing block is writing data to a buffer, the corresponding destination processing block is able to read data from the buffer. The buffers are realized as the dual port memory that supports simultaneous writing and reading.

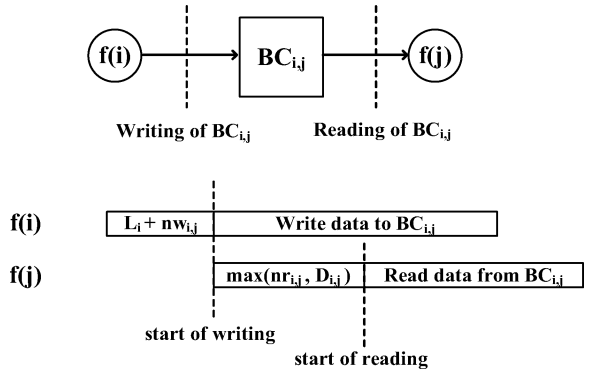


Fig. 3. Write and read timing of $\mathrm{BC}_{i,j}$.

TABLE I
Operational Dependency Derived From Fig. 4

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

As illustrated in Fig. 3, the buffer between the producer processing block $f(i)$ and the consumer processing block $f(j)$ is denoted as $\mathrm{BC}_{i,j}$. The primary parameters which determine the buffer controller structure and overall physical realization are represented as logic latency (L_i), write offset $(nw_{i,j})$, read offset $(nr_{i,j})$, block size $(M_{i,j})$, and delay factor $(D_{i,j})$ [11]. The write offset $nw_{i,j}$ 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 $\mathrm{BC}_{i,j}$ to the start of reading data from $\mathrm{BC}_{i,j}$ when the writing speed of processing block $f(i)$ and the reading speed of processing block $f(j)$ are matched. However, if the former is slower than the latter, processing block $f(j)$ does not read valid data from $\mathrm{BC}_{i,j}$. For this, the delay factor $D_{i,j}$ is used to represent the rate mismatch between processing blocks $f(i)$ and $f(j)$.

B. Constructing a Buffer-Based Dataflow

A buffer-based dataflow is constructed by using the *buffer controller parameter table* which is extracted from the operational dependency of a dataflow and the offsets of fan-ins and fan-outs of processing blocks. In Table I, $e_{i,j}$ represents the edge from the source $(f(i))$ to the destination $(f(j))$. $\mathrm{start_write}_{i,j}$ represents the start time of writing data through $e_{i,j}$ and $\mathrm{start_read}_{i,j}$ is the start time of reading data through

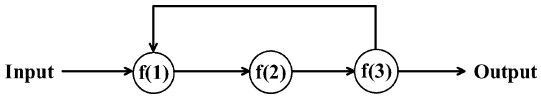


Fig. 4. Example dataflow.

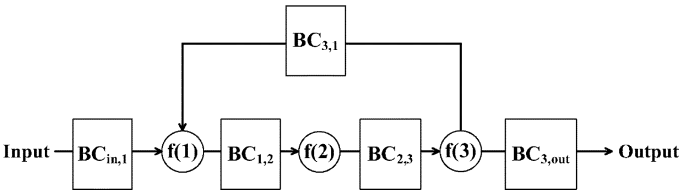


Fig. 5. Buffer-based dataflow derived from Fig. 4.

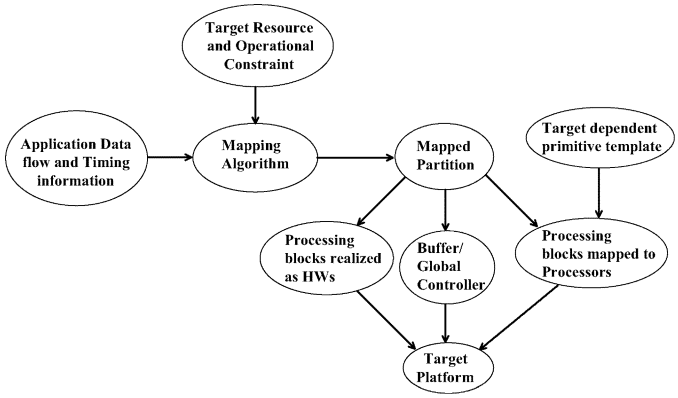


Fig. 6. Overall flow of proposed mapping methodology.

$e_{i,j}$. Since writing data to $e_{i,j}$ precedes reading data from $e_{i,j}$, $\text{start_write}_{i,j} < \text{start_read}_{i,j}$. In Table I, $f(1) - f(3)$ represents the operational dependency between the fan-in and fan-out edges of processing blocks. In the operational dependency of $f(3)$, $\text{start_write}_{3,1}$ is removed because $e_{3,1}$ is a feedback loop. If $f(1)$ reads the data generated by $f(3)$ in the current iteration period, the dataflow falls into the *deadlock* situation because $f(1)$ and $f(3)$ keep waiting for the data generated by each other.

Fig. 5 shows the buffer-based dataflow converted from the dataflow in Fig. 4. Since the buffer controllers in Fig. 5 have the same operational characteristics as the edges in Fig. 4, start_writes and start_reads of Table I are represented with the primary parameters introduced in Section II-A. In the buffer controller $\text{BC}_{i,j}$, the *start* signals are realized with the primary parameters introduced in Section II-A as follows:

$$\text{start_write}_{i,j} = L_i + nw_{i,j} + \text{start}_i \quad (1)$$

$$\text{start_read}_{i,j} = \text{start_write}_{i,j} + \max(nr_{i,j}, D_{i,j}) \quad (2)$$

$$\text{stop_write}_{i,j} = \text{start_write}_{i,j} + M_{i,j} \quad (3)$$

$$\text{stop_read}_{i,j} = \text{start_read}_{i,j} + M_{i,j}. \quad (4)$$

In (1), start_i is the time value in which $f(i)$ begins reading data from the previous buffer controller through its fan-in port. Equation (2) reflects the rate mismatch between the source processing block $f(i)$ and the destination processing block $f(j)$. 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 $\text{BC}_{i,j}$, the data are continuously being written to (or read from) $\text{BC}_{i,j}$ until the size of transferred data reaches $M_{i,j}$. Equations (3) and (4) represent the end time of writing and reading data to/from $\text{BC}_{i,j}$, respectively.

In the buffer-based dataflow, the buffer memory size of $\text{BC}_{i,j}$, $\text{MEM}(\text{BC}_{i,j})$ is given by

$$\text{MEM}(\text{BC}_{i,j}) = \min\{M_{i,j}, (\text{start_read}_{i,j} - \text{start_write}_{i,j})\}. \quad (5)$$

In (5), when the reading of $\text{BC}_{i,j}$ starts before the end of writing data to $\text{BC}_{i,j}$ (i.e., $\text{start_read}_{i,j} < \text{start_write}_{i,j} + M_{i,j}$), $\text{MEM}(\text{BC}_{i,j})$ is determined by the difference between $\text{start_read}_{i,j}$ and $\text{start_write}_{i,j}$. In this case, while $f(i)$ is writing data to $\text{BC}_{i,j}$, $f(j)$ can read data from $\text{BC}_{i,j}$. However, if the reading of $\text{BC}_{i,j}$ starts when the writing of $\text{BC}_{i,j}$ is completed, $\text{MEM}(\text{BC}_{i,j})$ is $M_{i,j}$.

C. Our Approach and Objectives

When a dataflow is synthesized in the target platform having multi-core processors and programmable logics such as Xilinx Virtex FPGA devices [1], it is difficult to synchronize data transfers between processors/between a processor and a hardware logic 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. Since it is difficult to synchronize data transfers between processors (or between a processor and a hardware logic), our mapping algorithm tries to map consecutive processing blocks to the same processor. In the proposed methodology, 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. 6 shows the overall flow of the proposed mapping methodology. In Section III, the mapping of processing blocks is characterized with primitive templates.

In Fig. 6, 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. 7.

