Stony Brook University

High Performance Partition Based Reconfigurable Platform for Multiple Concurrent Applications

A Dissertation Presented

by

Qi Qi

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Electrical Engineering

Stony Brook University

December 2015

Stony Brook University

The Graduate School

Qi Qi

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree,

hereby recommend acceptance of this report.

Sangjin Hong, Advisor

Professor, Department of Electrical & Computer Engineering

Peter Milder, Co-advisor

Assistant Professor, Department of Electrical & Computer Engineering

Leon Shterengas

Associate Professor, Department of Electrical & Computer Engineering

Hongshik Ahn

Professor, Department of Applied Mathematics and Statistics

This dissertation is accepted by the Graduate School

Charles Taber

Dean of the Graduate School

Abstract of the Dissertation

High Performance Partition Based Reconfigurable Platform for Multiple Concurrent Applications

by

Qi Qi

Doctor of Philosophy

in

Electrical Engineering

Stony Brook University

2015

Reconfigurable architectures, combining the benefits of flexibility and high performance, are suitable for embedded digital signal processing. However, it is critical to bridge the gap between application algorithms and their implementation. Further, low power design is critical, but it is difficult to migrate an existing algorithm into a data-centric application that is represented as a dataflow and to map this to a reconfigurable architecture. Thus, such a reconfigurable platform mapped from application dataflow graphs and an architecture-aware optimization algorithm become necessary.

This thesis proposes an efficient algorithm to optimize the clock frequencies of the processing elements in a reconfigurable architecture, finding the frequency configuration that minimizes the power consumed while meeting the application's timing

requirements. The algorithm takes as input a dataflow representing the intended application characteristics and the required timing constraint information, and optimizes the frequency configuration by dynamically exploiting correlation between frequencies and iteration time in consideration of parameter variation to avoid data collision or loss. Then it proposes a novel hardware reconfigurable platform divided into multiple partitions, where each partition is entirely buffer-centered consisting of a large number of heterogeneous processing elements operating with buffers through reconfigurable interconnect, to execute multiple concurrent applications. Depending on performance requirements, an application migrated from a dataflow graph can be mapped to more than one partition interacting through bridge buffers. To accommodate asynchronous clock configuration, this platform uses flexible hierarchical controller design. The controller considers execution flow and structural configuration separately but collaboratively for dynamic reconfiguration of the dataflow. The use of a tree structured controller makes the design scalable.

We model the proposed reconfigurable platform and hierarchical controller in SystemC, and implement the frequency optimization algorithm to provide clock frequencies that minimize power consumption to such platform. Experiments shows that this algorithm achieves power consumption that is typically equal to a simulated annealing-based method, while running 100 times faster on average. The SystemC simulations demonstrate the controller is able to load and execute applications with dynamic reconfiguration. Therefore, the system can map multiple processing elements onto a single core and switch between them during run-time.

All glory to, Kaylee

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my advisor, Sangjin Hong, for his help and guidance to my work during my studies.

I also thank my co-advisor, Peter Milder, for his advice and expertise. I appreciate his discussions and suggestions to my thesis and admire his nice personality.

Thank you to my father, Peidu Qi, for his pure love and support.

Thank you to my auntie and uncle, Ningjie Hu and Qigui Yu, for their encouragement and support.

Especially thank to my husband, Yufei Ren. I would not finish my study without him. These days are such wonderful memory for us.

Last but not least, thank to my daughter, Kaylee Ren, for blooming up and being demanding to me. She attracted me so much attention and made my study harder. I hope I can add another chapter about how to preserve her talent.

Chapter 1

Introduction

1.1 Introduction

Traditional design methods, such as DSP, ASIC, FPGA and Multiprocessor System-on-Chips, often have constraints on performance, reconfigurability and energy efficiency. An architecture taking care of these constraints in real-time signal processing applications is necessary. The DSP processors based sequential von Neumann or Harvard computation style of the programmable processors, support only limited parallelism required by the ever demanding applications, therefore are inherently slow [1]. Custom designed ASIC implementations achieve high speed and low power by exploiting parallelism and control flow. However, they lack reconfigurability in the sense that they are not robust when facing functional and structural change. FPGA has such flexibility with applications, but consumes big overhead due to the high cost of its reconfigurable interconnect. Multiprocessor System-on-Chips uses multiple programmable CPUs, which achieve high throughput by parallelism, but

rely on compiler to efficiently utilize multi-cores. Reconfigurable architectures process with very flexible computing fabrics, combining the merits of flexibility and high performance, are promising alternatives for embedded digital signal processing by demonstrating performance superiority [2] as well as energy efficiency [3].

Reconfigurable architectures are able to make changes to applications and control flow by loading a new function to the processing fabric during runtime. There are some limitations of the reconfigurable structure. Typically, the control structure is complicated and hard to manipulate, therefore, it costs a lot of time to configure the system. This thesis presents a new controller structure which allows more efficient reconfiguration. To rapidly modify the execution of the dataflow, it is necessary to design a controller structure providing an efficient spacial and temporal connectivity for the reconfigurable architecture.

Currently available reconfigurable technologies are classified into three categories: System level FPGAs, Embedded reconfigurable FPGAs and arrays of processing elements [4] - [6]. We focus on the improvement of reconfigurable technology in the arrays of processing elements area. MorphoSys, a well-known application of this type, is a coarse grained integrated reconfigurable system architecture targeted at inherent data-parallelism, high regularity and high throughput requirement applications [7]. It has faster speed compared to FPGAs, and also supports the execution and data load at the same time. RaPiD is able to map the algorithms in DSP applications into a datapath and a controlling program using static reconfiguration [8]. PipeRench is an architecture which is efficient in complex computation without the need of custom hardware design [9]. It has a compiler to configure the system quickly.

The design gap in existing technologies, from the algorithm of advanced applications to their implementation needs to be bridged. An application can be represented as dataflow, consisting of processing elements, buffers and interconnect. In our proposed data-centric partition based architecture, dataflow representable applications can be mapped into our structure. This architecture supports multiple applications executing concurrently with dynamic reconfiguration and straight forward access to the external I/O data. This architecture has the scalability to add more applications around the existing applications during post fabrication, therefore, saving cost especially when the applications are suffering from changes for different requirements. Also, it has the reconfigurability to configure the interconnections dynamically when necessary. Besides, this platform supports multi-width data by way of converting the data width of the processing element to a smaller size into the buffers and then back to the original size. At the same time, the interconnection overhead is reduced. By using different clock frequencies to improve the system speed and synchronize the data, the throughput is increased.

1.2 Background and Overview

1.2.1 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. Sec-

ond, each node in the dataflow executes a set of data elements per frame. Fig. 1-1 (a) shows a dataflow where a node is regarded as the source and destination of data. The 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 reconfiguration of the overall system.

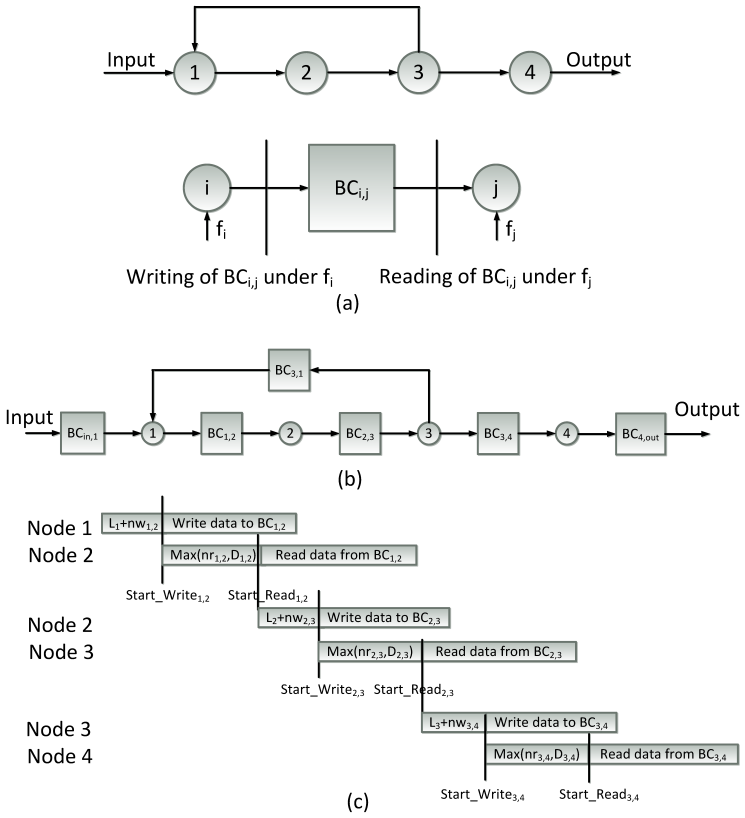


Figure 1-1: A dataflow example and timing.

Fig. 1-1 (b) gives a buffer-based dataflow graph [44], 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.

When the buffer between the producer node i and the consumer node j is represented as $BC_{i,j}$, the primary parameters which determine the buffer controller structure and overall physical realization are represented as logic latency (L_i), write offset ($nw_{i,j}$), read offset ($nr_{i,j}$), block size ($M_{i,j}$) and delay factor ($D_{i,j}$). The basic unit of all parameters is the clock cycle of a target realization. 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. For example, if node i reads data from $BC_{k,i}$, $nw_{i,j}$ is [(the start time of writing data to $BC_{i,j}$)−(the start time of reading data from $BC_{k,i}$)$-L_i$]. The read offset $nr_{i,j}$ is the offset between write into and read from a buffer. From the view point of $BC_{i,j}$, $nr_{i,j}$ is [(the time of sending the first data to node j)−(the time of receiving the first data from node i)] when writing speed of node i and reading speed of node j are matched. 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 delay factor $D_{i,j}$ represents the rate mismatch between nodes i and j, in case that the writing speed of node i is slower than the reading speed of node j, node j does not read the valid data from $BC_{i,j}$. The $nw_{i,j}$, $nr_{i,j}$ and $D_{i,j}$ are derived from the functional relation between nodes in a dataflow graph. These parameters characterize edges in a dataflow graph. Thus, when two applications consisting of same nodes have different connections (edges) between nodes, $nw_{i,j}$ and $nr_{i,j}$ of the same node in two applications may be different. The other parameters (L_i and $M_{i,j}$) represent the nodes'

characteristics. Since they are derived from the implementation of the elements, they have no dependency on the connections in a dataflow graph.

Fig. 1-1 (c) shows the timing of the buffer based dataflow. 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.

To control the read and write timing of the buffers, buffer controllers generate these control signals according to the given parameters. Generally, the write logic is decided by L_i and $nw_{i,j}$, while the read logic is decided by $D_{i,j}$ and $nr_{i,j}$. Various start signals are used to synchronize the system and they persist until the data in the block is effectively read or written. They have the following relationships:

$$start\_write_{i,j} = start\_time\_instant_i + L_i + nw_{i,j}, \tag{1.1}$$

$$start\_read_{i,j} = start\_write_{i,j} + \max(nr_{i,j}, D_{i,j}), \tag{1.2}$$

$$stop\_write_{i,j} = start\_writei, j + M_{i,j}, \tag{1.3}$$

$$stop\_read_{i,j} = start\_read_{i,j} + M_{i,j}. \tag{1.4}$$

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, represents the end time of writing and reading data to/from $BC_{i,j}$, respectively. $start_i$ is the time value in which node i begins reading data from the previous buffer controller through its fan-in port. 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 $M_{i,j}$.

To satisfy the requirement of real-time signal processing system working on sets of data as frames, the system is represented as dataflow graphs where each node executes concurrently [10]. Inserting buffers between nodes insulates the functionality of nodes. The reading and writing of the buffers are controlled by the controller, therefore, the timing relation can be easily determined. One platform consisting of processing elements, buffers and reconfigurable interconnection is proposed [44]. A reconfigurable interconnection topology is used to provide the connections between the buffers and the processing elements. The buffers are used for storage, as block level pipelined elements. Dual port buffers with simultaneous read-write capability is used as block level pipelining elements to increase the throughput.

Our proposed partition based platform allows for mapping from any type of dataflow graph with any number of external I/O of multiple applications executing concurrently. Each node in the dataflow has different operation speed. If all the applications are mapped into one partition, the control modules which are able to enable the multiple executions at the same time become complicated. Therefore, one application at least occupies one partition with separate control module. If the size

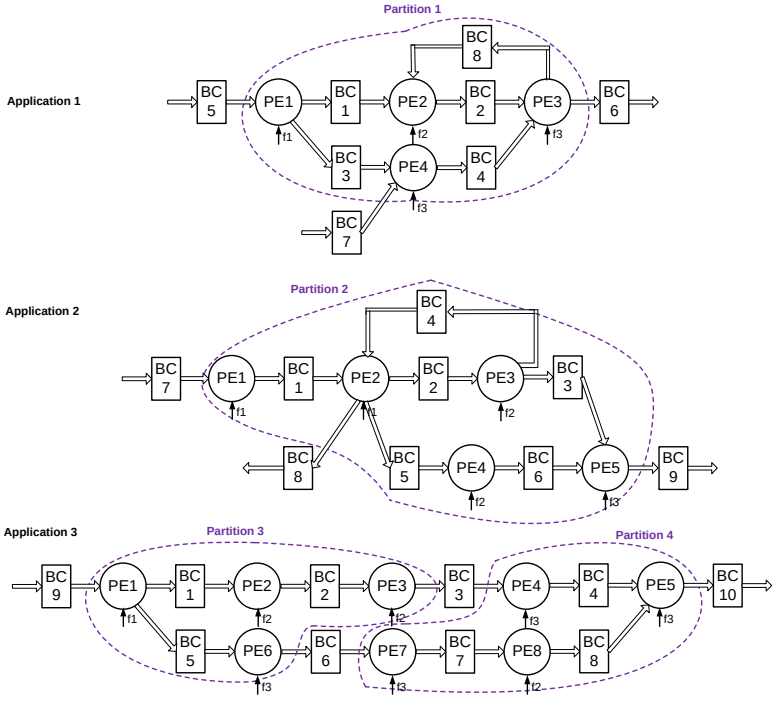


Figure 1-2: Established dataflow by inserting buffers.

of an application is too big to be accommodated by one partition, it has to be divided into several partitions interacting through the bridges. In Fig. 1-2, application 1 and 2 are mapped to partition 1 and 2 separately, while application 3 are mapped to partition 3 and 4. In the case when one partition can only accommodate six processing elements, application 3 has to be divided in two partitions.

1.2.2 Architecture Mapping and Overview

Fig. 1-3 shows the global view of the partition based platform and its interface with host processor. This thesis describes the platform and the controller design, and their interaction with host processor, including how it supports multiple applications executing simultaneously and the dynamic configuration for multiple applications.

The control structure in the global controller and the partition controller are the

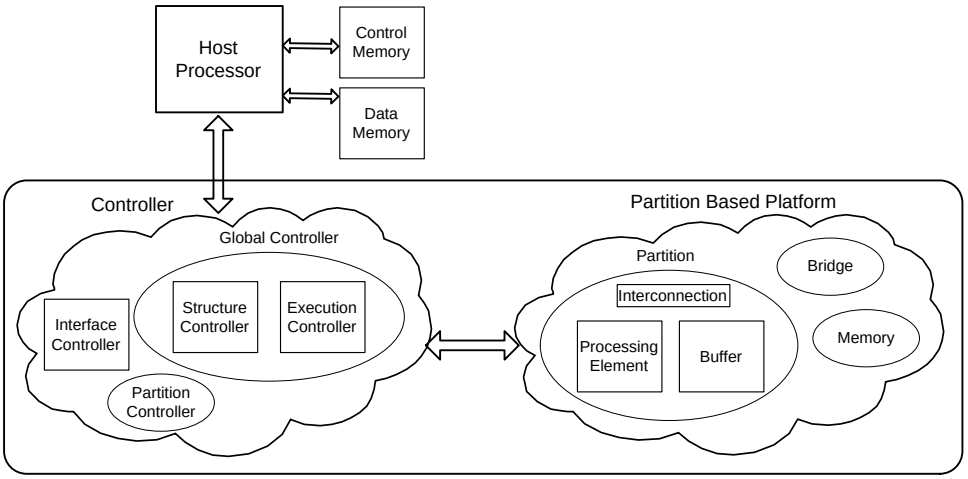


Figure 1-3: The global view of the partition based platform

same and can be divided into two parts: structural and execution. The partition or global structural controller is in charge of configuring the interconnection of the buffers or bridges. The partition or global execution controller are in charge of reading and writing the buffers or the bridges. By the interaction between the global controller and the partition controller, and between the structure controller and execution controller, synchronization is realized. The host processor gets access to the data from the control memory and data memory. The control memory saves the data configuring the interconnection, timing and clock frequencies. The data memory saves the external data necessary for the applications.

To improve the flexibility of a system, we propose the hierarchical control layers as illustrated in Fig.1-4 for the reconfigurable partition based architecture. There are two control layers separated with each other: Global control and partition control. The global controller gives the start signals to each partition. According to the start signal from global controller and the content in the partition memory, partition controllers

send the start read and write signals to the buffers. For instance, if partition 2 has an interconnection change, only the controller of partition 2 is changed, all the other control logic including the start signal from global controller and partition controller remains the same. Each control layer can be modified without any modifications of other layers. This reconfiguration mechanism results in the fast manipulation of the whole system once there is any change of the system.

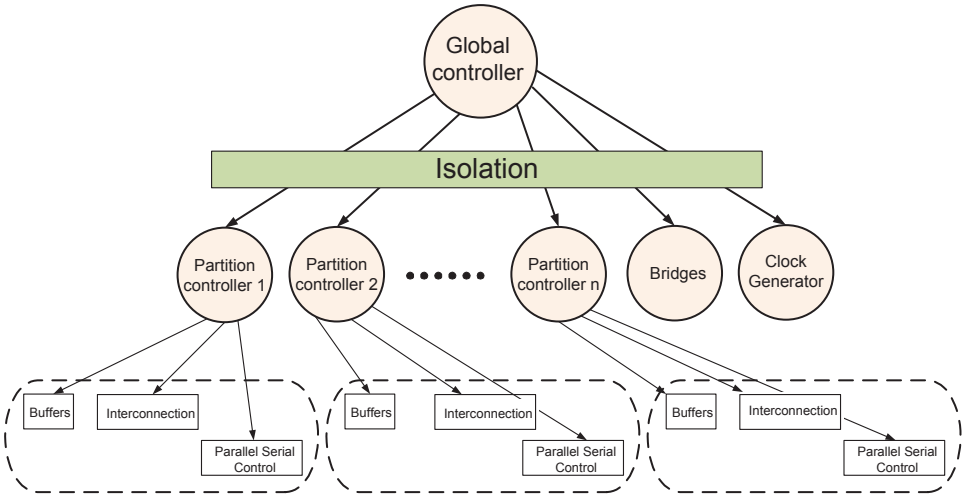


Figure 1-4: Hierarchical controller illustration.

Our proposed architecture as shown in Fig. 1-5, which divides the overall system into four partitions interacting through bridges, where each partition has an equal number of processing elements, buffers and its own partition controller. Not only is the partition able to communicate with another partition through the bridge, but also it can get access to the data from the outside of the platform through the bridge. This platform allows for the concurrent execution of several applications, and one application can make use of one partition or several partitions. We map the applications in Fig. 1-2 into this platform: application 1 uses partition 1 and bridge 1, application

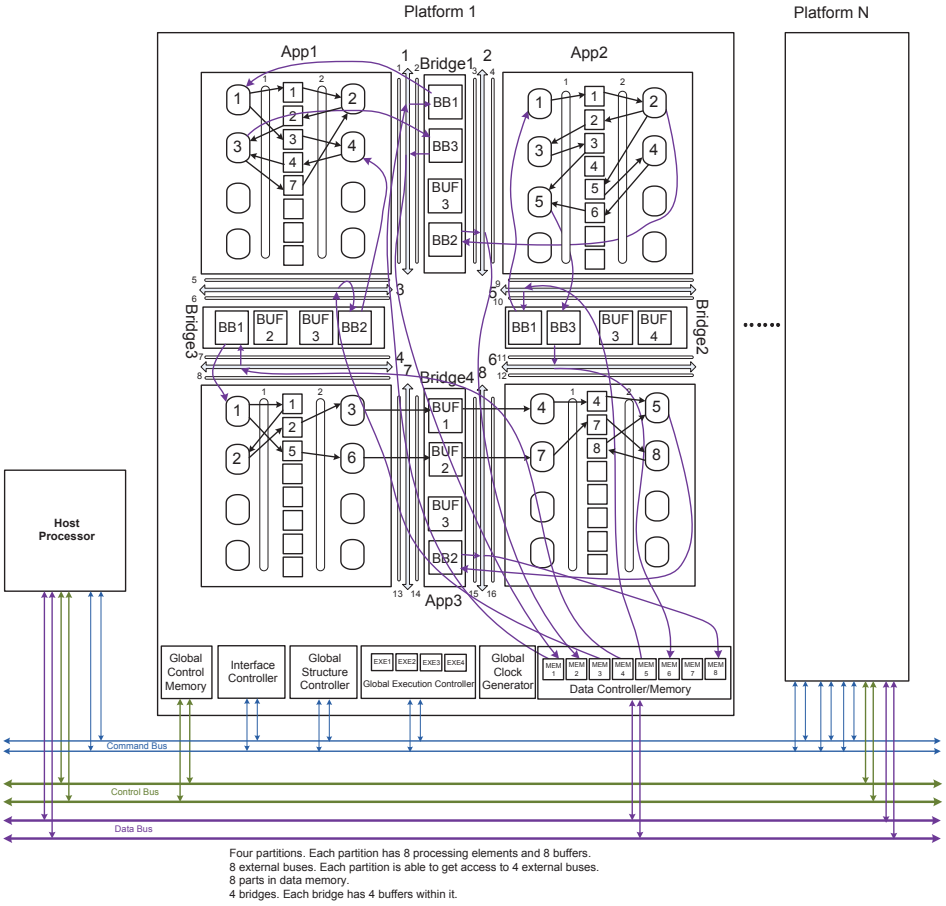


Figure 1-5: Mapping of the platform.

2 uses partition 2 and bridge 2. There are more number of nodes in application 3, therefore, to fully utilize the resources it is divided into partition 3 and 4. The buffers in bridge 4 substitute the original buffers in the dataflow.

1.2.3 Execution and Reconfiguration Overview

The controllers accept the command from the host processor and then convert the command into control signals to the platform. This platform is controlled by the hierarchical layers, which aim at the simplicity and reconfigurability for different applications. Compared to the structure where a global controller controls all the

processing elements directly, the hierarchical control layers are partially changed if the execution characteristic of a logic unit is changed. The global controller sends start signals to start the partition controllers, while the partition controllers send each buffer the start read and write signals. Each level of control is isolated from other levels, in the sense that they can be replaced without any modifications of others. More partitions are added to the platform through these bridges without affecting the existing partitions. Since it is expensive to tape out the ASIC, all the designs has to be fixed once the chip is produced. But this structure makes the changes of the platform easily, therefore, has the flexibility to the variations of the functions of the platform.

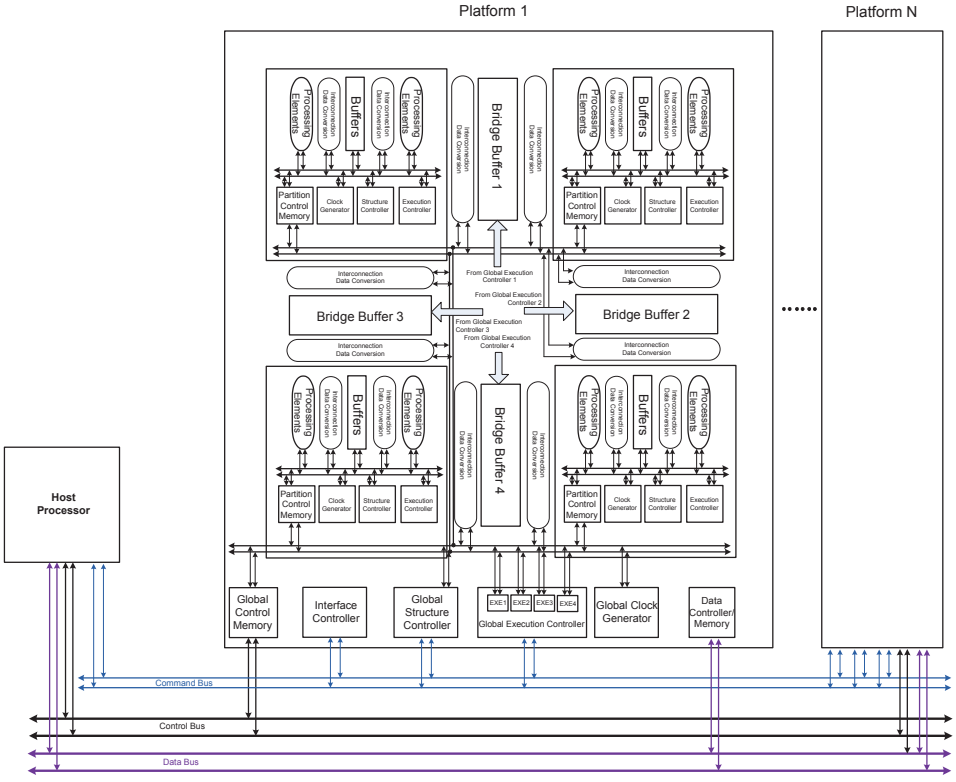


Figure 1-6: Overview of control modules.

Fig.1-6 gives the overview of the control structure. Basically, the control structure in the global controller and the partition controller are the same and can be divided into two parts: structural and execution. The partition or global structure controller is in charge of configuring the interconnection of the buffers or bridges. The partition or global execution controller are in charge of reading and writing the buffers or the bridges. By the interaction between the global controller and the partition controller, and between the structure controller and the execution controller, synchronization is realized.

1.2.4 Organization

The rest of this thesis is organized as follows. Chapter 2 explains a frequency selection algorithm to minimize power consumption of the reconfigurable architecture. Chapter 3 outlines a high performance partition based reconfigurable architecture for multiple concurrent applications. Chapter 4 describes the hierarchical controller design for such platform. Chapter 5 presents the conclusion and future work.

1.3 Related Work

Dataflow is an effective method to model digital signal processing (DSP) applications [11] - [12]. Modeling techniques and software synthesis techniques, such as parameterized dataflow modeling [13] and heterogeneous modeling [14], improves design reusability and dynamic reconfiguration. Many effective optimization techniques are developed to improve characteristics of system, such as static scheduling [15]. Schedul-

ing approach for dynamic dataflow application [16] decomposes dynamic dataflow so that existing optimization method can be used. [17] improves throughput rate by multiprocessor scheduling. [18] - [19] reduce memory requirement by buffer merging. Method of generating dataflow graphs obtained from benchmark applications with multiple input nodes is useful to bridge the gap from real applications to optimization methods [20].

Applications represented by dataflow graph can be mapped into our proposed reconfigurable architecture. Many approaches of mapping applications to coarse-grained reconfigurable architecture are proposed to maintain parallelism and resource sharing [21] - [23]. [24] proposes a methodology to derive processing elements and shows that application specific reconfigurable computing has performance benefits close to fully-custom ASIC based designs in addition to the intended reconfigurability. To support dynamic reconfiguration, ReKonf [25] is designed to evaluate various configuration of architectures and then make suitable reconfiguration decision. It is able to alter the configuration dynamically by controlling the switches and support dynamic configuration. MorphCache [26] dynamically tunes a multi-level cache hierarchy to allow significantly different cache topologies. It improves both average throughput and harmonic mean of speedups. ReMAP [27] demonstrates significant advantages of incorporating reconfigurability into future heterogeneous large-scale chip multiprocessors by accelerating and parallelizing applications.

A lot of research effort [28] - [30] have been made to reduce the power consumption while maintaining the system requirement. Many factors can be controlled to achieve the low power consumption of the system. One way of reducing the dynamic power

is by switching the active component to the low power state or shutting down the idle component [28]. Another way is to select the frequencies of the component to reduce the energy dissipation [29] - [30]. These algorithms are effective for applications of energy harvesting system and difficult to migrate to the data-centric applications which are represented as dataflow and mapped to the reconfigurable architectures. The reconfigurable architecture consists of arrays of heterogeneous processing elements, where general-purpose processors, field-programmable gate arrays (FPGAs) and application-specific integrated circuits (ASICs) are mapped. Several ways such as interconnection resource reduction [31] developed for buffer-based dataflow minimizes the energy consumption by the sharing methodology which reduces the buffer memory and the number of active bus. Another way is to redistribute the slack time and refine the task execution order [33] for multiple concurrent tasks. Simulated annealing [34] is a popular optimization method to solve our problem. Even though its runtime is greatly reduced compared to exhaustive search, it still consumes huge runtime especially when a system is consist of large amount of units. Clock distribution network places an important effect on the performance of VLSI system. Globally asynchronous locally synchronous (GALS) architecture trades off the easy setting of constraint for synchronous circuit with the efficiency of asynchronous circuit. Some design framework [35] help to choose design strategy to balance the trade-off in multiple clock domain. Dynamic voltage and frequency scaling (DVFS) scheme [36]- [40] is widely used to optimize power in multiple clock domain. Globally ratiochronous locally synchronous (GRLS) [41] demonstrates strong competitor to GALS in complexity, overhead and performance benefit while maintaining close energy consumption. Due

to the limitation of technology scaling to the voltage variation, dynamic frequency scaling (DFS) [42] - [43] provides good power saving by selecting the frequency of each processor.

Chapter 2

Frequency Selection Algorithm for Power-Efficient Multiple Data-Centric Applications

2.1 Introduction

Low power design is the key area in VLSI system design. Dynamic power management becomes critical with the technology scaling for high performance applications. This chapter proposes a method and algorithm for assigning clock frequencies to processing elements in reconfigurable architecture to minimize the power consumed while meeting the application's timing requirements. It uses the knowledge of the tradeoff between power and speed of the individual elements. Then we improve the solution to limited number of clock frequencies when applications are dynamically mapped. If a data-centric application can be represented by a dataflow, the proposed algorithm

gets the solution of the clock frequencies in faster speed than simulated annealing based algorithm.

Our algorithm uses GRLS and DFS to select the frequencies of the processing elements dynamically based on the applications. The new contributions of our approach are as follows. 1) Although a lot of research effort has been made to minimize the power consumption, there is limited work addressing the optimization for multiple data-centric applications mapping into the reconfigurable architecture. Our proposed algorithm applies to multiple heterogeneous applications if the application can be represented as dataflow graph. Once the timing constraint of the applications are given, our algorithm schedules the frequencies of the processing elements based on the analysis of dataflow graph representation. 2) Our algorithm considers the limited availability of clock frequencies in a platform and distributes them among multiple applications, which are mapped to the partition-based reconfigurable architecture. 3) Our algorithm achieves typically same power with simulated-annealing based method, while consumeing much less runtime than simulated annealing algorithm.

In the reconfigurable architecture platform, the clock frequencies of the processing elements need to be chosen to minimize the total power. However, we can not provide any clock frequency because the number of clocks is limitedly generated from the clock generator and the irrational clock frequencies can enlarge the clock skew to cause data loss or wrong data. So the problem becomes to how to select the operating frequencies of the processing elements with finite number of clocks. This problem becomes more complicated if multiple dataflow are designed in the platform with a single clock generator as shown in Fig. 2-1.

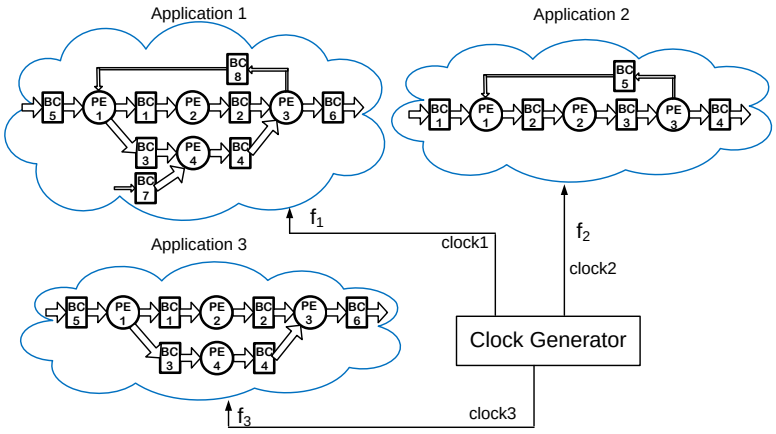


Figure 2-1: Providing clocks to the applications.

The power profile of a processing element describes the power consumption and operating frequencies with different implementation methods. Given the power profiles of the processing elements and the dataflow, we want to find the set of frequencies satisfying the required iteration constraint. We need to vary the speed of the processing elements so that the iteration period constraint can be satisfied. The first step is to choose the critical loop in the dataflow, and then choose the processing element within the loop to change the frequency.

Fig. 2-2 shows the static mapping flow of multiple applications to a buffer-centric reconfigurable architecture. Given the initial number of clock signals, the proper selection of the clock frequencies becomes critical. Local clock frequencies are generated based on a global clock. However, due to the characteristic of the typical clock generators, the generated clock frequencies are integer related. If the global clock frequency changes, then the generated local clock frequencies change. Clocks are limited and should be shared. In this thesis, we use power profile and its sensitivity to select frequencies to minimize power consumption.

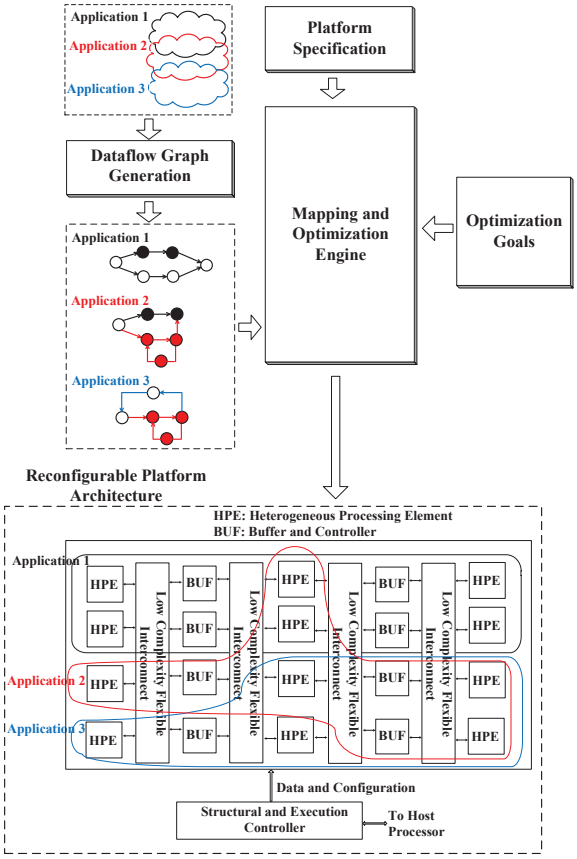


Figure 2-2: Overall mapping flow of multiple applications to a buffer-centric reconfigurable architecture.

2.2 Multi-rate Buffer Based Dataflow Characterization

2.2.1 Iteration Period and Clock Frequency

In this section, we introduce the correlation between the clock frequency and iteration period for an application. In Fig. 2-3, data are sent from one processing elements (PE) to the other by inserting buffer controllers (BC) to buffer the data transmission. Feed-forward or feedback path can be the critical path in an application. Choosing

possible lowest frequency for each processing element in a dataflow minimizes power consumption. High clock frequency of the processing element in critical path will reduce the total iteration time, however, increase the power consumption. Therefore, the correlation between the clock frequency and the iteration period of an application is useful to achieve an optimal clock distribution method.

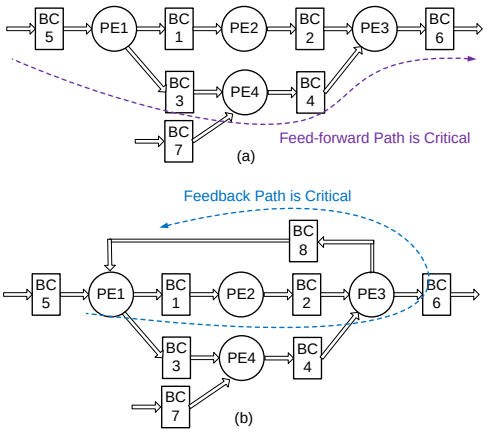


Figure 2-3: Types of Dataflow.

The critical path of an application determines its iteration time. When the feed-forward path is the critical path as shown in Fig. 2-3 (a), its minimum iteration time is illustrated in Fig. 2-4. The gray bar is the timing of first iteration and the green bar shows the data activity of second iteration. The minimum possible time to avoid data overlap of two iteration is iteration period (T). To eliminate the overlap of buffer activity in current period with the next one, the iteration time must surpass the maximum consumed time of the buffer activity in the path, i.e., M_6/f_3 in Fig. 2-4. Therefore, the iteration time of an application when the feed-forward path is critical is follows:

$$T_{iteration} = \max_{i=feedforward} \left\{ \frac{M_i}{f_{m,n}} \right\} \tag{2.1}$$

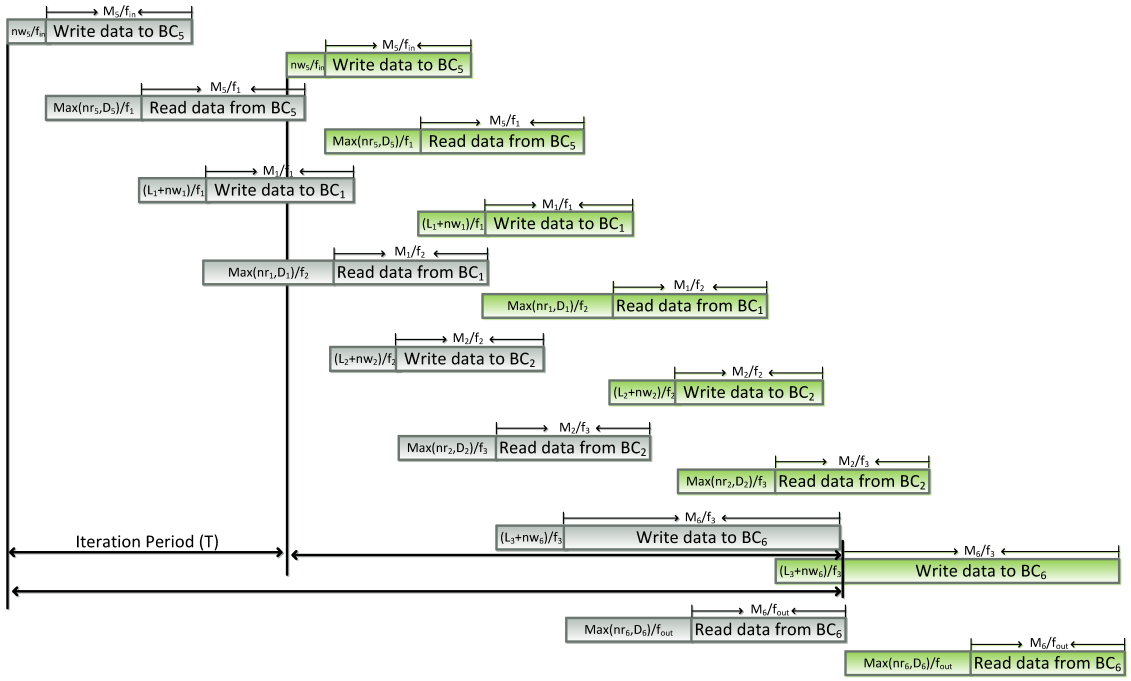


Figure 2-4: Iteration Time of Feed-forward Path.

