

國立交通大學

電子工程學系 電子研究所

碩士論文

應用於多週期通訊架構上達成互連資源共享之

通道與暫存器配置演算法

**A Register and Channel Allocation Algorithm for  
Interconnect Resource Sharing on  
Multicycle Communication Architecture**

研究生：洪玉如

指導教授：黃俊達 博士

中華民國九十六年六月

應用於多週期通訊架構上達成互連資源共享之  
通道與暫存器配置演算法

A Register and Channel Allocation Algorithm for  
Interconnect Resource Sharing on  
Multicycle Communication Architecture

研究生：洪玉如

Student：Yu-Ju Hong

指導教授：黃俊達博士

Advisor：Dr. Juinn-Dar Huang

國立交通大學

電子工程學系 電子研究所



A Thesis

Submitted to Department of Electronics Engineering & Institute of Electronics

College of Electrical and Computer Engineering

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of Master

in

Electronics Engineering & Institute of Electronics

June 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年六月

# 應用於多週期通訊架構上達成互連資源共享之 通道與暫存器配置演算法

研究生：洪 玉 如

指導教授：黃 俊 達 博士

國 立 交 通 大 學

電 子 工 程 學 系 電 子 研 究 所

## 摘 要

在深次微米的科技裡，連線的延遲不但已不能再被忽略，更隨著科技的進步而逐漸的主導了系統的時間延遲。為了處理不斷增加的時間延遲，一些先進的架構合成流程採用了可允許多週期通訊的分散式暫存器架構。建立在相同的原則之上，規律分散式暫存器架構接著被提出。這個架構利用高度規律的構造彌補分散式暫存器架構的缺陷，大幅提升了預測連線延遲的正確性。然而，規律分散式暫存器架構需要大量的互連資源，特別是大量昂貴的全局接線。在這篇論文中，我們針對規律式分散暫存器之整體資源共享架構，將通道與暫存器的分配轉成資料傳輸排程的問題以進而減少全局接線的使用量。一般而言，傳統全局繞線會儘量分散傳輸以避免擁塞，但我們提出了一個集中導向之傳輸路程排程器，藉由集中路徑來提高連線共享的可能性。另一方面，我們使用通道性的傳輸時間排程器，充分利用資料傳輸本有的寬裕時間以解決通道中的資源競爭。實驗結果顯示，比起原本的規律分散式暫存器架構，我們的演算法在應用於大型的設計時，能節省高達 81.7% 的全局連線以及 8.2% 的暫存器。

# **A Register and Channel Allocation Algorithm for Interconnect Resource Sharing on Multicycle Communication Architecture**

Student: Yu-Ju Hong

Advisor: Dr. Juinn-Dar Huang

Department of Electronics Engineering & Institute of Electronics  
National Chiao Tung University

## **Abstract**

In deep submicron technology, wire delay is no longer negligible and is gradually dominating the system latency. Several state-of-the-art architectural synthesis flows adopt the distributed register (DR) architecture to cope with this increasing latency by allowing multicycle communication. To further improve the accuracy of delay estimation, the regular distributed register (RDR) with highly regular structure was later proposed. However, the RDR architecture suffers from extra overhead on interconnect resources, especially on the costly global wires. In this thesis, we formulate channel and register allocation on the RDR-global resource sharing (RDR-GRS) architecture as data transfer scheduling for minimizing global wires. Contrary to the traditional congestion-avoiding routers, we present a concentration-oriented path scheduler to concentrate the data transfers such that the probability of wire sharing increases. Also, a channel-based time scheduler is developed to resolve contentions for wires in the channel by exploiting the slacks of transfers. Experimental results show that, for large-scale designs, our algorithm can averagely reduce 81.7% wires and 8.2% registers compared to the previous work on the RDR architecture.