In Fig. 7, $f(i)$ 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 $\text{BC}_{i,j}$. In order to synchronize data transfers between each processing block (i.e., $f(i)$ and $f(j)$) and $\text{BC}_{i,j}$, the global controller generates $W_\text{BC}_{i,j}$ and $R_\text{BC}_{i,j}$. $W_\text{BC}_{i,j}$ is the signal which enables $f(i)$ to write data to $\text{BC}_{i,j}$, whereas $R_\text{BC}_{i,j}$ is the signal which initiates $f(j)$ to read data from $\text{BC}_{i,j}$.

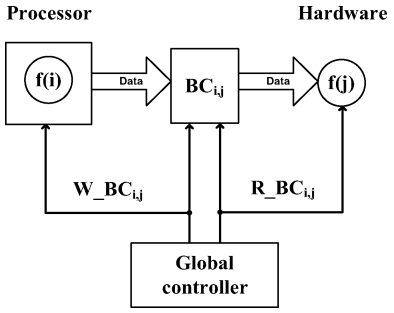


Fig. 7. Synchronization of data transfers through a global controller and a buffer controller.

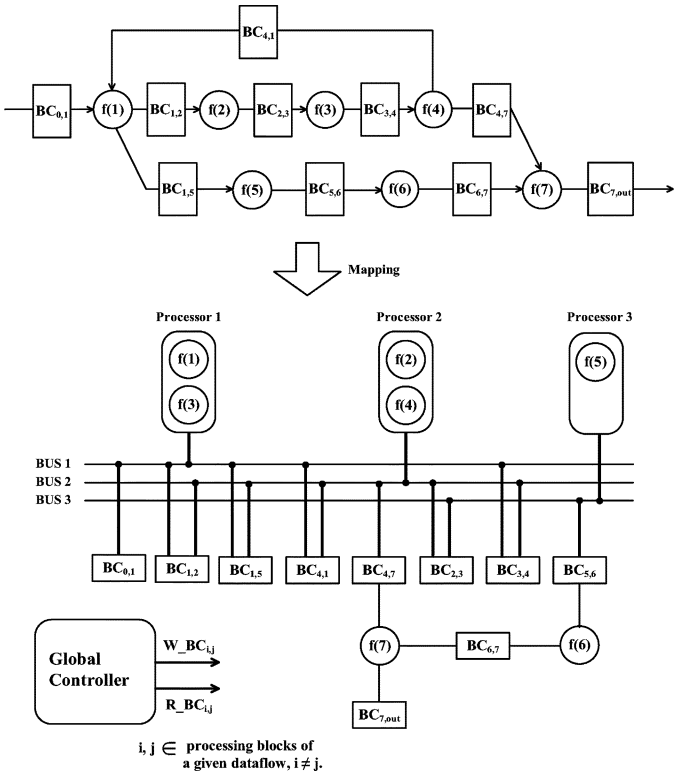


Fig. 8. Mapping a buffer-based dataflow to a target platform.

III. Mapping Characterization

Fig. 8 illustrates that the buffer-based dataflow transformed from the dataflow of Fig. 1 is mapped to a target platform having multi-core processors and hardware logics. In Fig. 8, 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 a processing block either mapped to a processor or realized as a hardware logic and a buffer controller. Therefore, the mapped processing blocks need the entities for the data transfers with buffer controllers outside the processors. Section III-A defines primitive templates as the entities for the data transfers with buffer controllers. Section III-B investigates the mapping effect when processing blocks are mapped by using the primitive templates.

```
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)

```
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)

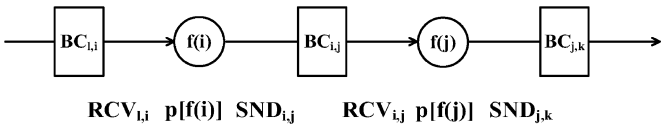
Fig. 9. Pseudo codes of $SND_{i,j}$ and $RCV_{i,j}$.

A. 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. 9.

In Fig. 9, $RCV_{i,j}$ corresponds to the fan-in operation of $f(j)$, and $SND_{i,j}$ represents the fan-out operation of $f(i)$. $R_\mathrm{BC}_{i,j}$ is the signal initiating $RCV_{i,j}$ to read data from $\mathrm{BC}_{i,j}$. $W_\mathrm{BC}_{i,j}$ is also the signal initiating $SND_{i,j}$ to write data to $\mathrm{BC}_{i,j}$. Both $R_\mathrm{BC}_{i,j}$ and $W_\mathrm{BC}_{i,j}$ are generated by a global controller. The "read" and "write" functions indicate target-dependent bus operations. 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. 10.

In Fig. 10, $p[f(i)]$ and $p[f(j)]$ are the programs corresponding to $f(i)$ and $f(j)$, respectively. $\mathrm{END}_R_\mathrm{BC}_{l,i}$ ($\mathrm{END}_R_\mathrm{BC}_{i,j}$) is the signal to notify $p[f(i)]$ ($p[f(j)]$) that $RCV_{l,i}$ ($RCV_{i,j}$) is completed. Both $\mathrm{END}_R_\mathrm{BC}_{l,i}$ and $\mathrm{END}_R_\mathrm{BC}_{i,j}$ are generated by a global controller. Since the




```
Input_of_p[f(i)] = RCV_{i,i} ( size_of_M_{i,i} );

function p[f(i)] (Input_of_p[f(i)] ) {
    while( END_R_BC_{i,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});
```

Fig. 10. Pseudo template codes of a given buffer-based dataflow.

```
function 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
```

Fig. 11. Pseudo-code of $\mathrm{JOINT}_{i,j}$.

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 $\mathrm{Input_of_}p[f(i)]$. 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. 11 shows the pseudo-code of $\mathrm{JOINT}_{i,j}$.

In Fig. 11, compared with *RCV* and *SND* operations, there is no signal from a global controller because *JOINT* does not use any bus for data transfers. 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.

In the buffer-based dataflow where all processing blocks are realized as hardware logics, $R_\mathrm{BC}, W_\mathrm{BC}$ and $\mathrm{END}_R_\mathrm{BC}$ correspond to $\mathrm{start_read}, \mathrm{start_write}$ and $\mathrm{stop_read}$, respectively. In case $f(i)$ is mapped to a processor, L_i is replaced with the execution time of the program $p[f(i)]$. In addition,

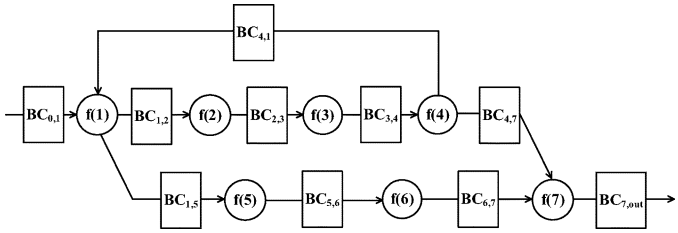


Fig. 12. Buffer-based dataflow constructed from the dataflow of Fig. 1.

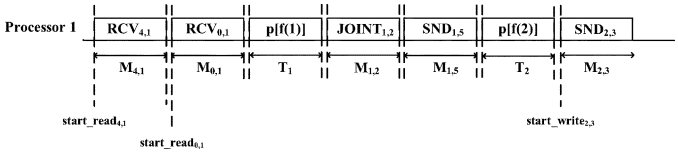


Fig. 13. Execution timing when $f(1)$ and $f(2)$ of Fig. 12 are mapped to processor 1.

$\mathrm{start_write}_{i,j}$ is delayed to (end of program $p[f(i)]+1$). Therefore, (1) changes as follows:

$$\mathrm{start_write}_{i,j} = T_i + \mathrm{start}_i + 1 \tag{6}$$

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 (1)–(4), (6), Section III-B compares the case in which processing blocks are implemented as hardwares with the case where processing blocks are mapped to a processor architecture.