When the feedback path is critical path as shown in Fig. 2-3 (b), its minimum iteration time is illustrated in Fig. 2-5. The read of buffer 5 in next iteration period must arrive later than the read of buffer 8 in current period, so only the time consumed by the buffers within the feedback loop counts toward the iteration time. Therefore, the iteration time of an application when the feedback path is critical is follows:

$$T_{iteration} = \sum_{i=feedback} \left\{ \frac{L_m + nw_i}{f_m} + \frac{max\{nr_i, D_i\}}{f_n} \right\} \tag{2.2}$$

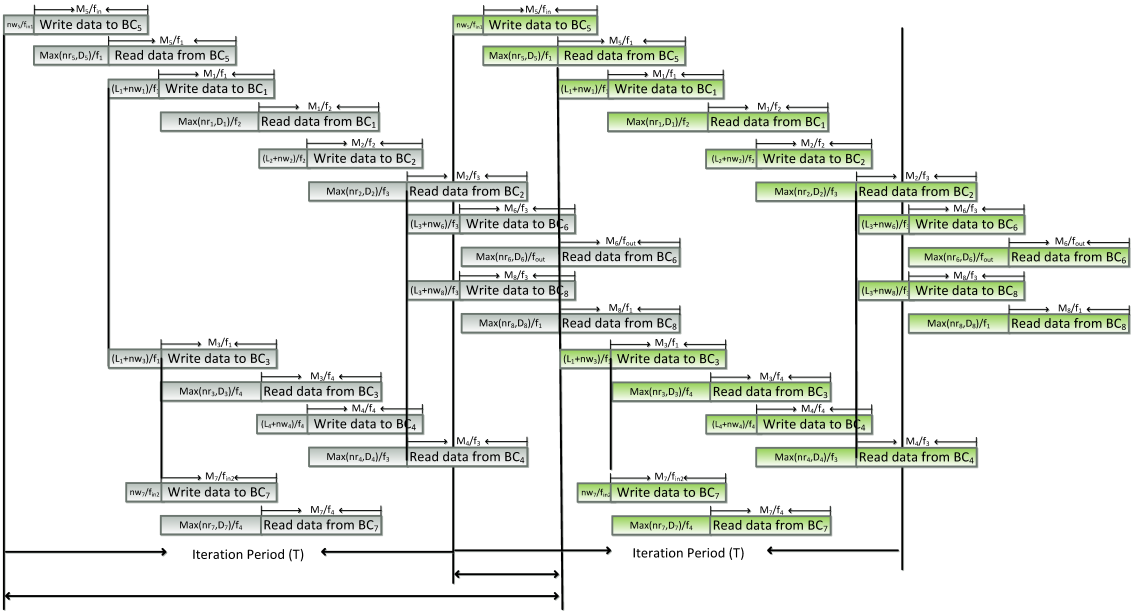


Figure 2-5: Iteration Time of Feedback Path.

Given an application represented by a dataflow graph with initial parameters and frequencies, the parameters need to be changed when the operating frequencies change. The parameters is defined above in Section 1.2.1 in Chapter 1.Now we discuss the characteristics of the dataflow and how the parameters are changed as a result of frequency change.

Constraint 1. Given multiple input terminals in a processing element, the delay factors of the buffers in the input paths might be changed when operating frequencies are changed to guarantee simultaneous data arrival time.

$$\sum_{i=path1} \left\{ \frac{L_m + nw_i}{f_m} + \frac{max\{nr_i, D_i\}}{f_n} \right\} = \sum_{j=path2} \left\{ \frac{L_p + nw_j}{f_p} + \frac{max\{nr_j, D_j\}}{f_q} \right\} \quad (2.3)$$

f_m and f_p are the frequencies of the processing elements writing data into buffer$_i$ or buffer$_j$. f_n and f_q are the frequencies of the processing elements writing data from buffer$_i$ or buffer$_j$. For the feed-forward path in Fig. 2-3 (a), there are multiple solutions of changing the delay factors of buffers in the input paths to $PE3$ to satisfy Constraint 1. Given the frequencies of the processing elements, we calculate the left and right sides of Constraint 1. The path with smaller iteration time consumes less time for the data to arrive at $PE3$. Therefore, we need to increase the delay factors of the buffers in this path to guarantee simultaneous data arrival time. All delay factors of the buffers in the faster path are increased to satisfy Constraint 1.

Constraint 2. If the write speed of a buffer is slower than the read speed, the delay factor needs to be big enough to allow the complete write of the block of data.

When considering Constraint 1, we tend to increase the delay factor of the buffer with slowest read speed in the faster path, since it might also be increased to satisfy Constraint 2. To verify if Constraint 2 is satisfied, we calculate the minimum requirement of the delay factor of a buffer with rate mismatch.

As shown in Fig. 2-6 (a), we assume the write speed f_i of buffer$_k$ is smaller than the read speed f_j. The minimum delay factor D_k of buffer$_k$ happens when the write of the last data in the block finishes at the same time with the read of the last data as shown in Fig. 2-6 (b). The minimum delay factor is calculated as follows:

$$\frac{M_k}{f_i} = \frac{max\{nr_k, D_k\}}{f_j} + \frac{M_k}{f_j} \tag{2.4}$$

$$D_k = M_k \cdot \left(\frac{1}{f_i} - \frac{1}{f_j}\right) \cdot f_j \tag{2.5}$$

$$D_k = M_k \cdot \left(\frac{f_j}{f_i} - 1\right) \tag{2.6}$$

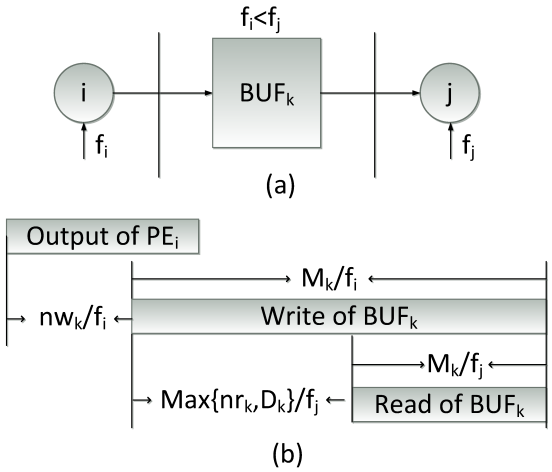


Figure 2-6: Speed Mismatch.

Constraint 3. The total delay time of feedback path must be higher than the time consumed by the read of M_i data into the path.

For the feedback path in Fig. 2-3 (c), Fig. 2-7 shows the possible data conflict when the read of buffer 5 requires so long time that it has not finished before the read data of buffer 8 in next iterative cycle arrives. In the critical case, the total delay time of the feedback path satisfies the following equation:

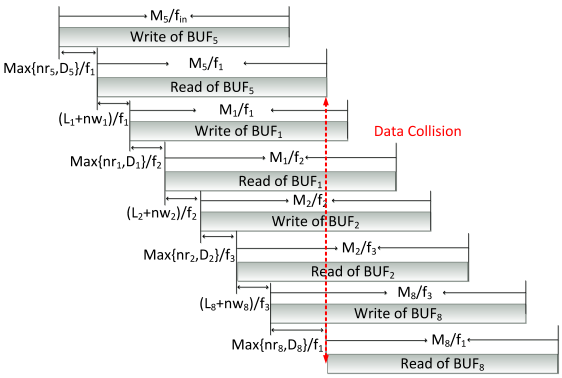


Figure 2-7: Block Size Causing the Data Collision.

$$\sum_{i=feedback} \left\{ \frac{L_m + nw_i}{f_m} + \frac{max\{nr_i, D_i\}}{f_n} \right\} = \frac{M_j}{f_q} \tag{2.7}$$

f_m and f_p are the frequencies of the processing elements writing data into buffer$_i$ or buffer$_j$. f_n and f_q are the frequencies of the processing elements writing data from buffer$_i$ or buffer$_j$. i represents all of the buffers in the feedback path, j represents the buffer writing data into the feedback path. To satisfy Constraint 3, we need to increase the delay factors of the buffers in feedback path. Considering Constraint 2, we pick up the buffer with slowest read speed to increase its delay factor to the value satisfying Equation 2.7.

Fig. 2-8 shows an example combining all the constraints together. Given the initial parameters of the buffers and the frequencies of the processing elements in the application, we need to adjust the parameters of the buffers once the frequencies are changed. To satisfy Constraint 1, path 1 and path 2, path 3 and path 4 should consume the same iteration time. Path 1: $PE1$ - $BC2$ - $PE2$ - $BC3$ - $PE3$; Path 2:

$PE1$ - $BC9$ - $PE6$ - $BC10$ - $PE3$; Path 3: $PE1$ - $BC2$ - $PE2$ - $BC3$ - $PE3$ - $BC4$ - $PE4$; Path 4: $PE1$ - $BC11$ - $PE7$ - $BC12$ - $PE4$. The delay factors of all the buffers need to be checked if the write speed is slower than the read speed. Therefore, all of the buffers apply to the Constraint 2. To satisfy Constraint 3, the two feedback loop, Path 4 and Path 5, need to be checked. Path 4: $PE1$ - $BC2$ - $PE2$ - $BC3$ - $PE3$ - $BC4$ - $PE4$ - $BC8$; Path 5: $PE3$ - $BC4$ - $PE4$ - $BC5$ - $PE5$ - $BC7$. The consumed time in Path 4 should surpass the data activity time in $BC1$, M_1/f_1. The consumed time in Path 5 should surpass the maximum value of data activity time in $BC3$ and $BC10$, $max\{M_3/f_3, M_{10}/f_3\}$. The iteration time of this application is the minimum one when we analyze the two feedback paths and the feed-forward path separately.

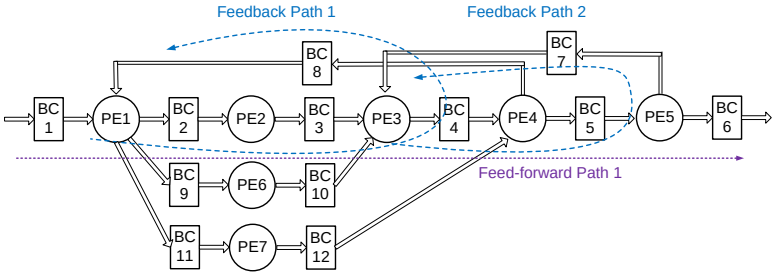


Figure 2-8: An example combining all cases together.

2.2.2 Iteration Period and Power Consumption

The implementation of a processing element decides the power consumption. To evaluate our proposed algorithm, we assume power is linear function of operating frequencies for one implementation method. As shown in Fig. 2-9, each implementation method has a highest operating frequency, say f_{T1}, f_{T2} and f_{T3}. They are the highest operating frequency of the implementation of type 1, 2 and 3 respectively. To save

the power, the optimal way of implementing a function is shown in bold.

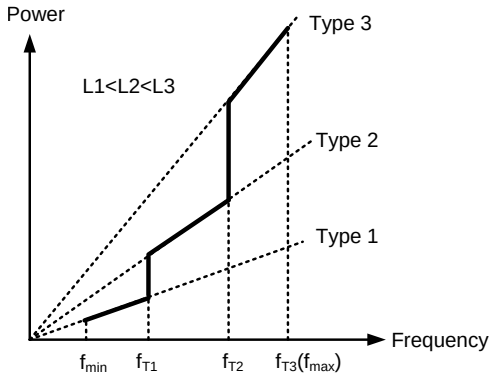


Figure 2-9: Power profile of processing element.

The following equations calculate the power, energy and the constraint of the iteration period. The total power of the application is the sum of the power consumption of each processing element, which is the linear function of the operating frequency. To get the minimum power, we need to choose the lowest frequencies and the simplest method of implementation. The constraint of iteration period limits the parameters of the buffers and frequencies of the processing elements in the critical path.

$$P = \sum_i P_i = \sum_i k_i \cdot f_i, \tag{2.8}$$

$$T_{iteration} < T_{constraint}, \tag{2.9}$$

Considering the buffer based dataflow as shown in Fig. 2-3, each processing element operates under their own frequency f_i and corresponding buffer parameters L_i, nw_i, nr_i and D_i. The parameters represent the number of clock periods with respect to the corresponding clock frequency. So the absolute time of these parameters is

divided by the frequency f_i.

For the dataflow in Fig. 2-3 (b), we will show how the iteration time changes with frequencies. The power profiles of the processing elements and the total power are given in Fig. 2-10. Fig. 2-10 (a) gives the simplified power profiles, each processing element is implemented in an unified method whatever the operating frequency is. Fig. 2-10 (b) gives the typical generic power profiles of the processing elements in the critical path. Fig. 2-10 (c) gives how the total power of the application changes with the change of frequency of one processing element.

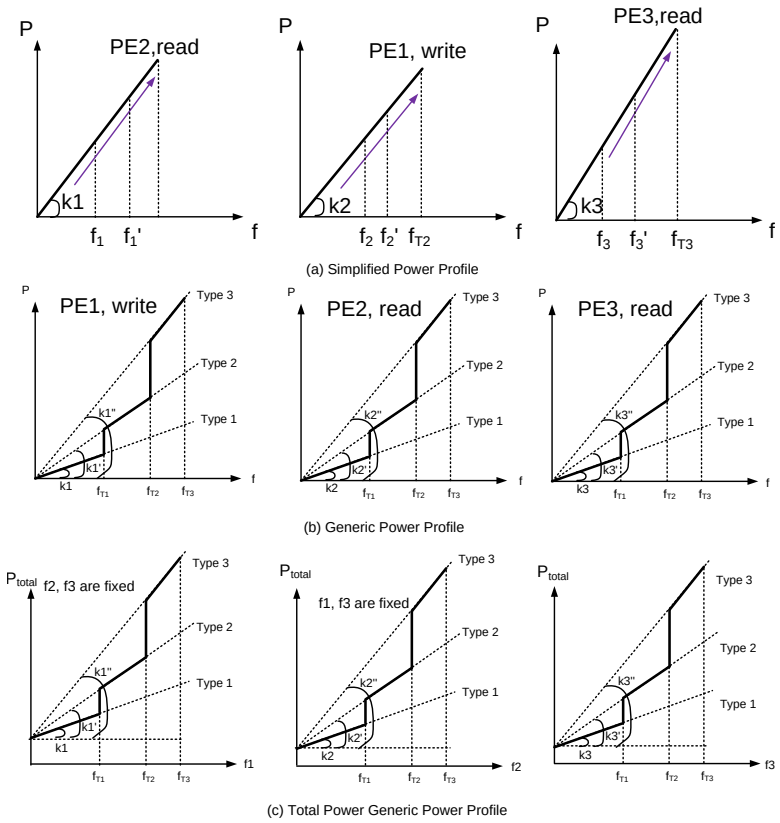


Figure 2-10: Characteristic of the Power Profile.

The slope of the power profile of each region describes how much power dissipation is necessary if the frequency increases. We calculate $\frac{\Delta T}{\Delta P}$ to get how much iteration

time variation is necessary to save power if the frequency of the processing element changes at f_i. It is related to the operating frequencies of the processing elements, the slopes of power profiles and parameters of buffers, such as logic latency L_i of the processing element, the write offset nw_m of the buffer writing data into the processing element, the read offset nr_n and the delay factor D_n of the buffer reading data from the processing element. If the parameters become larger, say delay factor D_n increases because of rate mismatch or multiple path, then the result becomes smaller. We want the absolute value of $\frac{\Delta T}{\Delta P}$ to be big, that is, low operating frequency, small slope of the power profile and large parameters to make it efficient to sacrifice power for iteration time.

Fig. 2-11 gives the illustration of how the iteration time variation changes with the operating frequencies of the processing elements for simplified and generic power profiles. $\frac{\Delta T}{\Delta f}$ is derived by deviation of the expression of iteration time by frequency. It describes how much iteration time change with a small frequency change. $\frac{\Delta T}{\Delta P}$ is only related to the operating frequency but not dependent on the slopes of the power profiles. Therefore, $\frac{\Delta T}{\Delta P}$ keeps the same for the simplified linear power profile and non-linear power profile.

Fig. 2-12 give the figure of how $\frac{\Delta T}{\Delta P}$ changes with the operating frequency of the processing element. We can get that to sacrifice power consumption to satisfy the iteration constraint, we want $\frac{\Delta T}{\Delta P}$ as small as possible, that is slope of the power profile and the operating and threshold frequencies as low as possible, while the parameters are as large as possible. For example, if the delay factor of the buffer is enlarged because of the rate mismatch, then the sensitivity of the $\frac{\Delta T}{\Delta P}$ becomes smaller,

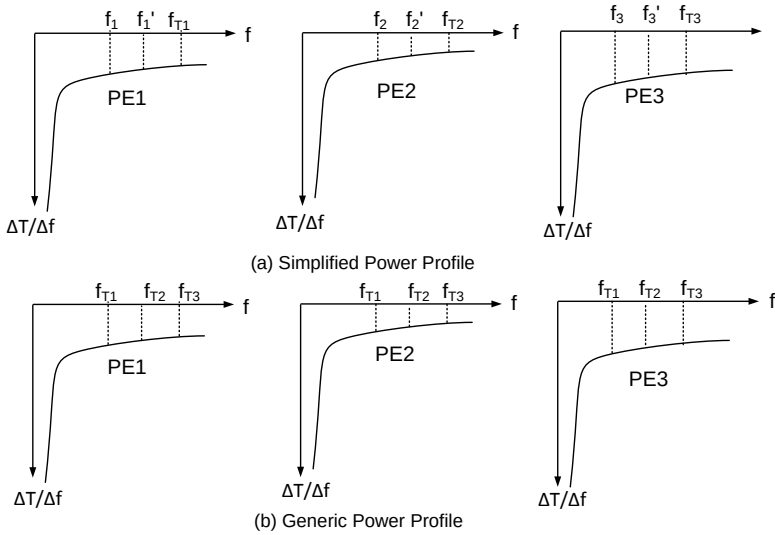


Figure 2-11: Iteration Time Change Rate.

therefore, helps to save power while maintaining the required iteration constraint.

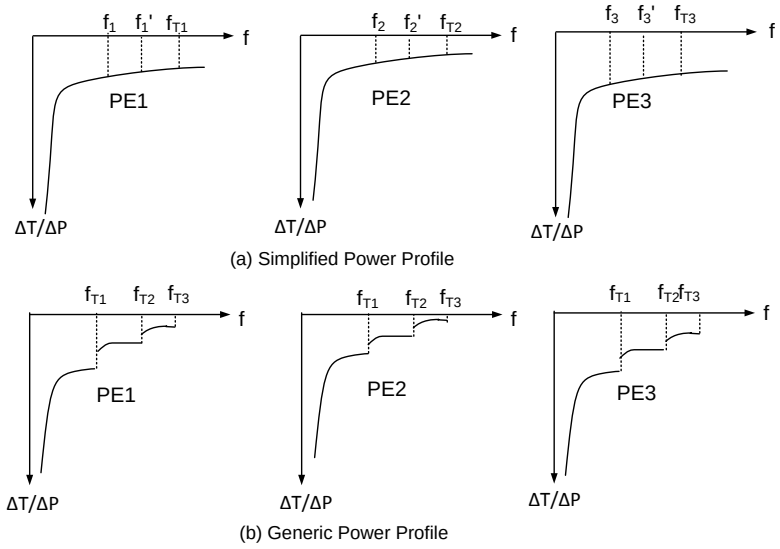


Figure 2-12: Iteration Time Change Rate over Power.

In this section, we shows the calculation of iteration time and power consumption of a dataflow graph. The iteration time of an application is related to frequencies of each processing elements, parameters that control the timing. Parameters are adjusted for frequency configuration to satisfy the constraints of an application. We

model simplified and non-linear power profiles to calculate power consumption. Then we show how much iteration time we need to sacrifice to gain power reduction. This knowledge will be used in our proposed algorithm in next section.

2.3 Frequency Selection for Low Power

2.3.1 Frequency Selection Algorithm

There are many algorithms available to assign the clock frequencies under a timing constraint. However, the runtime of these algorithms, such as simulated annealing based method, can be huge with the increased number of units in an application. In this section, we discuss the method of selecting frequencies for each processing element to minimize total power consumption while satisfying the iteration time constraint with less runtime. Our goal is to adjust the clock frequencies so that the iteration time is equal to the iteration constraint, while maintaining minimum total power. To select the frequencies of processing elements, the iteration time of each loop is calculated in the application based on the given initial frequencies. Then the frequencies of the processing elements in the critical path change to achieve minimum power. The frequencies of the processing elements in non-critical paths vary to reduce power before the non-critical path becomes critical path.

In the optimization process, first we choose the processing element to change frequency and then we decide how much the frequency changes. First we calculate the difference of current iteration time and target iteration time. If the difference of cur-

rent iteration time and target iteration time is out of range from $-\delta$ to 0, the iteration goes on to increase or decrease the frequencies of the processing elements. In order to adjust the clock frequencies, we have to select the one with smallest contribution factor $\frac{\Delta T}{\Delta P}$ so that we sacrifice least power consumption to get the iteration constraint. $\frac{\Delta T}{\Delta P}$ of one processing element is calculated when the frequencies of other processing elements are fixed and there is a minor change of frequency of this processing element. This result is calculated based on current frequencies of all processing elements and dataflow representation. If current iteration time is bigger than the iteration constraint, then the frequencies of the processing elements is increased. The frequency of the processing element with maximum $\frac{\Delta T}{\Delta P}$ is reduced by Δf. However, if current iteration time is smaller than the iteration constraint, we slow down the clock frequencies as much as possible to save the total power. The frequency of the processing element with minimum $\frac{\Delta T}{\Delta P}$ is increased by Δf. The the new $\frac{\Delta T}{\Delta P}$ are updated based on the changed frequencies. If the dataflow needs to change the delay factor because of rate mismatch or multiple paths, the new current iteration time is updated. Apply the current optimal frequency and the difference of the iteration time and iteration constraint and go back to the new iteration until the iteration time is close to the iteration constraint.

Fig. 2-13 shows the steps of how the clock frequencies of the processing elements is adjusted. In the initial case, iteration time calculated from initial frequencies is bigger than target time. Based on $\frac{\Delta T}{\Delta P}$ when the frequency of one processing element changes, we choose to increase f_1. After f_1 is increased to $f_1^{'}$, we calculate $\frac{\Delta T}{\Delta P}$ when $f_1^{'}$ becomes current frequency of the first processing element. We will choose the

processing element with maximum $\frac{\Delta T}{\Delta P}$, which is the second processing element. This iteration continues until step 5, where current iteration time is less than target time. Based on minimum $\frac{\Delta T}{\Delta P}$, the frequency of first processing element decreases to $f_1^{'''}$. Then in step 7, the frequency of third processing element is increased to $f_3^{'''}$ where current iteration time is close to target time and the iteration stops.

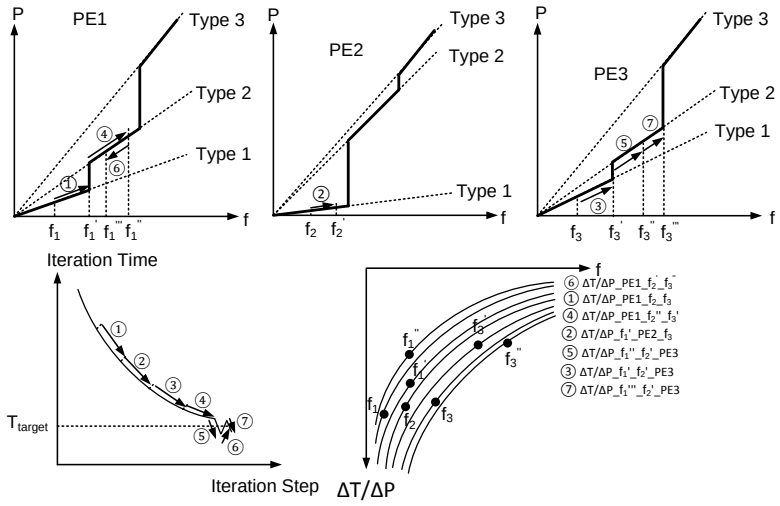


Figure 2-13: Steps of Frequency Adjustment.

In the proposed algorithm, the selection of step size is critical. If the step size is big, the system works in fewer steps, but may not be able to exactly converge on the constraint; If the step size is small, the system will be able to get to the constraint, but it takes many more steps, slowing down the algorithm. δ is a small variable that defines when the algorithm ends if the length from current iteration time to target time is within δ. If the current iteration time is smaller and closer than δ to the target iteration time, our algorithm terminates. If the step size is constant and large, our algorithm may have big change of oscillation around the target iteration time. Therefore, we choose the step size proportional to the distance from current

frequency to estimated final frequency. When it comes to the end of the algorithm, the step size becomes smaller. The chosen of the coefficient k is related to runtime. If k is big, the runtime is smaller and there is high chance of oscillation. If k is small, the oscillation is effectively eliminated, but it consumes more runtime. Also, we can choose k under different power profiles. If the slope of current power profile is high, we choose smaller k in case of the frequency is increased or decreased too much. In our algorithm, k factor is high when current frequency is far from estimated final frequency. k factor becomes low when current frequency stays close to estimated final frequency. If several $\frac{\Delta T}{\Delta P}$ are the same, k factor should be very small to guarantee frequency do not go far away from optimal. Since we need to change all the frequencies at this time. To eliminate oscillation, we decrease k factor each time the iteration time crosses over the target time, where we are from trying to decrease frequencies to trying to increase frequencies, or from trying to increase frequencies to trying to decrease frequencies. In this way, iteration time can go close to target time without oscillation.

Algorithm 1 gives the summary of the algorithm assigning clock frequencies. In the search process, we decide to increase or decrease the frequencies of the processing elements by the value of $\frac{\Delta T}{\Delta P}$, based on the difference of current iteration time and the iteration constraint. Then we calculate the new difference and continue with the iteration until current iteration time is smaller and close to the iteration constraint. Initially k is set up based on slopes of power profile for initial frequencies. k is between 0 and 1. If the slope is high, k is small. If the slope is low, k is big. Each time iteration time goes across target time, k is reduced by half. If $\frac{\Delta T}{\Delta P}$ for all processing elements

Algorithm 1 Find clock frequencies to achieve minimum power consumption

```
1: OPTIMAL_FREQUENCY(x: difference; ini: initial; opt: optimal )
2:
3: Initial setting of k based on slope of power profile;
4: while (T_itr − T_target < − δ or T_itr − T_target > δ) do
5:     if T_itr − T_target < − δ then
6:         Pick up the PE which has minimum ΔT/ΔP;
7:         If all ΔT/ΔP are 0, k = 0.01, change all frequencies at the same time;
8:         If there are equal ΔT/ΔP, choose the one with highest slope;
9:         k = k * 0.5 when T_itr goes across T_target
10:        f_i = f_i - k*(f_iteration − f_final);
11:        if f_i < f_c and f_i−1 > f_c then
12:            f_i = f_c
13:        else
14:            f_i = f_i
15:        end if
16:        Update L, nw, nr, D, M based on Equation 2.3 to 2.7;
17:    else
18:        Pick up the PE which has maximum ΔT/ΔP;
19:        If all ΔT/ΔP are 0, k = 0.01, change all frequencies at the same time;
20:        If there are equal ΔT/ΔP, choose the one with lowest slope;
21:        k = k * 0.5 when T_itr goes across T_target
22:        f_j = f_j + k*(f_iteration − f_final);
23:        if f_i−1 < f_c and f_i > f_c then
24:            f_i = f_c
25:        else
26:            f_i = f_i
27:        end if
28:        Update L, nw, nr, D, M based on Equation 2.3 to 2.7;
29:    end if
30: end while
```

are all zeros, then k is chosen very small, such as 0.01, to make sure all the frequencies only change a small amount. This algorithm discards unnecessary combinations of frequencies by the value of $\frac{\Delta T}{\Delta P}$, therefore, it achieves the result in high speed.

Evaluation and Discussion

In this section, we use five cases as shown in Fig. 2-14 to show the result of our proposed algorithms. For each case, the power profiles of processing elements have different steep slopes and multiple discontinuities. The first case is an application with single feed-forward path. The second case is an application with multiple feed-forward paths. The third case is an application with single feedback path. The fourth case is an application with mix of single feedback path and multiple feed-forward paths. The fifth case is an application with mix of multiple feedback path and multiple

feed-forward paths.

We implement our proposed algorithms and simulated annealing-based algorithms as a baseline in Matlab, and show that the simulated annealing algorithm is very slow. Given an application, we extract a function that represents the iteration time by the dataflow representation. Also, we set up the constraints that the timing parameters should change if given frequencies are changed. A function of calculating power consumption is provided for each application. These functions setting up an application are also provided to simulated annealing-based algorithm. Then we compare the power and runtime of our proposed algorithm and simulated annealing-based algorithm.

For the feed-forward path with 1 processing element, we try to see how the start point affect the result. 1MHz to 5 MHz are the start points of the frequencies of PE1. We get the result that start points do not affect the selection of frequency. We can set up how far away of the iteration time is from target time (1%, 5% and 10%) and then the iteration stops. When final iteration time is 1% away from target time, total power consumption of our algorithm is 1% more than simulated annealing. If 1% is increased to 10%, our power consumption is 10% more than simulated annealing. Setting up final iteration time to be 1% away from target time is sufficient to achieve comparable result to simulated annealing. For single feed-forward path with 7 processing elements as shown in Fig. 2-15, allowing 1% difference of iteration time will cause power consumption within 1% more than simulated annealing. From Table 2.1, we can tell how the runtime changes with different k factor. Simulated annealing consumes longest time. If we use small constant step size, the average runtime is around half of

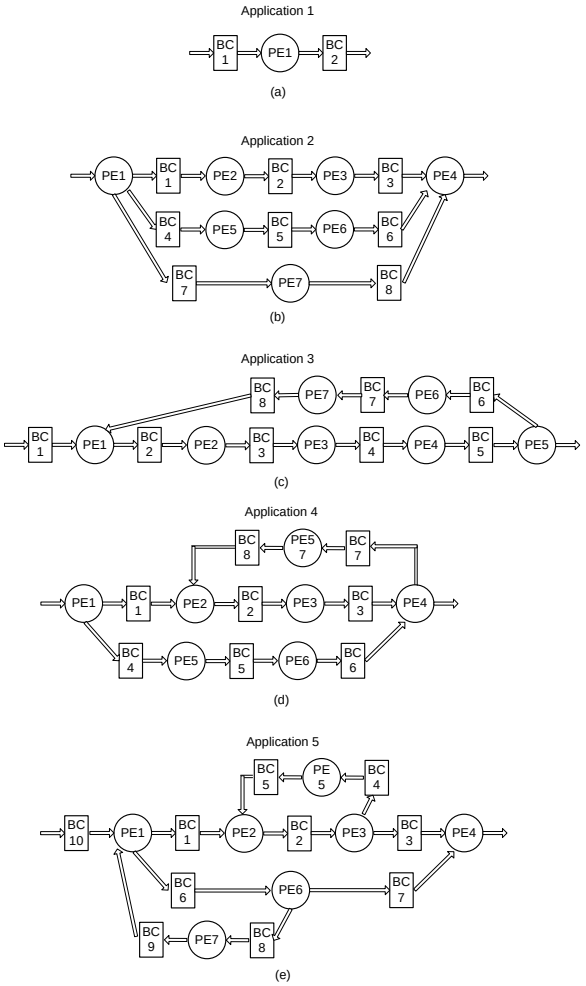


Figure 2-14: Four applications as test example.

simulated annealing. However, the runtime is greatly reduced if we use variable step size. 0.6, 0.3, 0.2 are k factors when slopes are different. 0.6 is the k factor we choose when slope is smaller than 3. 0.3 is k factor when slope is between 3 and 8. 0.2 is k factor when slope is more than 8.

Table 2.1: Runtime of proposed algorithm and simulated annealing

	Proposed Algorithm (s)								
$StepSize$	constant step = 0.001	0.1, 0.1,0.1	0.2, 0.2,0.2	0.3, 0.3,0.3	0.4, 0.4,0.4	0.5, 0.5,0.5	0.6, 0.5,0.4	0.6, 0.3,0.2	simulated annealing
$T(average)$	31.24s	1s	0.7131s	0.5468s	0.3898s	0.3259s	0.3221s	0.4875s	57.28s

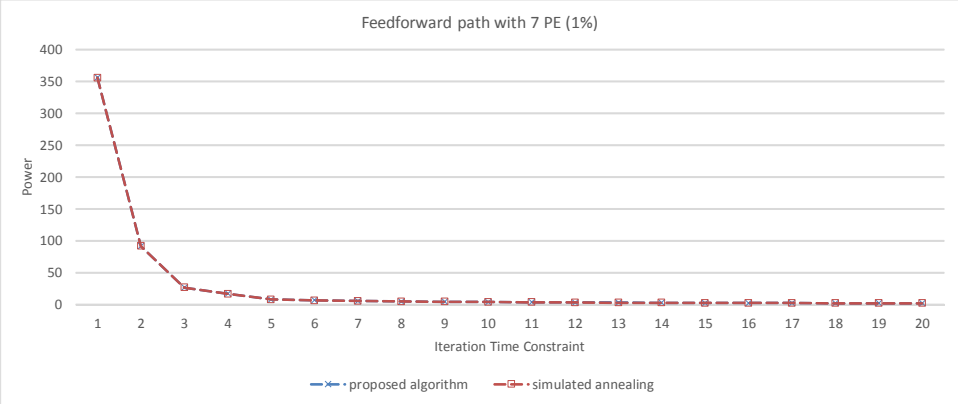


Figure 2-15: Single feed-forward path with 7 PEs.

When multiple feed-forward paths give data to one processing element in case 2, data should arrive at the same time. Delay factor (D) is increased for the path that consumes less time to guarantee multiple data arrival for a processing element. We choose k factor to be 0.6, 0.3 and 0.2 initially. The average runtime is 0.61s. This value is slightly higher than the case with single feed-forward path because of adjust of parameters due to multiple fan-in consumes some time.

Application 3 is single feedback path. Application 4 and 5 have multiple feed-forward paths and single feedback path. Table 2.1 gives the runtime comparison of our proposed algorithm and simulated annealing for application 2 to 5. Our proposed algorithm is 10 times faster than simulated annealing on average. Compared to applications with simple dataflow structure, applications with complicated dataflow path consumes more time, but still saves more time than simulated annealing algorithm. Fig. 2-16 and Fig. 2-17 shows the power consumption is close to simulated annealing and the starting points have little impact on power consumption for application 4 and 5.

Table 2.2: Runtime of proposed algorithm and simulated annealing for applications

	Proposed Algorithm (s)					Simulated Annealing (s)				
Application	1	2	3	4	5	1	2	3	4	5
$T(average)$	0.4875	0.6100	0.5224	4.5400	1.0631	57.2800	57.7856	59.0666	30.9024	32.2000

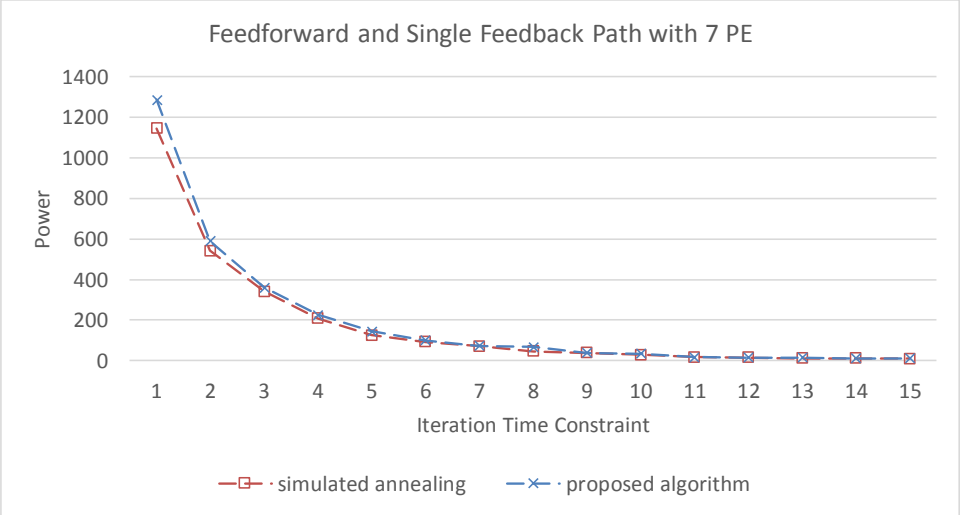


Figure 2-16: multiple feed-forward paths and single feedback path in case 4.

From above results, we get that our algorithm achieves comparable power consumption when feed-forward path and feedback path both exist in dataflow and maintains much less runtime. For the case when there is only feed-forward path or feedback path, our algorithm gets same power consumption while consumes much less runtime.

2.3.2 Limited Number of Clock Frequency

Algorithm 1 selects the clock frequencies for an application to minimize power consumption. The algorithm produces N frequencies for N processing elements. However, the clock generator in hardware implementation can not provide any value of clock frequency. Also, there might be read or write of wrong data if the clock frequencies are non-integer related and the clock skews are enlarged. Therefore, we change

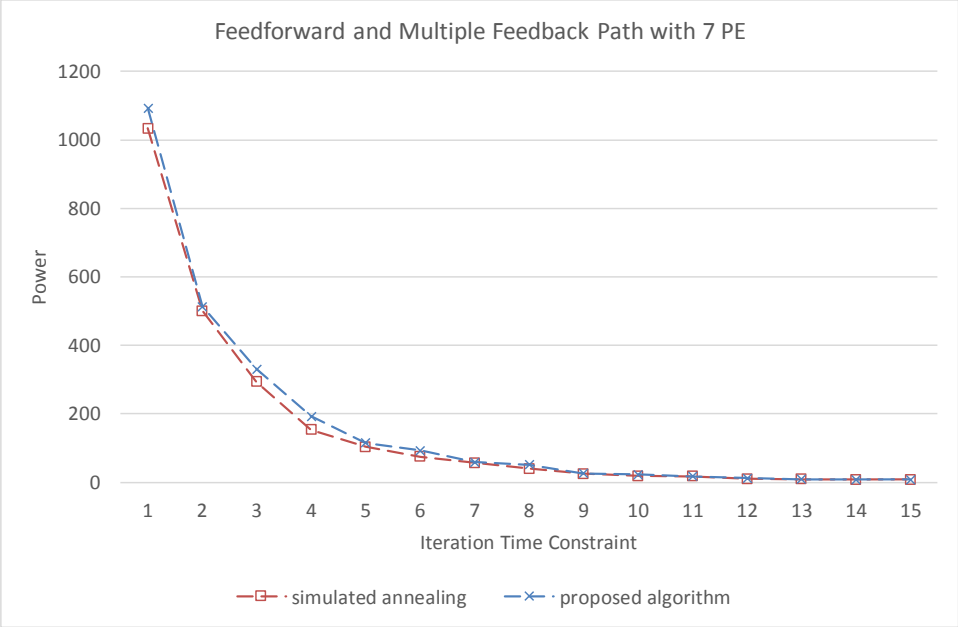


Figure 2-17: multiple feed-forward paths and multiple feedback paths in case 5.

the result from Algorithm 1 to the case when limited number of clock frequencies M is smaller than the number of processing element N and the clock frequencies should be integer related.

Given a dataflow where N processing elements require up to N frequencies, f_1, f_2, ..., f_N, to achieve the low power operator. However, if the available number of clock frequencies $N^{'}$ is smaller than N, then some of the processing elements must share the same frequencies. Then the problem becomes a grouping problem. The objective is to group one or more frequencies so that the total number of clock frequencies is $N^{'}$ while maintaining the minimum power.

Algorithm 2 gives the summary of the algorithm with limited clock numbers. For illustration, consider a dataflow with N processing elements, where f_1, f_2, ..., f_N is calculated by Algorithm 1 to get the minimum total power. Given the dataflow

Algorithm 2 Find clock frequencies to achieve minimum power with limited number of clock frequencies

```
1: // ∀ BC_{i,j} in a Given Buffer Based Dataflow
2: LIMITED_NUMBER_OF_CLOCK
3:
4: while N_current > N_limit do
5:     for i = 1 to N − 1 do
6:         for j = i + 1 to N do
7:             diff_{i,j} = | f_i - f_j|;
8:         end for
9:     end for
10:    f_min = min{diff_{i,j}};
11:    for f_p = f_k to f_k + f_min do
12:        f_m = f_p;
13:        f_n = f_p;
14:        if iteration time from f_1, ... , f_N < target time;
15:        save f_1, ... , f_N
16:        calculate power from f_1, ... , f_N;
17:    end for
18:    power_min = min{power};
19:    save the frequencies with power_min;
20: end while
```

with p processing elements and k different clock frequencies, we group the frequencies with minimum difference and replace them by new frequency between them. For each grouping method, the substituting frequency ranges from the minimum to the maximum values of the original frequencies. We select the one with minimum total power while maintaining the iteration time constraint in these cases. The grouping repeats until the clock number is reduced to k. The grouping method reduces the possible cases of selecting the frequencies to much less possibilities based on the result of Algorithm 1.

Evaluation and Discussion

Fig. 2-18 shows how the total power changes with the limitation of the number of clock frequencies for application 5 as shown in Fig. 2-14 under different iteration time constraints, and the comparison with simulated annealing. The total power increases if the available number of clocks are reduced. If several applications are mapped to

a platform and the total number of clocks of the applications is not enough in the platform, we need to reduce the number of clocks for some of the applications. When the number of clock is limited, our result is close to simulated annealing. The total power increase with the decrease of the available number of clocks. If the number of the optimal frequencies happens to be less than the number of processing element, the limitation of number of clocks do not have effect on the total power.