# 誌 謝

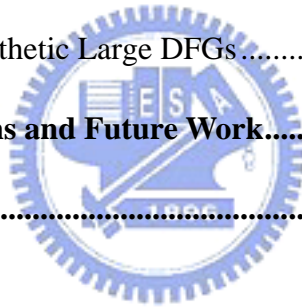
感謝我的指導教授黃俊達老師，在兩年碩士生涯裡，引領我探索學術研究的領域。老師總是耐心的指正我的錯誤，讓我能不斷的進步，培養建立正確的研究方向及態度。感謝我的口試委員－清大資工林永隆教授、交大電子周景揚教授及陳宏明教授－在百忙之中蒞臨指導，你們寶貴的意見是我在研究上精進的最好動力與依據。感謝這二年來所有和我分享快樂時光的朋友，謝謝你們也在我徬徨的時候鼓勵、陪伴我。不論之後你們選擇了什麼樣的道路，希望你們每一個人都能堅定的邁向人生的下一階段，永遠保持樂觀。最後，感謝我的父母親，永遠支持我的決定、對我充滿信心，讓我能保有勇氣追求自己的夢想。我會繼續朝著這樣的方向前進，相信你們會為此感到開心和驕傲。



# Contents

摘 要 .....	i	
Abstract .....	ii	
誌 謝 .....	iii	
Contents.....	iv	
List of Figures .....	vi	
List of Tables .....	viii	
<b>Chapter 1</b>	<b>Introduction .....</b>	<b>1</b>
1.1	Centralized Register (CR) Architecture and Synthesis .....	1
1.2	Distributed Register (DR)-Based Architectures .....	3
1.3	Related Works .....	7
1.4	Thesis Organization.....	8
<b>Chapter 2</b>	<b>Channel and Register Allocation.....</b>	<b>10</b>
2.1	Channel and Register Allocation Problem .....	10
2.2	Regular Distributed Register-Based Architectures.....	12
2.2.1	RDR .....	12
2.2.2	RDR-pipe.....	13
2.2.3	RDR-GRS .....	14
2.3	Motivational Example .....	15
<b>Chapter 3</b>	<b>Proposed Algorithm.....</b>	<b>18</b>
3.1	Problem Formulation.....	18
3.1.1	Architecture Specification .....	18
3.1.2	Data Transfer Set.....	19
3.1.3	Transfer Scheduling.....	21

3.2	How to Increase the Probability of Wire Sharing.....	23
3.3	Concentration Phase .....	26
3.4	Iterative Rescheduling Scheme .....	28
3.4.1	Channel-Based Time Scheduler .....	31
3.4.2	Reroute Procedure .....	35
3.4.3	Channel Control Mechanism.....	36
3.5	Pseudo Code of Our Algorithm.....	38
<b>Chapter 4</b>	<b>Experimental Results .....</b>	<b>39</b>
4.1	Experiment Setup .....	39
4.1.1	DFG Generation .....	39
4.1.2	Scheduled and Bound DFG .....	40
4.2	Part I: MediaBench.....	42
4.3	Part II: Synthetic Large DFGs.....	45
<b>Chapter 5</b>	<b>Conclusions and Future Work.....</b>	<b>50</b>
<b>References.....</b>		<b>52</b>