B. Mapping Effects

Fig. 12 is the buffer-based dataflow constructed from the dataflow of Fig. 1 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 follows:

$$\begin{aligned} &\mathrm{start_write}_{2,3} \geq \mathrm{start_read}_{1,2} + L_2, \\ &\mathrm{start_read}_{1,2} > \mathrm{start_write}_{1,2}, \\ &\mathrm{start_write}_{1,2} > \max(\mathrm{start_read}_{4,1}, \mathrm{start_read}_{0,1}) + L_1. \end{aligned} \tag{7}$$

From (7), the minimum $\mathrm{start_write}_{2,3}$ for the hardware realization is given by

$$\begin{aligned} &\min(\mathrm{start_write}_{2,3}) \text{ for hardware} \\ &\quad = L_2 + \min(\mathrm{start_read}_{1,2}) + 1 \\ &\quad = \max(\mathrm{start_read}_{4,1}, \mathrm{start_read}_{0,1}) + L_1 + L_2 + 3. \end{aligned} \tag{8}$$

Fig. 13 shows the execution timing from the input of $f(1)$ to the output of $f(2)$ when $f(1)$ and $f(2)$ of Fig. 12 are mapped to processor 1. In Fig. 13, since two successive processing blocks, $f(1)$ and $f(2)$, are mapped to the same processor, $\mathrm{JOINT}_{1,2}$

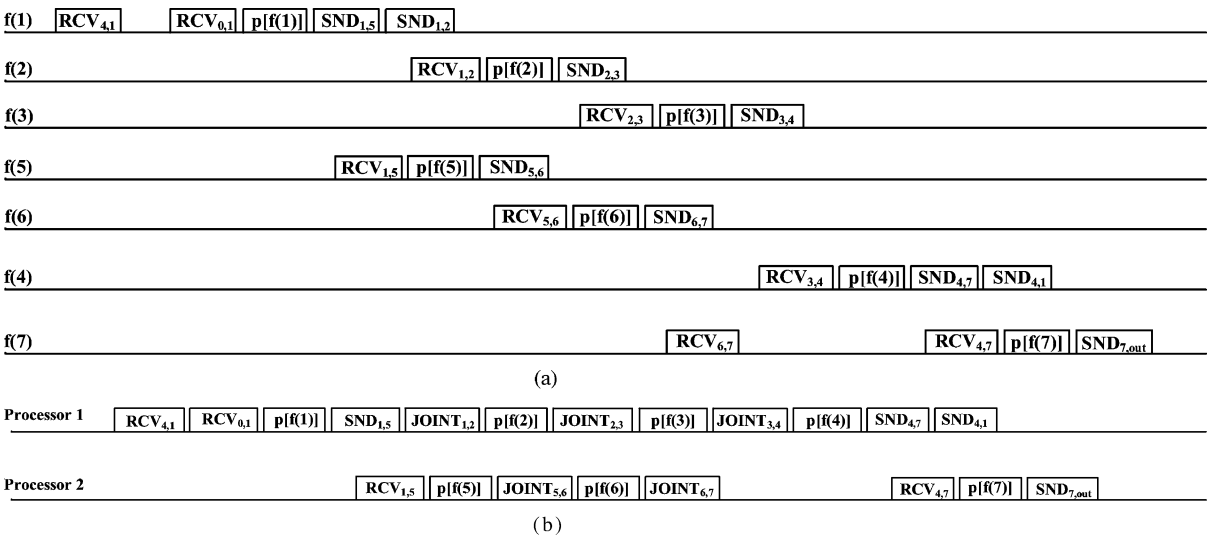


Fig. 14. Execution timing representation to map processing blocks to processors. (a) Execution timing for the mapping of processing blocks in Fig. 12. (b) Execution timing when processing blocks of Fig. 14(a) are mapped to processors.

is used for the data transfer between $f(1)$ and $f(2)$. From the execution timing, the minimum $\mathrm{start_write}_{2,3}$ for processor 1 is given by

$$\begin{aligned}&\min(\mathrm{start_write}_{2,3}) \text{ for processor}\\&\quad = \mathrm{start_read}_{4,1} + T_1 + T_2 + M_{4,1} + M_{0,1} + M_{1,5}\\&\qquad + M_{1,2} + 6. \end{aligned} \tag{9}$$

Compared with (8), all previous execution times (including data transfers) reflect to $\mathrm{start_write}_{2,3}$. 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 $\mathrm{start_read}_{0,1}$ before $RCV_{4,1}$ is completed. The timing mismatch may lead to the wrong operations of the programs running on processor 1. In Section IV, we discuss the synchronization issue and propose a solution.

IV. Resource Utilization and Synchronization

A. 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. 12. Fig. 14 shows the execution timing representation of processing blocks.

In Fig. 14(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 $\mathrm{f}(i)$, EXE_j: execution times of $f(j)$]

$$\begin{aligned}&\max(\mathrm{end}(\mathrm{EXE}_i), \mathrm{end}(\mathrm{EXE}_j)) - \min(\mathrm{start}(\mathrm{EXE}_i),\\&\quad \mathrm{start}(\mathrm{EXE}_j)) > (\mathrm{end}(\mathrm{EXE}_i) - \mathrm{start}(\mathrm{EXE}_i))\\&\qquad + (\mathrm{end}(\mathrm{EXE}_j) - \mathrm{start}(\mathrm{EXE}_j)) \end{aligned} \tag{10}$$

where $\mathrm{start}(\mathrm{EXE})$ and $\mathrm{end}(\mathrm{EXE})$ correspond to the start and end times of *SND*, *RCV* and functional execution, respectively. Equation (10) represents that two execution times are non-overlapped if the difference between the maximum $\mathrm{end}(\mathrm{EXE})$ and the minimum $\mathrm{start}(\mathrm{EXE})$ 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 $p[f(i)]$), $\mathrm{start}(\mathrm{EXE})$ and $\mathrm{end}(\mathrm{EXE})$ of (10) are translated to

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

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

The number of processors is further reduced by using *JOINT*. In Fig. 14(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 $\mathrm{JOINT}_{1,2}$, $f(1)$, 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. 14(b). If the number of processors in the target platform is 1, the execution timing of Fig. 14(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

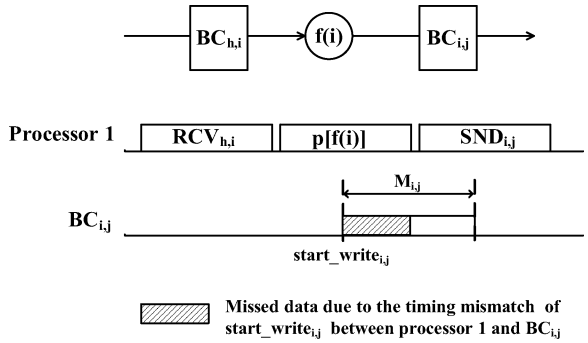


Fig. 15. Processor 1 receives $\mathrm{start_write}_{i,j}$ during the execution of $p[f(i)]$.

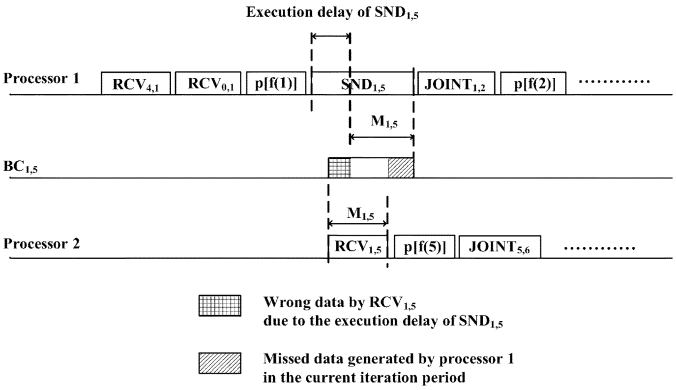


Fig. 16. When the execution time of $SND_{1,5}$ takes longer than $M_{1,5}$ in Fig. 14(b).

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. 15.