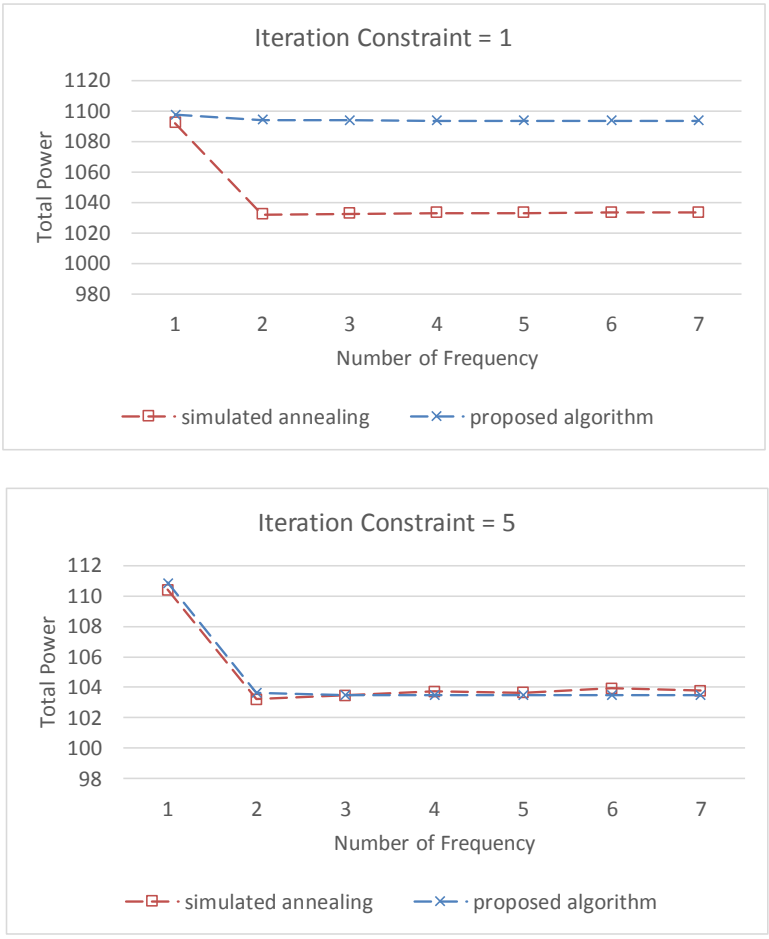


Figure 2-18: Relationship of total power and number of available clocks.

2.3.3 Frequency Selection Algorithm for Multiple Applications

In this section, we give the solution to arrange the clock frequencies of the processing elements in multiple applications. The total power increases if the available number of clocks are reduced. If several applications are mapped to a platform and the total number of clocks of the applications is not enough in the platform, we need to reduce the number of clocks for some of the applications. As shown in Fig. 2-19, three applications, each of them under their own iteration constraint T_i, share N clocks provided by the clock generator. By applying Algorithm 1, we get the number of clock frequency N_1, N_2, N_3 and the frequencies for each application. If N is smaller than the sum of N_1, N_2, N_3, we need to reduce the total number of clock frequencies to N.

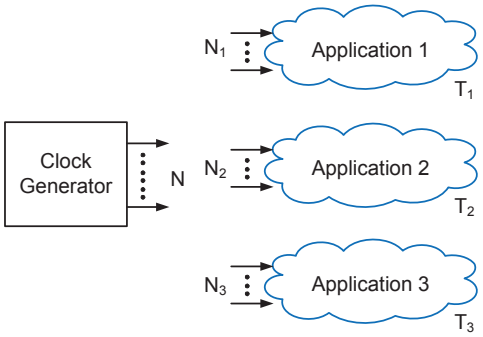


Figure 2-19: Arrangement of Clock Frequencies.

Fig. 2-20 (a) shows that the power consumption is less or the same if more clock frequencies are allowed. Once the number of clocks achieves the one with minimum power consumption, it is meaningless to increase the number. In Fig. 2-20 (b), it shows the power consumption of many possible solutions when total clock number

is limited. Fig. 2-20 (c) shows the total power of these applications with different assignment of clock numbers. Our goal is to find the allowed clock number distribution that consumes minimum total number. We do not need to exhaust all the combination of number of clock frequencies, but only use these combination when the optimal number is achieved to calculate the minimum total power of all the applications. For combinations of number of clocks, there is an optimal solution that achieves minimum power while all the iteration time constraints are satisfied. Our target is to find this optimal solution.

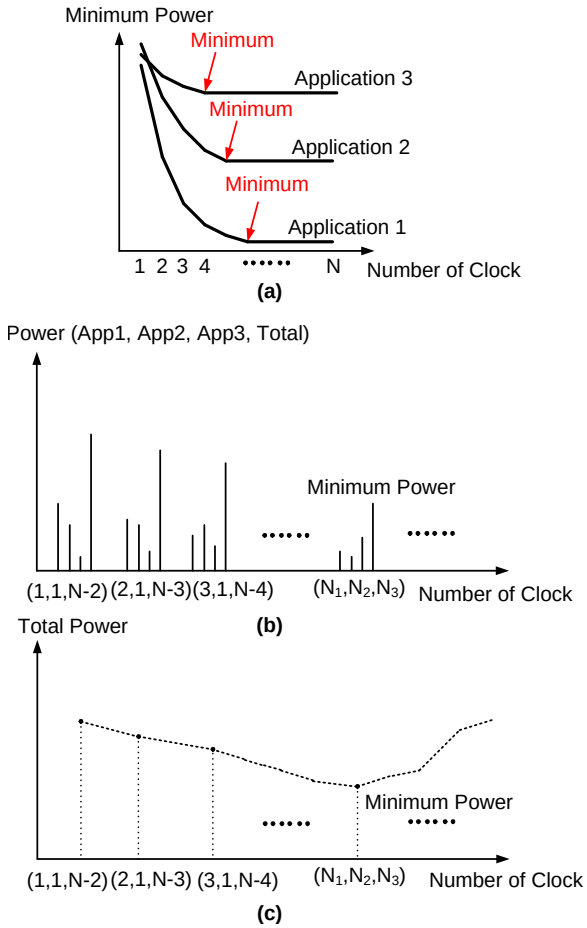


Figure 2-20: Arrangement of clock frequencies for each combination.

If we apply Algorithm 1 to three applications separately, the clock frequencies

corresponding to the number of clock in each application are independent as shown in Fig. 2-21. There is a chance that one application is using a clock frequency that is also used by another application. However, we are able to share the same clock frequency in hardware implementation. Therefore, our approach is to treat the three application as a whole application and reduce the number of clock frequencies using the grouping method in Algorithm 2. Before using the grouping method, we get the optimal frequencies for each application by applying Algorithm 1. $f_{1,1}$, $f_{1,2}$ and $f_{1,N1}$ are the optimal frequencies for application 1; $f_{2,1}$, $f_{2,2}$ and $f_{2,N2}$ are the optimal frequencies for application 2; $f_{3,1}$, $f_{3,2}$ and $f_{3,N3}$ are the optimal frequencies for application 3. Then we group these frequencies treating them as a single application. In this way, we allow the internal grouping between applications. Several applications can make use of the same clock frequency and it is only counted as one clock. This approach provides more available clock frequencies if the total number of clock is fixed.

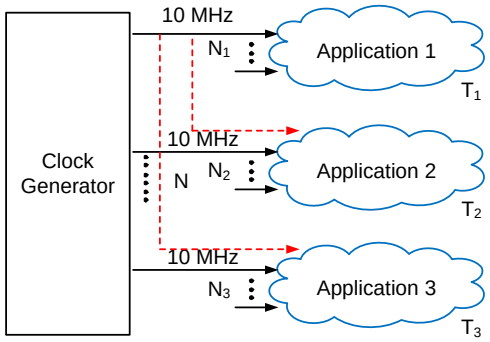


Figure 2-21: Clocks are generated independently.

Then we consider the case when the clock is arranged and then more applications arrive as shown in Fig. 2-22. Suppose N_1 is the optimal number of clock frequencies for application 1. If a second application comes, then N-N_1 clocks are available. N_2

is the optimal number of clock frequencies for application 2. If the available clock frequencies are enough to satisfy the optimal requirement of application 2, then N_2 clocks are used. However, in the case when the remaining clocks are not enough to satisfy the optimal requirement of application 2, we need to reduce the number of clock frequencies than the optimal value to reserve for application 2. In this case, Algorithm 2 is used to decide how many clock frequencies are used by application 1 and 2.

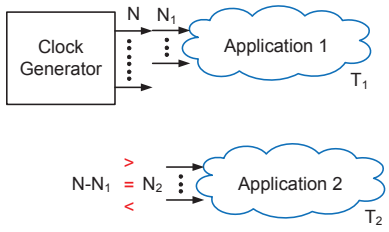
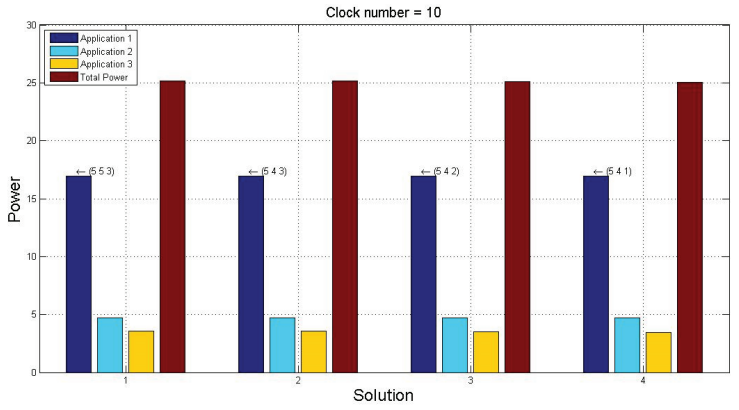


Figure 2-22: Incremental Mapping.

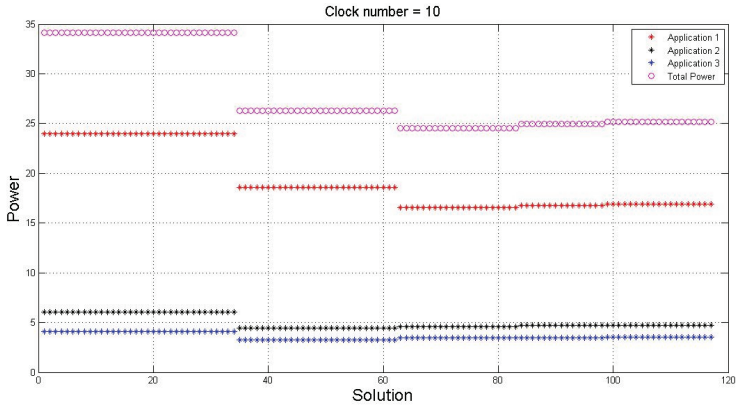
Evaluation and Discussion

Fig. 2-23 shows the clock assignment when three applications are mapped and only 10 clocks are available. Each application can use at most 7 clocks. Fig. 2-23 (b) gives 117 possible solutions with exhaustive possible combination and the minimum power is 24.5585. However, grouping method only searches 4 possible combinations as shown in Fig. 2-23 (a) and achieve the minimum power as 25.0685. We sacrifice 2% power consumption to reduce the search time of exhaustive algorithm.

We test the function with dynamic mapping. We achieve the assignment of the clocks when single and multiple applications are mapped. Fig. 2-24 (a) shows how the total power changes with the addition of the applications. Application 1 is used



(a)



(b)

Figure 2-23: Total power of applications.

in the first case. With the add of application 2, the available number of clocks for application 1 is reduced. At this time, application 1 consumes higher power than the case when single application 1 is mapped. If three applications are mapped, the optimal assignment changes. If two applications are mapped, 5 clocks and 3 clocks are assigned to application 1 and 2 separately can achieve the minimum total power. If three applications are mapped, the best solution of the number of clocks assigned to the applications is (4,3,1). Fig. 2-24 (b) shows the frequency change with the added

number of applications. When 2 applications are mapped, changing frequency of $f2$ will achieve minimum power. If 3 applications are mapped, 6 iteration steps with each changes one frequency of the processing element are enough to achieve the minimum power. Therefore, in the case of dynamic change of the adding of applications, the number of iteration steps is greatly reduced while maintaining acceptable minimum total power.

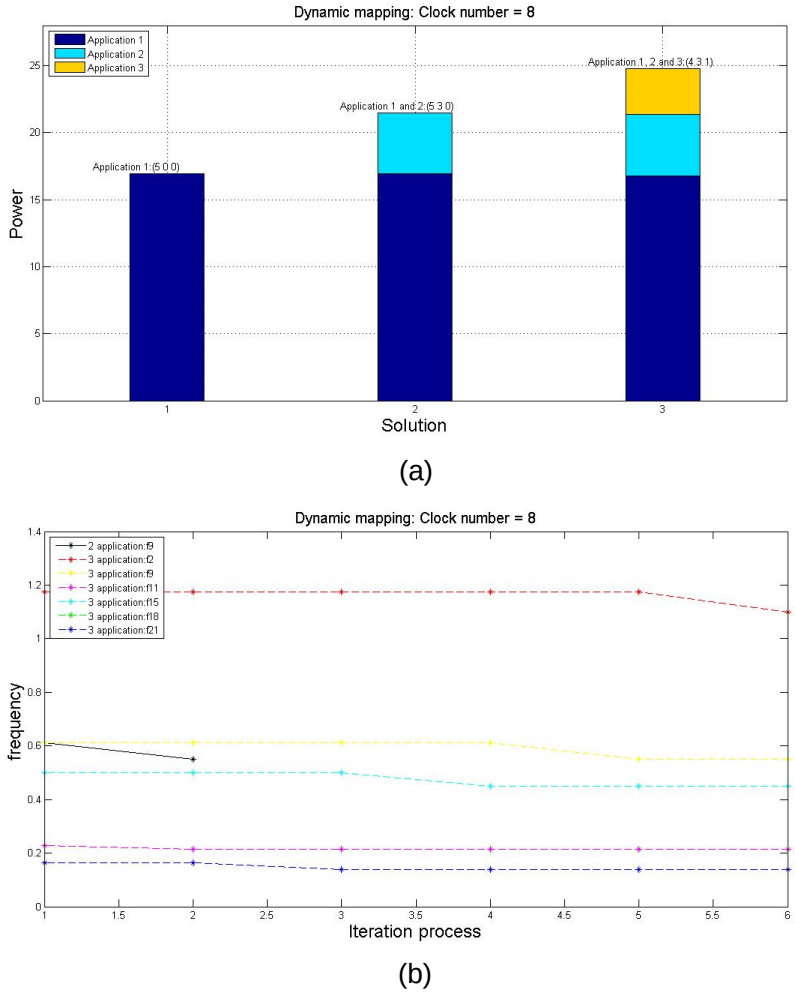


Figure 2-24: Incremental mapping.

Table 2.3 shows the average runtime of our proposed algorithm and simulated

annealing with different iteration time constraint under clock number limitations. The speed of our algorithm is 300 times faster than simulated annealing. With the clock number limitations, our algorithm consumes little more time than that without clock limitations. However, simulated annealing requires twice of the time when there is no limitation of the clock number.

Table 2.3: Runtime of proposed algorithm and simulated annealing with clock number limitations

	Proposed Algorithm (s)							Simulated Annealing (s)						
Clock Number	1	2	3	4	5	6	7	1	2	3	4	5	6	7
$T(average)$	0.103	0.102	0.102	0.102	0.102	0.102	0.102	48.5	39.3	33.7	28.2	24.8	21.9	17.9

2.4 Conclusions

In this chapter, we proposed a frequency selection algorithm targeting multiple data-centric applications to minimize power consumption and maintain system iteration time constraint. The proposed algorithm adjusts the frequencies of the processing elements in the application by exploiting the characteristic of the dataflow representation. We makes use of variable step size to change the frequencies by exploring the the relationship between power consumption and iteration time. Also, we introduce the limitation of the available clock frequencies and how they are mapped to platform with limited number of clock dynamically. The simulation result shows that given the dataflow representation of the applications, the algorithms can achieve typically same result with simulated annealing based method while consuming much less runtime.

Chapter 3

High Performance Partition Based Reconfigurable Platform for Multiple Concurrent Applications

3.1 Introduction

The requirement of real-time signal processing calls for the application of reconfigurable architecture, which integrates the flexibility similar to FPGA and the high performance characteristic of ASIC. FPGAs include fine-grained reconfigurable interconnect, which allows a great amount of flexibility, but also adds significant overhead in terms of area, power, and time. Custom designed ASIC implementation has the advantages of high speed and low power, but they lack of scalability in the case that there are functional changes. Reconfigurable architecture, combining the merits of flexibility and high performance, is more suitable for embedded digital signal pro-

cessing by demonstrating performance superiority [2] as well as energy efficiency [3]. There are some limitations of the reconfigurable structure. The control structure is complicated and hard to manipulate, therefore, it costs a lot of time to configure the system. Without our proposed controller structure, it wastes lots of resources and time to reconfigure the system. To manipulate rapid modification of the execution of the dataflow, it is necessary to design a controller structure providing an efficient spacial and temporal connectivity for the reconfigurable architecture.

This thesis proposes a novel hardware reconfigurable platform consisting of multiple partitions to execute multiple concurrent applications. Depending on the performance requirements, an application migrated from a dataflow graph can be mapped to more than one partition interacting through bridge buffers. A partition, however, hosts a single application, increasing the overall flexibility of the architecture. Each partition within the proposed architecture is entirely buffer centered consisting of a large number of heterogeneous processing elements operating with buffers through reconfigurable interconnect. The buffers are used for storage and as block level pipelining elements to increase the throughput. Furthermore, these buffers achieve isolation among the processing elements, facilitating the key objectives such as dynamic reconfiguration and multiple clock domains for the heterogeneous processing elements. The interconnect architecture supports sufficient concurrency within a configuration, while simultaneously permitting the timing-sharing of the resources across the configurations. The multi-bit applications are supported by way of converting the diverse data sizes into the normalized smaller size, therefore, reduces the overhead of the interconnect architecture. Another important capability of the proposed architecture

is the significantly high memory bandwidth to accommodate large amounts of I/O data commonly encountered in multimedia applications. A handshaking mechanism is introduced to concurrently read data from and provide data to the global data memory. It demonstrates that the implementation of reconfigurable platform with multi-bit applications is feasible in SystemC model.

3.2 Partition Architecture Organization

3.2.1 Partition Datapath

Partition

Partitions are the basic cell in our proposed platform. The partition module integrates eight processing elements, buffers, interconnection and data conversion modules. Partition size is decided by the size of mapped applications. If the partition size is bigger than the applications, resources are wasted even when one application is mapped into one partition. If the partition size is so small that the applications has to be divided into many partitions, the global controller becomes complicated due to the coordination between large number of partitions.

Fig.3-1 shows the partition module. It integrates eight processing elements, buffers, interconnection and data conversion modules. The partition execution controller controls the start time of read and write of the buffers and the sizes of the data conversion modules according to the content from the partition memory. The structure controller decides the interconnection and data conversion modules. Whereas the start

of the partition structure and execution controller comes from the signal outside of the partition, the global structure and execution controller. Compared to multiple global controllers controlling the buffers, interconnection and data conversion modules without partitioning, this design approach facilitates dynamic reconfiguration and executing multiple applications simultaneously.

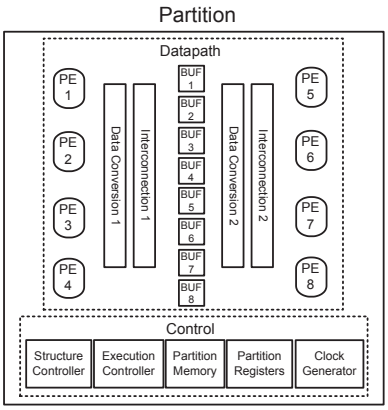


Figure 3-1: Partition module.

Processing Element Fabric

The processing element fabric is a collection of processing elements operated under their own clock as shown in Fig. 3-2. There are M processing element cells and $4M$ input and output ports distributed on both sides of the fabric for the convenience of interconnection. However, there are more total numbers of ports for the processing elements than the number of the ports of the fabric. It makes applications with more than two input and output ports possible to be mapped into the platform. One processing element can use any of the ports on one side in the fabric by the $2M$ multiplexers and $2M$ demultiplexers. The setting of the multiplexers and demultiplexers

is decided by the value saved in the $4M$ registers, where the address bus selects the register and the control bus gives the setting value of the the multiplexers and demultiplexers. The size of the output data from the processing element can be different from the size of the input data. Another M registers are used to save the input and output data size of each processing element. Each processing element is operated under its own clock frequency. Therefore, at most M processing elements with no more than $4M$ ports, $2M$ of them are input and the other $2M$ are output, can be mapped into the processing element fabric.

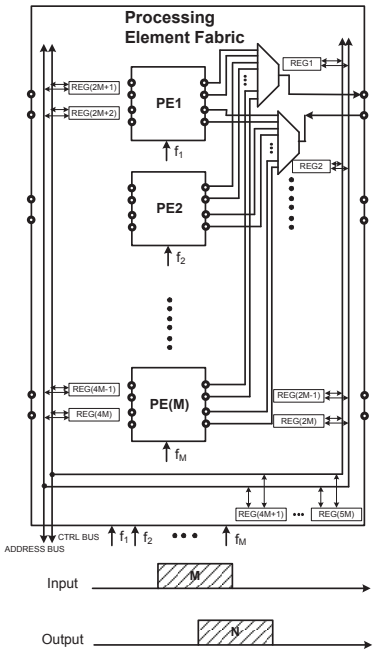


Figure 3-2: Processing element fabric.

Buffer Fabric

To isolate the data between two processing elements, the buffer fabric is introduced as shown in Fig. 3-3. The data size of the buffer is smaller than that in the processing

elements since it reduces the interconnection and makes full use of the resources. The read and write speed of the buffer can be different, which is decided by the clock frequencies of the processing elements. The buffers are controlled by the start read and write signals. Once the start read signal comes, the buffer reads data. Once the start write signal comes, the buffer sends out the data. However, the input data size of the buffer is the same to the output data size of the buffer, whereas they might be different in the processing element.

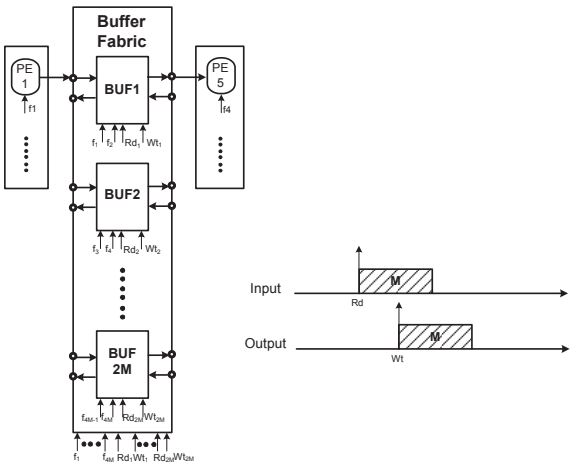


Figure 3-3: Buffer function.

Data Conversion

In the processing element fabric, the data size of the input in one processing element can be different from the that of the output. If we use different sizes of buffers between the processing elements, the flexibility of the system is ruined. To maintain flexibility of the system, we may choose the maximum size of the required sizes of buffers. But it will waste a lot of resource, especially when there is a big variation of the data size. Moreover, interconnection complexity is a significant problem. Without data

conversion module, the data size of the interconnection module is W bit. With the $2N$ ports, the total number of wires on one side of the buffer is $2NW$. Therefore, large value of W causes complex interconnection and low speed. To provide the flexibility while minimizing resource waste, we propose the data conversion module which changes all data sizes to a common denominator size. They converts the bigger size of data into a uniform smaller size of data going into the buffers, and then back to the original size in the processing elements. After the conversion of the data from W bit to W_A bit, the total number of wires on one side of the buffer is reduced to $2NW_A$. It improves the flexibility since any data size is supported by converting them to the buffer size. To achieve the same throughput of the variable word width application with the case where no data conversion is applied, multiple frequencies of clocks to the processing elements and buffers are provided. We increase the clock frequencies of the buffer, f_2, to balance the total consumed time of the buffer, since the serial data in the buffer consumes more clock cycles than that without the data conversion modules.

$$f_2 = f_1 \cdot \frac{W_A}{W_2} \tag{3.1}$$

The data conversion modules consist of parallel serial conversion and serial parallel conversion modules, which convert the W_1 bits data from the processing elements to W_A bits data sent to the buffers, and then to W_2 bits data entering another processing element as shown in Fig. 3-4. Each parallel to serial or serial to parallel module is

operated under the faster clock frequency of processing element. Within each of them, $2M$ registers are used to save the input and output data size.

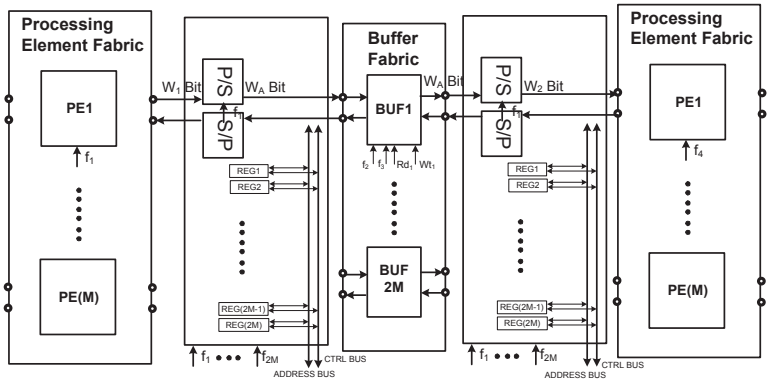


Figure 3-4: Data conversion module.

Flexible Interconnection

The interconnection module, consisting of multiplexers, connects any processing element in the processing element fabric with any buffer in the buffer fabric. Given the application, the interconnection modules within the partition set the interconnection between the processing elements and the buffers as shown in Fig. 3-5. They make it possible for any of the $2M$ processing elements connected with any of the $2M$ buffers. The content of the address and data bus is from outside of interconnection module. The address bus selects the registers which sets the multiplexer, while the control bus provides the value of the select signal of the multiplexer.

In the interconnection module, there are $4M$ multiplexers, choosing from any output of the processing elements to the buffer, or choosing from any output of the buffer to the processing element. Each buffer has two input and output ports. To address the $4M$ multiplexers, the address bus size is $log_2(4M)$ in each partition. For

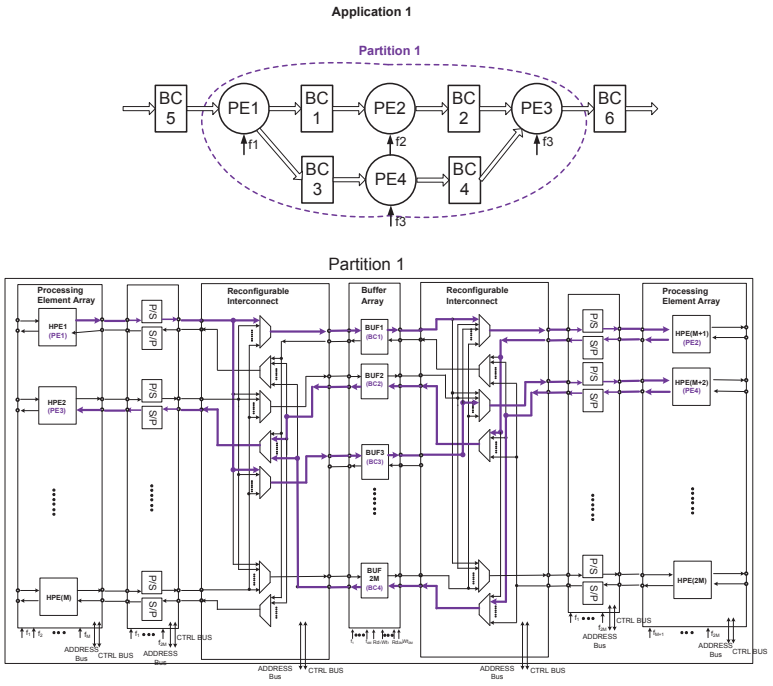


Figure 3-5: Interconnection module.

each multiplexer, it chooses from any of the M input as the output. Therefore, the control bus data size of the interconnection module is $log_2(M)$. To control the timing of the processing elements and buffers, the address size of processing element array is $log_2(2M)$ and that of buffer array is $log_2(2M)$. The interconnections between the partitions and the bridges follow the same style with that within the partition.

If the partition size is small, then the number of multiplexers within the partition is small, but results in larger number of multiplexers in the interconnection module around the bridges. Therefore, the balance of the complexity of the interconnection module is the main points of choosing the partition size.

3.2.2 Partition Control

Partition Clock Network

Fig. 3-6 shows the partition clock network. To provide multiple clock frequencies to the partitions and the bridges, the clock divider generates required clocks based on global clock frequency. Once the clock generator is enabled, the global select signal decides which global input clock is selected to generate the output signal. The output clock frequency equals to the global clock frequency divided by the divisor. The output select signal selects which output is currently being configured.

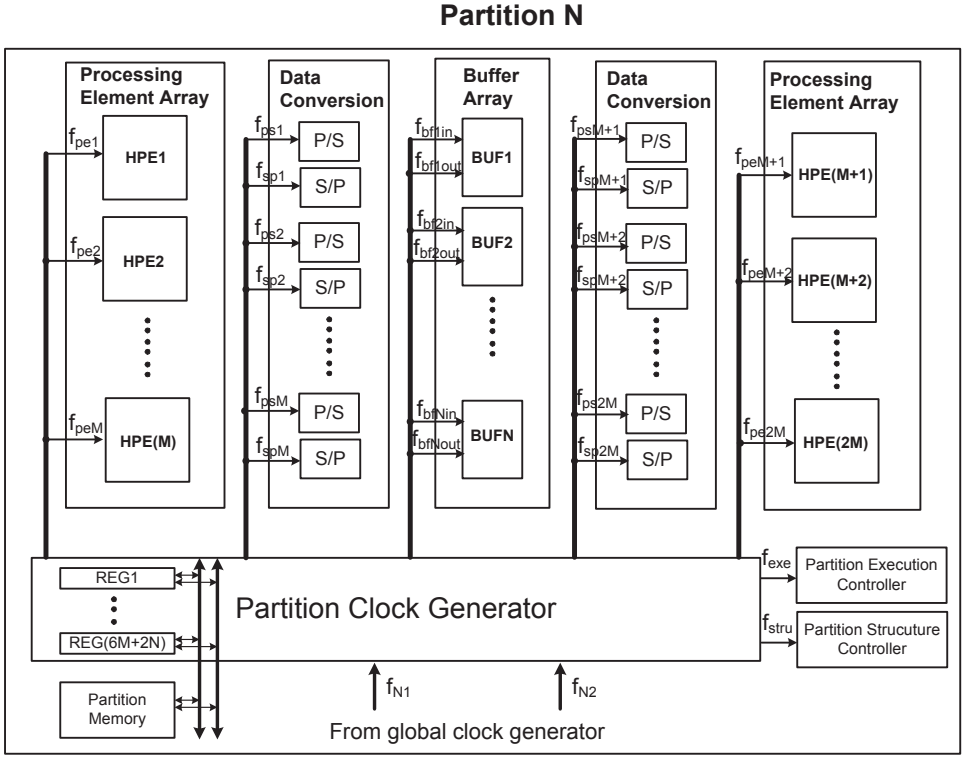


Figure 3-6: Partition clock network.

Within the partition, a single clock generator provides the clocks to the processing elements, the buffers, the data conversion modules and the partition execution

controller, based on the global clocks from the global clock generator. The clock frequencies of the processing elements are decided by the requirement of the applications. Normally we constrain the clock frequencies of the buffers and the data conversion modules the same to avoid inaccuracy. Each partition has its own partition clock generator, since this hierarchy balance the clock path between different partitions. Within the partition, the clocks are generated based on the unique global clocks, thus improves the accuracy of the clock frequencies.

Execution Controller

Fig. 3-7 shows the function of the partition execution controller. If the counter receives the enable signal from global execution controller, the counter counts the current time operated on the execution clock, and then sends the value to the finite state machine. The finite state machine gets access to the partition memory to fetch the address and data indicating the time when the buffers should start read and write based on the program saved in the partition memory. When the program memory is read, it sends the control values to the start read and write logics of the buffers. The benefit of using program is that it is able to generate several start signals at the same time to initiate several buffers, without comparing the current time with the required start time frequently. The size of the program memory is proportional to the clock frequency of the execution controller. Therefore, high clock frequencies could result in large size of program memory.

However, the clock frequency of the execution controller can not be too low. Suppose one processing element and the following processing element are operated

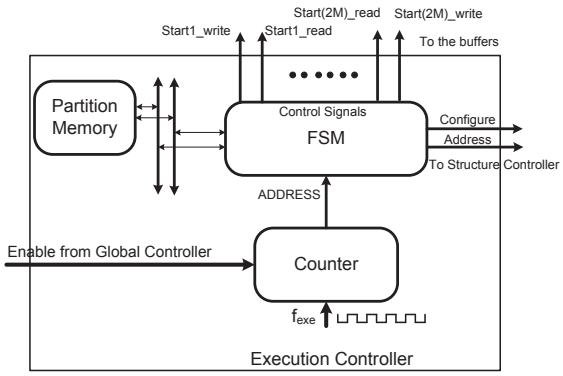


Figure 3-7: Execution controller.

on the clock of f_1 and f_2 separately. The buffer between them receives start signal from the execution controller which is operated on the clock of f_3. As shown in case 1 of Fig. 3-8, when the clock of the execution controller is smaller than the highest clock of the processing elements, the first read and written data may be lost. In case 2, the first written data may be lost. But in case 3, both the read and the written data are correct. Therefore, the minimum clock frequency of the execution controller should be chosen as the highest speed of the processing elements to avoid losing the first several read and written data of the buffers.

The read and write signals of the buffers are generated based on the content of the execution program. The structure of the program containing the control information is shown in Fig. 3-9. Each bit indicates the start read and write signal of the buffers in the partition. Besides the control information during the normal execution and communicating with the external, an additional bit is used to decide which case the current control signal is generated in. If the current signal is generated under the external case, then the execution controller has to check if the data is ready in the

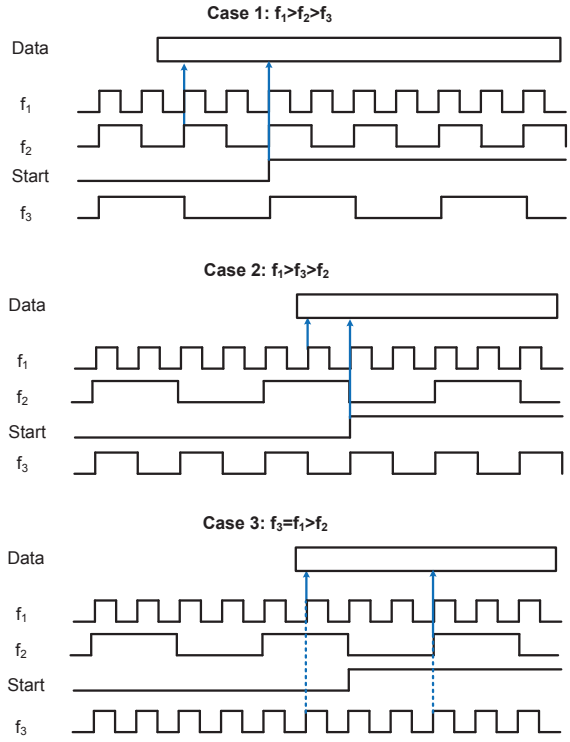


Figure 3-8: Clock frequency of the execution controller.

data memory. If the data is ready, it can be read or written successfully. If not, then the execution controller does not generate the according signals, therefore, the data will not be read or written from or into the data memory.

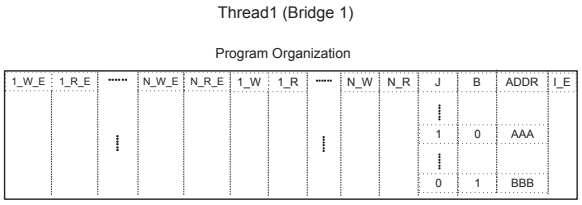


Figure 3-9: Control memory to get access to external data.

The execution program changes with the operating clock frequencies. As shown in Fig.3-10 (a), the processing elements are operated under different frequencies of clocks. Based on the assumption that $f1 > f3 > f2$ and the execution controller is

always operated under the highest speed, $f1$, we show how the program is modified. This dataflow is cut into two cases as shown in Fig.3-10 (b). For a buffer, the write speed is fast and the read speed is slow, or the write speed is slow and the read speed is fast.

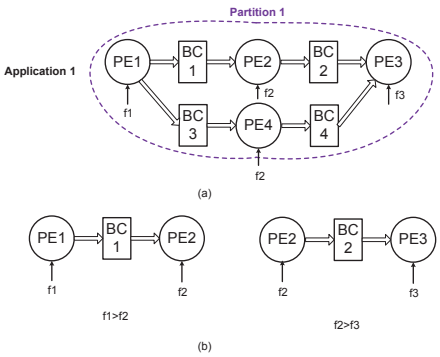


Figure 3-10: Dataflow with multi-rate clocks.

The following equations give the minimum required read offset when the write speed is f_1 and the read speed is f_2 (f_1 is less than f_2).

$$nr > M \cdot \left(\frac{1}{f_1} - \frac{1}{f_2} \right) \tag{3.2}$$

$$nr_{min} = \left[M \cdot \left(\frac{1}{f_1} - \frac{1}{f_2} \right) \right] + \frac{1}{f_2} \tag{3.3}$$

In the case when the write speed if faster than the read speed and $f1$ is the integral multiple of $f2$,

$$k = \frac{f_1}{f_2} \tag{3.4}$$

we add dummy zeros into the original program as shown in Fig.3-11 (a), where $k-1$ lines of zeros are added.

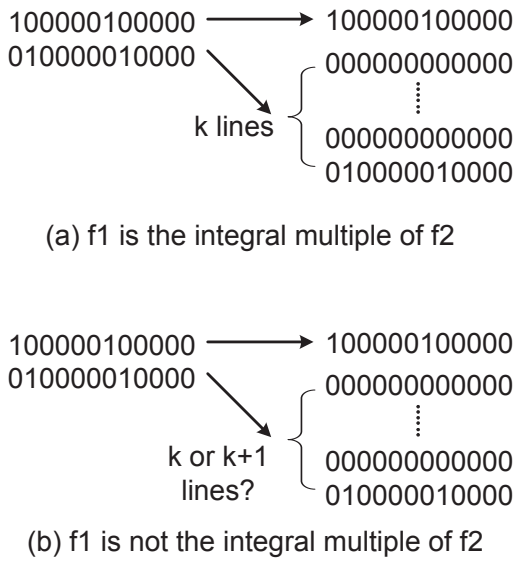


Figure 3-11: Program modification in multi-Rate application.

In the case when $f1$ is not the integral multiple of $f2$,

$$k = \left[\frac{f_1}{f_2}\right] \tag{3.5}$$

If one line changes to k lines, we take the risk of reading one useless data. If one line changes to $k+1$ lines, we take the risk of losing one useful data. One way to eliminate both risks is as following. Since we can not make the data come earlier, we choose the way of 1 line becoming $k+1$ lines for the first $\frac{F}{f_2}-1$ reading and 1 line becoming $k-\frac{F}{f_2}$ lines for the next $\frac{F}{f_2}$ reading, where F is the least common multiple of $f1$ and $f2$. At the same time, make sure each data holds longer than the following

value.

$$T = \left(\frac{1}{f_2} - \frac{1}{f_1}\right) \cdot \left(\frac{F}{f_1}\right) \tag{3.6}$$

For the case as shown in Fig.3-10 (b), where the write speed is faster than the read speed, dummy lines of zeros are added to avoid the wrong data to be read as shown in Fig.3-12.

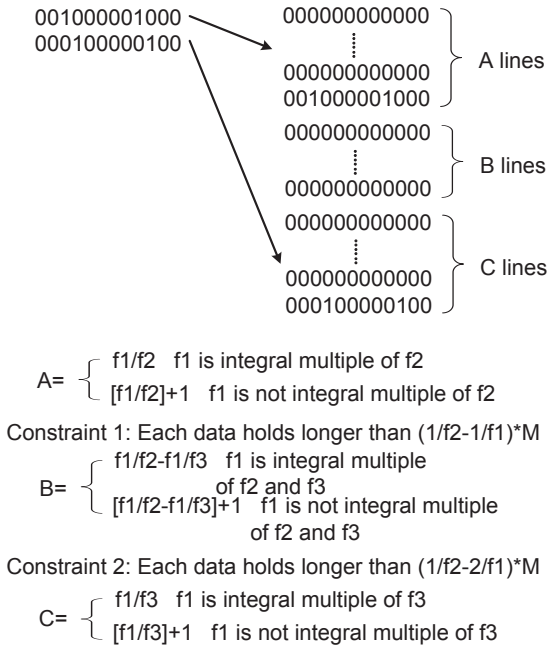


Figure 3-12: Program modification when read speed is faster.