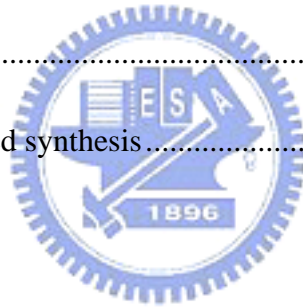


# List of Figures

Figure 1-1. Conventional architectural synthesis flow.....	2
Figure 1-2. Centralized register based architecture.....	3
Figure 1-3. Distributed register based architecture. ....	4
Figure 1-4. 2x3 cluster array RDR architecture. ....	5
Figure 2-1. (a) A scheduled and bound DFG of DCT; (b) placed FUs on a 3x3 cluster array RDR-based architecture. ....	11
Figure 2-2. Global communication in the RDR architecture.....	13
Figure 2-3. Global communication in the RDR-pipe architecture. ....	14
Figure 2-4. Global communication in the RDR-GRS architecture. ....	15
Figure 2-5. The demand of interconnect resources implemented with RDR/MCAS.....	16
Figure 2-6. The demand of interconnect resources implemented with RDR-pipe/MCAS-pipe.....	17
Figure 2-7. The demand of interconnect resources implemented with RDR-GRS/ILP..	17
Figure 3-1. Specification of a 2x3 RDR-GRS architecture. ....	19
Figure 3-2. A data transfer and the associated parameters. ....	20
Figure 3-3. Transfer path scheduling.....	22
Figure 3-4. Transfer time scheduling. ....	22
Figure 3-5. (a) Data transfers routed in a distributing way and (b) a concentrating way; (c) the sources and destinations of the data transfers. ....	25



Figure 3-6. (a) The adding sharing scores according to $tr_1$ ; (b) the final sharing scores.	26
Figure 3-7. Major components of iterative rescheduling scheme. ....	28
Figure 3-8. Channel resolving order of $tr_1$ .....	29
Figure 3-9. Partial pseudo code of iterative rescheduling scheme. ....	30
Figure 3-10. (a) Tight timing schedules of $tr_1$ ; (b) before and after timing adjustment.	32
Figure 3-11. Feasible set of cycles for $tr_3$ to use $ch_{2,5}$ . ....	33
Figure 3-12. Feasible cycles of data transfers at $ch_{2,5}$ . ....	34
Figure 3-13. (a) The updated sharing score; (b) before and (c) after the rerouting.....	35
Figure 3-14. Pseudo code of our transfer scheduling algorithm. ....	38
Figure 4-1. DFG generation .....	40
Figure 4-2. Flow of simplified synthesis.....	41



# List of Tables

Table 1. Scheduled and Bound DFGs on 3x3 Architecture..... 42

Table 2. Interconnect Resources Demand of DFGs ..... 44

Table 3. Interconnect Resources Demand of DFGs Normalized to RDR ..... 44

Table 4. Scheduled and Bound Synthetic DFGs on 10x10 Architecture..... 46

Table 5. Interconnect Resources Demand of Synthetic DFGs ..... 47

Table 6. Interconnect Resources Demand of Synthetic DFGs Normalized to RDR ..... 48

Table 7. Interconnect Resources Demand of Synthetic DFGs Normalized to RAND .... 49



# Chapter 1

## Introduction

With the scale evolution of fabrication technology, the delay of a long wire is no longer negligible due to large resistance-capacitance delay, coupling effect, inductance, and high operating frequency, etc. [1]. Full chip communication in a single cycle becomes harder to achieve as the gap between wire and gate performance continues to grow. In this chapter, we focus on the impacts of long global interconnects on architectural synthesis. The multicycle communication architectures based on distributed register files are introduced. Moreover, alternative design methodologies and methods proposed to tackle this problem at different levels are discussed. Finally, the organization of this thesis is given in the last section.



### 1.1 Centralized Register (CR) Architecture and Synthesis

Architectural synthesis is composed of a series of tasks to synthesize the given design in the behavior level to the register transfer level (RTL). The actual synthesis flow varies with the target architecture and the design style. Figure 1-1 gives an example of a simple conventional architectural synthesis flow. First, a design written in high-level language such as C language is compiled to generate an internal representation of the design, usually a control data flow graph (CDFG), for the synthesis system. The operations in the CDFGs are then assigned to the feasible functional units (FU) and timing slots. Next, the synthesis system generates the interconnection and associated control signals of data transfers to detail the cycle behavior of the circuits.