In Fig. 15, processor 1 receives $\mathrm{start_write}_{i,j}$ while $p[f(i)]$ is running. Processor 1 starts running $SND_{i,j}$ when $p[f(i)]$ is finished. However, $\mathrm{BC}_{i,j}$ starts receiving data from processor 1 as soon as it gets $\mathrm{start_write}_{i,j}$ from a global controller. Therefore, $\mathrm{BC}_{i,j}$ misses data due to the timing mismatch of $\mathrm{start_write}_{i,j}$ between processor 1 and $\mathrm{BC}_{i,j}$.

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 $p[f(i)]$. Furthermore, since *RCV*, *SND* and JOINT are also programs running on processors, the execution times of *RCV*, *SND* and *JOINT* must be estimated. In Fig. 14(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 $\mathrm{BC}_{1,5}$. Therefore, processor 2 does not execute its functions (i.e., $p[f(5)], p[f(6)]$ and $p[f(7)]$) correctly. Fig. 16 illustrates the case.

In Fig. 16, 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 $RCV_{1,5}$ is completed. Therefore, processor 2 receives wrong data from $\mathrm{BC}_{1,5}$ 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 $\mathrm{BC}_{1,5}$. In order to determine the ends of data transfers, the execution times of $SND_{i,j}, RCV_{i,j}$ and $\mathrm{JOINT}_{i,j}$ are estimated as their maximum values (i.e., $T_{\max}(SND_{i,j}), T_{\max}(RCV_{i,j})$ and

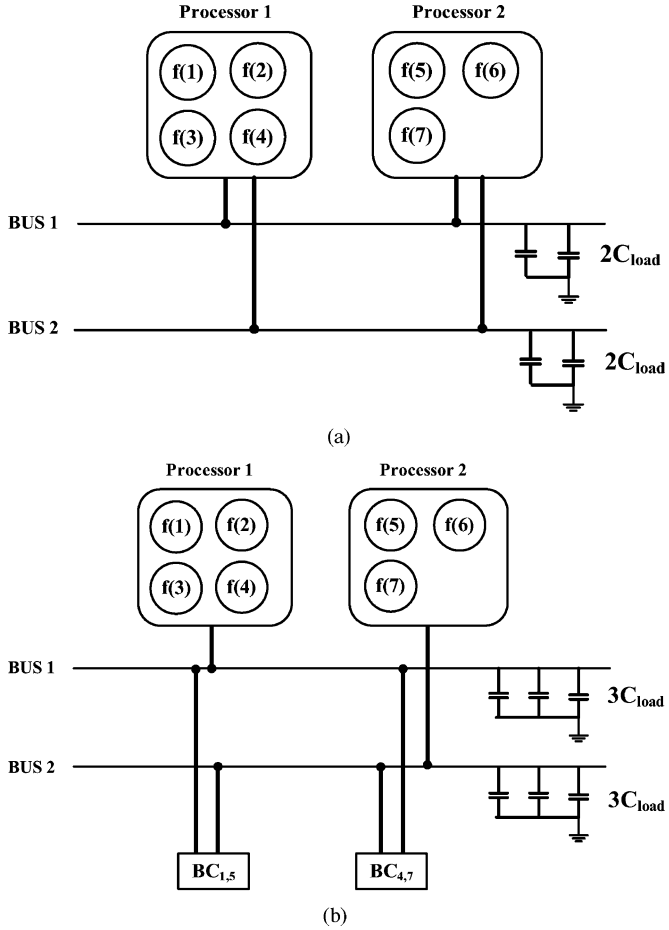


Fig. 17. Considering power consumption for interconnection between processors when the processing blocks of Fig. 14(a) are mapped to two processors. (a) Mapping processing blocks without buffer controllers. (b) Mapping processing blocks with buffer controllers.

$T_{\max}(\mathrm{JOINT}_{i,j})$). In addition, the estimated execution times are reflected to *start* and *stop* signals as follows:

$$\begin{aligned}
\mathrm{stop_write}_{i,j} &= \mathrm{start_write}_{i,j} + T_{\max}(SND_{i,j}) \\
\mathrm{start_read}_{i,j} &= \mathrm{stop_write}_{i,j} + 1 \\
\mathrm{stop_read}_{i,j} &= \mathrm{start_read}_{i,j} + T_{\max}(RCV_{i,j}) \\
\mathrm{stop_joint}_{i,j} &= \mathrm{start_joint}_{i,j} + T_{\max}(\mathrm{JOINT}_{i,j}) \quad (11)
\end{aligned}$$

where $\mathrm{start_write}_{i,j}$ and $\mathrm{start_read}_{i,j}$ are generated by a global controller and other signals are not generated but estimated to determine start_write and start_read. If $SND_{i,k}$ starts when $\mathrm{JOINT}_{i,j}$ is completed, $\mathrm{start_write}_{i,k}$ is $(\mathrm{stop_joint}_{i,j} + 1)$. However, in case actual execution times take longer than estimated, the signals of (11) fail to synchronize data transfers. In Section IV-C, we propose a scheme to synchronize data transfers by revising the signals of (11).

B. Power Consumption and Bus Utilization of Mapping

Fig. 17 illustrates the interconnection between processors when the processing blocks of Fig. 14(a) are mapped to two processors. C_{load} represents the port loading capacitance of a bus. In Fig. 17(a), buffer controllers are not used for mapping. In Fig. 17(b), two buffer controllers $\mathrm{BC}_{1,5}$ and $\mathrm{BC}_{4,7}$ are attached

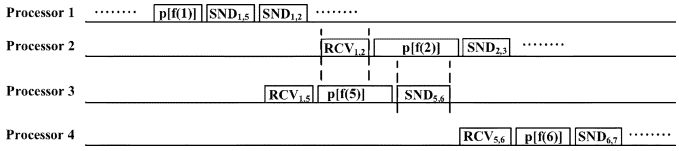


Fig. 18. Estimated execution times when processing blocks of Fig. 12 are mapped to four processors.

to the buses for synchronizing data transfers between $f(1)$ and $f(5)$ and between $f(4)$ and $f(7)$, respectively. However, these buffer controllers increase the port loading capacitance of the buses and can lead to additional dynamic power dissipation.

For example, when $f(1)$ (mapped to processor 1) sends data to $f(5)$ (mapped to processor 2), the dynamic energy consumptions of the mapping shown in Fig. 17(a) and (b) are $2C_{\text{load}}V^2fM_{1,5}$ and $6C_{\text{load}}V^2fM_{1,5}$, respectively, where C_{load} is the load capacitance of the buses, f is the operating frequency of the buses and V is the supply voltage. In Fig. 17(b), since the data transfer from $f(1)$ to $f(5)$ is realized as both $SND_{1,5}$ and $RCV_{1,5}$, the port loading capacitance, $3C_{\text{load}}$, doubly contributes to the energy consumption.

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. 18 shows the execution timing when processing blocks of Fig. 12 are mapped to four processors.

In Fig. 18, 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 $\text{BC}_{1,2}$. In addition, $p[f(2)]$ of processor 2 does not use the bus for $SND_{5,6}$ of processor 3. Therefore, processors 2 and 3 share the same bus because the execution times of *SND* and *RCV* between them are non-overlapped.

However, the execution times of Fig. 18 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.