Structure Controller

The resources in a partition are shown in Fig. 3-13 (a). The structure controller gets access to the address and data in partition memory and then controls the configura-

tion of the clock frequencies of all modules, the data conversion and interconnection modules by the values saved in the partition registers. The execution controller gets access to the address and data in the partition memory and then converts them into start signals to the buffers to control the timing of the application. Fig. 3-13 (b) shows the memory content in the partition memory, including the interconnection modules, the execution program and the clock frequencies of all the modules.

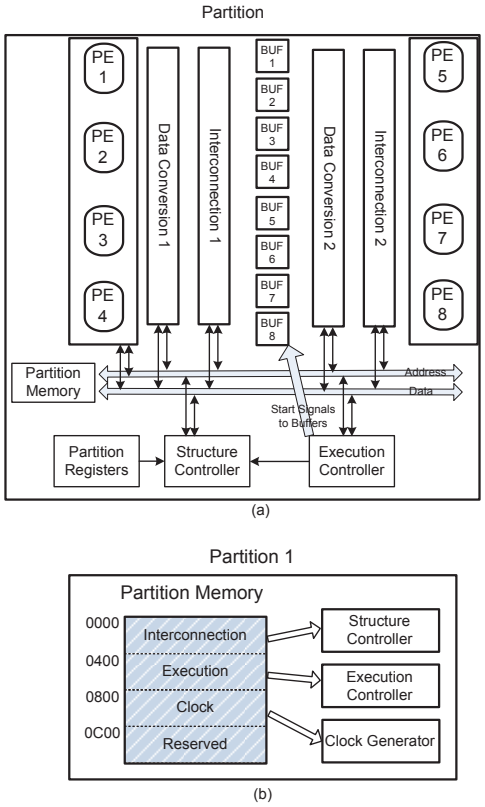


Figure 3-13: Resources in partition.

Structure and Execution Controller Interaction

The interaction between the partition execution and structure controller only happens when there is a branch in the execution program. If there is a branch in the execution

program, the execution controller suspends its operation and then sends the structure controller a reconfigure signal and the reconfiguration address. After the structure controller gets the reconfiguration signal and address, it starts the reconfiguration by the address.

In the initial case, the execution controller is triggered by the start and resume signals from the global execution controller. The structure controller is trigged by the configure signal from global structure controller. Each time after the structure controller finishes the configuration, it feeds back the global structure controller a finish signal.

3.2.3 Partition Datapath and Control Interaction

The processing element fabric, data conversion module and the interconnection module get the address and control data through the address and control bus. The structure and execution controller get the program from the partition memory through the buses. The buffer fabric gets the start signals of the execution directly from the execution controller.

3.3 Global Architecture Organization

3.3.1 Global Datapath

Global datapath consists of four partitions interacting through the bridges. The global data memory saves the data sent to the applications to process or from the applica-

tions after being processed. This platform allows for mapping of the applications into one partition or multiple partitions. The data is able to go within the partitions, between the partitions, or between the partition and the outside data memory through the bridges. The direction of the data from one partition to the other, or to the data memory is set by the interconnection module in the bridge. There are eight external buses in the platform, each of them can be used to read the data into or write the data out of our platform to the global data memory.

Bridge

When one application makes use of several partitions as shown in Fig. 3-14 (a), the bridges are necessary for the communication between the partitions. As shown in Fig. 3-14 (b), buffer 3 and 6 are substituted by the buffers in bridge 4. While buffer 9 and 10 are substituted by the buffers in bridge 3 and 4 to get access to the data outside. The bridges consist of multiple buffers, which are able to send data from one partition to another under the control signals from the global controller.

As shown in Fig. 3-14 (c), the bridge includes the buffers and two interconnection modules, the same with those within the partition. Another two interconnection modules indicate if it is connecting the data between two partitions, or between the partition and the external data. The buffers within the bridges have input and output ports on their both sides, which makes it convenient for the data to go both directions. The direction of the data is set by the multiplexers around the ports under the control of the global controller. Also, the data is converted to smaller size of data in the buffer through the data conversion modules.

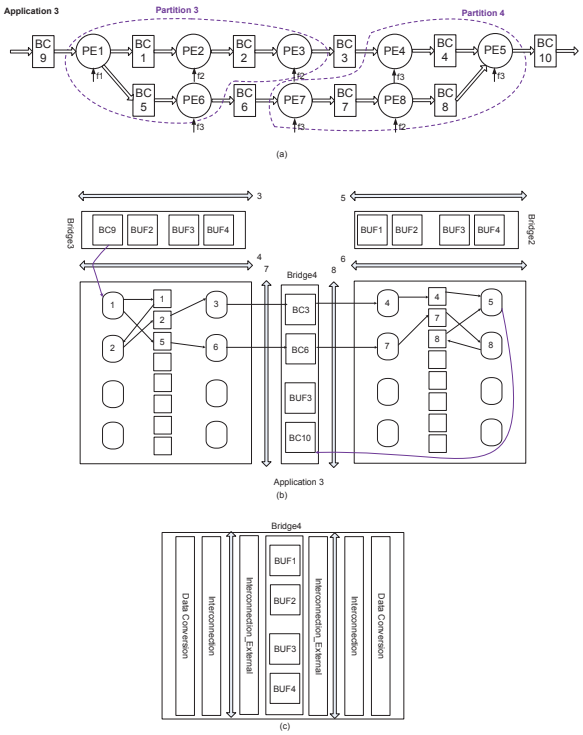


Figure 3-14: Bridge modules.

External Data Access

In order to obtain data from or deliver data to the data memory of the platform, buses connected to external I/O is provided for the exchange of the data through the bridges. Fig. 3-15 shows the dataflow when data comes from or is delivered out of the platform, where all the data exchange happens through the bridges. The buffers in the bridge substitute the buffers between the application and the outside data.

Before the data goes into the platform, initially the host processor writes the input data to some block in the data memory. There are eight blocks and status registers in the data memory, each of them communicates with the partition through its own bus. Therefore, the read and write of the four bridges to the external can happen concurrently. Each block is divided into two parts to make sure the writing

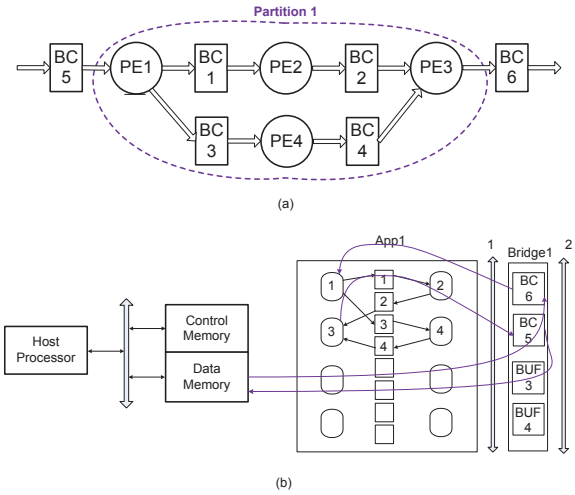


Figure 3-15: Dataflow with exchange of external data.

of the data into the data memory and the reading of the data by the applications can happen concurrently. The status register of each block indicates if the data in the block is ready to be read or written by the partitions. Once the data is ready to be fetched by the platform, host processor sets the status register to be "Readable". Then once the global execution controller starts to read the data into some bridge, it checks if the data is available first and then read the data into the processing element in the partition. In the other side, once the data is ready to be written out of the platform into the data memory, host processor sets the status to be "Writable". Then the global execution controller check the status of the data memory if it is ready to receive data. If it is ready, a start write signal is generated to the bridge from the global execution controller to make sure the data is written into the data memory. Later, host processor gets these data as the processing result.

Fig. 3-16 shows the structure of the bridges when connecting to the external I/O. Eight buses, all of them can be used as input bus or output bus, are placed

next to the four bridges to avoid the conflict when several applications exchange the data with the outside at the same time. The size of the bus is the same with the converted size of data to save the area occupied by the bus. While the size of the data memory is the integer multiple of the size of the bus to make full use of the resources. The bridge is connected with the external I/O through two buses, so that the processing elements in one partition can get access to the platform through one bus without conflict when the adjacent processing element in another partition is communicating with the outside of the platform through another bus. For the input path, a de-multiplexer is used to select the buffers through which the data comes into the partition. The two fan-in multiplexers are used to select if the input comes from the outside or from the processing elements in the partition. For the output path, M de-multiplexers are used to send the output data to another partition or to the outside through the external I/O. One M fan-in multiplexer is used to choose through which buffer in the bridge, the data is sent to the outside.

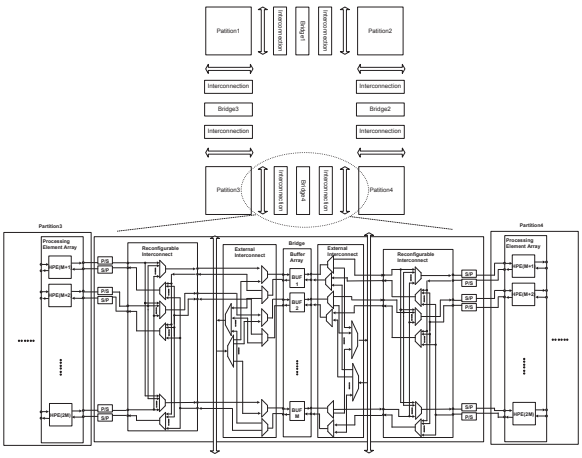


Figure 3-16: Bridges connected to external I/O.

The application is able to get the data from outside of the platform and then send

the processing result out of the platform. Since there are only two buses can be used by one partition, at most two processing elements in a partition can get access to the outside data at the same time. If one processing element is reading or writing data, then other processing elements in the bridge have to make use of the other bus to get access to the data outside. Also, this external buses makes it possible for two applications get access to the data outside concurrently. If one application gets access to the data outside through one buffer in one bridge, it occupies one bus. Then the other application, making use of the adjacent partition, can get access to the data through another buffer in the same bridge. But it uses the other bus next to the bridge.

3.3.2 Global Control

Global control path is divided into two layers under the control of host processor. The higher layer of the controller is the global controller consisting of one global structure controller and four global execution controllers. The lower level of the controllers consists of four partition controllers, each of them is divided into partition structure controller and partition execution controller. The global structure controller reads the data from global memory and distributes the data to each partition memory according to the address, which defines the partition or bridge the data belongs to. The global structure controller configures its own interconnection of the bridges and sends start signals to each partition structure controller to request it to configure their interconnection, clock frequencies and the data width. The global execution

controller executes its own program for the bridges and triggers the partition execution controller to start to execute their program. They operate under the command from the interface controller. While the interface controller accepts command from the host processor and then issues them based on the status in the status register. Besides, the execution controller requests the structure controller to reconfigure the structure whenever there is a "Branch" in the program for the global execution controller or partition execution controller.

Within the partition, the control data is saved in the partition memory, from where the partition structure and execution controller reads the data. The mapping of the processing elements is initially set in the processing element fabric. The partition structure controller configures the interconnection modules to set up the interconnection between the processing elements through the buffers by the control data. It also decides the data width before and after it is converted to smaller size to reduce the interconnection. The partition execution controller gives the start read and write time to the buffers by the execution data in the partition memory.

The interface controller coordinates the operation between the host processor and the controllers in the proposed platform. The host processor sends the command to our proposed platform containing the information of the configuration and execution of the applications. The overall system consists of host processor, interface controller and the reconfigurable architecture. The host processor writes the data into global memory and then sends the command to the interface controller, writes the application information, starting and ending address to the registers in the interface module. The interface controller accepts the command and then sends it to the global con-

troller. According to the addresses in the registers in the interface controller and the command from the interface controller, the global controller controls the timing and interconnection of the reconfigurable architecture and the partition controllers. After the configuration of the platform, the global controller tells the interface controller that the command has been finished. Then the interface controller feeds the finish signal back to the host processor. The host processor checks the status in the registers in the interface module to make sure the last command has been finished.

The interaction between the global execution and structure controller happens when there is a branch in the execution program or the application finishes the reconfiguration and requests to execute again. If there is a branch in the execution program, the global execution controller suspends its operation and then sends the global structure controller a reconfigure signal and the reconfiguration address. After the global structure controller gets the reconfiguration signal and address, it starts the reconfiguration by the address. Each time when a partition structure controller finishes the reconfiguration, it changes the status register. Once the global structure controller detects one application finishes the reconfiguration of all the partitions, it requests the global execution controller to start the execution again. Then the execution resume with its previous execution. Both the global structure and execution controller are operated under the command from the interface controller. The interface controller saves the command from the host processor and decides if to issue the command based on the current status of the global controllers.

Global Execution Controller

The execution of the platform is operated under the control of the global and partition execution controller. There are equal number of threads to that of the partitions to make sure these partitions can be activated at the same time. Each thread controls one partition and one bridge, in the way of sending the enable signal to the partition and start read and write signals to the buffers within the bridge. In the case when one application makes use of several partitions, say partition 3 and partition 4, the thread 3 and 4 are activated. The thread 3 and 4 send the enable signals to the partition 3 and 4 separately to start the partitions, and the thread 4 also sends the start read and write signals to the bridge 3 at the time indicated from the global program memory.

Fig. 3-17 shows the timing of the execution controller of the three applications in Fig. 1-2 after they are mapped into our proposed platform. When one application only makes use of single partition, the global execution controller only enables the according partition. When one application makes use of more than one partition, the global execution controller not only enables the according partition, but also sends the start read and write signals of the bridges. Once one partition is enabled, the execution controller within the partition begins to send the buffers the control signals of the reading and writing. From this figure, we get that the timing of multiple partitions can overlap if they are not used by the same application.

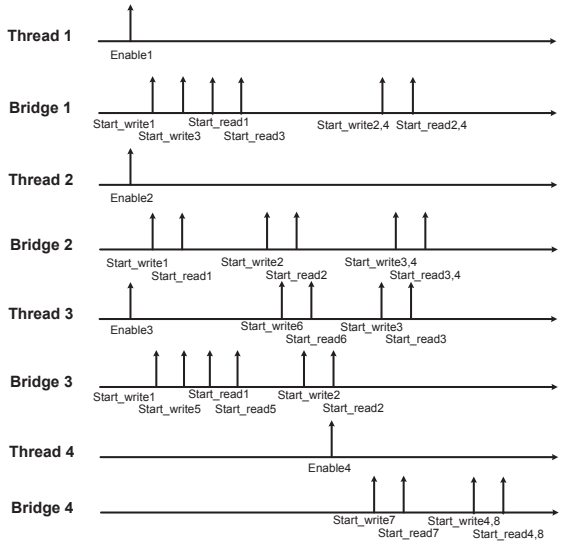


Figure 3-17: Timing of execution controller.

Global Structure Controller

The resources in the platform are shown in Fig. 3-18 (a). The host processor writes the external data into the global data memory, or fetches it from the global data memory. The global control memory is written by the host processor to load the applications into the platform. The interface controller accepts command from the host processor and then issues it to the global structure and execution controllers. The global data memory is used by eight data buses, while there is only one data and address bus for the global control memory. The global structure controller gets access to the address and data in the global control memory and then transfers them to the partition memories, the interconnection and clock frequencies of the bridges, the data conversion modules in the bridges. The global execution controller gets access to the address and data in the global control memory and then converts them into start signals to the bridges to control the timing of the bridges. Fig. 3-28 (b)

shows the memory content in the global control memory, including the data saved in the partition memory, the execution program of the bridge, the interconnection and clock frequencies of the bridge.

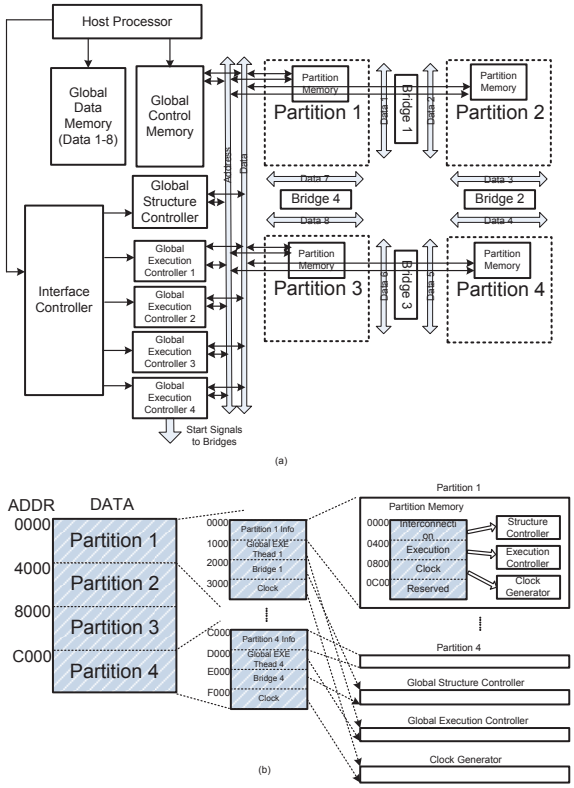


Figure 3-18: Resources in the platform.

Global Clock Network

To provide multiple clock frequencies to the partitions and the bridges, the clock generators are introduced to provide the clocks to the buffers within the partitions and the bridges. For the clock distribution in our proposed platform, we divide them into hierarchical layers to achieve the accuracy of the clocks by balancing the path of the clocks. From the global view, the global clock generator generates the

clocks for the partitions, the bridges and the global execution controller. The global clock network provides clocks to each bridge, two clocks used as the partition global clocks to each partition and four clocks to the global execution controller for the four threads. The clocks of the bridges are given directly from the global clock generator. For each buffer, two clocks are provided to control the writing and reading the buffer. To get better clock frequency accuracy, we tend to chose the strategy of providing all the bridges the same clock, including the reading and writing clocks. In this case, the clock frequency of all the bridges are set as the same by the global clock generator. But if the writing and reading speed has to be different required by the application, then we have to sacrifice the accuracy of the clocks. The clock frequencies of the partitions are set according to the requirement of the applications. The frequency of the global execution controller is set to the maximum value among the clock frequencies of the processing elements in the partitions to avoid losing data. The the clock frequencies is set one by one and finished in the initiation steps when the applications are configured. Therefore, this platform needs the same number of clock cycles with the clocks to finish the global setting.

Within the partition, a single clock generator provides the clocks to the processing elements fabric, the buffer fabric, the data conversion modules and the partition execution controller, based on the global clocks from the global clock generator. It improves the accuracy of the clock frequencies in the partition domain. The clock frequencies of the processing elements are decided by the requirement of the applications. Normally we constrain the clock frequencies of the buffers and the data conversion modules the same to avoid inaccuracy. Each partition has its own par-

tition clock generator to balance the clock path between different partitions. Once the clock generator is enabled, the global select signal decides which global input clock is based on to generate the output signal. The divisor defines the output clock frequency, which equals to the global clock frequency divided by the divisor.

The clock frequencies of the global clock generator are configured during the first configuration steps, together with the timing and interconnection information by the content saved in the global memory. During the execution of the platform, the clock frequencies are fixed. The reading and writing of the buffers are controlled under these clock decided by the application. But during the execution of the platform, if there is a request of using a different clock in a buffer, the execution has to be stopped and then it goes into the configuration mode to reconfigure the new clock frequencies. Since the reconfiguration command comes with the address range which needs the reconfiguration, it is possible to only reconfigure the modified clock. Therefore, the reconfiguration of the clocks includes the stop of the current application and the update of the modified clocks by the structure controller. The reconfiguration steps only reconfigures the modified part instead of configuring all the components, so multi-rate applications are supported and are flexible to change.

For the clock distribution in our proposed platform, we divide them into hierarchical layers to achieve the accuracy of the clocks by balancing the path of the clocks. From the global view, the global clock generator generates the clocks for the partitions, the bridges and the global execution controller. Global clock network provides $8n$ clocks to the four bridges, two clocks used as the partition global clocks to each partition and four clocks to the global execution controller for the four threads. The

clocks of the bridges are given directly from the global clock generator. For each buffer, two clocks are provided to control the writing and reading the buffer. To get better clock frequency accuracy, we tend to chose the strategy of providing all the bridges the same clock, including the reading and writing clocks. In this case, the clock frequency of all the bridges are set as the same by the global clock generator. But if the writing and reading speed has to be different required by the application, then we have to sacrifice the accuracy of the clocks. The clock frequencies of the partitions are set according to the requirement of the applications. The frequency of the global execution controller is set to the maximum value among the clock frequencies of the processing elements in the partitions to avoid losing some data. The setting of the clock frequencies is set one by one and finished in the initiation steps when the applications are configured. Therefore, this platform needs $8n + 9$ clock cycles to finish the global setting.

3.3.3 Global Datapath and Control Interaction

Fig. 3-19 shows the interaction between the global datapath and control modules. All of the modules get access to the address and control data bus. Each partition memory gets their data from the global memory. The bridges gets their control data of interconnection and data conversion from the global memory. The global structure controller sends the partition the configuration signal and address. The global execution controller sends each partition execution controller to start to execute. It also sends the signals to the bridges.

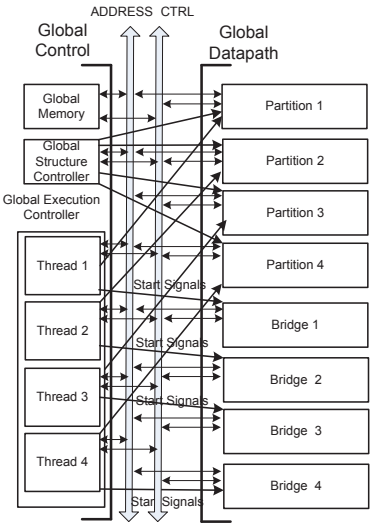


Figure 3-19: Global datapath and control interaction.

Fig. 3-20 shows the interaction between the global and partitions. The address and control data bus are shared among the global memory, global structure and execution controller, the global clock generator and the partition memories. The eight external data buses, two of them for each bridge, are connected to the data memory. Data is transmitted between the partitions and bridges. The global structure and execution controller sends the control signals to the partition controllers. The global clock generator sends the base clock to the partitions.

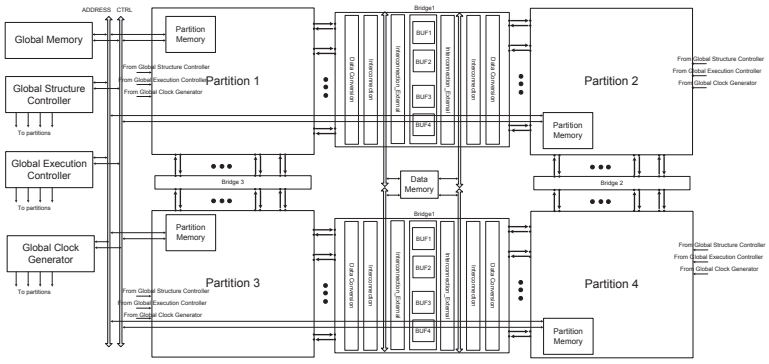


Figure 3-20: Global to partition interaction.

3.4 Configuration and Reconfiguration

3.4.1 Host Processor Interface Configuration

To operate our proposed platform, a host processor is in charge of writing application specific data into the memory and delivering the command to the platform. The communication between the host processor and the controllers in our platform is controlled by the interface controller. As shown in Fig. 3-21, the host processor writes the data into the global memory and then sends the command to the interface controller, writes the application information, starting and ending address to the registers in the interface module. The interface controller accepts the command and then sends it to the global controller. According to the addresses of the registers in the interface module and the command from the interface controller, the global controller controls the timing and interconnection of the reconfigurable architecture and the partition controllers. After the configuration of the platform, the global controller tells the interface controller that the command has been finished. Then the interface controller sends the finish signal back to the host processor. The host processor checks the status in the registers in the interface module to make sure the command has been finished.

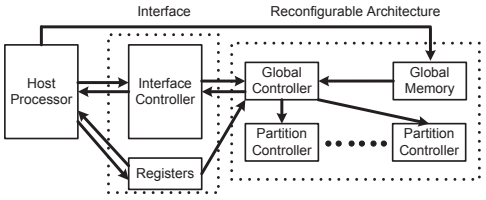


Figure 3-21: Overall structure including host processor.

There are six states in the interface controller as shown in Fig. 3-27. Before any application is loaded into the platform, the interface is in "Idle" status. When the system is in "Idle" status, there is only one way to activate it by "Load" command. After the load of the parameters, the system begins to execute. During the execution of the system, there are three possible command to stop it: "Suspend", "Halt" and "Stop". For the command of "Suspend", the current counting value will be reset to zero. If the command of "Execution" comes during the status of "Suspend", then the application starts to operate from the beginning. If there is a command of "Halt" during the execution, the current counting value will be saved in according registers. At this time, if there is a command of "Resume", the value in the registers will go on counting. If the command of "Stop" comes, then all the registers saving the structural and execution information are reset. The system can not execute again unless it is reloaded.

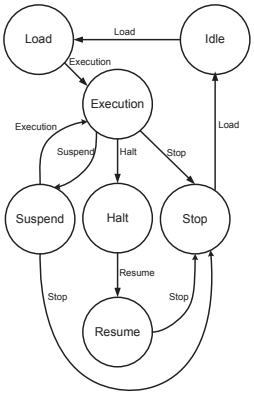


Figure 3-22: Status transition in interface controller.

3.4.2 Memory Mapping

From the view of host processor, it writes the program into a global memory space as shown in Fig. 3-23 by the address. This global memory space includes the control memory and the data memory, each of which consists of four parts of information related to each partition, bridge and those memory mapped from the partition memory. The control memory saves the information related to the control structure, while the data memory saves the data going into or out of the platform.

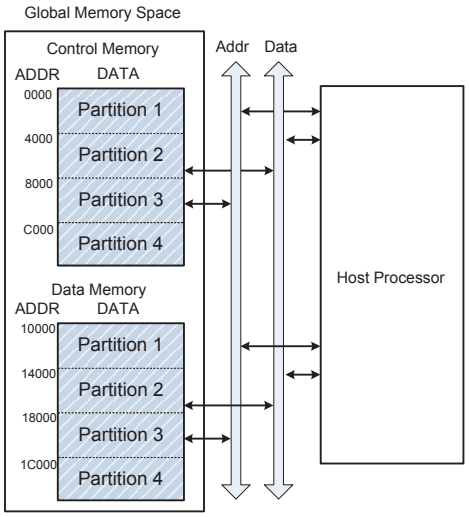


Figure 3-23: View from host processor.

Within our architecture, the global structure controller distributes the program in the control memory into the memories and registers within the partitions, global structure and execution controller as shown in Fig. 3-24. The partition related information which is saved into each partition memory includes the interconnection between the processing elements, the program indicating the execution and the clock frequencies. The program indicating the execution of each thread in the global exe-

cution controller and the interconnection of the bridges are saved into the registers in global execution and structure controller separately. The data saving the clock frequencies of the bridges is distributed to each registers in the bridge, providing them with the operated clock frequencies.

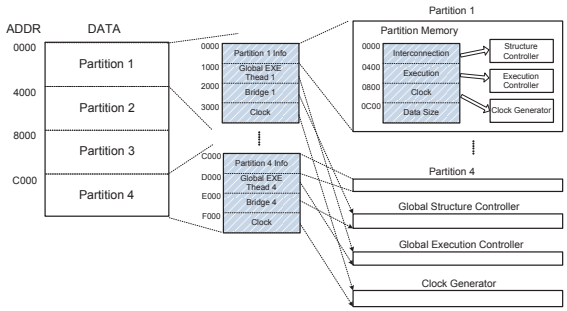


Figure 3-24: Control memory structure.

Within the partitions, the partition structure controller distributes the data saved in the partition memory into the registers within the partition structure and execution controller, and the registers saving the clock frequencies of the processing elements and the buffers.

Fig. 3-25 shows the mapping from global memory space to physical global memory and then to the scatted memories and registers within the partitions. The address in the global memory space consists of a base address "p" and the offset "d", while the address of the physical global memory consists of a base address "f" and the offset "d". The offset is the same for the global memory space and the physical global memory. The mapping table shows the mapping of the base address.

The mapping from the global memory space to the physical global memory is necessary, since the size of the global memory space can be larger than that of the

physical global memory. But only four partitions can be mapped at the same time. Therefore, this structure makes it possible for large amount of applications. Also, the mapping from the physical global memory to the memories and registers within the partitions is necessary, since the program of the application is contiguous in the global memory, while scatted in the memories and registers within the partition.

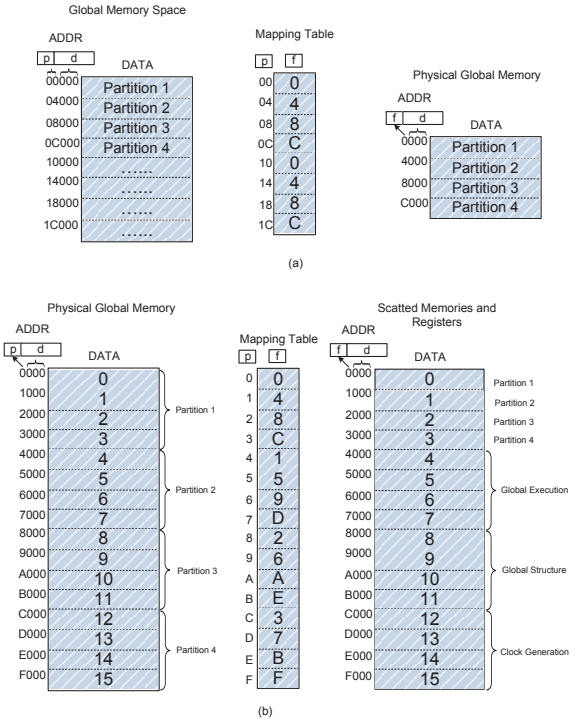


Figure 3-25: Mapping from physical global memory to scatted memories and registers within partitions.

Fig. 3-26 shows the distribution of the global memory space. By using three levels of data transfer, the data is saved into the registers physically located into different partitions. This hierarchical memory structure make it easy for the host processor to write the program into the architecture, since the host processor do not need to know the structure of the architecture. By distributing the data from the global memory to different partition memories and registers, it is easy for each application operate

properly, without compromising who gets access the global memory first.

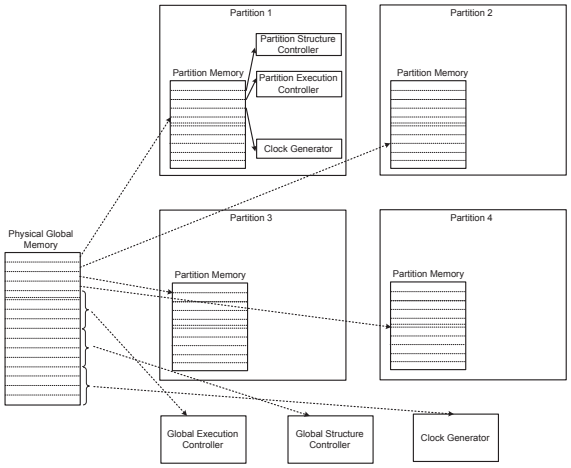


Figure 3-26: Memory structure overview.

The control memory is divided into four parts, each of them consists of the control information of the partition and the global control information of the adjacent bridge. Within the partition, the structural and execution content, together with the clock frequencies and the data sizes of each module is included. The structural content defines the interconnection within the partition and the reconfiguration information; The execution content gives the timing within the partition and the branch or jump address. The clock frequencies of the processing elements, buffers, parallel serial modules and the partition execution controller is saved in the global memory to realize the multi-rate application. Also, the input and output data sizes of the buffers within the partition and adjacent buffer are saved to realize the multi-bit application. The data format in the global execution controller is the same to that in partition execution controller, including the timing and the address. The data in global structure controller saves the interconnection of the bridge and the application related infor-

mation, including how many partitions are in each application and the configuration address of each partition. The global clock part includes the clock frequencies of the input and output data of the buffers in the bridge, and the clock frequencies of the global execution controller.

The data memory is divided into eight parts, each of them use one external bus. Within the eight parts, half of them is reserved for the read and the other half is reserved for the write. This division makes it possible for the activity of the buses and the read and write of one bus to be overlapped.

3.4.3 Application Configuration

The configuration of the applications starts from host processor. The host processor issues the command, writes the control data and external data into the memory and fills the tables in the interface controller with the information of the applications. There are two tables in the interface controller needed to be filled as shown in Table 3.1 and Table 3.2. Not only the host processor sends the command to load the applications, but also the host processor gives the address information of each partition and the application information to the interface controller as shown in Table 3.1. The application information from the host processor is saved in the interface controller as shown in Table 3.2. According to these address information, the global structural controller begins to distribute the control data to the partitions and bridges. After all the partitions finish the configuration, the global structural collects those finish signals and then send them to interface controller together with the bridge configu-

ration finish signal. Each time when the partition controllers and the bridges send the global controller finish signal, the global controller sends it back to the interface controller to update the status information in Table 3.1, the partitions are in "Load" status in this case. The interface controller collects the finish signals and checks the status register in the partitions if all the status of the partitions have been filled. If so, it will send the host processor a finish signal of the configuration.

Table 3.1: Status Table in Interface Controller

	Load	Execution	Stop	Suspend	Start Address	End Address
$P0$	X				00000000	00001111
$P1$	X				00010000	00011111
$P2$	X				00100000	00101111
$P3$	X				00110000	00111111
$BB0$					01000000	01001111
$BB1$					01010000	01011111
$BB2$					01100000	01101111
$BB3$					01110000	01111111

Table 3.2: Application Table in Interface Controller

Application	P0	P1	P2	P3	BB0	BB1	BB2	BB3
$App1$	X							
$App2$		X						
$App3$			X					
$App4$				X				

Fig. 3-27 shows the action of the global and partition controller corresponding to the command from the interface controller. When the system is in "Idle" status, there is only one way to activate it by the "Load" command. Once after the "Load" command, the program memory and the tables are filled in each partition and global structure and execution controller. The memory and the table in the partition structure controller save the interconnection configuration of the buffers within the partition and the start and end address of the memory during the whole configuration or partial configuration separately. The memory and table in the par-

tition execution controller save the start read and write time of the buffers within the partition, and the jumped or branched address during the iteration of execution or a partial reconfiguration. These information is distributed to each partition by the base addresses of the memory. The memory information in the global structure and execution controller are similar to that in the partition structure and execution controller except that it is configuring the interconnection and timing of the buffers in the bridges. Also, the table in the global structure controller saves the content of the applications, how many partitions are in one application, which bridge belongs to the application and the start and end address of the configuration of the bridge. The status register in the global structure controller is reset to "0".

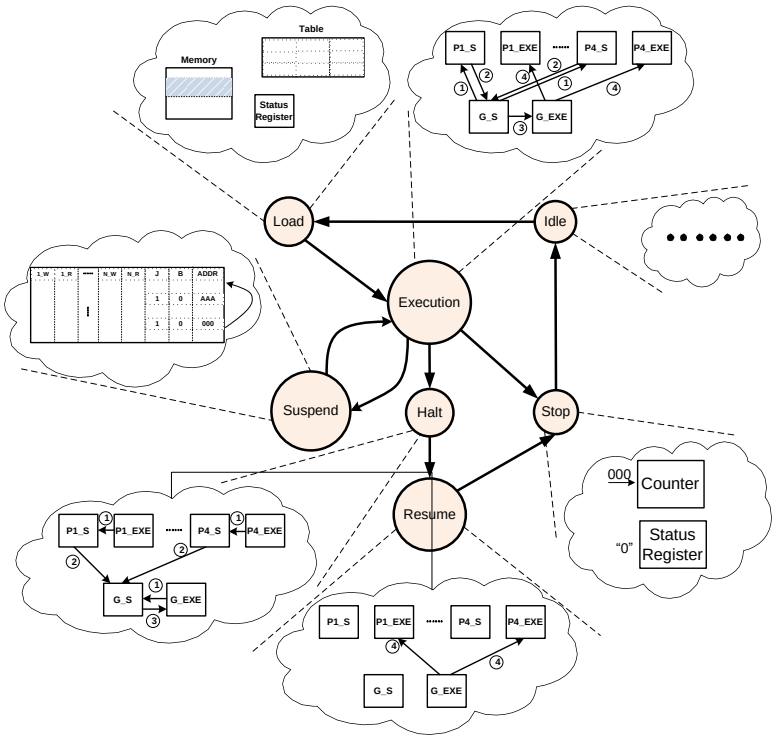


Figure 3-27: Commands from host processor.

After the load of these parameters, the system begins to execute. The global

structure controller sends each partition structure controller a start signal. After the partition structure controller gets this signal, it configures the whole partition, which is addressed "111" in the table. Each partition structure controller sends back a finish signal to the global structure controller after it finishes the configuration of its partition. After all the partitions feed back a finish signal, which is verified according to the table in the global structure controller, it sends several finish signals and a type signal "0" to according threads in the global execution controller, indicating the global execution controller to execute and the current status is initial configuration. Then each thread sends a start signal to trigger the partition execution controller.

During the execution of the system, there are three possible command to stop it: "Suspend", "Halt" and "Stop". For the command of "Suspend", the current counting address in the global and partition execution controller is reset to "0" and start to count after a command of "Execution" comes.

If during the execution, there is a "Halt" command, the current counting value will be saved in the registers. At this time, if there is a "Resume" command, the value in the registers will go on to count. If the command of "Stop" comes, then all the registers saving the structural and execution information are reset. The system can not execute again unless it has been reloaded.

The command of "Halt" comes when there is a request of partial reconfiguration of an application. The affected partition execution controllers and the global execution controllers saves the current address and then sends a configure signal and the address to according partition structure controllers and global structure controller. The partition structure then configures part of the interconnections within the par-

tition by the address. The global structure controller sets the status register to "1" and then configures the affected bridge by the address sent from global execution controller. At the end of configuration, the global structure controller collects the finish signals from partition structure controllers and then sends a resume signal to according threads in the global execution controller. Then the threads who get the resume signal trigger the partition execution controllers. These partition execution controllers go back to the previous end address and start to execute again.

Once the "Stop" command comes, all the counters in the controllers are reset to "0". The status register in the global structure controller is reset to "0". Then the system goes into "Idle" status.

3.4.4 Global Configuration

The resources in the platform are shown in Fig. 3-28 (a). The host processor writes the external data into global data memory, or fetches it from the global data memory. The global control memory is written by the host processor to load the applications into the platform. The interface controller accepts command from the host processor and then issues it to the global structure and execution controllers. The global data memory is used by eight data buses, while there is only one data and address bus for the global control memory. The global structure controller gets access to the address and data in the global control memory and then transfers them to the partition memories, the interconnection and clock frequencies of the bridges, the data conversion modules in the bridges. The global execution controller gets access to the address

and data in the global control memory and then converts them into start signals to the bridges to control the timing of the bridges. Fig. 3-28 (b) shows the memory content in the global control memory, including the data saved in the partition memory, the execution program of the bridge, the interconnection and clock frequencies of the bridge.

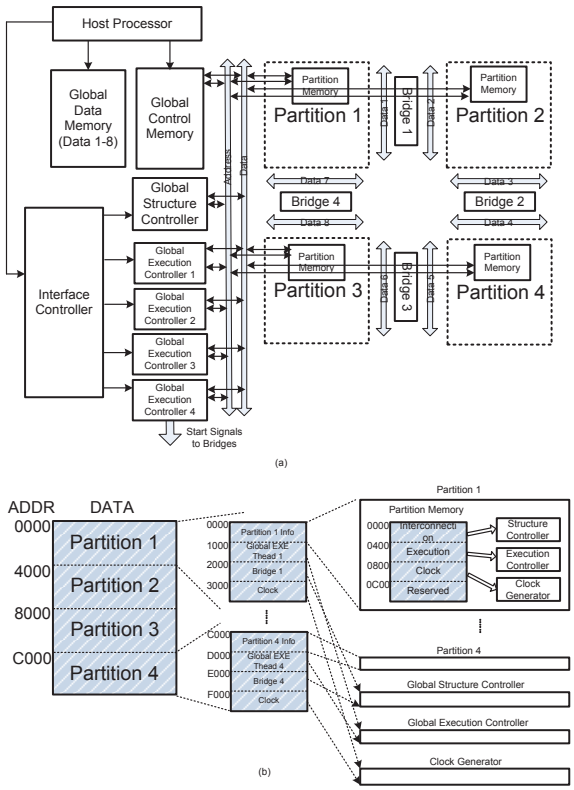


Figure 3-28: Resources in the platform.

The global structure controller configures the interconnection of the bridges, decides when to start the partition structure controller by the command from the global execution controller or an initiation configuration request, or sends the global execution controller the command by the status from the partition structure controller. As shown in Fig. 3-29, the FSM decides the output and the current address of the

counter according to the input and the information in the table.