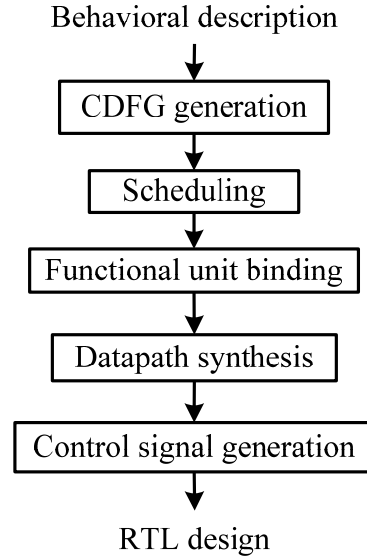


Figure 1-1. Conventional architectural synthesis flow.

The target architecture of conventional architectural synthesis is the CR architecture, as shown in Figure 1-2. All computational units, i.e., the FUs, fetch their operands from and store the computational results into this centralized register file through global interconnects. It is generally assumed that the FUs are able to access any registers within a clock cycle. Therefore, the system cycle time is mainly decided by the maximum sum of the execution time of the FUs and the associated global interconnect delays. However, though the cycle time shrinks as the chip manufacturing process advances, the wire delay does not scale well with the feature size. Hence, global wire delays gradually dominate and drastically lengthen the cycle time. If the synthesis system at this level simply overlooks the wire delays as before, serious impacts due to long interconnects may be exposed after physical floorplanning and potentially lead to worse performance because the timing closure is hard to achieve. To overcome this problem, various state-of-the-art synthesis flows perform preliminary floorplanning to obtain more accurate estimations of interconnect delays so that better synthesis outcomes can be expected [2][3][4]. Nonetheless, these synthesis flows can only limitedly alleviate the problem but is incapable

of stopping the trend of global wire delay's drastically lengthening cycle time.

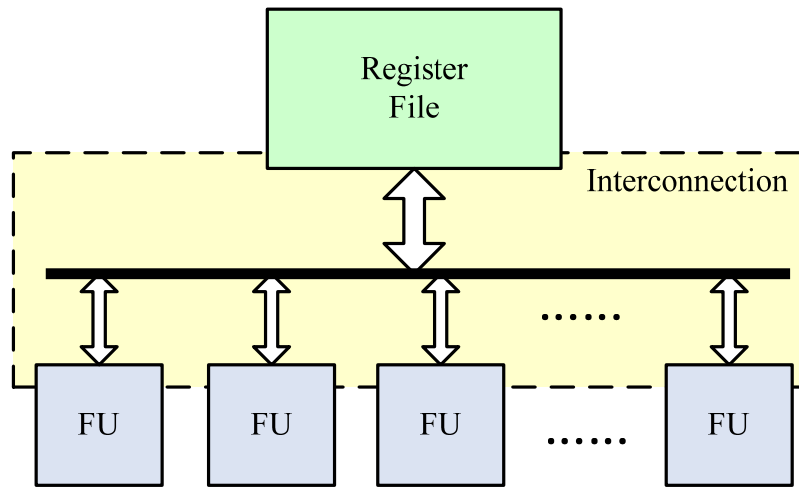


Figure 1-2. Centralized register based architecture.

## 1.2 Distributed Register (DR)-Based Architectures

The distributed register (DR) architecture is proposed to deal with the long interconnects in a completely different way than the previous CR architecture – allowing distributed local register files [5][6]. The distributed architecture partitions a whole chip into several clusters. Each cluster in the DR architecture contains functional units and local registers; a functional unit can only directly access the registers within the same cluster where it resides. The intra-cluster data transfers should be completed within a cycle, while the global data transfers, i.e., the inter-cluster transfers which go through global interconnects, are allowed to take multiple cycles. As a result, only intra-cluster wire delays contribute to the cycle time. A communication model of DR architecture is shown in Figure 1-3. The new paradigm not only prevents the long wire delays from increasing the cycle time but also enables parallel computation and communication. Instead of idling, the functional units can compute the global data transfer as long as the required operands are ready.