C. 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 $\text{initiate_start_write}_{i,j}$ signal to inform a global controller that $p[f(i)]$ is completed. As soon as the global controller receives $\text{initiate_start_write}_{i,j}$, it sends $\text{start_write}_{i,j}$ to both the processor and the buffer controller $\text{BC}_{i,j}$. Thus, the global controller needs the input port for receiving $\text{initiate_start_write}_{i,j}$ from the processor. When

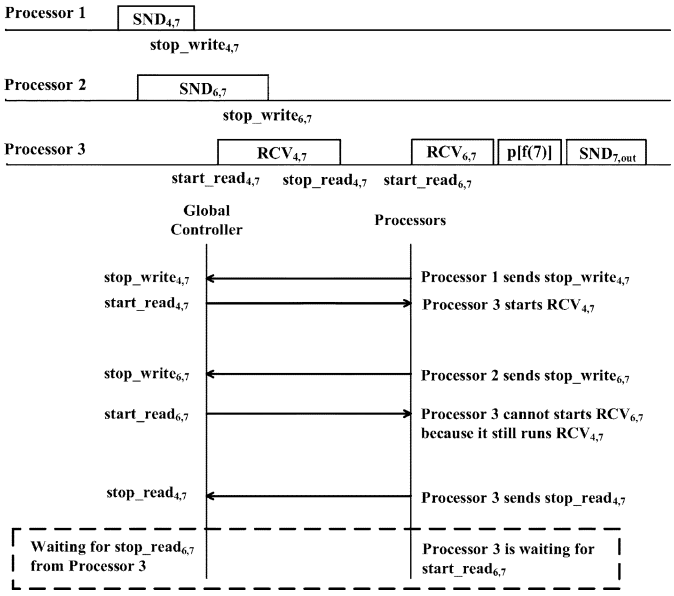


Fig. 19. Deadlock problem when start_read of multiple fan-ins is determined by stop_write.

the processor initiation scheme is applied, $\text{start_write}_{i,j}$ is changed to

$$\begin{aligned}\text{start_write}_{i,j} &= \text{initiate_start_write}_{i,j} + 1 \\ &= \text{start_of_}p[f(i)] + T_i(\text{actual}) + 1\end{aligned}$$

where $\text{start_of_}p[f(i)]$ 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 (i.e., $\text{initiate_start_write}, \text{stop_write}, \text{stop_read}$ and stop_joint) are used to determine *start* signals for data transfers. A global controller generates *start* signals when one cycle passes after receiving *stop* signals from processors. Thus, the number of *stop* signals of a global controller corresponds to the additional cycles incurred by using the processor initiation scheme.

However, in the case of $RCV_{i,j}$ in multiple fan-ins, if a global controller generates $\text{start_read}_{i,j}$ only by referring to $\text{stop_write}_{i,j}$ (i.e., $\text{start_read}_{i,j} = \text{stop_write}_{i,j} + 1$), the *deadlock* condition may occur. Fig. 19 illustrates the deadlock problem when $f(4), f(6)$ and $f(7)$ of Fig. 12 are mapped to three different processors.

As shown in Fig. 19, when $\text{stop_read}_{4,7} > \text{stop_write}_{6,7}$, $\text{start_read}_{6,7}$ $(= \text{stop_write}_{6,7} + 1)$ creates the deadlock problem indicated as the dotted line of Fig. 19. In order to prevent the deadlock problem, the global controller generates $\text{start_read}_{6,7}$ when it receives both $\text{stop_write}_{6,7}$ and $\text{stop_read}_{4,7}$.

Fig. 20 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. In the proposed scheme, a global controller is implemented in hardware logic, and its cost in terms of the number of ports is related to the number of the processors used for mapping. As shown in

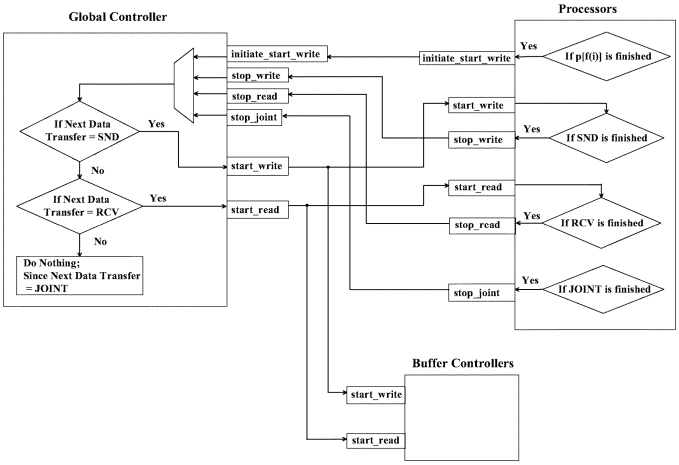


Fig. 20. Illustration of the signaling for processor initiation scheme.

Fig. 20, six input/output ports of a global controller are dedicated to one processor because the ordering of *start* and *stop* signals is fully sequentialized within a single processor. The number of ports in a global controller is therefore 6 times the number of processors used for mapping.

D. Processor-Hardware Coexistence of Mapping

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. 21.

Fig. 21 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 $\mathrm{BC}_{1,2}$. Thus, the execution time of $R_{1,2}$ takes from $\mathrm{start_read}_{1,2}$ to $\mathrm{stop_read}_{1,2}$. In Fig. 21(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. 21(a), $R_{1,2}$ *partially* receives the data which $SND_{1,2}$ writes to $\mathrm{BC}_{1,2}$ in the current iteration period because $R_{1,2}$ finishes before $SND_{1,2}$ ends. 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 $\mathrm{stop_read}_{2,3} < \mathrm{stop_write}_{2,3}$ as shown in Fig. 21(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* start when the writing of the buffer controllers are completely done. Therefore, even though $f(2)$ is implemented as a hardware logic, $\mathrm{start_read}_{1,2}$ is determined not by (2) but by (11) (i.e., $\mathrm{start_read}_{1,2} = \mathrm{stop_write}_{1,2} + 1$).

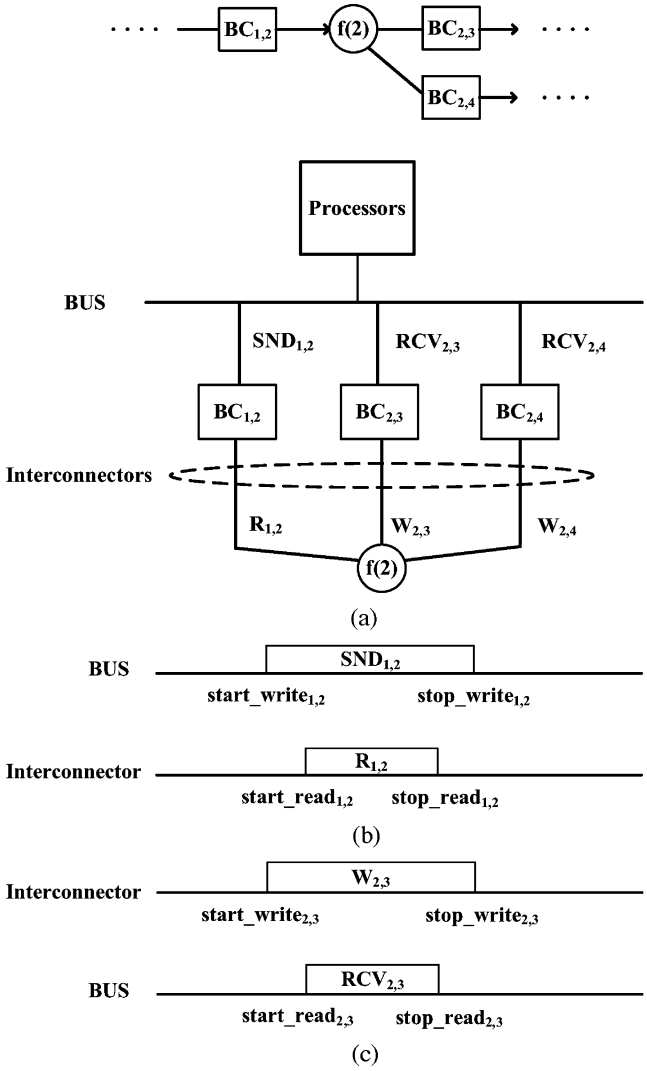


Fig. 21. Timing mismatch problem in the processor-hardware coexistence of mapping. (a) Only $f(2)$ is implemented as a hardware. (b) When the interconnect speed is faster than BUS speed. (c) When BUS speed is faster than the interconnect speed.