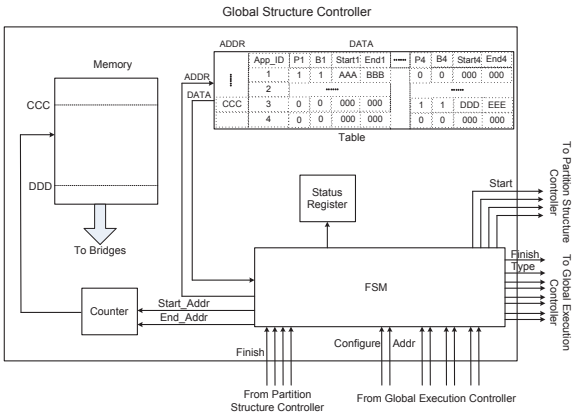


Figure 3-29: Global structure controller.

The global structure controller supports both the initiation configuration and reconfiguration. If the global structure controller initially configures the interconnection, the FSM reads the start and end address from the table, then sets the counter to configure the interconnection of the bridges and sets the status register to "0". When the value in the status register is "0", the FSM sends a start signal to start each partition structure controller to configure the interconnection of the partition. When the partition structure controllers send back finish signals to the global structure controller, the status register is changed to "1" and the FSM will check if each application has been finished by the application table. If one application has finished its configuration, the global structure controller sends according threads in the global execution controller a finish signal and the current type "resume". Normally, if the global execution controller decides to reconfigure the interconnection, each thread sends a configure signal and an address to the global structure controller. The information table lists each application and its occupying partitions, bridges and the start

and end address of the configuration of the bridges. The global structure controller fetches the start and end address of the bridges which need reconfiguration from the table, and then sets the counter to configure these interconnections.

The configuration of the interconnection of the bridges is shown in Fig.3-30. Fig.3-30 (a) shows the memory contents configuring the interconnection between the partitions. The base address and the offset address define which bridge the data is about, and which pair of multiplexers the data is configuring. The data is used as the selection value of the multiplexer. In the switches module, the multiplexers are divided into three levels: Mux5 and Mux6 are used to select the partitions; Mux1 to Mux4 are used to select the processing element: Mux1 and Mux2 select the input, while Mux3 and Mux4 select the output. Therefore, for one bridge six sets of multiplexers are needed to configure the interconnection, each set has the same number of multiplexers with the buffers. The Buffer ID bits will distinguish these sets of multiplexers as shown in Fig. 3-30(a). In one bridge, six N-bit registers are used to save the interconnection information. In this way, how the processing elements are connected through the bridges and communicate with the processing elements in the other partitions is configured.

3.4.5 Partition Configuration

The partition structure controller configures the interconnection of the buffers within the partition, after being triggered by the global structure controller or the partition execution controller. As shown in Fig. 3-31, the FSM fetches the start and end

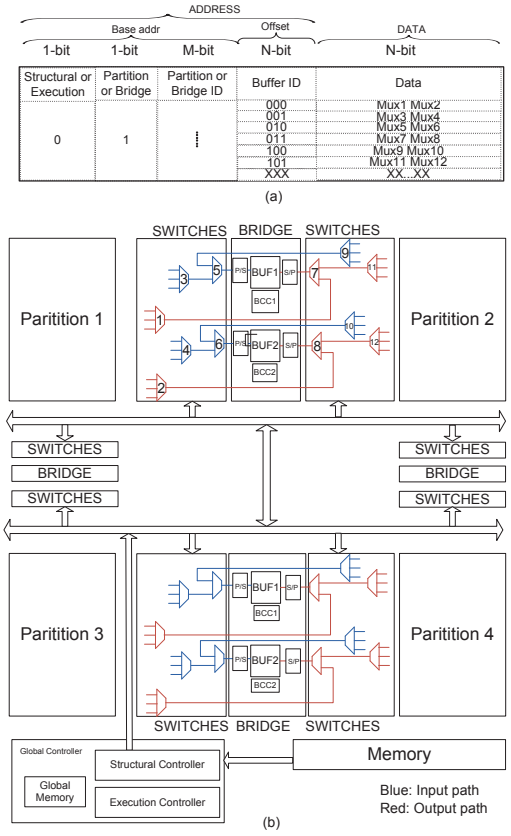


Figure 3-30: Global interconnection.

address of the configuration and sets the counter to the configured address in the memory.

The partition structure controller supports whole configuration and partial configuration. If the partition structure controller is trigged by the global structure controller, the FSM will read the start and end address located in "111", which denote the whole configuration. If the partition structure controller is trigged by the partition execution controller and the configuration address is "CCC", then the FSM will set the counter to count from "DDD" to "EEE", which is partial configuration of this partition. After the configuration of the partition, the FSM sends a finish signal to the global structure controller. If the configuration address from the parti-

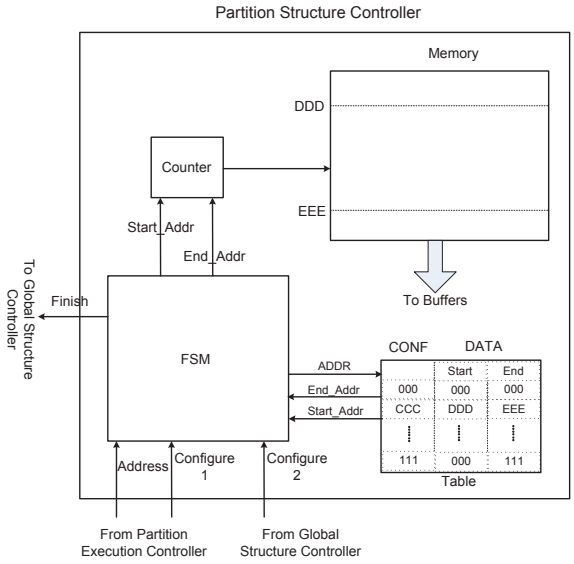


Figure 3-31: Partition structure controller.

tion execution controller is "000", then the FSM does nothing and then sends back a finish signal. This characteristic is used to synchronize the operation when several partitions in an application need configuration, while some other partitions in the application do not need the configuration.

The configuration of the interconnection within the partition is shown in Fig.3-32. Fig.3-32 (a) shows the memory contents configuring the interconnection between the processing elements and the buffers. The base address and the offset address define which data is saved in the memory of the partition structure controller, and which multiplexer the data is configuring. While the data is used as the selection value of the multiplexer. For buffer 1, Mux1 to Mux4 are used to configuring its interconnection. For example, if the data for Mux1 is "1000", then the processing element 11 is selected to connect with buffer 1.

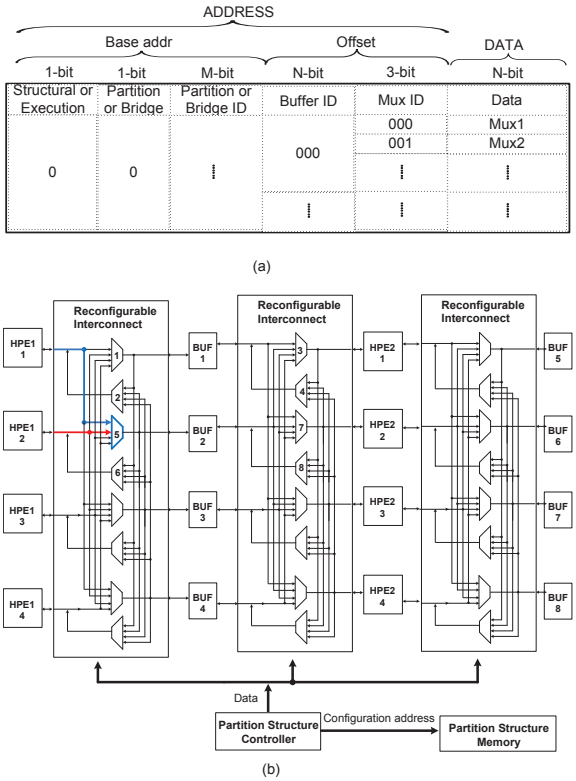


Figure 3-32: Partition interconnection.

3.4.6 Dynamic Reconfiguration

Our proposed platform allows for dynamic configuration during the execution of the applications. The request of dynamic configuration comes from buffer sharing scheme. If the buffer activity among some buffers do not overlap with each other, then these buffers can be shared. Fig. 4-13 shows the timing before and after buffer 1 and buffer 2 are shared by only using buffer 1.

After the end of the data access of buffer 1, buffer 2 is replaced by buffer 1. The start read and write previously sent to buffer 2 is provided to buffer 1 instead. Also the processing elements connected to buffer 2 are connected to buffer 1 now. Therefore, both the structure controller and the execution controller need to reconfigure current

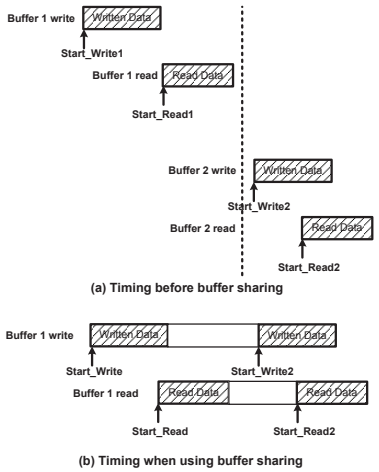


Figure 3-33: Timing before and after buffer sharing.

partition.

Given the application 1 shown in Fig. 3-34, Fig. 3-35 shows the buffer activity of this application when it is operated under a single clock frequency with data size of all the processing elements and the buffers 32-bit. Here we assume the logic latencies of all the processing elements are 2, so buffer 1 and 3, buffer 2 and 4 are active at the same time.

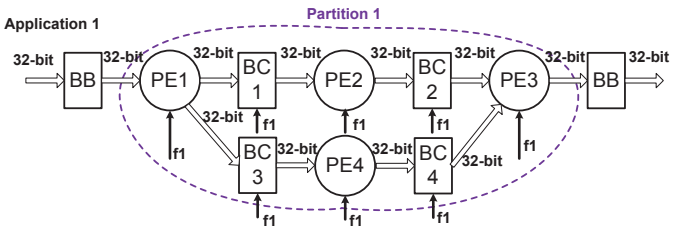


Figure 3-34: Dataflow of application 1.

Given the application 1 shown in Fig. 3-36 (a), Fig. 3-37 shows the buffer activity before and after buffer 1 and 2, buffer 3 and 4 are shared by buffer 1 and 3 separately. Here the data sizes of all the processing elements and the buffers are all 32-bit, and

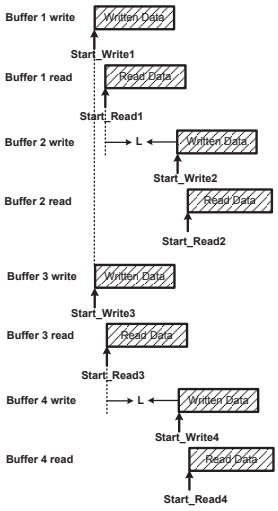


Figure 3-35: Buffer activity of application 1.

all of the logic latencies of the processing elements are 2.

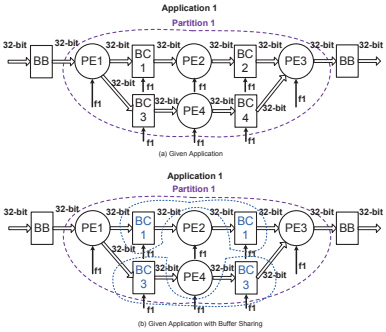


Figure 3-36: Dataflow of application 1 with buffer sharing.

Given the application 3 shown in Fig. 3-38 (a), Fig. 3-39 shows the buffer activity before and after buffer 1 and 2, buffer 3 and 6, buffer 7 and 8 are shared by buffer 1, 3 and 7 separately. Here the data sizes of all the processing elements and the buffers are all 32-bit, and all of the logic latencies of the processing elements are 10, in which case the buffer sharing scheme is applicable.

When the buffer sharing is applied for multiple partition based applications, it is only applied within the buffers in one partition, or in one bridge. The buffer from

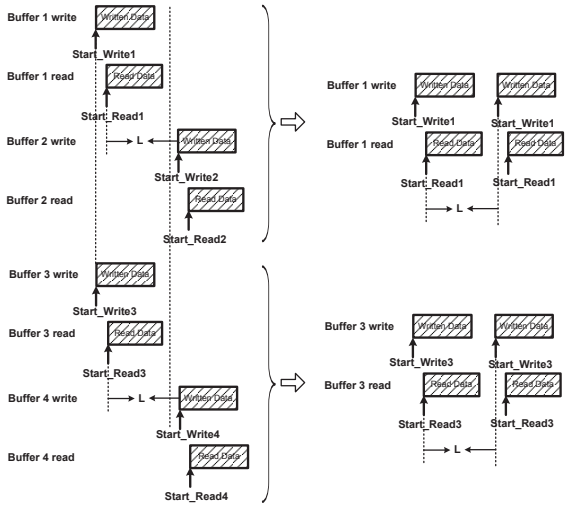


Figure 3-37: Buffer activity of application 1 with buffer sharing.

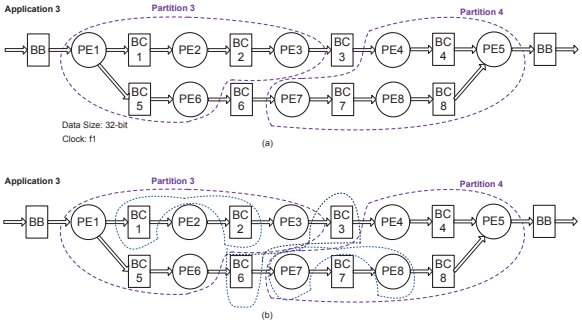


Figure 3-38: Dataflow of application 3 with buffer sharing.

one partition can not be shared with that in another partition, or bridge, or between different bridges.

Single Partition based Application

This section gives how the single partition application is mapped into the proposed platform. Fig. 3-44 shows the program distributed to each partition memory or registers in the global structure and execution controller. The memory in the partition structure controller contains the interconnection information between the partitions

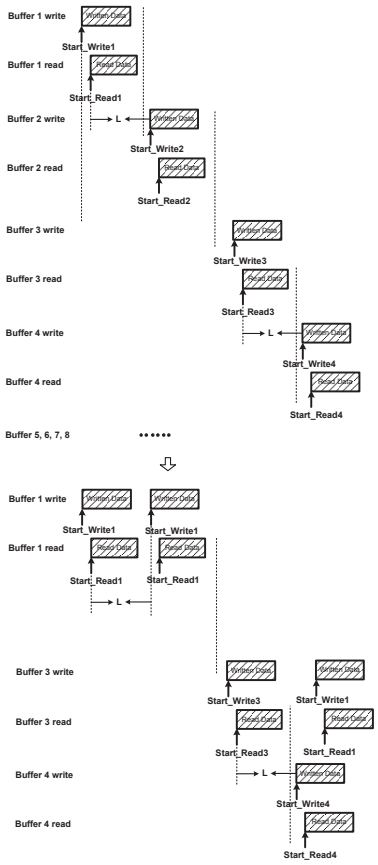


Figure 3-39: Buffer activity of application 3 with buffer sharing.

and the buffers by setting the multiplexers, while the configuration address and data saves the modified interconnection when there is a reconfiguration. For the partition execution controller, the memory saves the timing information of the buffers within the partition and the configuration address and the configuration address and data saves the modified interconnection within the partition structure controller. The program is executed one by one with each clock cycle and jumps to a fixed address if there is a request of reconfiguration. The first N-bit data is used to indicate when to enable the processing element. Since clock is continuously provided to the processing element, we disable the processing elements except the effective data is

going to be processed. The next $2N$-bit indicates the start read and write time of the buffers. The memory structure of the global controller is the same to that of the partition controller, except that global structure and execution controller configures the interconnection and timing of the bridges. By using the same format of memory in the execution controller, the programs in the partition execution controller and global execution controller branch at the same point, so that the reconfiguration is synchronized. Besides, the constitutions of the applications are saved in global structure controller and another status register saves if the current status is in initial configuration or reconfiguration.

The following content shows how the single partition application is reconfigured if there is a request of reconfiguration. The memory which needs the reconfiguration is shown in Fig. 3-41 when the buffer sharing scheme is applied. After a command from the interface controller indicating the initial configuration or requested by the global execution controller indicating the reconfiguration, the structure controller of partition 1 starts to configure the interconnection within it. Then the partition 1 executes after the configuration of type 1. At this time, if there is another command of reconfiguration together with the address from the interface controller, the program jumps the the branched address of type 2 and starts the reconfiguration with the shared buffer interconnection. With this reconfiguration where two buffers are active at different time, the partition executes again, jumping to the program with the second configuration. Therefore, if one buffer is used many times, then the same number of memory space, including the structure memory and the execution memory, is necessary to reconfigure the buffer.

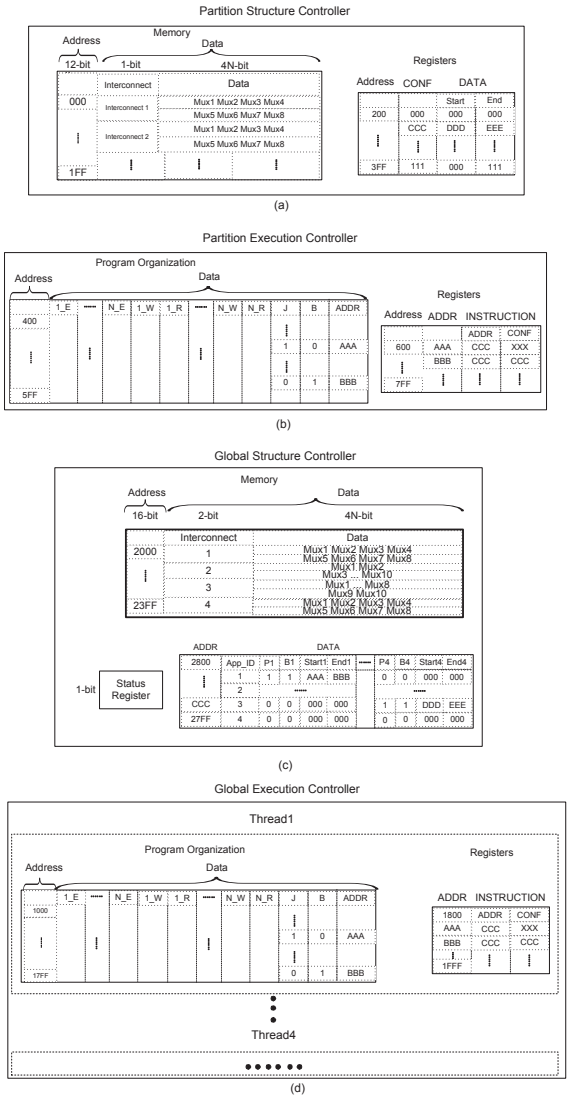


Figure 3-40: Content in control memory.

Multiple Partitions based Application

In the case when one application makes use of several partitions, such as application 3 shown in Fig. 3-42 where the data sizes are all 32-bit and the clock frequencies are all $f1$, not only partition 3 and 4 are mapped in the way shown in single partition application case, but also the information of the bridge is mapped into the memory and program in the global structure and execution controller. Table 3.3 shows the

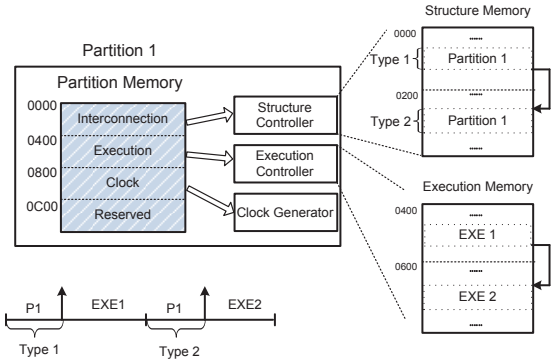


Figure 3-41: One partition based application in buffer sharing.

memory content in the global structure controller, where the buffers in the bridge 3 are configured. For the program in the third thread in the global execution controller, it decides the buffer activities in the bridge as shown in Table 3.4. For the fist ten cycles, there is no buffer activated in the bridge. Once the effective data comes out of partition 3, the buffers in bridge 3 starts to write and read the data. After the data goes through the partition 4, the program jumps to the first address. The numbers of zeros before and after the read and write of the buffers within the bridges are equal to the number of programs saved in the execution controllers of partition 3 and 4, through which the execution of different partitions is synchronized.

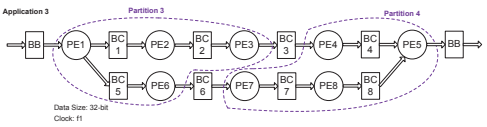


Figure 3-42: Dataflow of application 3.

Table 3.3: Memory in global structure controller

Address	Buffer ID	Data
0001000000000000	000	10001000
	001	01000100

Table 3.4: Program in global execution controller

Address	1E 2E 3E 4E 1W 1R 2W 2R 3W 3R 4W 4R	J	B	addr
0000100000000000	000000000000	0	0	XXXX
	...			
	100010100000	0	0	XXXX
	110001010000	0	0	XXXX
	...			
	000000000000	1	0	0000

The following content shows how the multiple partitions based application is reconfigured if there is a request of reconfiguration.

For the multiple partitions based application, the sequence of the reconfiguration is shown in Fig.3-43 when the buffer sharing scheme is applied. In the initial configuration, partition 3, 4 and the bridge configures their interconnection. Based on this configuration, application 3 starts to execute. At this time, if there is a request of reconfiguration due to buffer sharing or dynamic interconnection configuration, the counter jumps to the start address of type 2 and starts to configure partition 3 and 4 again. At the same time, the global structure controller does the same thing with the structure controllers of partition 3 and 4. It does the reconfiguration of the bridge 3. After the reconfiguration of application 3, it starts to execute. The reconfiguration of multiple partitions based application is similar to that of single partition based application, except that the global structure controller has to monitor when is the end time of the reconfiguration. Therefore, if part of the application do not need the reconfiguration, say partition 3, there is still a finish signal sent the the global structure controller indicating the end time of the reconfiguration.

The reconfiguration of the multiple partitions application is the same with that of single partition application, except that the interconnection and execution of the

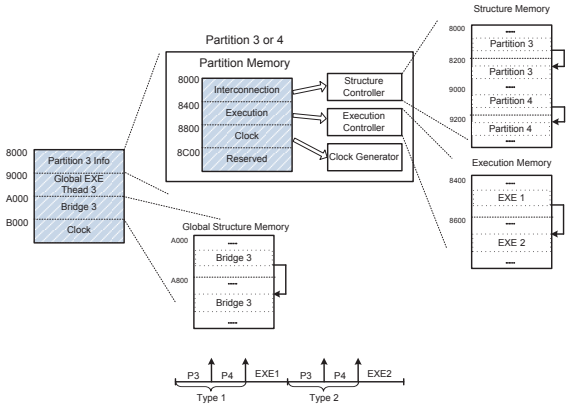


Figure 3-43: Reconfiguration of multiple partitions application.

bridges are changed. Table 3.5 and Table 3.6 shows the memory content of the global structure controller and execution controller when buffer sharing scheme is applies. The global structure controller configures buffer 3 first and then the global execution controller starts the buffer 3 to execute. Then there is a branch in the program so that buffer 3 is reconfigured, replacing buffer 6. Later the program in the execution controller memory jumps to the point where buffer 3 is activated again.

Table 3.5: Memory in global structure controller with buffer sharing

Address	Buffer ID	Data
0101000000000000	000	10001000
	...	...
0101100000000000	000	01000100

Table 3.6: Program in global execution controller with buffer sharing

Address	1E 2E 3E 4E 1W 1R 2W 2R 3W 3R 4W 4R	J	B	addr
0000100000000000	000000000000	0	0	XXXX
	...			
	100010000000	0	0	XXXX
	110001000000	0	1	1000
	...			
	100010000000	0	0	XXXX
	110001000000	0	0	XXXX
	...			
	000000000000	0	1	0000

Multiple Applications Mapping

When multiple applications are mapped into our proposed platform, the interface controller is in charge of writing the information of the applications into the platform. Table 3.7 shows the applications saved in the table in the global structure controller. With the information of the applications in the table, the global structure controller is able to check if the configuration of one application is finished and then sends a finish signal to the global execution controller to synchronize the application. If one application is being reconfigured, then the interface controller sends another request to reconfigure another application, it saves the next reconfigured application into the status register. By writing the information into the memories and the tables, one single application is allowed to be configuring while four applications are allowed to be executing at the same time.

Table 3.7: Table in global structure controller

Application	P1	B1	Start1	End1	P2	B2	Start2	End2	P3	B3	Start3	End3	P4	B4	Start4	End4
1	1	1	AAA	BBB	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	1	1	CCC	DDD	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	1	1	EEE	FFF	1	0	0	0

If one application is under reconfiguration, the global structure controller is in busy status. At this time, if it comes a command of the configuration of another application, then the interface controller detects the busy status of the application and saves the command into its queue. The interface controller issues the command of configuration of the second application until the global structure controller is free of the first application. However, if one application is busy with the configuration of

an application, then it comes the request of the branch from another application. The global structure controller stops the configuration and deals with the reconfiguration right way. Therefore, the command from the execution program has higher priority than the command from the host processor.

3.4.7 Application Reconfiguration

Single Partition Application

The request of reconfiguration comes from the buffer sharing or segmented buffer access. Initially, the interconnection and the block sizes of the reading and writing access in the buffers of the bridges and partitions are configured by the global and partition structural controller, according to the requirement of the applications.

The request of reconfiguration is issued by the execution program written from the host processor and configured by the structural controller as shown in Fig. 3-44. After the initial configuration of the platform, if there is a need of sharing some buffer, the execution program branches in the program both for the partition and for the global execution. Both the affected partition execution controller and the global execution controller branch to a new address "BBB". The global and partition execution controllers stop current execution and save next loading address after the reconfiguration finishes in the registers. The partition execution controller fetches the configuration address "CCC" by the address "BBB" from the address table and sends the address "CCC" to the partition structure controller to request the reconfiguration. However, the global execution controller fetches the address in the same way, but sends

the request and the address to the interface controller. The interface controller checks if the global structure controller is in busy status. If it is free, then the interface controller issues the request and address to the global structure controller. Then the global and partition structure controllers reconfigure the interconnection and the buffer size by the address sent from the execution controller. The global and partition structure controller gets the start and end address of the configuration "DDD" and "EEE" in the tables. Once these structure controllers get the new configuration address, they configure the structure aspect in the partition and the bridge. After the partition structure controller finishes the reconfiguration, it sends the global structure controller a finish signal. The global structure controller sets the current status table and compares with the application table in the interface controller. Once the global structure controller makes sure the reconfiguration of the single partition application is finished, it asks the global execution controller to resume with the application and sets its current status register to be "free". Then the according partition and global execution controllers resume from the loading address saved in the registers.

Compared to the partition execution and structure controller, the global execution controller takes one more cycle to send the reconfiguration request and address to the global structure controller. It will not cause the mismatch of the timing, since the interface controller waits for the finish of the reconfiguration of both the global structure controller and the partition structure controller. The coordination of the interface controller avoids the conflict in the global structure controller when multiple applications want to reconfigure at the same time. The precondition of applying buffer sharing is to make sure the non-overlap buffer activity. Since the platform needs time

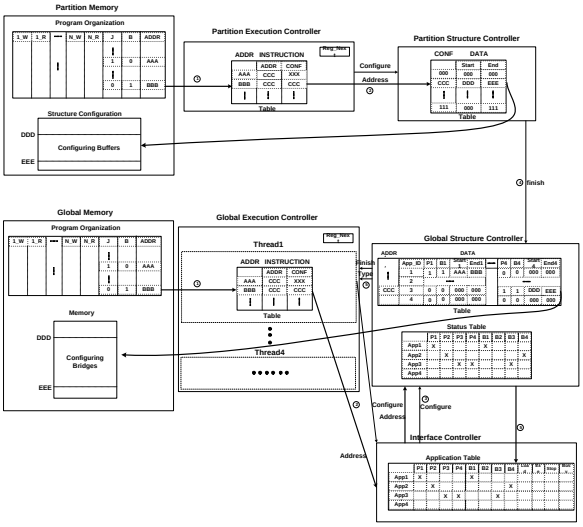


Figure 3-44: Memory mapping in the controllers.

to stop and reconfigure the application, we require more slack time to allow the reconfiguration of the modified buffers by the structure controller.

Multiple Partitions Application

For an application making use of multiple partitions, the reconfiguration is more complicated than the single partition application case. Assume application 3 makes use of partition 3 and 4. After the configuration of application 3, if there is a request of reconfiguration due to buffer sharing or dynamic interconnection configuration, Fig.3-45 shows the memory content in this case. The configuration of type 1 is configured firstly, and then each partition structure controller sends their execution controller a finish signal. After the finish signal from partition 3 and 4 are collected, the execution of type 1 starts. If there is a branch in the execution program requesting the reconfiguration, the start and end address is saved in the table in the global and structure controller to reconfigure the application.

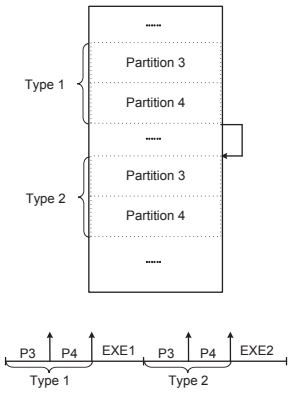


Figure 3-45: Reconfiguration of multiple partitions application.

As shown in Fig.3-46, when there is a reconfiguration request, the execution program in the partition memories of both partition 3 and 4 and the global memory branch. At this time, the global execution controller and the execution controllers of partition 3 and 4 suspend and save its next loading address. The partition execution controllers send their according structure controller a configuration signal and the configured address. But the global execution controllers sends the configuration signal and address to the interface controller and then the interface controller issues them to the global structure controller in the case when the global structure controller is free. Currently the global structure controller is free and the request comes from one application, therefore, the configuration request and address are sent to it. Once partition 3 or partition 4 finishes the reconfiguration, they send the global structure controller a finish signal. The global structure controller sets the status table and compare with the application table in the interface controller. The request and address of the reconfiguration from the global execution controller to the global structure controller goes one clock cycle later than that in the partitions. However, the global structure

controller will not asks the global execution controller to execute until all the partitions and the global structure controller finish the reconfiguration. The partition 3 and 4 configures the interconnection within them, and the global structure controller configures the interconnection of bridge 3. After all of them finish the configuration, the global structure controller sends the global execution controller a resume signal to resume the execution of partition 3, 4 and bridge 3 and clears the status register to zero.

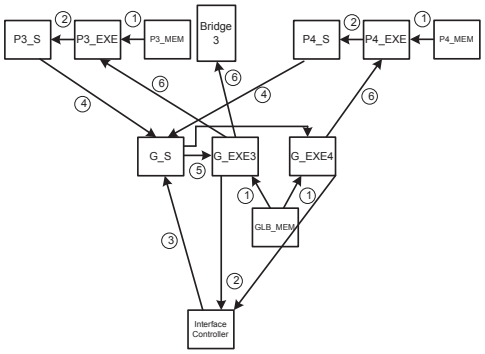


Figure 3-46: Synchronization in multiple partitions application.

Therefore, the synchronization of the operation of the partitions within one application is achieved through the global structure controller. The partition structure controllers do not trigger their partition execution controller directly, the partition execution controllers are controlled solely by the global execution controller. Even though if in the configuration of type 2, only partition 3 needs the reconfiguration, while partition 4 does not need any change, the execution controller of partition 4 still sends the structure controller of partition 4 a reconfiguration request and address and suspends the execution of partition 4. The difference is the structure controller of partition 4 configures nothing, but only sends the finish signal to the global structure

controller right away. Whenever if a partition needs to be reconfigured, the multiple partitions application is synchronized in this way.

3.4.8 Simultaneous Reconfiguration

The coordination of the simultaneous reconfiguration is controlled by the interface controller. The interface controller has two sets of queues to avoid the conflict with the commands from the host processor and the reconfiguration requests from the execution program. These queues buffers the command when the global structure controller is busy with one application, initial configuration or reconfiguration, then comes another request of loading of another application, or there is a request of "Branch" in the program.

As shown in Fig. 3-47, we assume the global structure is free and them the three application requires reconfiguration at the same time. Each partition execution controller sends the request to the partition structure controller. However, the four global execution controllers send the request of branch to the interface controller. The interface controller saves the reconfiguration requests in the first set of queue. Then it issues the requests one by one according to the current status register in the global structure controller. In the case when the global structure controller is busy with configuring one application, it sets its status to "Busy". Each time before the interface controller issues a command of reconfiguration, it checks the status of the global structure controller. If it is busy, the interface controller waits until the "Busy" status is clear.

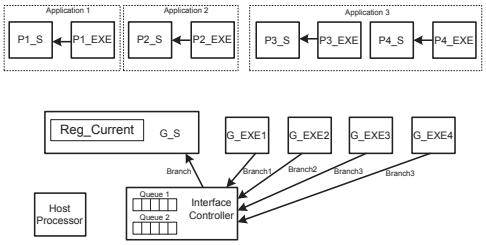


Figure 3-47: Concurrent reconfiguration.

The second queue is used to save the command of load from the host processor when the global structure controller is busy with the initial configuration or reconfiguration. Each time the host processor sends the command of load, it is saved into queue 2. However, queue 2 has lower priority than queue 1. The request from the execution program always has higher level of interrupt level than the command from the host processor. For example, if the global structure controller is busy with the reconfiguration of application 1 and then application 2, at this time the host processor sends a command to load application 3, this command of load is saved into queue 2 until application 1 and 2 finishes their reconfiguration and the global structure controller is set free. If the host processor wants to process the command of loading application 3 first, it has to stop the execution of application 1 and 2 first. The command of "Stop" is issued by the interface controller right away without checking the status of any register.

Fig. 3-48 summarizes the status registers and queues in the global structure controller and the interface controller. The global structure controller has a status table to record how many partitions and bridges have finished the configuration of each application. Once it detects some application finishes its configuration, the

global structure controller asks the according global execution controller to execute. The global structure controller also has a status register recording its current status. It is written by the global structure controller to indicate if it is busy or not, read by the interface controller to decide if it should issue the request of configuration. The interface controller has an application table within it. It is written by the host processor about the composition of each application, read by the global structure controller to check if it finishes the configuration. The interface controller also has two queues to deal with the case of multiple concurrent applications. The global structure controller has to deal with the application one by one. Each time when the interface controller gets a request of "Branch" from the global execution controllers, the interface controller puts these commands into queue 1 by the application table and then issues them one application after another. It is used to deal with the case when multiple applications request reconfiguration concurrently. Each time when the host processor sends the interface controller a command of "Load", it is pushed into queue 2. The command in queue 2 will not pop out until queue 1 is empty and the global structure controller is free. Queue 2 is used to deal with the case when the global structure controller is busy with reconfiguring some applications while the host processor requests to configure another application.

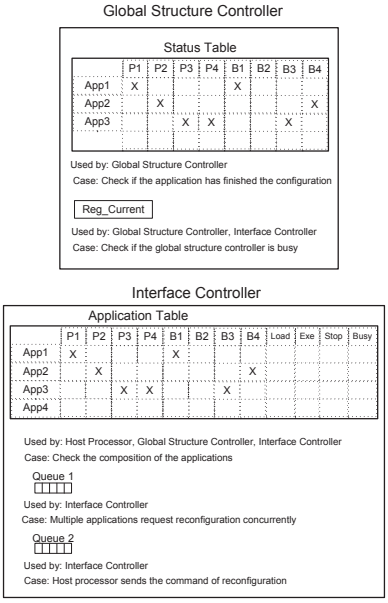


Figure 3-48: Status registers and queues in global structure controller and interface controller.

3.5 Execution Control

3.5.1 Execution Initiation of Single Partition Application

Fig. 3-49 shows how the global controller cooperates with the partition controller dynamically when one application only uses one partition. If the host processor issues the command of "Load", the interface controller checks if the global structure controller is busy with configuring other applications. If not, then the global structure writes the control data into the partition 1 memory and starts the structure controller of partition 1 to configure. After the structure controller of partition 1 finishes the configuration, it sends the global structure controller a finish signal. At this time, if the global structure controller finishes the configuration of bridge 1 of the interconnection to get access to the external data, then the global structure controller sets

the status of application 1 as "Loaded".

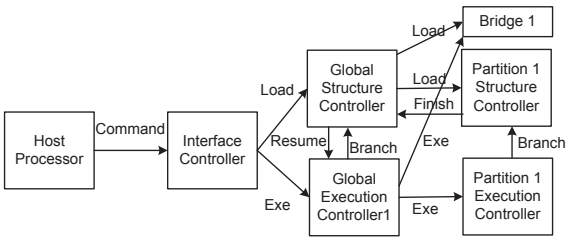


Figure 3-49: Single partition application operation.

If the host processor issues the command of "Execution", the interface controller checks if the application is loaded in the status register. If the application is loaded, then the interface issues the command to the global execution controller 1. The global execution controller 1 triggers partition 1 execution controller and bridge 1 to execute. If there is a "Jump" in the program, then the partition 1 execution controller direct the current execution address to the jumped address. If there is a "Branch" in the program, the partition 1 execution controller suspends current execution and requests the partition 1 structure controller to reconfigures the interconnection. At the same time, the global execution controller 1 also suspends current execution of bridge 1 and requests the global structure controller to reconfigure the bridge 1. Once the partition 1 structure controller finishes the reconfiguration, it sends a finish signal to the global structure controller. The global structure controller triggers the global execution controller 1 and then the global execution controller 1 triggers the partition 1 execution controller to resume with the execution of the program.

The global execution controller generates the start read and write signals for the bridges after being triggered by the global structure controller, and starts or resumes

the according partition execution controller by the signal from the global structure controller. As shown in Fig. 3-50, the global execution controller is made up of four threads, each of them controls the according partition execution controller and the adjacent bridge. Each thread reads the data from the program and then resets the counter to according address, preparing to execute the buffers in the bridge, or sends start configure signal and address to the global structure controller.

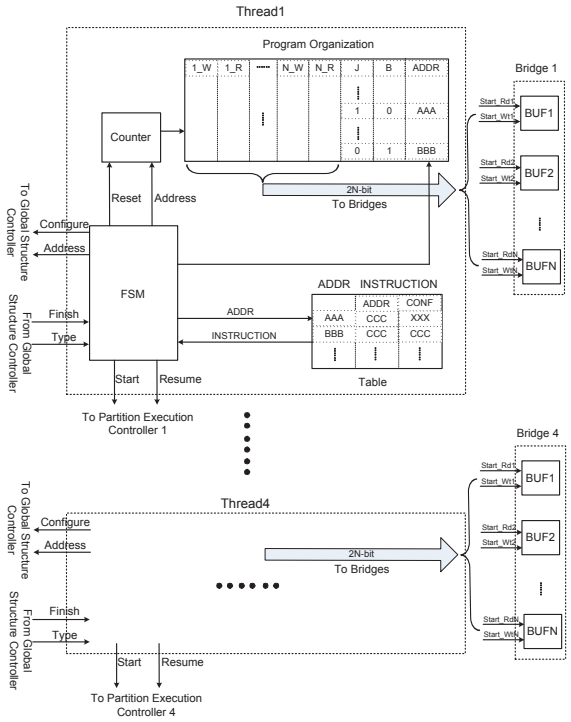


Figure 3-50: Global execution controller.

The behavior of each thread is similar to the partition execution controller, except that the program controls the buffers in the bridge instead of the partition, and it generates a start or resume signal to trigger the partition execution controller.

The division of the control of the partitions into separate thread brings the easy control when multiple applications work at the same time. When there is a transition

from one application to the other, then the according threads control the partitions and the bridges to start or stop without interrupting the other applications.

The partition execution controller generates the start read and write signals for the buffers within the partition once after being enabled by the start or resume signal from the global execution controller. As shown in Fig. 3-51, the Finite State Machine (FSM) within the partition execution controller reads the data from the program and then resets the counter to according address, or sends start configure signal and address to the partition structure controller.

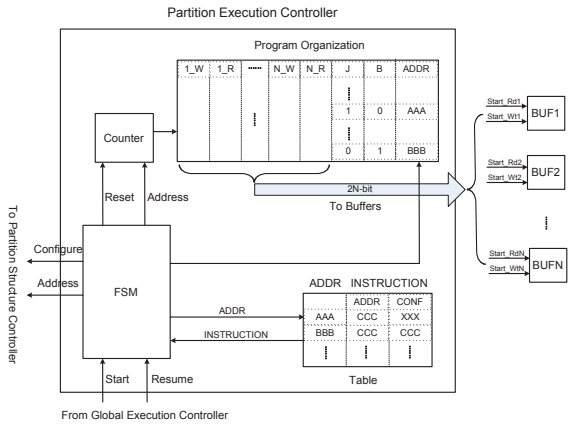


Figure 3-51: Partition execution controller.