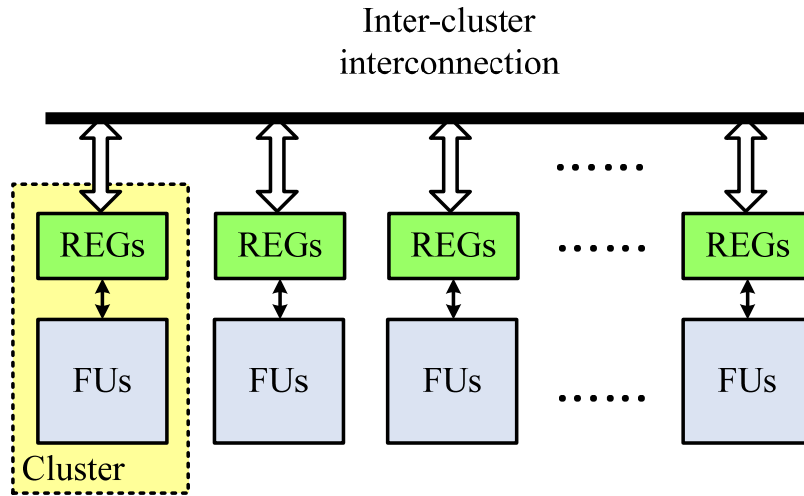


Figure 1-3. Distributed register based architecture.

Even though the DR architecture can keep the cycle time significantly smaller than the CR architecture when a large long wire delay is present, it may take more cycles to complete a data transfer due to newly introduced multicycle communication. Hence, the global interconnect delay can still be an overhead on the system performance in terms of the system latency. The system latency is a multiple of the cycle time and the total cycle count needed to complete all operations. Good scheduling, functional unit binding and placement are all indispensable to reduce the system latency. However, it is difficult to consider global wire delays when performing scheduling and binding since we do not know the actual delay until the final placement. Therefore, [5] performs binding then places the functional units driven by inter-clock slack. With the initial placement, [6] further applies the performance-driven scheduling with interconnect delay. Although these two works try to reduce the system latency in different aspects, they both rely on the interconnect delay information extracted from the preliminary placement. The rough result of coarse placement is usually far from the actual implementation obtained after floorplanning and routing. The inaccurate estimation of interconnect delay therefore

significantly limits the final quality of synthesis.

To eliminate inaccuracy of wire delay estimation, [7] proposes the regular distributed register architecture and the corresponding synthesis methodology named the architecture-level synthesis for multicycle communication (MCAS). The RDR architecture divides the entire chip into a 2-dimensional array of clusters (islands). Figure 1-4 shows the RDR architecture with 2x3 cluster array. Each cluster contains functional units, a local register file and a finite state machine for control signals. Due to the highly regular layout, the interconnect delay between each cluster pair can be correctly calculated and recorded into a table. With this look-up delay table, MCAS can perform resource allocation and binding, simulated annealing (SA)-based coarse placement with scheduling-based timing analysis followed by the post-layout scheduling with rebinding in a more accurate fashion. The regularity allows the distributed register architecture to beat the centralized architecture not only in cycle time but also in overall system performance.