E. Mapping Algorithm

From the execution timing representation such as Fig. 14(a), our proposed mapping algorithm creates the mapped partition. Fig. 22 shows the overall procedure of the proposed mapping algorithm.

Algorithm 1 Mapping processing blocks to processors

1: //$G = (V, E)$: a buffer-based dataflow,
2: //$V =$ the set of processing blocks, $E =$ the set of buffer controllers.
3: //$V = V_p \bigcup V_h$,
4: //$V_p =$ the set of processing blocks mapped to processors,
5: //$V_h =$ the set of processing blocks realized as hardware logics.
6: //$E_{i,j} =$ the buffer controller between V_i and V_j.
7: //np = number of processors (its initial value is 0.)

```
8: //processor[np] = array of the processors which V_i and V_j are mapped to.
9: //non_overlap(a,b) = check if execution times of a and b are non-overlapped,
10: //non_overlap(a,b) corresponds to (10).
11: //non_overlap_BUS(a,b) = check if bus operations of a and b are non-overlapped.
12: //V_i, V_j ∉ V_h and V_i, V_j ∈ V_p
13:
14: //Mapping V_i to processors
15: MAPPING(V_i)
16:
17: for all (V_i) do
18:     for all (V_j ∈ processor[np]) do
19:       if (non_overlap(V_i, V_j)) == TRUE then
20:           add V_i to processor[np];
21:        else
22:          if (E_{i,j} ∈ E) then
23:            remove SND_{i,j} from V_i;
24:            remove RCV_{i,j} from V_j;
25:            //Check if V_i is mapped to processor[np] by using JOINT_{i,j}
26:            if(non_overlap(V_i, V_j) == TRUE) then
27:              add V_i to processor[np];
28:              add JOINT_{i,j} to processor[np];
29:             else
30:              //In case of overlap, restore V_i and V_j
31:               add SND_{i,j} to V_i
32:               add RCV_{i,j} to V_j
33:              //V_i is assigned to new processor.
34:               add V_i to processor[np++];
35:             end if
36:           else
37:             //Since E_{i,j} does not exist,
38:             //V_i is assigned to new processor.
39:              add V_i to processor[np++];
40:            end if
41:          end if
42:        end for
43: end for
44: //Bus sharing among processors
45: for all (V_i ∈ processor[np], V_j ∈ processor[np+1]) do
46:     //If SNDs and RCVs are non-overlapped between two processors,
47:     //the processors share the same bus.
48:     if (non_overlap_BUS(V_i, V_j) == TRUE) then
49:       assign BUS of processor[np] to processor[np+1];
50:       remove BUS of processor[np+1];
51:     end if
52:     np++;
53: end for
```

In Fig. 22, "Mapping processing blocks to processors" box corresponds to Algorithm 1. When the number of processors provided by a target platform is smaller than that of the mapping found by Algorithm 1, 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 1 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. 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.

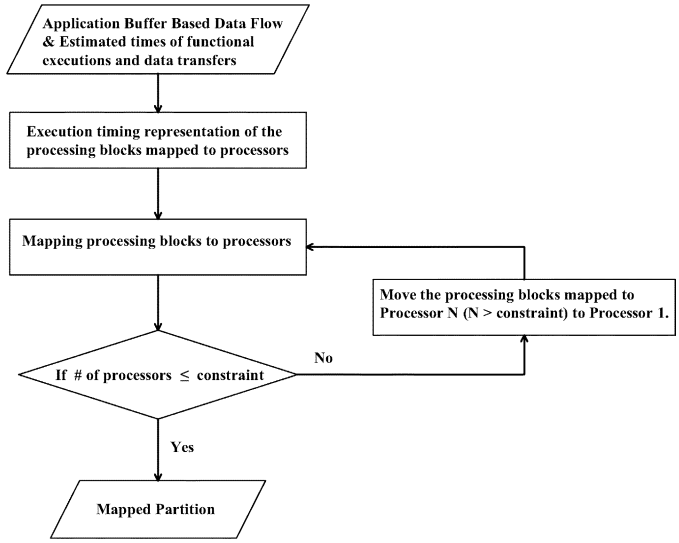


Fig. 22. Proposed mapping algorithm.

F. Template Code Generation

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 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 $p[f(i)]$, $\mathrm{JOINT}_{i,j}$ and $p[f(j)]$ run on processor 1, the output argument of $p[f(i)]$ is passed to the input argument of $\mathrm{JOINT}_{i,j}$ because the subscript i of $\mathrm{JOINT}_{i,j}$ represents $p[f(i)]$. In the same way, the output argument of $\mathrm{JOINT}_{i,j}$ is passed to the input argument of $p[f(j)]$ because the subscript j of $\mathrm{JOINT}_{i,j}$ 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. 23 shows the procedure to generate the template code.

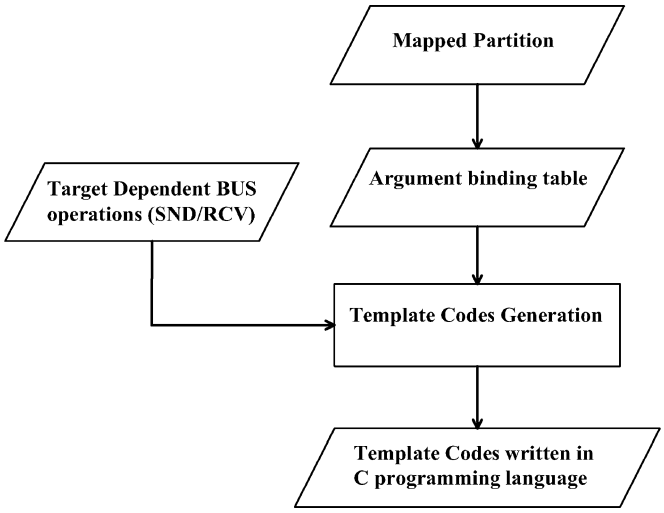


Fig. 23. Procedure of template codes generation.

TABLE II
BUFFER CONTROLLER PARAMETERS OF FIG. 12

	nr	D	M	$start_write$	$start_read$
$BC_{0,1}$	1	0	16	8	9
$BC_{1,5}$	1	0	16	37	38
$BC_{5,6}$	1	0	32	49	50
$BC_{1,2}$	1	0	24	51	52
$BC_{6,7}$	1	0	32	59	60
$BC_{2,3}$	1	0	24	67	68
$BC_{3,4}$	1	0	16	83	84
$BC_{4,7}$	1	0	16	99	100
$BC_{4,1}$	1	0	8	102	103
$BC_{7,out}$	1	0	30	115	116

TABLE III
ESTIMATED $T_i(\max)$ OF THE PROCESSING BLOCKS IN FIG. 12

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

V. 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. 22 is realized through the C programming language. The template code generation in Fig. 23 is validated in the Xilinx ISE environment.

A. Evaluation of the Proposed Mapping Algorithm

Our evaluation uses the buffer-based dataflow of Fig. 12 and targets the platform consisting of multi-core processors at 400 MHz and buses at 100 MHz. Table II lists the buffer controller parameters of Fig. 12.

The estimated execution times of processing blocks are shown in Table III. In addition, the execution times of *SND*s and *RCV*s are also estimated as twice as the corresponding M values in Table II. Fig. 24 shows that mapping is changed according to the estimation of $SND_{1,2}$.

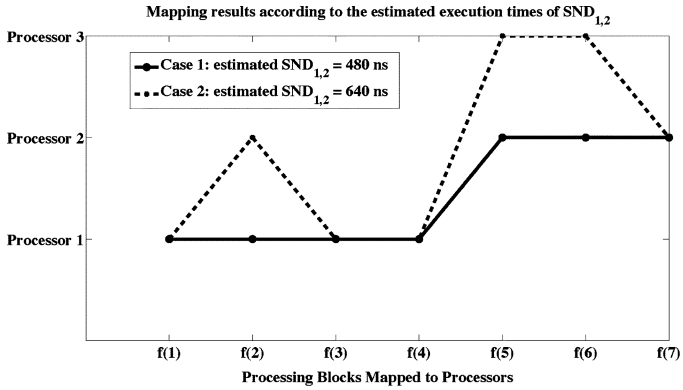


Fig. 24. Mapping cases according to the estimation of $SND_{1,2}$.