The first $2N$ bit of the program organization indicates the start read or write time of the buffers. The "Jump" bit is used to form a loop in the program. However, the "Branch" bit is used in the case when there is a reconfiguration request during the execution. If there is a "1" in the "Jump" bit, the FSM reads the address in the program, "AAA" in this case, and then look for the reset address in the table, "CCC" in this case. In the next clock cycle, the FSM reset the counter to "CCC". If there is a "1" in the "Branch" bit, the FSM stops the counter first and then reads

the address in the program, "BBB" in this case. Then the FSM look for the address "CCC" and the configuration data "CCC" in the table. At this time, the FSM saves the address "CCC" and sends the partition structure controller a configure signal and configuration address "CCC". After the global execution controller sends the resume signal, the FSM resets the counter to the saved address "CCC". If the global execution controller sends a start signal, then the FSM resets the counter address to "0". The program is executed from the first address.

3.5.2 Execution Initiation of Multiple Partitions Application

Fig. 3-52 shows how the global controller cooperates with the partition controller dynamically when one application uses two partitions, partition 3 and 4. If the host processor issues the command of "Load" and the global structure controller is free, the global structure controller configures bridge 3, partition 3 and 4 structure controllers. After partition 3 and 4 finishes the configuration, they send the global structure controller finish signals. Then the global structure controller sets the status of this application as "Loaded".

If the host processor issues the command of "Execution" and the interface detects the application is loaded, then it issues the command to global execution controller 3 and 4. The global execution controller 3 triggers bridge 3 and partition 3 execution controller, the global execution controller 4 triggers partition 4 to execute the program saved in their memory. If there is a "Jump" in the program, then the partition 3 and 4 execution controllers direct the current execution address to the jumped address.

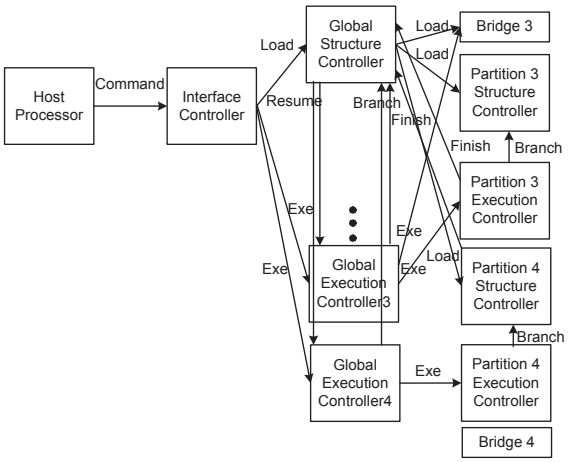


Figure 3-52: Multiple partitions application operation.

If there is a "Branch" in the program, the partition 3 and 4 execution controllers suspend current execution and request the partition 3 and 4 structure controllers to reconfigure the interconnection. At the same time, the global execution controller 3 also suspends current execution of bridge 3 and requests the global structure controller to reconfigure the bridge 1. Once the partition 3 and 4 structure controllers finish the reconfiguration, they send finish signals to the global structure controller. After the global structure checks the reconfiguration of the application is finished, the global structure controller triggers the bridge 3, the global execution controller 3 and 4 to resume to execute. Therefore, the resume of the execution of all the partitions within an application is control under the global structure controller, in which way the synchronization of an multiple partitions application is realized.

How the interface controller coordinates with the global controller and the host processor is shown in Fig. 3-53. For the other command, this architecture will use the same process as shown in Fig. 3-53. If the interface controller accepts the "Halt" command, the partition will save its current parameters in its registers and go on

with the execution when the "Resume" command comes. But during the "Suspend" period, the partition will not save the current parameters. After the command of "resume", the register as shown in step 8 will be written back into execution mode.

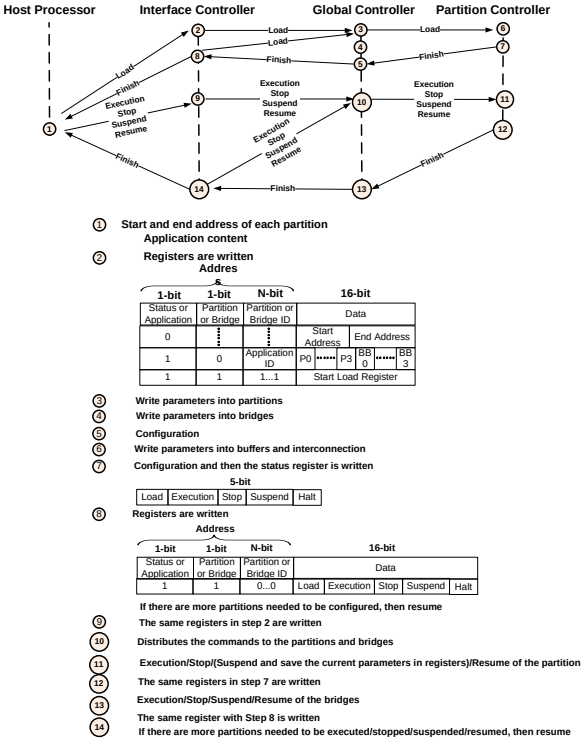


Figure 3-53: Command processing.

Fig. 3-54 shows the timing when multiple applications are configured and executed. During the configuration of application 1, the structural controller writes the data into the registers from the memory. Then the structural controller generates the according control signals to the buffers, either within the partition or for the bridges. At the same time, the status registers are indicated in "Load" status. Once the host processor sends the command to execute the application 1, the execution controller checks if the current time matches to the start time. If it does, then a start signal is generated to trigger the execution of related partitions and bridges. When it comes

the command of "Stop", the execution controller stops the ongoing application and then writes the new status into the register indicating the current "Stop" status.

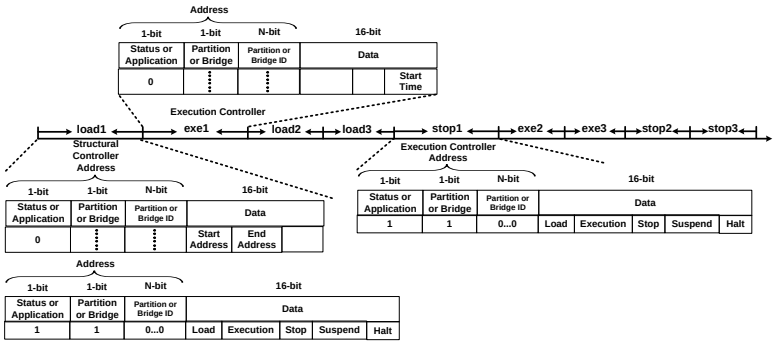


Figure 3-54: Configuration and execution of multiple applications.

Fig. 3-55 shows the timing during the loading of the program. Firstly, the host processor write the program into the global memory by applications, then the host processor sends the command of "Load" to the interface controller. The interface controller checks if the global structure controller is in the status of "busy". If it is busy, the interface controller saves the commands into a queue and then sends out the command after the global structure controller is free. The global structure and execution controllers are loaded, and then the global structure controller requests the according partition controllers to load and sets its status to "Free". Once after the global controllers and the partition controllers finishes the load, the global structure controller marks the loading of the current application finishes. Then it is possible for the command of "Execution" to execute, or the command will be saved into the queue in the interface controller until the finish of the loading of the application. When one application, say application 3, uses multiple partitions, partition 3 and partition 4, the global structure controller will only mark the finish of application 3 after both

partition 3 and 4, and the global controller finish the loading.

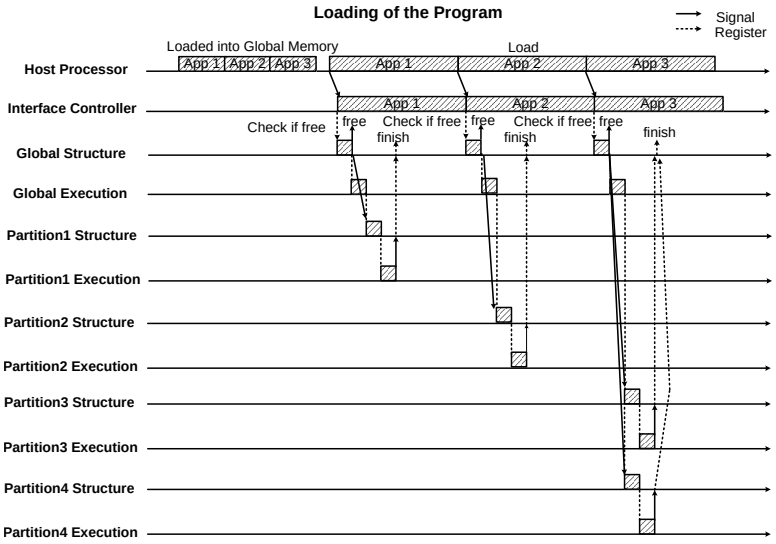


Figure 3-55: Timing of loading.

Fig. 3-56 shows the timing of the execution of the applications. After the interface controller receives the command from the host processor, it checks if the current application has been loaded. It only issues the command of "Execution" after the application is loaded, or it saves the command in a queue until the finish of the loading. For the global execution controller, several applications can be executing since it has multiple threads. The according partition execution controller starts and stops to execute by the request from the global execution controller. Once there is a command of "Stop" from the host processor, the interface controller issues the command without condition immediately.

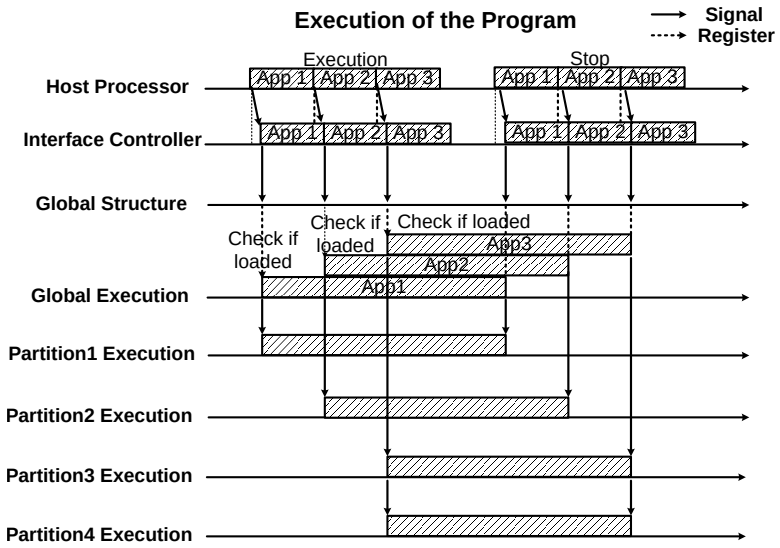


Figure 3-56: Timing of execution.

3.5.3 External Data Access

The processing elements get access to the external through the bridges. Fig. 3-57 shows the structure of the bridges when connecting to the external I/O. Eight buses, either of them can be used as input bus or output bus, are placed next to the four bridges to avoid the conflict when several applications exchange the data with the outside at the same time. The size of the bus is the same with the converted size of data to save the area occupied by the bus. While the size of the data memory is the integer multiple of the size of the bus to make full use of the resources. The bridge is connected with the external I/O through two buses, so that the processing elements in one partition can get access to the platform through one bus without conflict when the adjacent processing element in another partition is communicating with the outside of the platform through another bus. For the input path, a de-multiplexer is used to select the buffers through which the data comes into the partition. The

two fan-in multiplexers are used to select if the input comes from the outside or from the processing elements in the partition. For the output path, M de-multiplexers are used to send the output data to another partition or to the outside through the external I/O. One M fan-in multiplexer is used to choose through which buffer in the bridge, the data is sent to the outside.

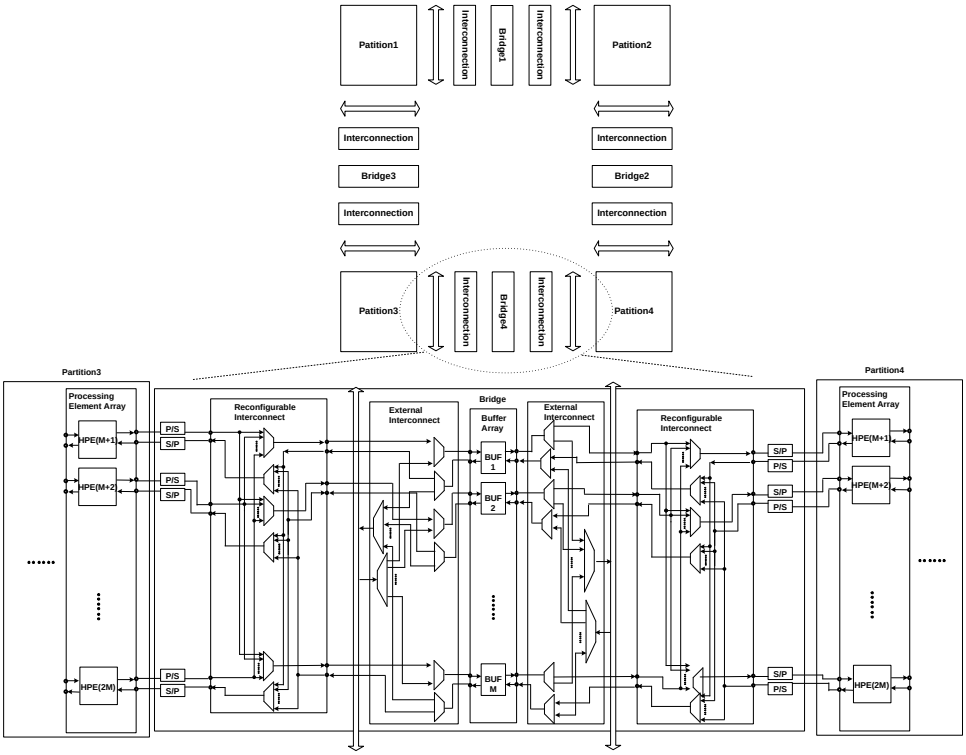


Figure 3-57: Bridges connected to external I/O.

Fig. 3-58 shows the way the external data is accessed. The interconnection modules between the bus and bridge sets if the data in the bridge is connected to the bus. The data controllers decides which bank of the data memory to use and the read or written address of the data memory. Once there is a command of execution from the host processor, the global execution controllers execute the execution program. It generates the start read and write signals which are not only sent to the bridges,

but also sent to the data controller to set the start address of the data memory. If the data controller detects the request of getting access of the external data, The data controller will choose the first available bank. The data controller saves the address of the bank of data memory for the use of reading or writing of the external data. This makes it possible for the concurrent access to all the data buses. Within each bank of the data memory, it consists of two parts, one of them for the read data and the other for the written data. It makes it possible for the share of the data memory cells of reading and writing. Each time when the data is read, it is read from the first part of the block. While the written data is saved in the second part of the block. After the read or write of the data memory bank, the interface controller changes the status registers in the host processor and writes or fetches the data into or from the data memory. If the first two bit is "01", this bank is ready to be read. If it is "10", this bank is ready to be written. If the value of the bank is "1", it is ready for the operation. The registers in interface controller also keeps the record of the interconnection between the bridge and the bus. The first 8-bit register gives the status of the buffers in the bridge. If it is "00", the buffer is used to isolate data within the platform. If it is "01", the buffer is fetching data from the external bus. If it is "10", the buffer is writing data into the external data memory. The data memory is divided into eight parts, each of them is connected to the external bus.

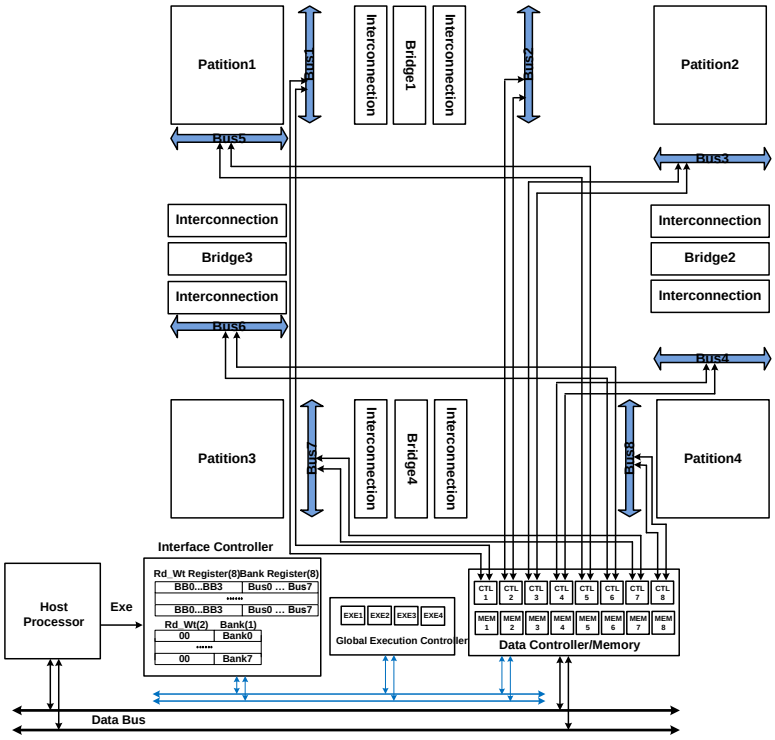


Figure 3-58: Data memory.

3.6 Architecture Evaluation

3.6.1 Single Application in Single Partition

In this section, we evaluate our four partition based platform by mapping the three applications shown in Fig. 1-2 and the function of the hierarchical controllers for the given three applications. The data in the global memory space and data memory is represented by the script files "Glb_Mem_Space" and "Data_Memory". Each time when several applications are mapped into the platform, the host processor writes the control data into the global memory space once. While the host processor can write the data into the data memory dynamically whenever it is necessary for the mapped applications. The platform is modeled by SystemC, and we verify the operation and

the function of the controllers by the log files and the resulted waveforms as shown in Fig. 3-59.

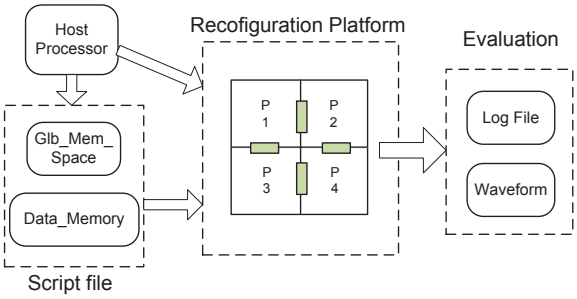


Figure 3-59: Platform implementation.

Three applications as shown in Fig.3-60 are mapped into the four partition based platform shown in Fig.3-61. Two of them make use of only one partition, and the other one uses two partitions. Each application uses the buffers in the bridge and the external buses to get access to the data outside. Each application processes data of different sizes, 16-bit, 24-bit, 32-bit and 64-bit, operated under a single clock frequency or multiple frequencies. The first application makes use of single partition and two external buses. Since two buffers share the same external bus, say bus 1, this application operates as non-overlapped iteration. The second application makes use of a single partition and three external buses, each buffer uses one external bus. Therefore, it is possible to operate as overlapped iteration. It is operated under two clock frequencies. For the third application, it makes use of two partitions and two external buses. It is illustrated as non-overlapped iteration and operated under single clock.

For the first and second single partition based applications as shown in Fig.3-60, we evaluate their functionality. For application 1, it is operated under 1 MHz

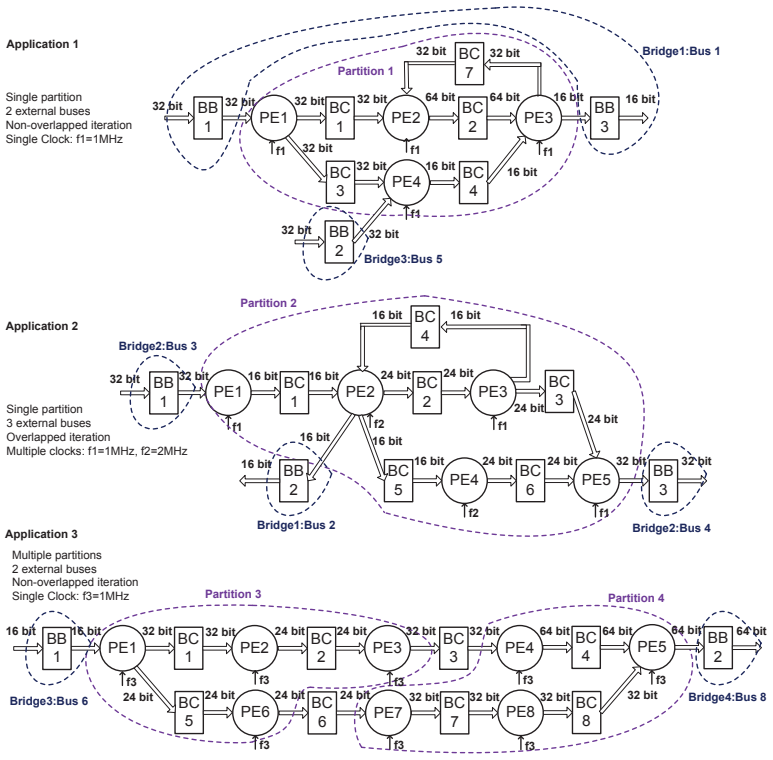


Figure 3-60: Given applications.

frequency, with different input and output data size for the processing elements. The first application is operated under single clock, while the second application is operated under multiple clocks.

The execution program which sets the buffer activity is shown in Table 3.8. Each line of program is executed every clock cycle. The total iteration time is the number of cycles necessary for the processing elements to finish the process, plus the read and write delays in the longest direct path. After one iteration, the program branches to reconfiguration and then execute again.

If we want to normalize the buffer size to 8-bit. To maintain the same throughput, the clock frequencies of the processing elements and the buffers are summarized in Table 3.9. Since the processing elements are all operated under 1 MHz, the read

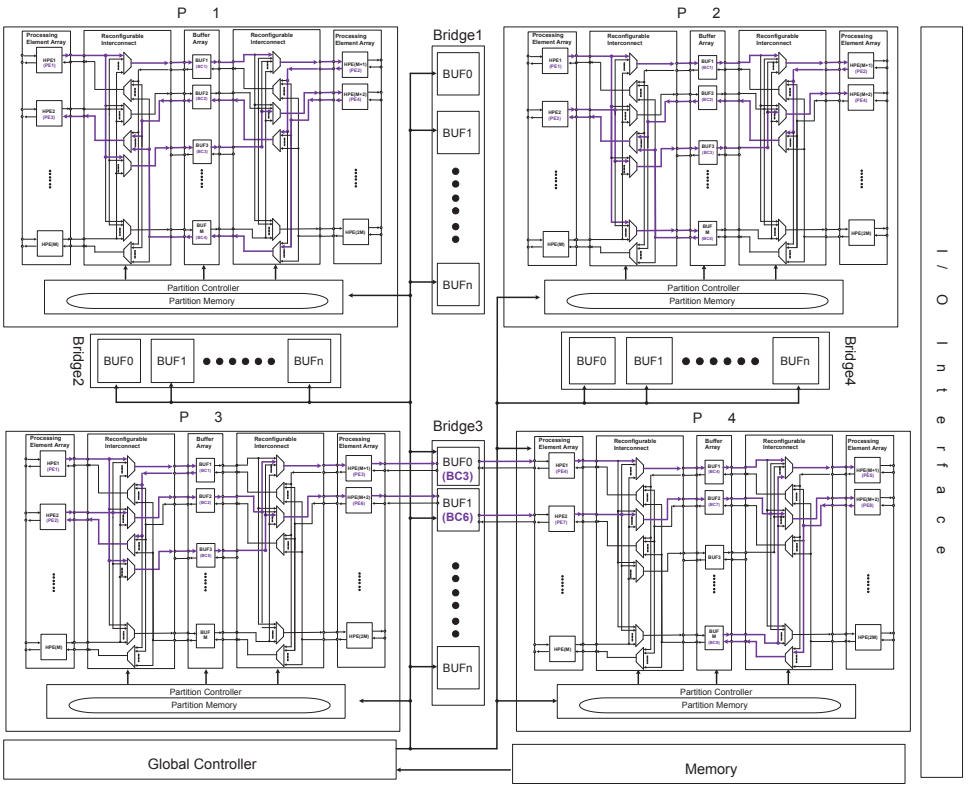


Figure 3-61: Four partition based reconfigurable platform.

and write speed of each buffer are the same. Table 3.10 shows the content programming the clock frequencies, including the clock frequencies of the processing elements, buffers, data conversion modules, the structure and execution controllers.

The new timing of the buffers are changed as shown in Table 3.11 under the assumption that each data conversion module consumes 1 us to finish the conversion. In this case, the total iteration time is 161 us. Compared to the original case, the total consumed time increases 7 us, which is the expense of the data conversion modules in the longest direct path.

The data configuring the interconnection is shown in Table 3.12. Address 0 configures the selection signals of the 4-input multiplexers from the left side processing

Table 3.8: Execution program of application 1

Partition_1_Exe	0000000000000000	0	0	0000
	0000000000000000	0	0	0000
	...	...	...	...
	1000100000000000	0	0	0000
	0100010000000000	0	0	0000
	...	...	...	...
	0010001000000000	0	0	0000
	0001000100000000	0	0	0000
	...	...	...	...
	0000000010000000	0	0	0000
	0000000001000000	0	0	0000
	0000000000000000	0	1	0000
Global_Exe	10000000 00000000 00000000 00000000	0	0	0000
	01000000 00000000 00000000 00000000	0	0	0000
	...	...	...	...
	00000000 00000000 00000010 00000000	0	0	0000
	00000000 00000000 00000001 00000000	0	0	0000
	...	...	...	...
	00000000 00000000 00000000 00000000	0	0	0000
	00000000 00000000 00000000 00000000	0	0	0000
	...	...	...	...
	00000000 00000000 00100000 00000000	0	0	0000
	00000000 00000000 00010000 00000000	0	0	0000
	00000000 00000000 00000000 00000000	0	1	0000

Table 3.9: Clock frequencies in data conversion case

Data_conversion	frequency(MHz)
$PE1$	1
$PE2$	1
$PE3$	1
$PE4$	1
$BC1_r$	4
$BC1_w$	4
$BC2_r$	8
$BC2_w$	8
$BC3_r$	4
$BC3_w$	4
$BC4_r$	2
$BC4_w$	2
$BC7_r$	4
$BC7_w$	4
$BB1_r$	4
$BB1_w$	4
$BB2_r$	4
$BB2_w$	4
$BB3_r$	2
$BB3_w$	2

element fabric to the buffers, while address 1 configures the selection signals of the 8-input multiplexers from the buffers to the left side processing element fabric. In the same way, address 2 and 3 configures the interconnection between the buffers and the right side processing element fabric. For bridge 1 and 3, address 0 configures the connection from the left side processing element fabric to the buffers in the bridge. Address 1 configures the connection from the buffers in the bridge to the left side processing element fabric. The first two 4-bit data in address 2 represent the selection

Table 3.10: Clock frequencies in data conversion case

Partition_1	000	0000	0	8MHz
	001	0000	0	8MHz
	010	0000	0	1MHz
	...	...	...	...
	010	0111	0	1MHz
	011	0000	0	4MHz
	011	0000	1	4MHz
	011	0001	0	8MHz
	011	0001	1	8MHz
	011	0010	0	4MHz
	011	0010	1	4MHz
	011	0011	0	2MHz
	011	0011	1	2MHz
	011	0100	0	4MHz
	011	0100	1	4MHz
	...	...	...	...
	011	0111	0	1MHz
	100	0000	0	4MHz
	100	0001	0	8MHz
	100	0010	0	4MHz
	100	0011	0	2MHz
	100	0100	0	4MHz
	...	...	...	...
	100	0111	0	1MHz
Bridge_1	000	0000	0	8MHz
	001	0000	0	8MHz
	010	0000	0	4MHz
	010	0000	1	4MHz
	010	0001	0	2MHz
	010	0001	1	2MHz
	...	...	...	...
	010	0011	0	1MHz
	010	0011	1	1MHz
	011	0000	0	4MHz
	011	0000	0	2MHz
	...	...	...	...
	011	0011	0	1MHz
Bridge_3	000	0000	0	4MHz
	001	0000	0	4MHz
	010	0000	0	1MHz
	010	0000	1	1MHz
	...	...	...	...
	010	0011	0	4MHz
	010	0011	1	4MHz
	011	0000	0	1MHz
	...	...	...	...
	011	0011	0	4MHz

signals from the buffers in the bridge to the external IO bus. The data in address 3 configures if the data goes into the processing element fabric in partition or the external IO. Address 4 to 7 configures in the same way between the buffers in the bridge and the right side processing element fabric in the partition.

3.6.2 Single Application in Multiple Partitions

In this section, we verify the functionality of the multiple partitions communication through the bridges. We pick up the third application, the dataflow of which is shown in Fig. 3-62. These processing elements and buffers are supposed to operate at 100

Table 3.11: Timing of buffers and bridges in data conversion case

	start_time(ns)	start_write(ns)	start_read(ns)
$BC1$	54	54	55
$BC2$	107	107	108
$BC3$	54	54	55
$BC4$	107	107	108
$BC7$	160	161	161
$BB1$	0	1	2
$BB2$	54	54	55
$BB3$	160	161	161

Table 3.12: Interconnection configuration in the four partition based platform

$Partition\_1$	Address0	1000	0000	1000	0000
	Address1	00000000	01010000	00000000	00000000
	Address2	10000000	00100000	00000000	00000000
	Address3	0000	1000	0000	0100
$Bridge\_1$	Address0	0000	0100	0000	0000
	Address1	1000	0000	0000	0000
	Address2	0100	1000	0000	0000
	Address3	0110	1001	0000	0000
	Address4	0000	0000	0000	0000
	Address5	0000	0000	0000	0000
	Address6	0000	0000	0000	0000
	Address7	0000	0000	0000	0000
$Bridge\_3$	Address0	0000	0000	0000	0000
	Address1	0000	0001	0000	0000
	Address2	0000	0001	0000	0000
	Address3	0000	0000	0000	0101
	Address4	0000	0000	0000	0000
	Address5	0000	0000	0000	0000
	Address6	0000	0000	0000	0000
	Address7	0000	0000	0000	0000

MHz, with different input and output data size as shown in the figure. The data configuring the interconnection is shown in Table 3.13. The parameters configuring the timing of the buffers are shown in Table 3.14. Based on the parameters, the start read and write time of the buffers are shown in Table 3.15 under the assumption that all the processing elements cost 50 cycles to process the data. The total iteration time is 2.29us for this application. After application 3 is divided into partition 3 and partition 4, the timing parameters keep the same, except that buffer 3 and buffer 6 are substituted by the bridges.

After the data conversion modules are added to make full use of the resources and reduce the interconnection cost, the clock frequencies are changed to different values accordingly. But in the implementation of the data flow, it costs more to generate

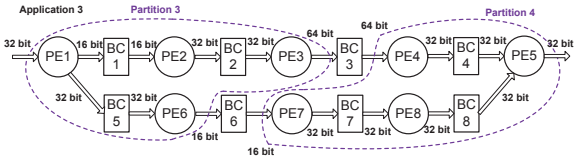


Figure 3-62: Multiple partitions in one application.

Table 3.13: Interconnection configuration of multiple partition

Partition_1	Address0	1000	0100	1000	0000
	Address1	0000	1000	0000	0000
	Address2	0100	0010	0000	0000
	Address3	0000	0000	0000	0000
	Address4	1000	0100	0000	0000
	Address5	0000	0000	0000	0000
	Address6	1001	0100	0000	0000
	Address7	0000	0000	0000	0100

the accurate different clocks. Therefore, we want the clock frequency of the buffers the same to relax the requirement of the accuracy. So all the clock frequencies of the buffers are increased to their least common multiple and the parameters are also changed accordingly. Table 3.16 shows the frequencies of the clocks.

To achieve the same buffer speed, we should change the speed of the processing element together with the parameters to make sure the correct transition of data. For one buffer, if the speed of writing is bigger than the speed of reading, then there is no problem. But in the case when the speed of writing is less than the speed of reading, the read offset (nr) should be recalculated. An example of the case is buffer 3. The processing element 1 writes data into buffer 3 at 200 MHz, while processing element 4 reads the data at 400 MHz. To make sure the right data is read from the buffer 3, the read offset(nr) is increased at least to 82.5ps.

After the clock frequency is increased, clock jitter becomes significant as shown in Fig. 3-63. Without the clock jitter, the effective data can be sampled as the case

Table 3.14: Parameters in given application for multiple partitions

	BC or BB	L	nw	nr	D	M
Given_application	BC1	10	1	1	0	32
	BC2	15	1	1	0	32
	BC3	15	1	1	0	32
	BC4	10	1	1	0	32
	BC5	15	1	1	0	32
	BC6	10	1	1	0	32
	BC7	15	1	1	0	32
	BC8	10	1	1	0	32

Table 3.15: Timing of buffers and bridges for multiple partitions

	start_time(ns)	start_write(ns)	start_read(ns)
$BC1$	500	610	620
$BC2$	1120	1280	1290
$BC3$	1790	1950	1960
$BC4$	2460	2570	2580
$BC5$	500	660	670
$BC6$	1170	1280	1290
$BC7$	1790	1950	1960
$BC8$	2460	2570	2580

with lower clock frequency. But if the fast speed clock comes earlier, the useless data will be saved into the registers in the buffer. To avoid the possibility to get the first useless data, we increase the read offset by one to make sure the first data is effective.

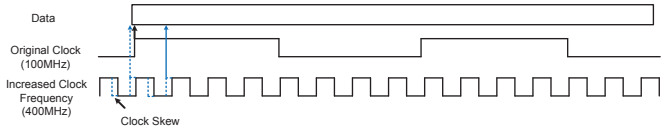


Figure 3-63: Different read and write speed.

3.6.3 Multiple Applications

Load of Applications

The following steps in the log file show how the applications are loaded into the four partition based platform:

- The host processor writes the program related with application 1 into the global

Table 3.16: Clock frequencies in normalized buffer frequency case

$Data\_conversion$	frequency(MHz)
$PE1$	200
$PE2$	100
$PE3$	100
$PE4$	400
$BC1$	800
$BC2$	800
$BC3$	800
$BC4$	800

memory, from 0x0000 to 0x3FFF. The control information of partition 1 and bridge 1 is written into the global memory.

- The host processor writes the program related with application 2 into the global memory, from 0x4000 to 0x7FFF. The control information of partition 2 and bridge 2 is written into the global memory.

- The host processor writes the program related with application 3 into the global memory, from 0x8000 to 0xFFFF. The control information of partition 3 and 4, bridge 1 is written into the global memory.

- The host processor sends command to load application 1.

- The interface controller receives the command of load and checks if the global structure controller is busy. If it is busy, the interface controller saves the command in a queue until the global structure controller is set free.

- Once the interface controller issues the command of load, the memory and registers in the global structure and execution controller are written with the program and tables of application 1.

- The program and tables of partition 1 are written into the memory and registers

within partition 1.

- The status registers in application 1 is set, indicating the finish of loading application 1.

- The same loading process repeats for application 2 and 3.

Fig.3-64 shows the verification by the waveform. The host processor gives the command of writing the data into the global memory and loading them into the platform. The global structure controller and each partition structure controller starts to load the data into their memory and registers after the interface controller issues the command. Once after each application finishes its loading, the status register is set by the global structure controller.

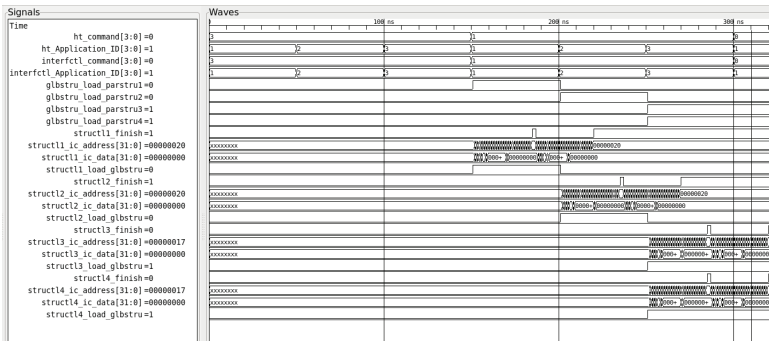


Figure 3-64: Load of applications.

Execution of Applications

The following steps in the log file show the execution of the applications after they are loaded into the memories and the registers:

- The host processor sends command to execute application 1.

- The interface controller receives the command of execution and checks if the application has been loaded. If it is not loaded, the interface controller waits. If the application is loaded, then the interface controller issues the command.
- The status registers in application 1 is set, indicating application 1 is executing.
- Partition 1 is executing.
- Program branches in partition 1. The global structure reconfigures the interconnection. At the same time, the execution controller of partition 1 suspends the execution and requests the structure controller of partition 1 to reconfigure.
- The global structure controller finishes the reconfiguration and then sets its status register that it is free.
- The structure controller of partition 1 finishes the reconfiguration and then sends the global structure controller the signal indicating it has finished the reconfiguration.
- The global structure controller checks if the reconfiguration of application 1 is finished. If yes, it request the global execution controller to resume the execution of application 1 again.
- The global execution controller triggers the execution controller of partition 1 to resume the execution.
- Program jumps to the first address in partition 1. The execution controller of partition 1 controls the current executed program to the jumped address.

- The same execution process repeat for application 2 and 3. The execution of application 2 and 3 overlap with that of application 1, except that the global structure controller has to reconfigure the interconnection of one application at a time. If it is busy reconfiguring one application, the other application has to wait until it is free.

Fig.3-65 shows the execution of the applications. Once the host processor sends the command of executing some application, the global execution controller triggers the according execution controller of the partition. It also gives the start read and write signals of the buffers in the bridge if an application makes use of several partitions. If the current executed program has a branch, the structure controller gets the reconfiguration address and data and then reconfigures the interconnection. Once the host processor sends command to stop the execution of the application, the global execution controller stops according partition execution controller.

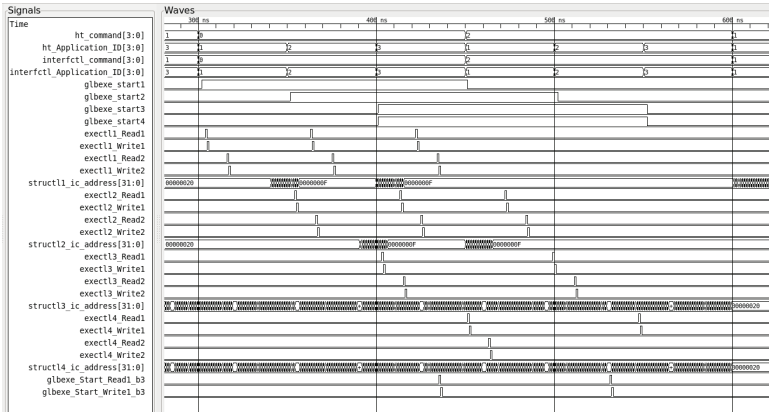


Figure 3-65: Execution of applications.

The reason of dividing the control of the partitions into threads is to make it possible for the multiple applications executing at the same time. Table 3.17 shows

the timing of the most difficult case when all the three applications start running at the same time, including the enable time of each thread and the start read and write time of its following bridge buffers. In the case when there is a transition from application 1 to application 2, the enable signal of the application 2 is delayed under the control of the execution controller. Table 3.18 shows the timing in the case when application 1 and application 3 start at the same time, then application 2 starts after application 1 finishes. Therefore, the host processor controls the execution time of the applications simply by giving the desired parameters to the program memory.

Table 3.17: Timing of the global execution controller in Case 1

	Enable(ns)	start_write(ns)	start_read(ns)
$Thread1$	0	*	*
$Thread2$	0	*	*
$Thread3$	0	1280,1950	1290,1960
$Thread4$	1790	*	*

Table 3.18: Timing of the global execution controller in Case 2

	Enable(ns)	start_write(ns)	start_read(ns)
$Thread1$	0	*	*
$Thread2$	2000	*	*
$Thread3$	0	1280,1950	1290,1960
$Thread4$	1790	*	*

Fig. show the data and control of this platform after the three applications are mapped into our proposed platform. Once there is an command indicating the configuration of the platform, the three applications are loaded. Then they start to execute at the same time, and the third application which consumes most time ends as the last one.

External Data Access

The following steps in the log file show the execution when the platform needs to get access to the data outside:

- The host processor sends command to execute application 1.
- The interface controller receives the command of execution and checks if the application has been loaded. If it is not loaded, the interface controller waits. If the application is loaded, then the interface controller issues the command.
- The status registers in application 1 is set, indicating application 1 is executing.
- The host processor writes data into the first half of block 1 for the application 1 to read and then sets the status of the block "Readable".
- Partition 1 is executing. It reads the external data from the data memory and then executes.
- The host processor sets the status of block 1 "Writable".
- Application 1 checks if block 1 has been set as "Writable". If it is ready, then application 1 writes the data into the second half of block 1.
- The same execution process of application 2 and 3 overlap with application 1, since they do not share the bus.