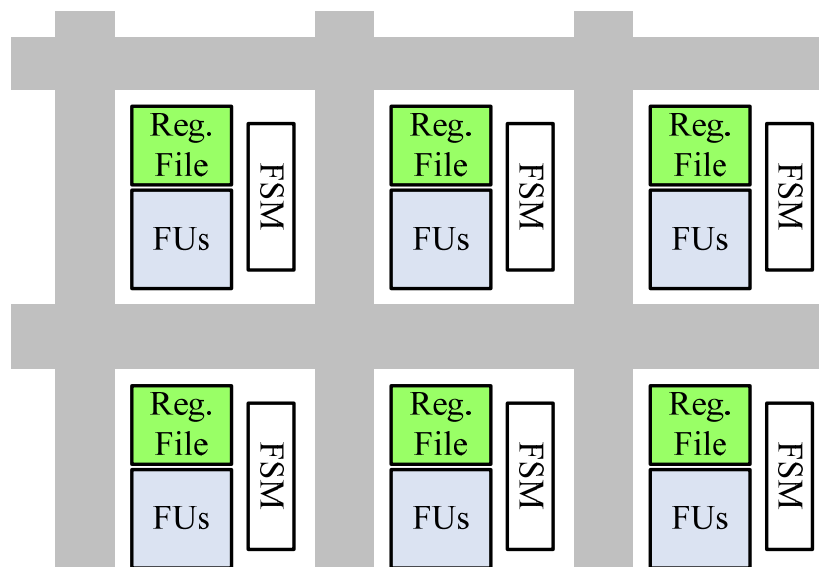


Figure 1-4. 2x3 cluster array RDR architecture.

However, DR-based architectures usually need more registers and wires compared to CR architectures. One of the reasons is that the same data may be demanded by several clusters concurrently. In the CR architecture, all the data is stored in the centralized register file, so no extra register is needed to store repeated data. On the other hand, in the DR architecture, each cluster demanding the data should store the data in its own register file such that the functional units in that cluster can fetch. Besides, by carefully arranging the data to the registers, the centralized registers can usually be more efficiently utilized than the distributed ones. Furthermore, the DR-based architectures either require dedicated long global interconnects to hold the data for multiple cycles until the transfer is over, or require extra registers to pipeline the multicycle interconnection. For the former, the number of required global wires is lower-bounded by the maximum number of concurrent data transfers. For the latter, a massive amount of pipeline registers could be introduced but the reduction of global wire is still uncertain. As a result, the DR-based architecture usually suffers from a significant overhead on interconnect resources, especially on the expensive global wires. It is reported that the DR architecture needs on average 100% more registers and 46% more global wires than the CR architecture [7]. Therefore, the issue of minimizing highly-demanded interconnect resources in the DR architecture should be seriously addressed.

One of the DR-based architectures, the RDR-Global Resource Sharing architecture (RDR-GRS) inherits all the aforementioned advantages of the RDR architecture and adopts a more sophisticated interconnect model to allow global sharing of interconnect resources (both global wires and registers) [8]. Due to the high flexibility that the RDR-GRS architecture provides, it serves as an excellent architecture for exploring channel and register allocation problems. In this thesis, we present a channel and register allocation algorithm to schedule the data transfers such that the costly global wires can be minimized through channel sharing.



### 1.3 Related Works

Before ending this chapter, we discuss several research works proposed to tackle the long interconnects at different levels in the design process or using different design methodologies.

There has been plenty of research works addressing the long interconnect delay problem in the lower level of circuit design. At the layout level, optimal buffer insertion and wire sizing is adopted to help reduce wire delays [9]. Retiming with physical floorplanning or placement is performed at the logic level to minimize clock period [10][11][12][13]. At the register-transfer level, long wires are pipelined to boost up the potential clock frequency [14]. Though applying these techniques helps better the system performance significantly, they are subject to various limitations, either inherently or due to the lack of automated tool supports. As the gap between interconnect and gate delay keeps growing, seeking for a systematical methodology at a higher level of abstraction is definitely a must.

One of the alternative design methodologies is asynchronous design. By using event-triggering instead of periodic clock-triggering, asynchronous design is able to function correctly under arbitrary wire delay. The robustness, immunity of clock skew, and better technology migration potential make asynchronous design suitable to be used for large systems. However, it is still far from mature with no performance guarantee and barely any applicable automatic tools yet. Globally-asynchronous locally-synchronous systems (GALS) [15] are targeted to preserve the advantages from both asynchronous and synchronous design. A GALS design partitions the chip into several blocks; each block employs its own local clock, while the inter-block data exchanges follow a full handshake protocol. Nevertheless, adaptation of the interfaces of the synchronous blocks to self-timed environment needs asynchronous wrappers which result in extra overhead on both area and

performance.