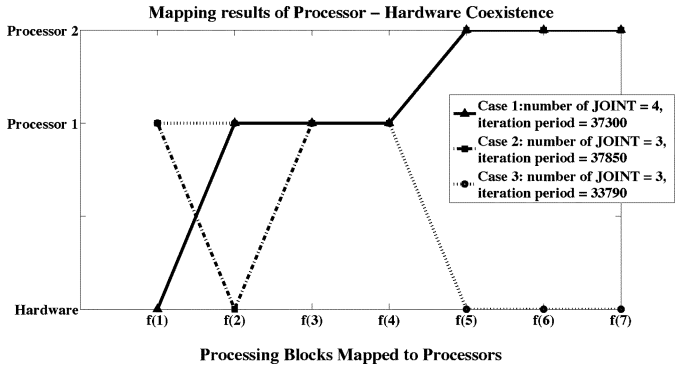


Fig. 25. Mapping results of processor-hardware coexistence.

In Fig. 24, if the actual execution time of $SND_{1,2}$ is always within 480 ns, the mapping result corresponds to case 1. In this case, $\mathrm{BC}_{1,2}, \mathrm{BC}_{2,3}, \mathrm{BC}_{3,4}, \mathrm{BC}_{5,6}$, and $\mathrm{BC}_{6,7}$ are mapped to *JOINT*s. Thus, the total number of buffer controllers to be realized as hardwares is 5 ($=$ 10 BCs in Table II -5 *JOINT*s). When the actual execution time of $SND_{1,2}$ vary from 480 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 cases 1 to 2. Case 2 uses one more processor to have the same iteration period with case 1. If $SND_{1,2}$ is observed as 480 ns in most iteration periods and it is 600 ns in one particular iteration period, then case 2 uses one more processor only for that particular iteration period. In this case, the maximally estimated execution time may waste the system resource. In addition, since the mapping of case 2 has 2 *JOINT*s, the total number of buffer controllers implemented as hardware is 8. If some processing blocks are realized as hardware logics, the iteration period is further reduced. Fig. 25 shows the mapping results when some of processing blocks in case 1 of Fig. 24 are realized as hardware logics.

In Fig. 25, 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

TABLE IV
NUMBER OF PROCESSORS REQUIRED FOR THE MAPPED PARTITION WHEN THE PROPOSED MAPPING METHODOLOGY IS APPLIED TO THE APPLICATION DATAFLOW GRAPHS (DFGS) IN [12], [15], AND [16]

	# of processing blocks	# of processors for mapping	Total execution time	Buffer memory
SIRF [12]	6	2	8904 ns	132 bytes
TX MC-CDMA [16]	7	2	47684 ns	96 bytes
IPv4 [15]	13	3	25390 ns	220 bytes
RX MC-CDMA [16]	13	5	280958 ns	29624 bytes

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.

For a target platform consisting of 400-MHz multi-core processors and 100-MHz buses, we apply the proposed mapping methodology to various applications such as sample importance resample (SIRF) [12] IPv4 forwarding [15], multi-carrier code division multiple access (MC-CDMA) transmitter and receiver [16]. In the mapping, the cycles of sending and receiving data are estimated as $2M$, where M is the block size. Table IV summarizes the results we obtained.

In Table IV, the number of processors in both SIRF and TX MC-CDMA is determined by the number of data paths from the input to output. SIRF has a single input, a single output, and two data paths from the input to output. Similar to Fig. 12, the number of processors in SIRF is two. TX MC-CDMA also has two data paths because it has a single input and two outputs. Thus, the number of processors in TX MC-CDMA is also two. However, in the cases of IPv4 and RX MC-CDMA, the number of processors are determined by the number of processing blocks having multiple fan-ins/fan-outs because the processing blocks having multiple fan-ins/fan-outs are mapped to different processors in order for their execution times to be non-overlapped with each other. Moreover, as the number of processors for mapping increases, more buffer controllers are required for data transfers between processors, and thus the buffer memory size also increases, as shown in Table IV.

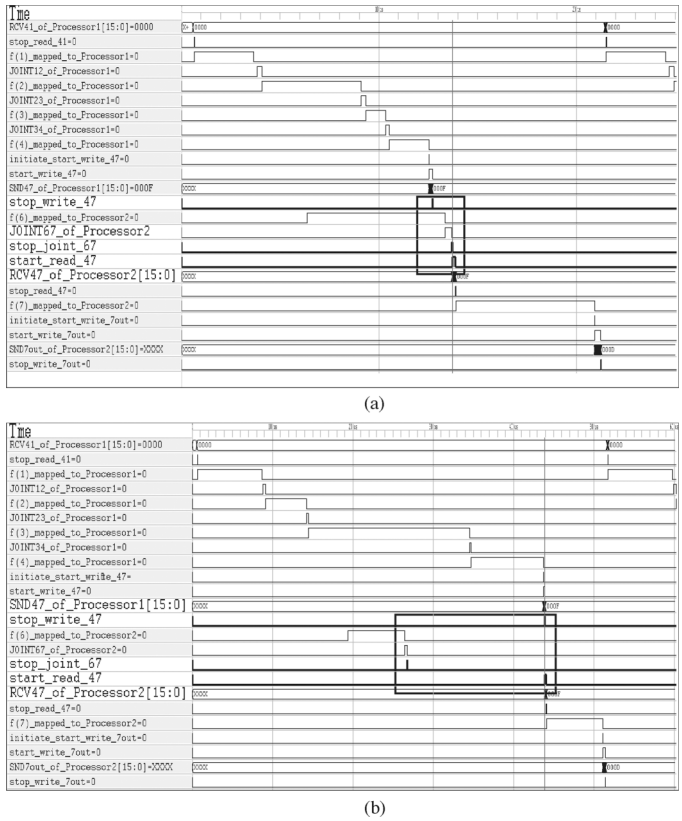


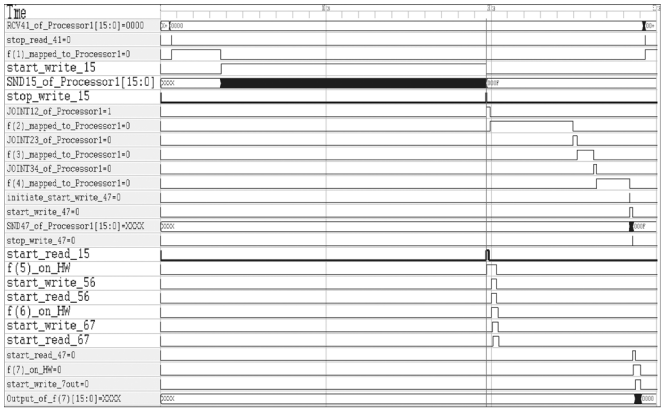
Fig. 26. Simulation results of processor initiation scheme. (a) $\mathrm{stop_joint}_{6,7}$ determines $\mathrm{start_read}_{4,7}$. (b) $\mathrm{stop_write}_{4,7}$ determines $\mathrm{start_read}_{4,7}$.