Fig. shows the execution when the applications get access to the external data. Once the host processor sends the command of executing some application, the host

processor writes the data into the data memory first. Then after the application finishes processing these data, it writes the data back into the data memory if the block is ready to be written.

Conflict Handling

Fig.3-66 shows the case when there is a conflict of the commands of configuration. When application 1 is still configuring, it comes the command of configuring application 2, the interface controller will not issue the command until application 1 finishes its configuration.

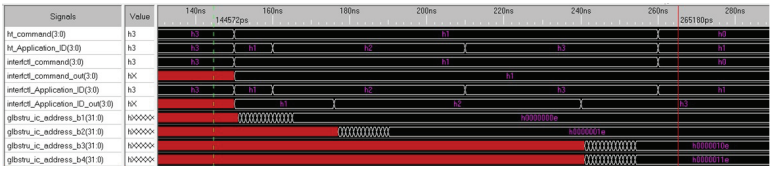


Figure 3-66: Conflict of the configuration commands.

Fig.3-67 shows the case when there is a conflict between the reconfiguration request and the command of configuration. When application 3 is doing the reconfiguration, then it comes the command of configuring application 3 from the host processor, the interface controller will push the command from host processor into the queue and then issues them after the reconfiguration of application 3.

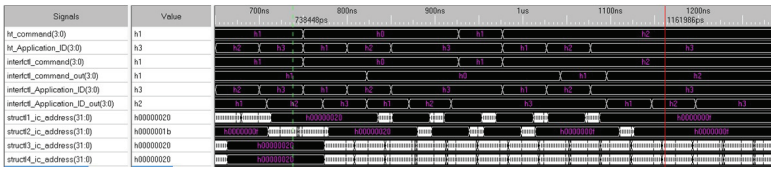


Figure 3-67: Conflict between the reconfiguration request and command of configuration.

Discussion

The simulation result shows the multiple applications work successfully in three scenarios.

Scenario 1: load application 1. Load and configure the interconnection and timing parameters of application 1. Then load the execution parameters from the global memory and execute application 1.

Scenario 2: Load application 2. After the configuration and execution of application 1, there is no longer any parameter about it from the global memory. So application 1 keeps running without being interrupted due to the running of application 2. Then it is the turn of application 2. Load and configure the interconnection and timing parameters of application 2. Then load the execution parameters from the global memory and execute application 3.

Scenario 3: Before the configuration and execution of application 3, the parameters given from the global memory indicates all the timing parameters in application 2 are "0". Then application 2 is stopped, following the configuration an execution of application 3: Load and configure the interconnection and timing parameters of application 3. Then load the execution parameters from the global memory and execute application 3.

The first configuration time of above description consumes 60 clock cycles. With the number of partitions and processing elements varies, the consumed time will change. Here we consider the reconfiguration time of several cases.

Case 1: Interconnection within partition 1 is changed.

Case 2: The timing parameters within partition 1 is changed.

Case 3: The interconnection of bridge 1 is changed, while all the other partitions and the bridge timing parameters keep the same.

Case 4: The timing parameters of bridge 1 is changed.

Case 5: The interconnections within partition 1 and 2 and their timing parameters are changed. The timing parameters in bridge 1 and 2 are changed.

Table 3.19 gives the reconfiguration time and steps as to the above cases. From the result, we get that Case 1 to Case 4 get 83.3% to 56.7% reconfiguration time reduction, while Case 5 only get 10% reduction. Therefore, in the case that there are only a few modifications of the platform, the hierarchical controller saves much reconfiguration time.

Table 3.19: Reconfiguration time and steps

	Case 1	Case 2	Case 3	Case 4	Case5
$First\_time\_configuration\_time$	60	60	60	60	60
$Reconfiguration\_time$	10	26	17	11	54
$Execution\_steps$	11	1,10	5,10	6,10	1,2,6,7,10,11

There is tradeoff when dividing the partitions: Big partition size causes complicated interconnections and wasted resources, while small partition division increases the number of total partitions in a fixed application, therefore, makes the control structure complicated.

In the case that the size of one application is too big to be realized by one partition, this application is mapped to several partitions. The maximum allowed size of the partition is decided by the width of the registers within the partition. There are two constraint when dividing the partitions: First, the total number of buffers, the same

to the number of the processing elements, is less than 2^N, where N is the width of defining the buffer ID. Secondly, the number of the buffers in the bridges is less than 2^N, where N also defines the width of the address offset. The size of the data in the memory decides the maximum allowed iteration time. In our proposed platform, each parameter(L, nw, nr, D or M) is 8-bit. Therefore, the maximum allowed iteration time is $2^8 + 1$. If an application of larger iteration time is mapped to our proposed platform, the data size should be increased.

The total iteration time of one application might be changed to satisfy the requirement of multiple applications. For application 1, the input and output of the external data happens through bus 1. When the output data is ready, but the bus is still occupied for the reading of the input data, then the data is hold to wait until the bus is set free. Therefore, the total iteration time is extended by the waiting time 1.

For the case of application 2 and 3, once the data is ready to be write out of the platform, the host processor waits for the whole block of data to come out and then write them into the data memory. If the host processor is busy with the reading and writing of one application, another application is ready to read or write the data to external, then this application has to wait until the single data bus of the host processor is free. In this case, the iteration time of this application is also extended.

In conclusion, the iteration time of one application might be extended because of two reasons. One is that only one bus is assigned to the application, so it has to wait for the finish of the reading of the data from the external and then starts to write data outside. This happens when the time of reading a block of data is bigger than the iteration time. The other case is the reading or writing of the external data

conflicts with those of another application after the mapping. Then the application has to wait for the free time of the single bus to the host processor. In this case, the iteration time of an application is extended.

After some applications are mapped to the platform, it is capable to add more applications. But the existing applications have to be reconfigured to satisfy the following constraint resulted from the platform:

- **The application uses adjacent partitions:** This platform supports mapping of the applications into adjacent partitions. Since they can only communicate through the buffers in the bridge.

- **Enough buffers in the bridge:** The buffers in the bridge can be used both as the buffers within an application and as the buffers of the data outside.

- **Enough bus to get access to the data outside:** The bus can be used both as either input or output bus at a time. There are two bus next to one partition, so one partition can get access to the data outside through two buffers concurrently. The execution time of an application has to be extended to satisfy this constraint.

- **Single bus for the host processor:** There is only one single bus for the host processor to write the data into the blocks in the data memory. Before these data is used or sent by the applications, the host processor should have time to finish the writing or reading of the data.

- **Action of the host processor and the applications concurrently:** Each

block of data memory corresponds to one of the eight buses. However, they are divided into two parts so that the single bus connected to the host processor and the eight buses next to the bridges can be used at the same time.

3.7 Conclusion

A partition based reconfigurable platform for multiple applications executing concurrently is presented. Both functional and architectural aspects of the system are modeled. This architecture simultaneously considers system performance and architectural models for the evaluation. Two models are closely linked by a set of common parameters which affect both the performance and the hardware complexity. Reconfigurable controller for supporting multiple applications in partition based platform is presented. Our proposed hierarchical control layers supporting multiple size data are efficient for rapid reconfiguration. The reduced control bus architecture is useful to reduce the interconnect resources. This control structure is only suitable for a single application within one partition. In the case that multiple applications use the same partition, the control signals should change smoothly between applications, without affecting the function of previous applications.

Chapter 4

Hierarchical Controller Design for Rapid Manipulation of Partition Based Reconfigurable Platform

4.1 Introduction

Modern application-specific systems often demand easy reconfigurability, which allows the system to adapt to the nature of the computation being carried out [1], [2], [4], [10]. Current reconfigurable computing systems enable us to map complex tasks to different cores, minimizing hardware resources and simultaneously increasing the processing speed [3], [7] - [9].

Mapping program flows onto a multi-core architecture presents a challenge to designers and this is the reason why there exists a large design gap in reconfigurable architecture logic [31] - [32]. Most systems consists many processing elements and

mapping these to multiple processors requires designing efficient controllers [44].

Mapping a design dataflow onto such a multi-core architecture can prove to be an ardous task, since synchronizing data transfers between multiple processing blocks is a problem owing to different latencies and execution times. While it is possible to estimate the execution times and map the dataflow on the basis of such approximations, wrong results may be produced if the actual execution times are even slightly different. Many sophisticated modeling techniques exist that enable us to map such designs onto hardware [45] - [50]. However, these techniques often seem inadequate when it comes to complex designs: non-trivial design issues such as flexibility of mapping methodology, complexity of controlling architecture and dynamic reconfiguration remain unsolved. Moreover, such techniques focus primarily on functional modeling, and place less emphasis on the controller design, which often scales exponentially in complexity with a linear increase in the complexity of the design.

To deal with these issues, we propose a design methodology based on buffer-based dataflow (BBDF) where the data access is globally synchronized by a controller that handles the data transfers between multiple processing elements. BBDF is a transparent representation that bridges the gap between the algorithmic description of a design and its structural implementation with a buffer-centric perspective, as opposed to the conventional processing element-centric perspective. The proposed methodology is built around the principle of representing the algorithm as a BBDF and by using a powerful controller that can handle all the buffer parameters and interconnections, thereby making reconfiguration a straightforward process. The controller design of key importance here: because of the nature of logic architectures, the designer does

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

Mapping program flows onto a multi-core architecture presents a challenge to designers and this is the reason why there exists a large design gap in reconfigurable architecture logic. Most systems consists many processing elements and mapping these to multiple processors requires designing efficient controllers. In order to synchronize the data transfers at the level of a dataflow graph, we use the buffer-based dataflow for mapping processing elements to processors. Our methodology creates a mapped partition from the buffer-based dataflow representing an application, the resource constraint of a target realization and estimated times for functional executions and data transfers. Our mapping algorithm tries to map consecutive processing blocks to the same processor to increase efficiency. In the proposed methodology, the data

transfers of processing blocks mapped to processors are realized as target-dependent primitive templates.

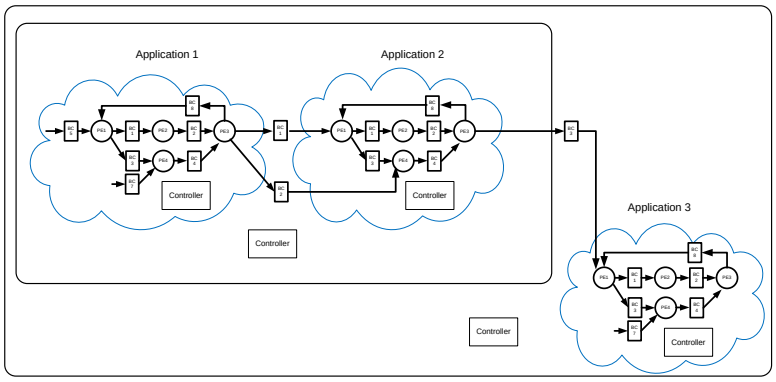


Figure 4-1: Illustration of multi-level execution controller structure.

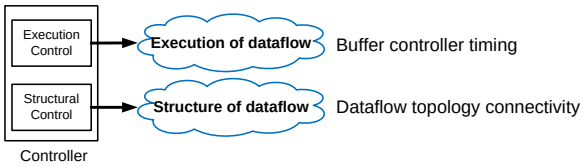


Figure 4-2: Orthogonal controller for execution flow control and interconnect topology configuration.

Apart from simple dynamic reconfiguration, our design efficiently handles adjustment of the iteration period of the algorithm, and solves multi-rate dependencies as well. The iteration period has to be often adjusted according to the needs of the designer. Our methodology takes care of this requirement by increasing or decreasing processing speed of the elements depending on the required iteration period. Multi-rate dependency, which is due to the processing elements operating at different clock frequencies with respect to one another, is also solved in the same way. The use of global and buffer controllers in our design facilitates the separation of the structural and temporal aspects of the dataflow and simplifies program size as well as scalability.

4.2 Mapping and Characterization

4.2.1 Buffer Based Operations

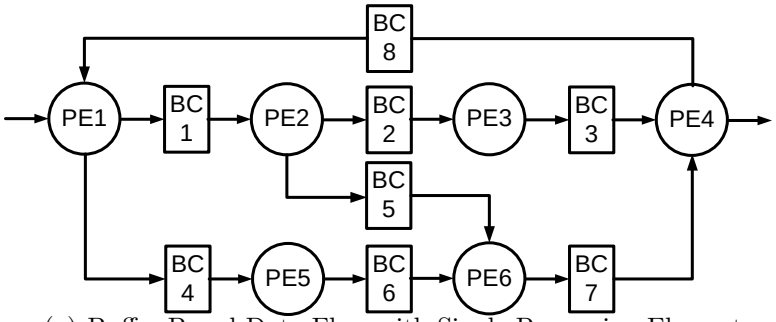
Application-specific systems are usually complex designs with multiple processing elements and a number of datapaths and sequential elements. In general, all algorithms and designs can be represented as dataflows governed by the relation

$$Y = P(X) \tag{4.1}$$

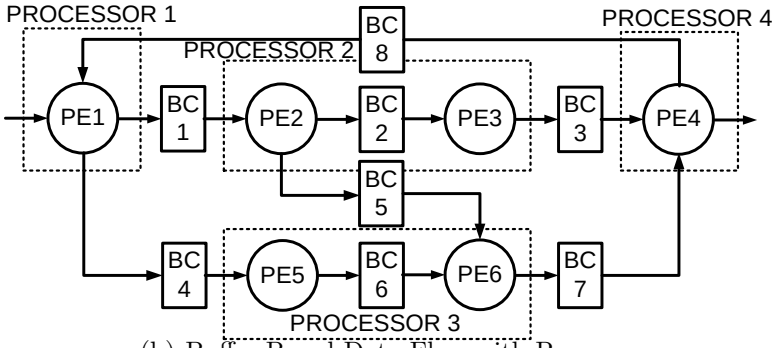
where an input X and an output Y are finite blocks of data, and P is the representation of the processing element (PE). X and Y are sequentially consumed and produced as data blocks, and their sizes may be different.

Fig. 4-3(a) shows the buffer-based dataflow, obtained when the processing elements are separated by buffers to isolate their functionalities. Each processing elements includes both the functionality as well as the storage elements required for proper functioning. Processing elements in the dataflow execute the required program or algorithm on a finite set of data in every iteration period. Typical dataflows have multiple inputs, outputs, and even feedback elements. Inclusion of the buffer controller enhances the reconfigurability of the entire system. The buffers can be realized as dual-port memory, since they allow access to reading and writing to all processing elements connected to them. However, reading and writing operations must be performed keeping in mind the execution times and latencies of each processing element. The various processing elements can be divided into processors to be mapped onto

the target architecture.



(a) Buffer Based Data Flow with Single Processing Element



(b) Buffer Based Data Flow with Processor

Figure 4-3: Two different realization of buffer based data flow. (a) A function is mapped to an processing element. (b) Multiple functions are mapped to a processor.

Fig. 4-4 is the design realization where several processing elements are mapped to one processor and multiple processors are used, based on the dataflow obtained in Fig. 4-3(a). The division of processing elements into multiple processors or cores is indicated in the figure. This division is based on simple optimization techniques. If processing elements in the same sequential data path are mapped to the same processor, the processor can process the data sequentially from one processing element to the other without unnecessarily wasting any time to wait for the previous processing element to finish. The global controller synchronizes the data transfers between multiple processors. The mapped partition has the global timing information to make

Table 4.1: Buffer Controller Parameters in Processing Element Realization

	L	nw	nr	D	M	start_write	start_read
$BC1$	10	1	1	0	32	11	12
$BC2$	10	10	1	0	32	32	33
$BC3$	15	12	1	0	32	60	61
$BC4$	15	5	1	0	32	20	21
$BC5$	30	6	1	0	32	48	49
$BC6$	20	7	1	0	32	48	49
$BC7$	20	7	1	0	32	76	77
$BC8$	10	1	1	0	32	72	73

sure the correct frames are being read or written between different processor cores (or between a processor and some hardware logic).

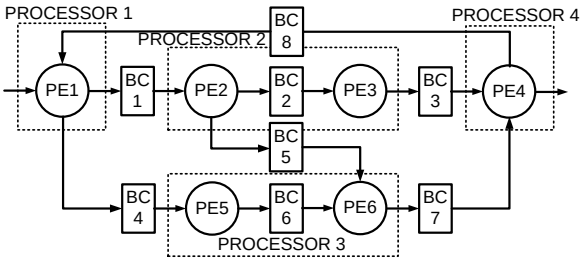


Figure 4-4: Buffer-based dataflow mapped to a multi-core system.

The buffer-based dataflow obtained in Fig. 4-3(a) has to be mapped to the target realization. Fig. 4-5 shows this mapping where the processing elements are placed on the two sides of buffers, connected by the reconfigurable interconnects. In the figure, we only show the result of connected elements.

Using the previously derived relationships between the start signals, we now try to approximate the read and write times for each buffer and these are illustrated in Table 4.1. These times are based on the assumption that the execution time of each processing element is 50 cycles. The buffer controller requires this information to

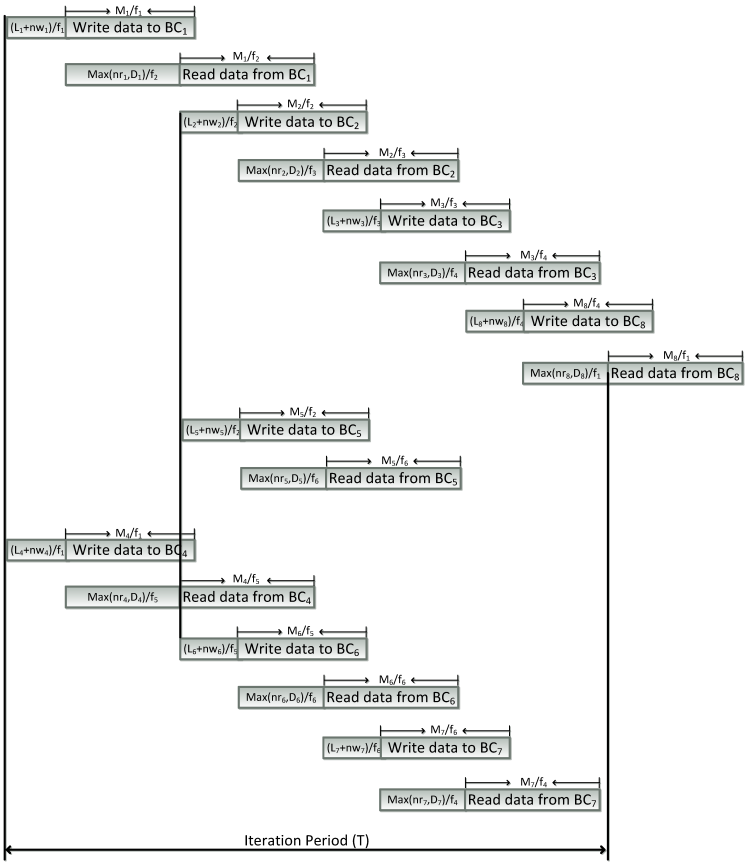


Figure 4-5: Mapped realization with interconnects and buffers, and processing elements realized as hardware logic.

make sure that read and write operations are correct. In the dataflow of Fig. 4-3(a), data coming out of buffers 5 and 6 should arrive at processing element 6 at the same time, as per the program flow. To make sure that processing element 6 will process the correct data frame, the parameters $nw_{1,4,5,6}$ and $nr_{1,4,5,6}$ are calculated to get the same *start_read* times for buffers 5 and 6. The same technique is used to calculate the other read and write times. The principle used is that when there are several fan-ins to a processing element, the effective data coming out of those fan-ins should arrive at the processing element at the same time.

4.2.2 Multi-Rate Support and Iteration Period Control

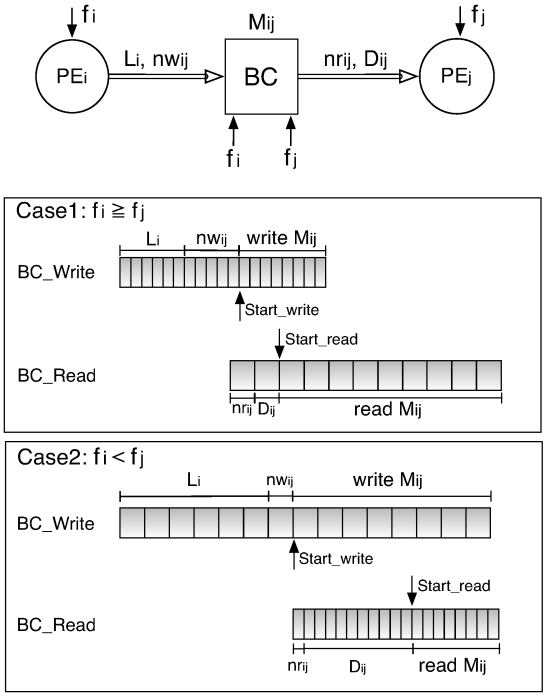


Figure 4-6: An example of a BBDF and buffer activity when different clock rates are specified for the processing elements.

While most designs usually have processing elements operating at the same fre-

quencies, this is not always the case as required by the designer. Fig. 4-6 shows a situation where the write operation to a buffer by PE_i and the read operation from the same buffer by PE_j occurs at different clock rates. In order to support such situations, the delay factor $D_{i,j}$ is used to synchronize the different elements within the buffer controller. We define f_i and f_j as the clock rates (in Hertz) of PE_i and PE_j respectively. When f_i is greater than f_j, the slower process does not have to wait as long as there is at least one block of valid data in the buffer. In this case, there is no overflow, since the next data block is not generated before the current data block is completely used (read, in this case), and therefore the delay factor is not needed. However, if f_i is less than f_j, the faster process has to wait in order to prevent data underflow, until enough data is written to the buffer. In this case, the following equation gives the minimum delay.

$$D_{i,j} = \left\lceil \left(\frac{M - nr_{i,j}}{f_i} - \frac{M - nr_{i,j} - 1}{f_j} \right) \times f_j \right\rceil \tag{4.2}$$

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

$$start\_write_{i,j} = \left[(L_i + nw_{i,j}) \times \frac{f_G}{f_i} \right] \tag{4.3}$$

and

$$start\_read_{i,j} = start\_write_{i,j} + \left[(nr_{i,j} + D_{i,j}) \times \frac{f_G}{f_j} \right] \tag{4.4}$$

The key benefit of enabling straightforward multi-rate support is that we can assign an arbitrary clock rate to any processing element while satisfying the requirements of the iteration period. When high-speed processing is necessary for a processing element, we set it to operate at a higher clock frequency in a critical section of the design. Similarly, all elements are by default assigned the lowest clock-speed that satisfy the iteration period requirements. This makes our architecture power-aware and minimizes the power footprint of our methodology. Fig. 4-7 illustrates two possible cases of iteration period given a buffer based dataflow. As shown in the figure, many iteration periods are made possible by simply varying the timing of the *start* control signals, and the other control signals are handled by the buffer controller. Individual buffer speeds may vary as long as the fan-in constraints, which are enforced such that the original execution characteristics are not modified, are satisfied.

Fig. 4-8 illustrates read and write operations being performed at different speeds. Even though the data block size $M_{i,j}$ may be too large for the buffer $BC_{i,j}$, the actual storage required by the implementation is not. For each buffer, the *start_write* and *start_read* signals are separated by t_{offset}. The storage requirement for each buffer is thus given by

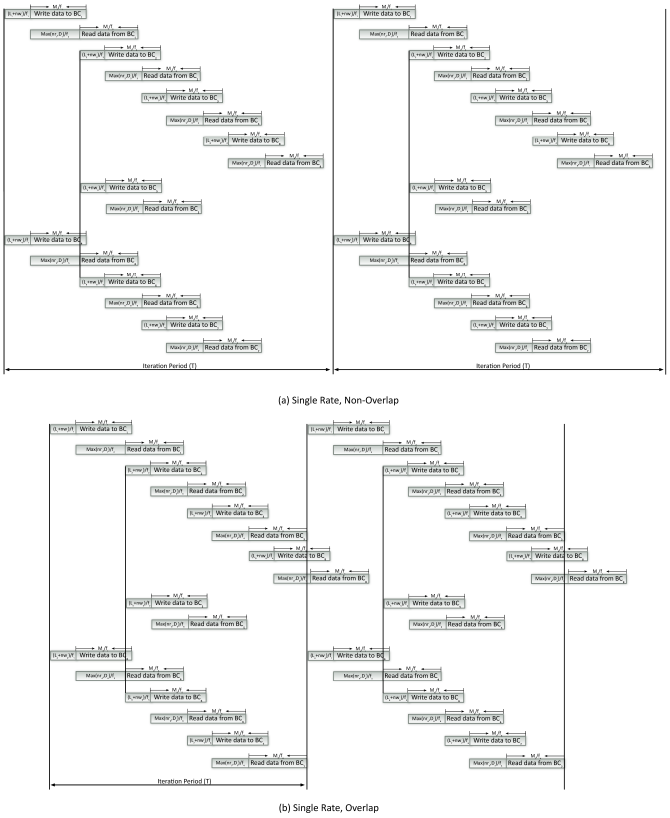


Figure 4-7: Illustration of iteration period adjustment by *start* signal manipulation.

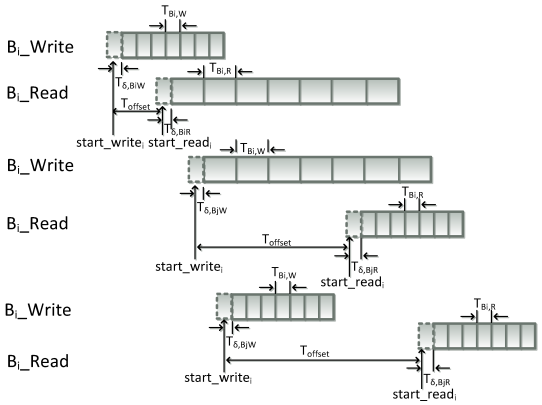


Figure 4-8: Buffer activities at different speeds, illustrating that the storage requirements depend on the signal timings .

$$storage\_size_i = \left\lceil \frac{start\_read_i - start\_write_i}{T_{wi}} \right\rceil \tag{4.5}$$

where T_{wi} is the time required to successfully complete a write operation. Hence, the total amount of data storage required by the application depends on the control signals generated by the buffer controller. In the event that t_{offset} is larger than the entire buffer activity duration, the storage requirement will be limited by the block size $M_{i,j}$.

4.2.3 Execution Controller Structure

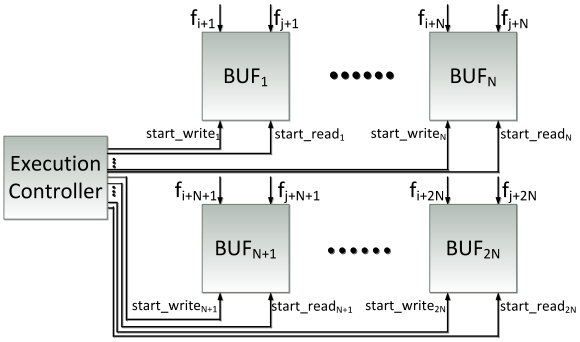


Figure 4-9: Execution controller, its connection and signals.

Fig. 4-9 shows the structure of the execution controller. The execution controller generates bit patterns for controlling the buffer controllers. Every two bits of information describes the activation signals for the write and read operations respectively. For example, the pattern "10" would start the write operation but not the read operation for a particular buffer controller. Similarly, a sequence such as "01010000" would only start the read operations for the first two buffers, while the other two buffers

are left unactivated. The bit patterns are thus stored as programs in the program memory as a large sequence of 1s and 0s. Thus, the execution timing of the data flow is easily controlled by modifying the program memory contents. As an example, if we want to delay the iteration period of the algorithm by a hundred cycles, we simply insert one hundred 0s into the program content.

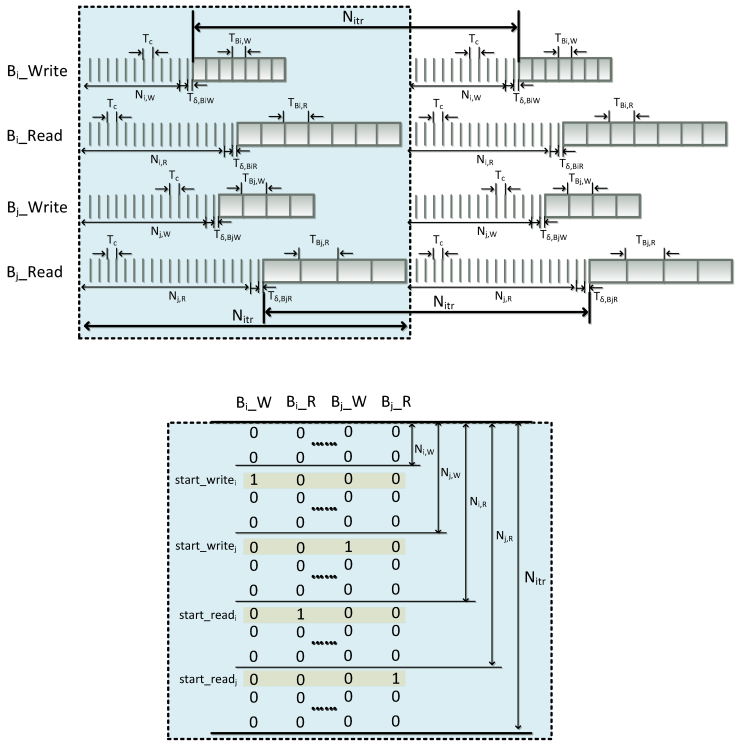


Figure 4-10: Timing diagram and corresponding program structure for iteratio period with non-overlapping buffer execution.

Fig. 4-10 shows the timing diagram of an example BBDF with different iteration period requirements and non-overlapping buffer activity. For these timing diagrams, their corresponding program content structure is shown below. The execution program can be generated by extracting any portion of the timing as long as the length of the interval is equal to the iteration period. Since this example has only two buffers,

the width of the program content is four. In addition, the number of non-zero rows is at most four, since there are four instance where the *start* signals are generated. If two or more buffer start times are identical, two rows may contain multiple 1s. Assuming the controller frequency as shown in the figure, the total length of the program N_{itr} is

$$N_{itr} = T_{itr} \times f_{controller} \tag{4.6}$$

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

Fig. 4-11 illustrates the buffer activity timing and its corresponding program structure where buffer activities overlap across iterations. In this case, the program is generated by selecting the interval where are buffer activities are shown (this period is indicated with a shaded box).

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

$$start\_read_i - T_{ri} < \frac{k_{ri}}{f_{controller}} < start\_read_i \tag{4.7}$$

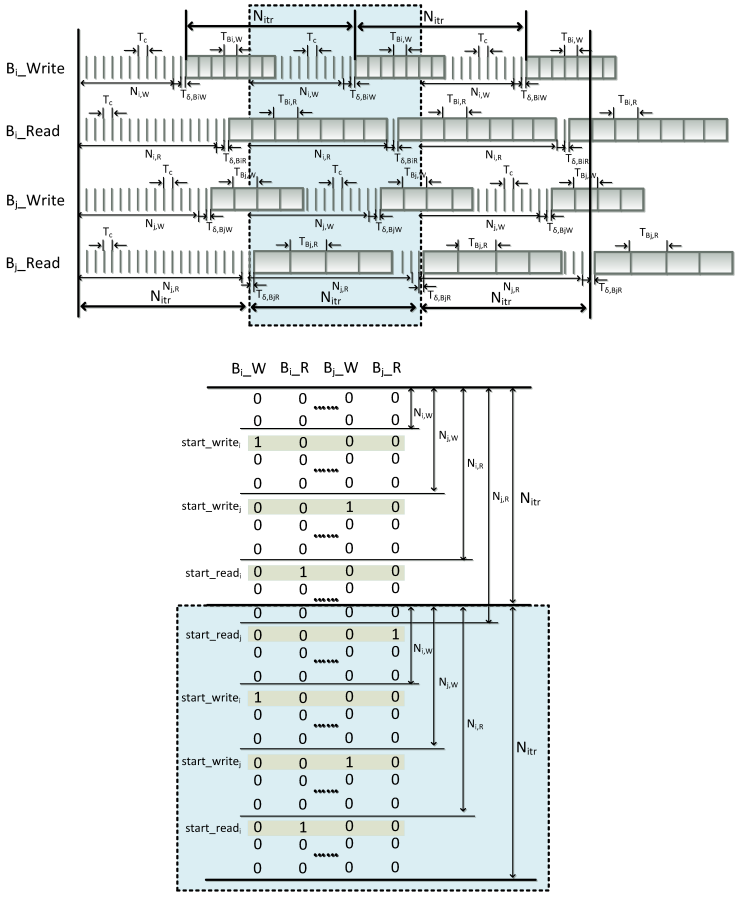


Figure 4-11: Timing diagram and corresponding program structure for iteration period with overlapping buffer execution.

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

$$start\_write_i - T_{wi} < \frac{k_{wi}}{f_{controller}} < start\_write_i \tag{4.8}$$

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

4.2.4 Joint Execution-Structural Control

If read/write activity amongst some buffers do not overlap with each other, these buffers can be replaced by one buffer in certain special cases. There are a number of scenarios where this concept can be used to reduce the total number of buffers. Consider the case of Fig. 4-12, where processor 1 is connected to processors 2 and 3 through buffers 1 and 2. In this case, the path of the dataflow is decided by processor 1. The data will be written to buffer 1 or 2 depending on the computational decision made by processor 1. In any case, both the buffers can never be active at the same time. Once the processor has decided its dataflow path, the buffers lying on the other path are effectively never used in the same cycle.

Taking advantage of this mutual exclusiveness, we replace buffers 1 and 2 with a single buffer labelled buffer 1, as shown in Fig. 4-12(b). As soon as the dataflow path is decided upon, the processor will send this information to the global controller,

which will then configure this shared buffer using the technique of dynamic path selection and handle the structural reconfiguration. Fig.4-13 shows the timing before and after buffer 1 and buffer 2 are shared by using a single buffer labelled buffer 1. Buffer sharing can also be implemented for select periods of time.

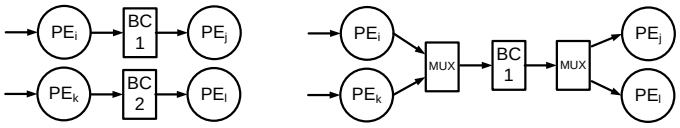


Figure 4-12: The concept of buffer sharing. If the buffer activities of two paths do not overlap, the same buffer controller can be used.

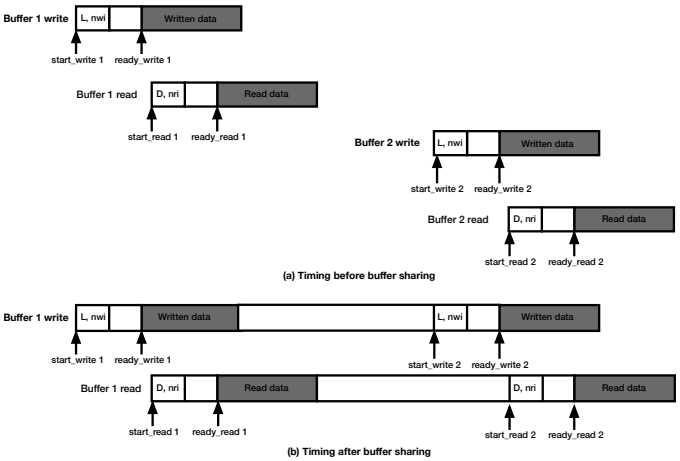


Figure 4-13: Timing before and after buffer sharing.

The *start_read* and *start_write* signals from the execution controller to buffer 2 are sent to buffer 1 instead. Similarly, the processors connected with buffer 2 are connected to buffer 1 instead during the buffer sharing period. These connections and control signals have to be handled by the structural controller.

The structural controller is shown in use in Fig. 4-14. The purpose of this controller is to configure the data flow by managing interconnections as well as the buffer

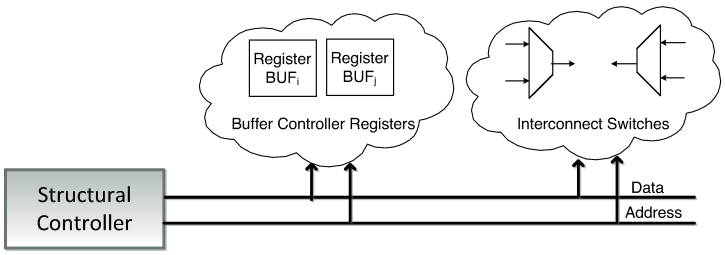


Figure 4-14: Structural controller and its connection to the buffer controllers and interconnect switches. The registers within the buffer controllers and the switches are address mapped.

controllers. All registers in the buffer controllers and interconnection multiplexers are mapped into memory, and therefore reconfiguring the data flow is simply a matter of writing the new configuration content to the registers. The size of an address depends on the number of buffer controllers and the interconnection complexity. The program memory of the structural controller stores multiple configurations, which allow for different data flows to be constructed by writing the corresponding control program content to the registers.

Fig. 4-15 illustrates the duration of the time that the buffer selection must be completed. Because of this additional requirement, the selection of the controller frequency must consider additional conditions

$$start\_write_i + M_i \cdot T_{wi} < \frac{k_{si}}{f_{controller}} < start\_write_j \tag{4.9}$$

where T_{wi} is the time required for the buffer write operation, and k_{si} is some integer constant. The index i is the buffer that has completed the activity and the index j is the buffer that needs to be set.

To maximize the buffer sharing, minimizing the buffer activity overlaps by selecting higher operating frequencies for the processing elements.

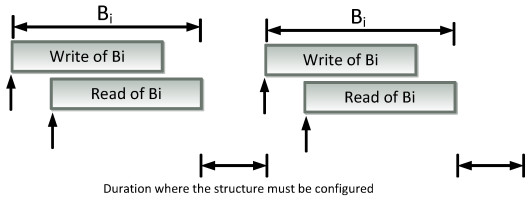


Figure 4-15: The range of the time which the structure must be reconfigured.

Fig. 4-16 illustrates the integrated controller with the execution flow of the buffer control and the structure modification.

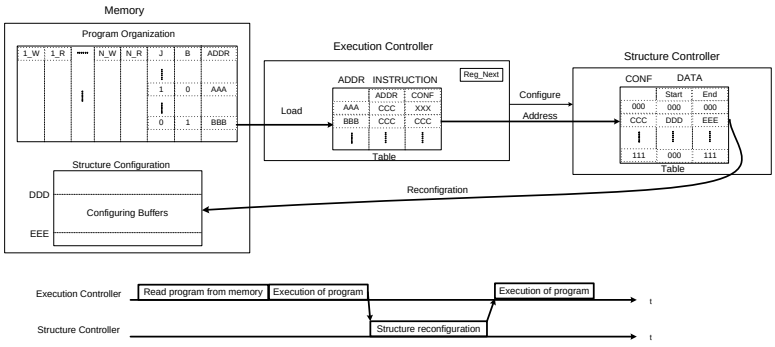


Figure 4-16: Connection of the processor and the buffers using a bus.

4.3 Asynchronous Hierarchical Execution Controller

4.3.1 Asynchronous Data Access

In all real designs, however, execution latency is always present. The latency may occur during the operation due to the fact that processing elements may not have a

fixed execution time. Regardless of the reason, dealing with this latency is critical for correct operation of the design.

Fig. 4-17 illustrates the timing where the actual data generation and consumption do not correspond with the *start_write* and *start_read* timing signal. Because of this situation, the buffer may write the wrong data samples. Similarly, the data read by the buffer controller may not properly read by the processing element.

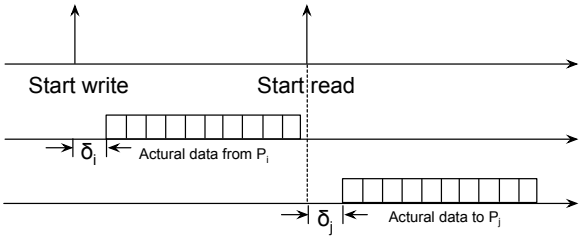


Figure 4-17: Illustration where the actual data generation and consumption may not match with the *start_write* and *start_read* timing.

In order to handle the asynchronous buffer activities, the additional handshake control signals are necessary as illustrated in Fig. 4-18. The *start_write* signal will be generated as before, but the actual writing process starts when the processing element provide *ready_write* signal prior to actually generating the data. Similarly, the *start_read* signal is generated by the controller but the actual reading process will starts when the consuming processing element provide *ready_read* signal. Hence, additional circuitry is necessary. The correct timing signals are illustrated in Fig. 4-19. Unlike the synchronous case, 1 clock cycles delay after the ready signal is introduced in the asynchronous case.