Synchronous design methodology is still most widely adopted by far, and is supported by numerous computer-aided tools. Authors in [16][17] propose the latency insensitive design (LID) with correct-by-construction methodology to provide multicycle communication while still conforming to conventional design toolsets. Given a synchronous design consisting of several modules, LID encapsulates each module by a shell, a wrapper implementing the latency insensitive protocol, allowing the module to stall for multiple cycles without affecting its internal states. After conventional placement, routing, and post-layout optimization which inserts relay stations as pipeline registers in the long channels, LID generates a functionally equivalent synchronous implementation that can tolerate arbitrary latency. Nonetheless, relay station insertion could severely worsen the system throughput which is determined by the critical loop (containing one or more relay stations). Special care should be taken to derive a good performance-driven synthesis flow.

Among the multicycle communication architectures, the RDR-based architecture is selected as the target architecture due to its accurate interconnect delay calculation, systematical exploration of cycle time and latency tradeoff, and the complete layout-driven synthesis flows on top of it. Though the extra wiring overhead of the RDR architecture may be serious issues as the design size grows, the concept of global sharing in RDR-GRS offers a chance for us to minimize the interconnect resources.

## 1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 introduces the channel and register problem, and gives an analysis of a communication model of several RDR-based architectures with examples to demonstrate the potential of interconnect resource

minimization in the RDR-GRS architecture. Chapter 3 defines the transfer scheduling problem and presents our channel and register allocation algorithm in detail. In Chapter 4, the experimental results compared to previous approaches are shown. Chapter 5 concludes this thesis and points out possible future work.



# Chapter 2

## Channel and Register Allocation

In this chapter, we introduce the channel and register allocation problem in RDR-based architectures. We analyze RDR, RDR-pipe [18], RDR-GRS and their associated resource allocation algorithms respectively. A motivational example comparing the demand of interconnect resources in the three RDR-based architectures is given at the end of this chapter.

### 2.1 Channel and Register Allocation Problem

The architectural synthesis for RDR-based architecture includes operation scheduling, FU binding, and FU-cluster mapping. After completing these tasks, the global communication demands among clusters in the target architecture become obvious, but the approach in fulfilling those communication demands varies in different architectures. Therefore, channel and register allocation is indeed an architecture-specific problem that decides how interconnect resources are allocated to data transfers.

The initial inputs are scheduled and bound data flow graphs and the placed functional units with topology information of the target architecture. Figure 2-1 shows an example of a data flow graph of discrete cosine transform (DCT) and the corresponding FU placement on the 3x3 cluster array RDR architecture. There are four available functional units in this example; two of them are arithmetic logic units (ALU) and the other two are multipliers. The horizontal lines in Figure 2-1(a) divides the total latency required to complete the operations in this data flow graph into time slots (cycles). Each operation (node) of the data

flow graphs is scheduled to be executed at a timing slot by one of the functional units, ALU0, ALU1, MUL0 and MUL1. The edges between the nodes represent the data dependencies of the operations. An operation is ready to be executed only when all its required data (operands) arrive at the functional unit. For example, ALU1 cannot execute operation 11 until it receives the results of operation 7 and 9, which should be issued from MUL1. Figure 2-1(b) displays the exact placement of the functional units on the target 3x3 RDR-based architecture. Every functional unit is placed in a cluster where a register station (RS) resides.

A channel and register allocation algorithm takes the above inputs and decides how the clusters exchange data, i.e., what channels and registers are used during data transfers. The objective of the algorithm is to meet all pre-scheduled data arrival times while at the same time attempting to minimize the interconnect resources required for data transfers.