B. Processors Only

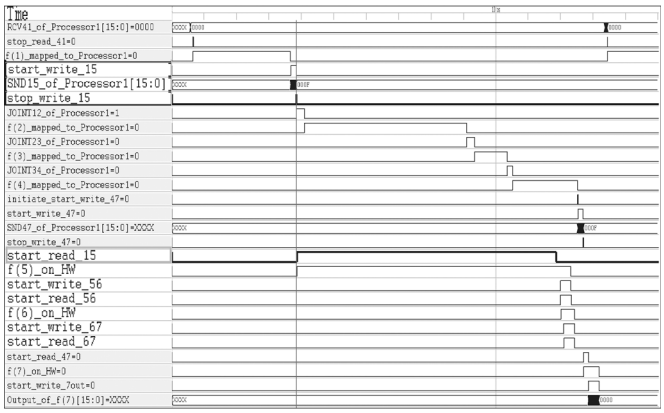
Fig. 26 shows the simulation results of the processor initiation scheme applied to the mapped partition of "case 1" in Fig. 24. In Fig. 26, a global controller generates *start* signals, and processors sends *stop* signals to the global controller. Due to the variation of execution times, $\mathrm{start_read}_{4,7}$ is enabled when either $(\mathrm{stop_joint}_{6,7} + 1)$ in Fig. 26(a) or $(\mathrm{stop_write}_{4,7} + 1)$ in Fig. 26(b). For the correct operation in both cases, the global controller generates $\mathrm{start_read}_{4,7}$ when it receives $\mathrm{stop_write}_{4,7}$ and $\mathrm{stop_joint}_{6,7}$ from processor 1 and processor 2. As a result, Fig. 26(b) shows that processor 2 starts running $RCV_{4,7}$ when it finishes $\mathrm{JOINT}_{6,7}$. In Fig. 26(b), processor 2 begins running $RCV_{4,7}$ when processor 1 ends $SND_{4,7}$.

C. Processor-Hardware Coexistence

Fig. 27 shows the simulation results when case 3 of Fig. 25 is synthesized. In Fig. 27, the timings of $\mathrm{start_write}_{5,6}$, $\mathrm{start_read}_{5,6}$, $\mathrm{start_write}_{6,7}$, and $\mathrm{start_read}_{6,7}$ 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 $\mathrm{start_write}_{1,5}$ and $\mathrm{start_read}_{1,5}$ ($\mathrm{start_write}_{4,7}$ and $\mathrm{start_read}_{4,7}$) are non-overlapped because the reading port of $\mathrm{BC}_{1,5}(\mathrm{BC}_{4,7})$ is connected to a processor bus and its writing port is connected to a hardware interconnect. Fig. 27(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



(a)



(b)

Fig. 27. Simulation results of processor-hardware coexistence. (a) When $SND_{1,5}$ takes longer than $R_{1,5}$. (b) When $R_{1,5}$ takes longer than $SND_{1,5}$.

TABLE V
ARGUMENT BINDING TABLE FOR PROCESSOR 1 OF CASE 3 IN FIG. 25

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

speed. In contrast, Fig. 27(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 $\text{start_read}_{1,5} = (\text{stop_write}_{1,5}+1)$ for the correct data transfer in the timing mismatch between a processor bus and a hardware interconnect.

D. Template Code Generation

From the mapped partition created by the mapping algorithm of Fig. 22, our methodology establishes the argument binding table to generate template codes. Table V lists the argument binding table for processor 1 of "case 3" in Fig. 25.

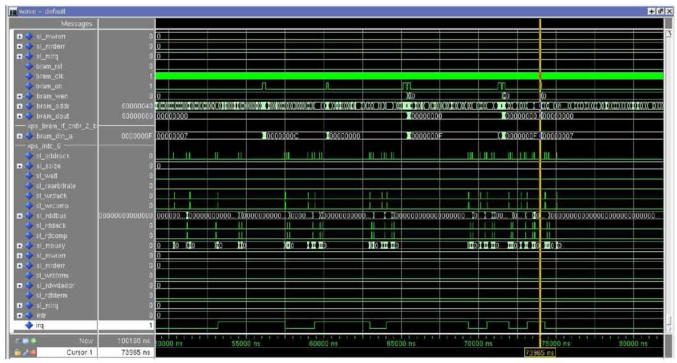
```
int* RCV(unsigned int size_of_M, u32 address_offset)
{
    XCache_InvalidateDCacheRange_revised(XPAR_XPS_BRAM_IF_CNTLR_2_BASEADDR\
            + address_offset, size_of_M*sizeof(int));
    memcpy(return_RCV, (int*)(XPAR_XPS_BRAM_IF_CNTLR_2_BASEADDR\
            + address_offset), size_of_M*sizeof(int));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

```
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)



(c)

Fig. 28. Generated template codes for Xilinx Virtex-5 FXT. (a) Target dependent *RCV* and *SND* subroutines. (b) Argument binding subroutine. (c) Simulation result in ModelSim 6.3c of Xilinx ISE 10.1

In Table V, "dependency" represents the direction of argument passing between 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. 28.

Fig. 28(a) shows *RCV* and *SND* primitive templates which are generated for the Xilinx Virtex-5 FXT target platform. Fig. 28(b) is the automatically generated template code based on Table V. Compared with the subroutines of Fig. 28(a), the argument binding subroutine does not contain the target specific library. When a target platform is changed, only *RCV* and *SND* primitive templates need to be rewritten. For example, in order to synthesize the template codes in Fig. 28(b) onto recent Virtex-5 devices, the primitive templates in Fig. 28(a) should be changed for soft IP cores in Virtex-5 because new Virtex-5

devices do not support hard-wired processors any more [1]. In Fig. 28(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.

VI. Conclusion

In this paper, we 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. Through the evaluation, we demonstrated that the mapping by our proposed methodology is successfully working even with misestimated execution times.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

[14] R. Lauwereins, M. Engels, M. Ade, and J. A. Peperstraete, "Grape-II: A system-level prototyping environment for DSP applications," *IEEE Computers*, vol. 28, no. 2, pp. 35–43, Feb. 1995.

[15] N. R. Satish, K. Ravindran, and K. Keutzer, "Scheduling task dependence graphs with vairable task execution times onto heterogeneous multiprocessors," presented at the Presentation Slides Embedded Syst. Week, Atlanta, GA, Oct. 2008.

[16] J. Delorme, "An automatic design flow for mapping application onto a 2D mesh NoC architecture," *Integr. Circuit Syst. Des. Power Tim. Model., Optim. Simulation, LNCS*, vol. 4644, pp. 31–42, 2007.

Woohyung Chun (M'10) received the B.S. and M.S. degrees in electronic and electrical engineering from Hongik University, Seoul, Korea, in 1997 and 1999, respectively, and the Ph.D. degree in electrical and computer engineering from Stony Brook University-State University of New York, Stony Brook, NY, in 2009.

He is currently with System LSI Division, Samsung Electronics, Kiheung, Korea, where he is developing the architecture of mobile application SoCs. He also worked at Corecess and Comtec systems as an Embedded System Software Engineer. His research interests include reconfigurable SoC architecture, VLSI signal processing, hardware-software co-design, and low power VLSI design for mobile multimedia systems.

Sungroh Yoon (S'99–M'06) received the B.S. degree in electrical engineering from Seoul National University, Seoul, Korea, in 1996, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 2002 and 2006, respectively.

From 2006 to 2007, he was with Intel Corporation, Santa Clara, CA, where he participated in developing Atom and Core i7 microprocessors. Previously he held research positions with Stanford University, Stanford, CA, and Synopsys Inc., Mountain View, CA. He is currently an Assistant Professor with the School of Electrical Engineering, Korea University, Seoul, Korea. His recent research interests include emerging computer systems and biocomputation.

Sangjin Hong (SM'05) received the B.S. and M.S. degrees from the University of California, Berkeley, CA, and the Ph.D. degree from the University of Michigan, Ann Arbor, all in electrical engineering and computer science.

He is currently with the Department of Electrical and Computer Engineering, Stony Brook University—State University of New York, Stony Brook, NY. Before joining Stony Brook University, he worked as a Systems Engineer with the Computer Systems Division, Ford Aerospace Corporation. He also worked as a Technical Consultant with Samsung Electronics, Korea. His current research interests include the areas of low power VLSI design of multimedia wireless communications and digital signal processing systems, reconfigurable SoC design and optimization, VLSI signal processing, and low-complexity digital circuits.

Prof. Hong served on numerous Technical Program Committees for IEEE conferences.