One potential problem with such random latency is that undesirable timing violation and invalid buffer controller operation may be possible especially in a tight

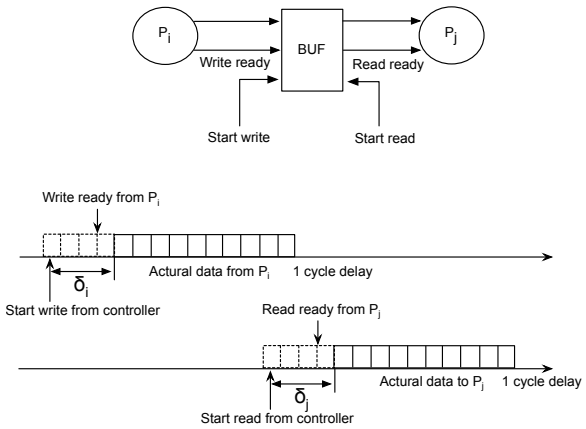


Figure 4-18: Buffer controller for correct data read/write operations.

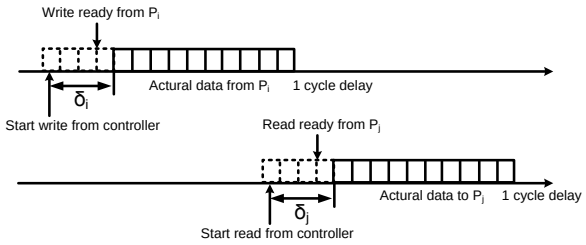


Figure 4-19: Buffer controller for correct data read/write operations.

buffer activity schedule. Two cases are illustrated in Fig. 4-20. The first case is the accessing the same buffer controller between the iterations. If the iteration period is aggressively selected, the operation of the buffer controller may not be completed before the next iteration. It is desirable to maintain T_{slack}. The second case is the sharing of the buffer controller within the same iteration. Similarly withe the other case, T_{slack} must be maintained.

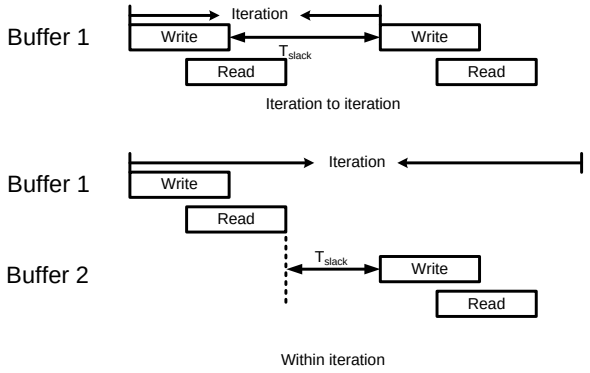


Figure 4-20: Invalid operation of the buffer controller due to the latency.

4.3.2 Dynamic Structural Reconfiguration

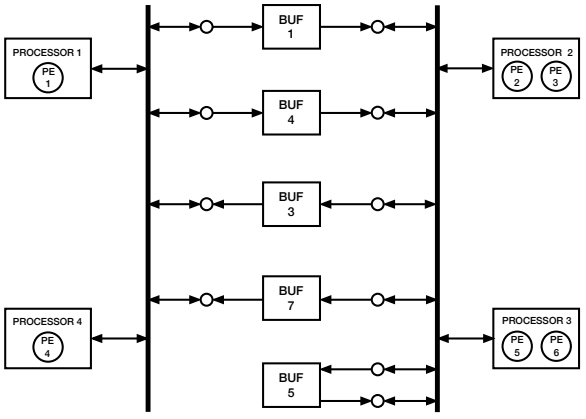


Figure 4-21: Illustration of multi-processor mapping where the dynamic structural reconfiguration as well as unknown delay need to be supported.

Fig. 4-21 shows the multi-processor mapping of the buffer-based dataflow shown in Fig. 4-3(a), using the buffer-based mapping shown in Fig. 4-5. The division of processing elements is based on the principles previously derived. The processors communicate via a shared bidirectional bus, which in turn is connected to the interconnect structure which enables data transfer to and from the buffers. This mapping method can encounter several problems in real systems. In the case that processing elements in one processor are on several different sequential data paths and rely on the outputs of the previous processing elements, the overall execution time will increase since certain processing elements will have to wait longer.

Fig. 4-22 illustrates the processor activity for all processing elements. The processor activity consists of both buffer reading and writing duration. The processing elements can be grouped to map to a processor if their activities do not overlap in time.

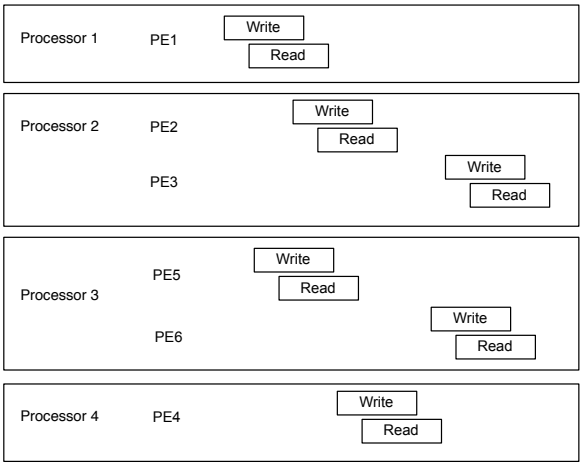


Figure 4-22: Illustration of processor activity timing.

There are two cases where we use the technique of dynamic path selection. One is the buffer sharing scheme, where the single buffer controller replaces the multiple

buffer controllers to operate through time multiplexing. The other case is that the processor itself decides its data path. When the buffer sharing scheme is used, the host processor calculates which buffers are to be replaced and sends this information to the controllers on the target design. If the path is to be decided by the processors, each processor only decides which buffer should be connected with it. In this scheme, the path is formed by the collective decisions of all the processors regarding their own path.

Due to buffer sharing, the topology must change during the execution of the dataflow. Especially, the buffer controllers connected to the Processor 1 and Processor 2 need to be configured before their usages.

4.3.3 Hierarchical Controller Design

Fig. 4-23 illustrates a hierarchical controller connections. The original dataflow is divided into a several groups where each group operates autonomously. Each group incorporates its own execution and structural controllers. The edges connecting these groups have buffer controllers. The buffer controllers specified as bridge buffer controllers, which are controlled by a global execution and structural controller. The local controllers and the global controller are synchronized for correct overall operations.

Fig. 4-24 shows the buffer activity control timing for the single controller case. As discussed in the previous section, the controller frequency and the control memory are selected considering the all of processing elements and buffer controllers.

Since the partitions only have a small subset of the dataflow, the frequency and

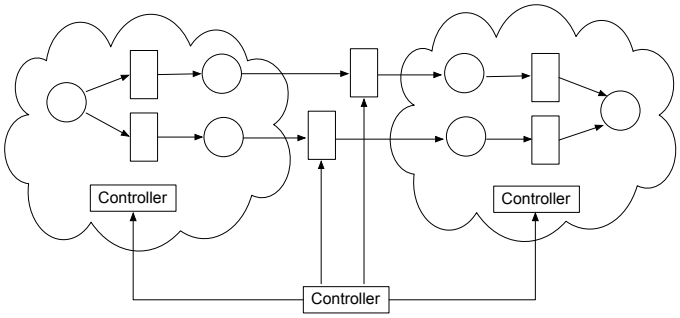


Figure 4-23: Illustration of hierarchical controllers. Each partition has it sown controller and the connection of the partitions is controller by the global controller.

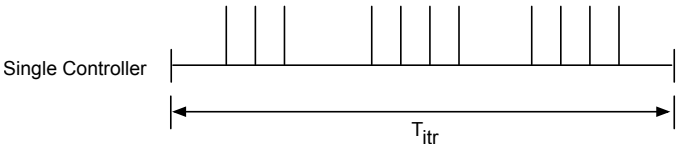


Figure 4-24: Illustration of buffer activity timing diagrams.

the control memory of the local execution controller only consider the small set of processing elements. Hence, the the program size of the local controller is smaller than the signle controller case.

Fig. 4-25 shows the buffer activity timing for the partitions. And the global controller handing the bridge buffer controllers is also illustrated. In order to properly synchronize the overall execution, the following conditions must be satisfied.

Fig. 4-26 illustrates the timing of the reconfiguration. The local controller timing is illustrated separately. Each local controller operates with its own controller frequency. The global controller timing indicates the activity signals for the bridge buffer controller. The timing is compared with that of the single controller case.

Fig. 4-27 illustrates the control signal interfaces between the local controllers and the global controller. Note that there is no direct connection between the global

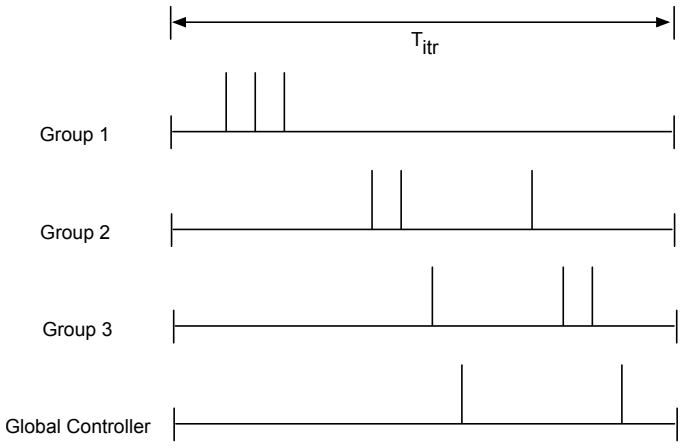


Figure 4-25: Illustration of buffer activity timing diagrams.

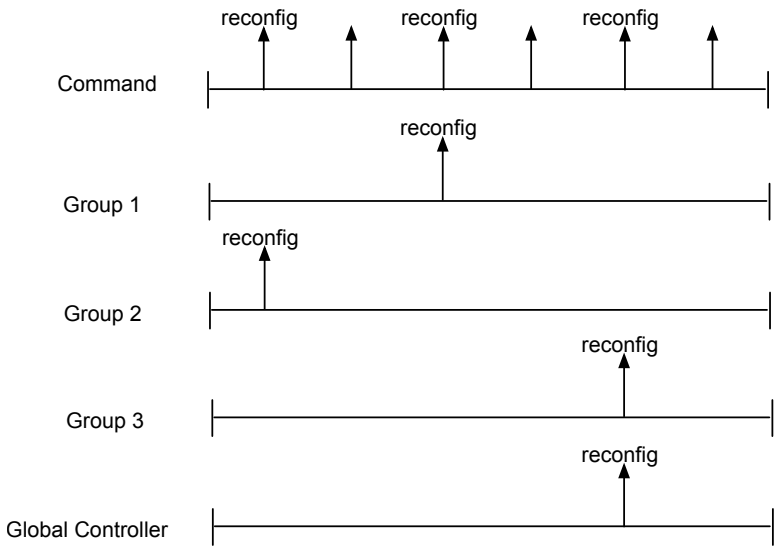


Figure 4-26: Illustration of multi-level structural reconfigration timing.

controller and the local controllers. The overall flow of the dataflow is maintained by controlling the bridge buffer controllers. Because of the elimination of the direct interfaces between the controllers, the scalable design is possible.

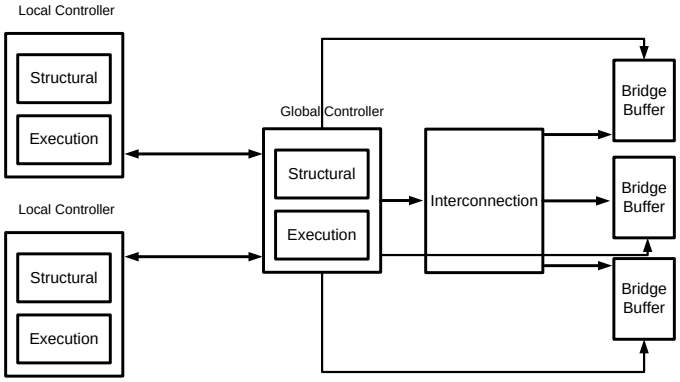


Figure 4-27: Illustration of multi-level structural reconfigration timing.

Fig. 4-28 illustrates the program size comparison between the single controller case and the multiple controller case. In both case, the length of the program is identical since they have the same controller frequency and the iteration period. However, the width of the single controller case is equal to the sum of the widths of the distributed controllers. Note that after the integration of the group including the controller, the group can be viewed as another processing element. Because the group maintains the processing element property, the design process is scalable.

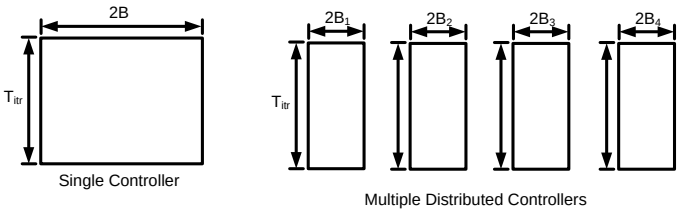


Figure 4-28: Illustration of the program size between the single controller and multiple distributed controllers.

4.4 Evaluation

4.4.1 Evaluation Setup

In the evaluation, a dataflow illustrated in Fig. 4-29 is used. The dataflow consists 14 processing elements and 16 buffer controllers. A feedback is included in the dataflow. The data flow is evaluated using the single controller. Then, the dataflow is divided into three partitions to demonstrate the hierarchical controller design. In the partitions, the feedback path is intentionally divided. The buffer sharing is also considered in the evaluation where the structural configuration is performed.

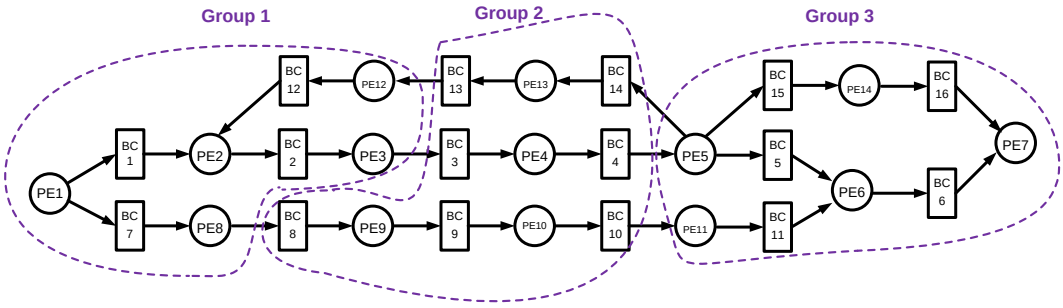


Figure 4-29: The topology of the buffer-based dataflow used for evaluation.14 processing elements and 16 buffer controllers are used. The original dataflow graph is divided into three sub dataflow graph for hierarchical controller illustration.

Fig. 4-30 illustrates the evaluation parameters for the processing elements and buffer controllers. Two sets of frequencies are used for the processing elements. The frequencies of the processing elements are chosen such that while these clock frequencies are obtainable by clock division from single clock, the resulting controller clock frequency is large causing the large controller memory requirement.

Similarly with the buffer controllers, two sets of the data block size are chosen as illustrated in the figures. The values are tabulated in Tables 4.2 and a timing

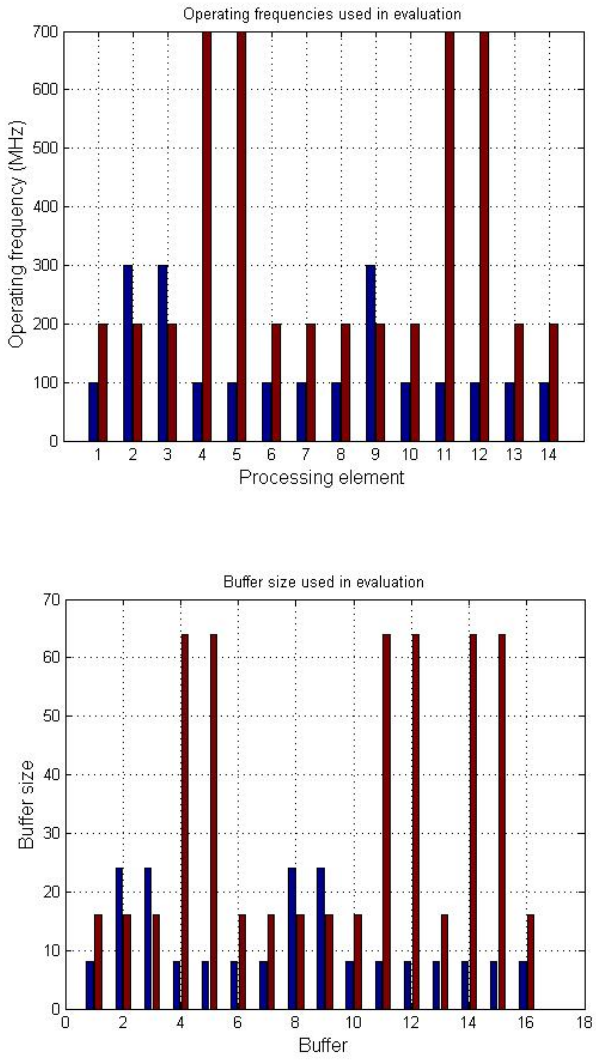


Figure 4-30: (a) Illustration of the operating frequencies of the processing elements. (b) Illustration of the data block size used for the buffer controllers.

diagram is shown in Fig. 4-31.

Table 4.2: Buffer controller configuration for the dataflow in Fig. 4-29.

Buffer	M	Write		Read	
		Time(ns)	f(MHz)	Time(ns)	f(MHz)
1	8,16	21,11	100,200	101,16	300,200
2	8,16	111,31	300,200	116,36	100,200
3	8,16	146,51	100,200	156,111	100,700
4	8,16	186,119	100,700	196,124	100,700
5	8,16	226,128	100,700	236,129	100,700
6	8,16	266,153	100,200	276,158	100,200
7	8,16	21,11	100,200	31,16	100,200
8	8,16	61,31	100,200	166,36	300,200
9	8,16	181,51	300,200	186,56	300,200
10	8,16	201,71	300,200	206,76	100,200
11	8,16	226,91	100,200	236,129	100,200
12	8,16	306,173	100,200	416,178	100,200
13	8,16	266,153	100,200	276,158	100,200
14	8,16	226,128	100,200	236,138	100,200
15	8,16	226,128	100,700	236,133	100,700
16	8,16	266,137	100,700	276,148	100,200

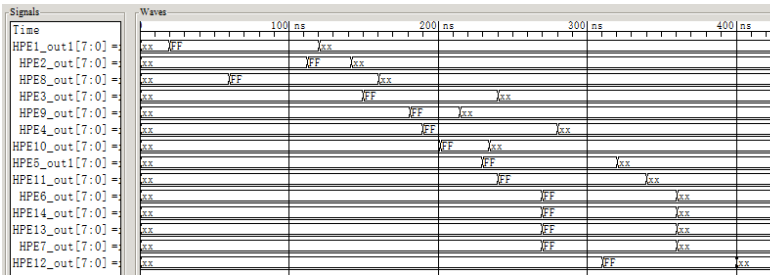


Figure 4-31: The buffer activity timing of the buffer-based dataflow used for evaluation.

4.4.2 Execution Controller Evaluation

In this section, we consider the execution of the dataflow with the single controllers.

The different iteration period requirements are considered. The start read and write

times of each iteration period requirement are illustrated in Fig. 4-32. For each iteration constraint, the controller speed and the control program memory size is illustrated in Tables 4.3. The controller frequencies are increased when the application is faster. The controller memory size maintains similar sizes. The timing for different iteration constraint are shown in Fig. 4-33.

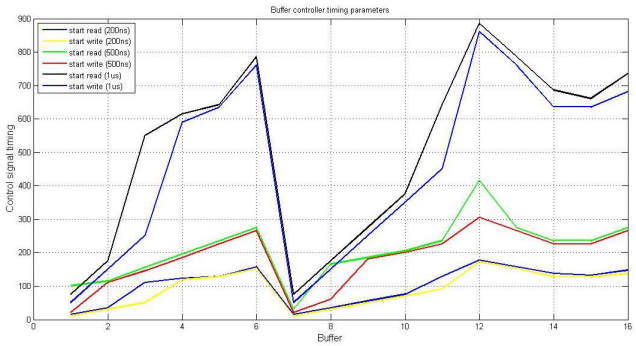
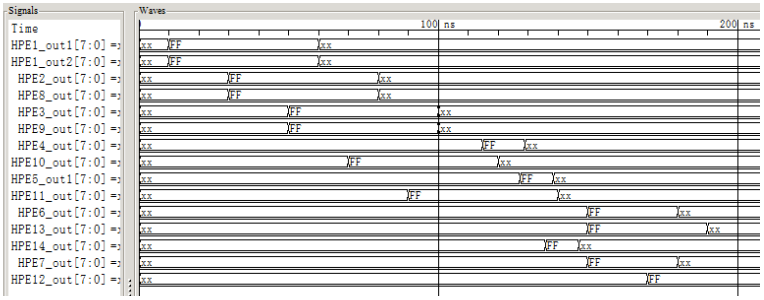


Figure 4-32: Illustration of the buffer controller timing parameters for three sets of iteration period requirements.

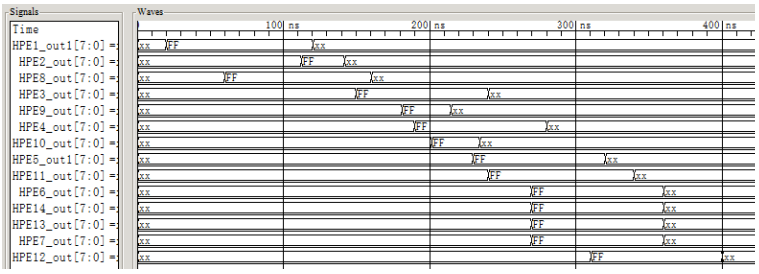
Table 4.3: controller frequency and the size of the program for different iteration period.

Iteration Time	200ns	500ns	1us
Controller Frequency	700MHz	300MHz	150MHz
Controller Memory Size	15.4K	15.1K	16.2K

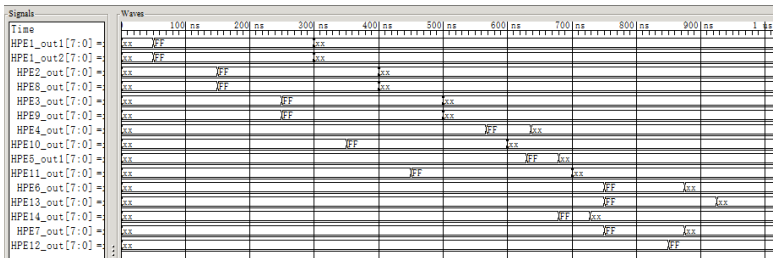
Instead of using the single controller as shown above, the hierarchical controller structure is used in the evaluation. As illustrated in Fig. 4-29, the original dataflow is divided into three groups. The feedback path is intentionally divided to demonstrate that the tightly coupled edges are also supported. Tables 4.4 illustrates the controller frequency and the size of the program for different iteration period. While the execution is identical to the single controller case, the controller memory size increased.



(a) Iteration time = 200 ns



(b) Iteration time = 500 ns



(c) Iteration time = 1 us

Figure 4-33: Buffer timing flow for the dataflow in Fig. 4-29.

So the memory size is increased for the flexibility of the controller design.

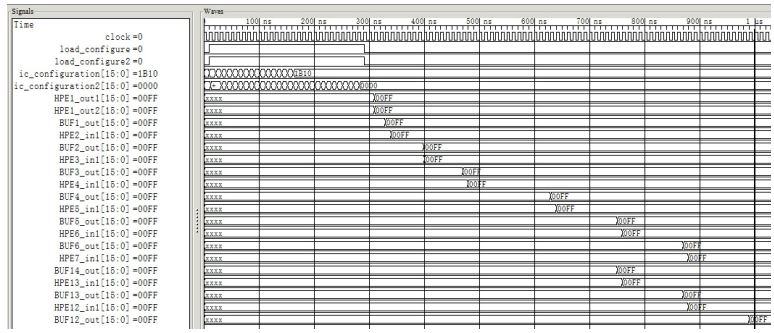
Table 4.4: controller frequency and the size of the program for different iteration period for hierarchical controller.

Iteration Time	200ns	500ns	1us
Controller Frequency - Group 1	700MHz	300MHz	150MHz
Controller Memory Size - Group 1	15.4K	15.1K	16.2K
Controller Frequency - Group 2	700MHz	300MHz	150MHz
Controller Memory Size - Group 2	12.9K	9.2K	11K
Controller Frequency - Group 3	700MHz	300MHz	150MHz
Controller Memory Size - Group 3	4.3K	3K	3.3K
Controller Frequency - Global	700MHz	300MHz	150MHz
Controller Memory Size - Global	15.4K	15.1K	16.2K

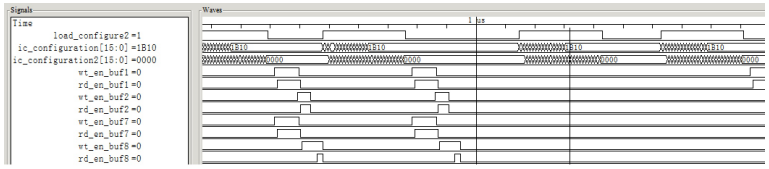
4.4.3 Buffer Sharing and Configuration

In this section, buffer sharing and dynamic reconfiguration of the dataflow considered in the evaluation. Fig. 4-34 illustrates the timing of the buffer controllers. The timing parameters are set so that the buffer sharing is maximized.

Fig. 4-35 illustrates the configuration timing signals that triggers the reconfiguration of the dataflow buffer controller usages. The configuration signals are generated right after the buffer usages. In the controller memory, in addition to the buffer controller timing, the configuration timing is also incorporated. When the hierarchical controller structure is used, all controllers are synchronized. Since it takes some time to configure, the all controllers stops the execution during a part of reconfiguration.



(a) Without buffer sharing



(b) With buffer sharing

Figure 4-34: Illustration of the buffer activity timing for maximizing the buffer sharing.

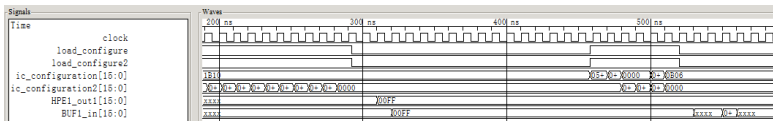


Figure 4-35: Configuration and reconfiguration of buffer sharing.

4.5 Conclusion

This chapter presents a flexible hierarchical controller design for asynchronous buffer based dataflows. The controller considers the execution flow and the structural configuration separately but collaboratively for dynamic reconfiguration of the dataflows. By allowing tree structured controller makes the design scalable. The buffer controllers that go between the processing elements isolates the execution hence making the flexible controller design process. The proposed controller supports traditional synchronous design as well as multi-core processors. The design methodology also includes the design of a top-level global controller, responsible for the configuration of the buffers and interconnections as well as path selection. The dynamic reconfigurability allows us to map multiple processing elements onto a single core and switch between them during run-time. The proposed design is evaluated with SystemC.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This thesis proposes a high performance partition based reconfigurable platform to execute multiple concurrent applications. Hierarchical controllers to support dynamic configuration of this platform is presented. A power-efficient frequency selection algorithm that minimizes power consumption and maintains the system performance requirement is used to provide the clock frequencies for the platform.

Chapter 2 proposes a fast power-efficient frequency selection algorithm that minimizes power consumption and maintains the system performance requirement for multiple data-centric applications efficiently. It utilizes variable step size scheduling techniques combined with dynamic frequency selection by exploiting the correlation between the frequencies and the iteration time to achieve minimum power under timing constraint. Then we improve the solution to limited number of clock frequencies when applications are dynamically mapped. If a data-centric application can

be represented by a dataflow, the proposed algorithm gets the solution of the clock frequencies in fast speed than simulated annealing algorithm. Experimental results show that it gets typically same results with simulated-annealing based method while running 100 times faster. Compared to the existing algorithms, the proposed algorithm has flexibility of applying to any kind of dataflow representation under various setting of timing constraint.

Chapter 3 proposes a partition based reconfigurable platform for multiple applications executing concurrently. This platform consists of processing elements and buffers interacting through a reconfigurable interconnect, dividing the applications into partitions so that they can execute concurrently. It uses control memory and data memory to save the control information and the processed data. Hierarchical controllers are used to manipulate large numbers of partitions and achieve rapid and dynamic reconfigurability. It also supports applications of different sizes of data within the partition by converting the parallel data into the serial smaller size of data in the data conversion modules within the partition and then back to the desired size. Also, the smaller size of data simplifies the interconnection between the processing elements. However, to provide the same throughput of the dataflow, multiple frequencies of clocks are necessary for the multi-rate applications. Therefore, we provide the clock distribution to the platform to minimize the inaccuracy of the clocks. We build this platform in SystemC model and simulation gives the functional verification of configuration and execution of multiple applications.

Chapter 4 presents a flexible hierarchical controller design for asynchronous buffer based dataflows. The controller considers the execution flow and the structural config-

uration separately but collaboratively for dynamic reconfiguration of the dataflows. By allowing tree structured controller makes the design scalable. The buffer controllers that go between the processing elements isolates the execution hence making the flexible controller design process. The proposed controller supports traditional synchronous design as well as multi-core processors. The design methodology also includes the design of a top-level global controller, responsible for the configuration of the buffers and interconnections as well as path selection. The dynamic reconfigurability allows us to map multiple processing elements onto a single core and switch between them during run-time.

5.2 Future Work

The applications considered in this thesis are dataflow examples containing feedback and feed-forward paths. The future topic can be to map real signal processing applications to the proposed reconfigurable architecture as if they can be represented as buffer based dataflow. Methods of dividing the partitions to maintain the highest speed of the system is an interesting topic. A smaller number of partitions have large interconnection within them but keep higher global speed. A larger number of partitions have fast operating speed but cause complicated global interconnections. We can balance this tradeoff according to applications in the future. Another topic related to this platform is how to do the mapping of the applications to achieve efficiency and lowest total iteration time. In future research, control architecture applicable in multiple applications within one partition is interesting. When the current applica-

tion changes from one to the other, the controller within the shared partition will decide the right start time and timing parameters. Furthermore, the reconfigurable architecture will be able to reduce power consumption by switching the applications and operating frequencies between these partitions.

Bibliography

[1] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast Prototyping of Data Path Intensive Architectures", IEEE Design and Test, Vol 8, No. 2, pp. 40-51, 1991.

[2] R. Tessier and W. Burleson, "Reconfigurable Computing and Digital Signal Processing: A survey", Journal of VLSI Signal Processing, Vol 28, No. 1-2, pp. 7-27, 2001.

[3] A. Abnous, *Low Power Domain Specific Processors for Digital Signal Processing*, Ph. D. thesis, University of California, Berkeley, 2001.

[4] A. Pelkonen, K. Masselos and M. Cupk, "System-Level Modeling of Dynamically Reconfigurable Hardware with SystemC", *International Parallel and Distributed Processing Symposium*, Apr 2003.

[5] Reiner Hartenstein, "A Decade of Reconfigurable Computing: a Visionary Retrospective", *Proceedings of Design, Automation and Test in Europe*, 2001.

[6] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk and P.Y.K. Cheung, "Reconfigurable Computing: Architectures and Design Methods", *IEE Proceedings of Computers and Digital Techniques*, Vol. 152, pp 153-207, 2005.

[7] H. Singh, M. Lee; G. Lu, F. J. Kurdahi, N. Bagherzadeh and E. M. C. Filho, "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications", *IEEE Transctions on Computers*, Vol. 49, No. 5, pp. 465-481, May 2000.

[8] C. Ebeling, D. C. Cronquist and P. Franklin, "RaPiD - Reconfigurable Pipelined Datapath", *Lecture Notes in Computer Science, Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*, Vol. 1142, pp. 126-135, 1996.

[9] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler", *IEEE Computer*, Vol. 33, No. 4, pp. 70-77, Apr 2000.

[10] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow", *Proceedings of the IEEE*, Vol. 75, No. 9, pp. 1235-1245, 1987.

[11] J. B. Dennis, "First Version of a Data Flow Procedure Language", *Symposium on Programming*, pp. 362-374, 1974.

[12] Ivan Corretjer, Chia-Jui Hsu, and Shuvra S. Bhattacharyya, "Configuration and Representation of Large-Scale Dataflow Graphs using the Dataflow Interchange Format", *IEEE Workshop on Signal Processing Systems Design and Implementation*, 2006.

[13] Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya, "Parameterized Dataflow Modeling for DSP Systems", *IEEE Transactions on Signal Processing*, Vol. 49, NO. 10, pp. 2408-2421, 2001.

[14] J. T. Buck and R. Vaidyanathan, "Heterogeneous Modeling and Simulation of Embedded Systems in El Greco", *Proceedings of the International Workshop Hardware Software Codes*, 2000.

[15] E. A. Lee, D. G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing", *IEEE Transactions on Computers*, 1987.

[16] William Plishker, Nimish Sane, Shuvra S. Bhattacharyya, "A Generalized Scheduling Approach for Dynamic Dataflow Applications", *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2009.

[17] S. Banerjee, T. Hamada, P. M. Chau, and R. D. Fellman, "Macro Pipelining based Scheduling on High Performance Heterogeneous Multiprocessor Systems", *IEEE Transactions on Signal Processing*, Vol. 43, NO. 6, pp. 14681484, 1995.

[18] Praveen K. Murthy, "A Buffer Merging Technique for Reducing Memory Requirements of Synchronous Dataflow Specifications", *Proceedings of 12th International Symposium on System Synthesis*, 1999.

[19] Simone Casale Brunet, Marco Mattavelli and Jorn W. Janneck, "Buffer Optimization Based on Critical Path Analysis of a Dataflow Program Design", *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2013.

[20] Sharad Sinha and Wei Zhang, "SynDFG: Synthetic Dataflow Graph Generator for High-level Synthesis", *Asia Symposium on Quality Electronic Design (ASQED)*, 2015.

[21] Ganghee Lee, Seokhyun Lee, and Kiyoung Choi, "Automatic mapping of application to coarse-grained reconfigurable architecture based on high-level synthesis techniques", *International SoC Design Conference (ISOCC)*, 2008.

[22] Zhaotong Li, Zheng Huang, Shuai Chen, Xuegong Zhou, Wei Cao and Lingli Wang, "A modeling and mapping method for coarse and fine mixed-grained reconfigurable architecture", *International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, 2012.

[23] Yoonjin Kim, Mary Kiemb, Chulsoo Park, Jinyong Jung, Kiyoung Choi, "Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain specific optimization,", *Proceedings in Design, Automation and Test in Europe Conference*, 2005.

[24] Deepak Sreedharan and Ali Akoglu, "A hybrid processing element based reconfigurable architecture for hashing algorithms,", *IEEE International Symposium on Parallel and Distributed Processing*, 2008.

[25] Rajesh Kumar Pal, Kolin Paul and Sanjiva Prasad, "ReKonf: A Reconfigurable Adaptive ManyCore Architecture,", *IEEE International Symposium on Parallel and Distributed Processing with Applications*, pp. 182 - 191, 2012.

[26] Shekhar Srikantaiah, Emre Kultursay, Tao Zhang, Mahmut Kandemir, Mary Jane Irwin and Yuan Xie, "MorphCache: A Reconfigurable Adaptive Multi-level Cache hierarchy,", *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 231242, 2011.

[27] Matthew A. Watkins and David H. Albonesi, "ReMAP: A Reconfigurable Heterogeneous Multicore Architecture,", *IEEE/ACM International Symposium on Microarchitecture*, pp. 497 - 508, 2010.

[28] Y. Lu, L. Benini and G. D. Micheli, "Low-Power Task Scheduling for Multiple Devices," *In Proceedings of the International Workshop on Hardware/Software Codesign*, pp. 39-43, May. 2000.

[29] S. Irani, S. Shukla, and R. Gupta, "Algorithms for power savings," *In Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 37-64, 2003.

[30] Shaobo Liu, Jun Lu, Qing Wu and Qinru Qiu, "Harvesting-Aware Power Management for Real-Time Systems with Renewable Energy", *IEEE Transactions on VLSI Systems*, vol. 20, NO. 8, pp. 1473-1486, August 2012.

[31] Woohyung Chun, Sungroh Yoon and Sangjin Hong, "Energy-Aware Interconnect Resource Reduction Through Buffer Access Manipulation for Data-Centric Applications", *IEEE Transactions on VLSI Systems*, vol. 19, NO. 5, pp. 818-831, May 2011.

[32] Woohyung Chun, Sungroh Yoon and Sangjin Hong, "Buffer Controller-Based Multiple Processing Element Utilization for Dataflow Synthesis", *IEEE Transactions on VLSI Systems*, May 2010.

[33] Jiong Luo and Niraj Jha, "Static and Dynamic Variable Voltage Scheduling Algorithms for Real-Time Heterogeneous Distributed Embedded Systems," *In Proceedings of the 15th International Conference on VLSI Design*, pp. 719-726, 2002.

[34] Ihor O. Bohachevsky, Mark E. Johnson and Myron L. Stein, "Generalized Simulated Annealing for Function Optimization," *Technometrics*, Vol. 28, NO. 3, August 1986.

[35] Emil Talpes and Diana Marculescu, "Toward a Multiple Clock/Voltage Island Design Style for Power-Aware Processors", *IEEE transactions on VLSI systems*, Vol. 13, No. 5, pp. 591-603, May 2005.

[36] Greg Semeraro, David H. Albonesi, Steven G. Dropsho, Grigorios Magklis, Sandhya Dwarkadas and Michael L. Scott, "Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture", *35th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 356-367, 2002.

[37] Greg Semeraro, Grigorios Magklis, Rajeev Balasubramonian, David H. Albonesi, Sandhya Dwarkadas and Michael L. Scott, "Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling", *Proceedings. Eighth International Symposium on High-Performance Computer Architecture*, pp. 29-40, 2002.

[38] Grigorios Magklis, Greg Semeraro, David H. Albonesi, Steven G. Dropsho, Sandhya Dwarkadas and Michael L. Scott, "Dynamic Voltage and Frequency Scaling for A Multiple-Clock-Domain Microprocessor", *IEEE Micro*, Vol. 23, Issue. 6, pp. 62-68, 2004.

[39] Qiang Wu, Philo Juang, Margaret Martonosi and Douglas W. Clark, "Voltage and Frequency Control with Adaptive Reaction Time in Multiple-Clock-Domain Processors", *11th International Symposium on High-Performance Computer Architecture*, pp. 178-189, Feb 2005.

[40] Umit Y. Ogras, Radu Marculescu, Diana Marculescu and Eun Gu Jung, "Design and Management of Voltage-Frequency Island Partitioned Networks-on-Chip", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 17, Issue. 3, pp. 330 - 341, Mar 2009.

[41] J. M. Chabloz and A. Hemani, "Distributed DVFS Using Rationally-Related Frequencies and Discrete Voltage Levels", *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design*, pp. 247 - 252, 2010.

[42] Thiago Raupp da Rosa, Vivian Larrea, Ney Calazans and Fernando Gehm Moraes, "Power Consumption Reduction in MPSoCs through DVS", *25th Symposium on Integrated Circuits and Systems Design*, 2012.

[43] Jing Chen, Tongquan Wei and Jianlin Liang, "State-Aware Dynamic Frequency Selection Scheme for Energy-Harvesting Real-Time Systems", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 22, pp.1679 - 1692, 2014.

[44] Junhee Mun, Shunghan Cho and Sangjin 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, June 2007.

[45] J. Lee, K. Choi and N.D. Dutt, "An Algorithm for Mapping Loops onto Coarse-Grained Reconfigurable Architectures", *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, pp. 183-188, 2003.

[46] Y. Guo, G.J.M. Smit, P.M. Heysters and H. Broersma, "A Graph Covering Algorithm for a Coarse Grain Reconfigurable System", *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, pp. 199-208, 2003.

[47] Y. Guo, G.J.M. Smit, H. Broersma, M.A.J. Rosien and P.M. Heysters, "Mapping Applications to a Coarse Grain Reconfigurable System", *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, pp. 221-235, 2003.

[48] Y. Guo, C. Hoede and G.J.M. Smit, "A Column Arrangement Algorithm for a Coarse-grained Reconfigurable Architecture", *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2006.

[49] Nikhil Bansal, Sumit Gupta, Nikil Dutt, Alex Nicolau and Rajesh Gupta, "Interconnect-Aware Mapping of Applications to Coarse-Grain Reconfigurable Architectures", *Proceedings of Field Programmable Logic and its Applications*, pp. 891-899, 2004.

[50] Z. Huang, S. Malik, "Exploiting Operational Level Parallelism through Dynamically Reconfigurable Datapash", *DAC*, 2002.