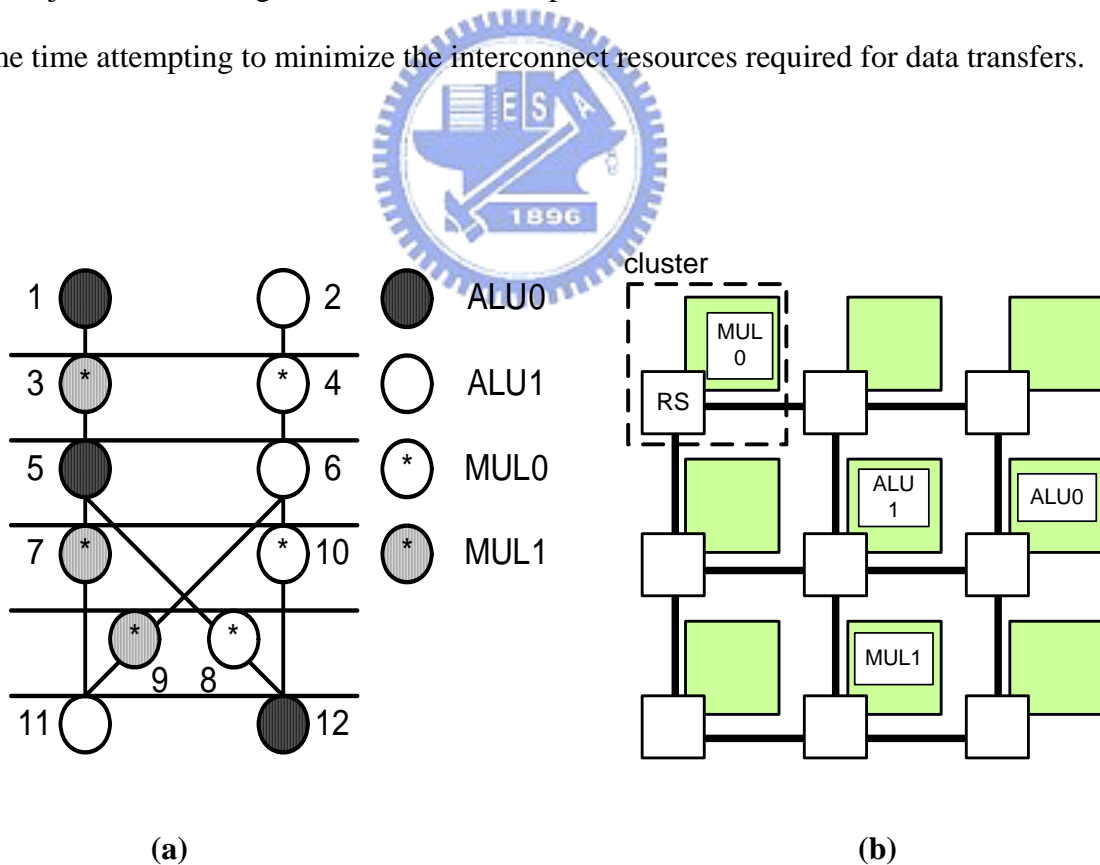


Figure 2-1. (a) A scheduled and bound DFG of DCT; (b) placed FUs on a 3x3 cluster array RDR-based architecture.

## 2.2 Regular Distributed Register-Based Architectures

RDR-pipe and RDR-GRS are both evolved from RDR to reduce the interconnect resources. In this section, we compare the global communication models and the associated interconnect resource allocation algorithms for RDR, RDR-pipe and RDR-GRS.

### 2.2.1 RDR

In [7], RDR/MCAS divides registers in the local register station into  $n$  banks where the register in bank  $n$  is designated for data transfers which take  $n$  cycles to complete. Every time a data transfer between a pair of clusters is initiated, a dedicated wire connecting the two clusters is assigned. This dedicated wire holds the data for as long as it needs to transfer and therefore is exclusively occupied by the data transfer throughout the transmission, which requires one or multiple cycles. Figure 2-2 shows a simple example of global communication in RDR. The left side of the figure is the scheduled, bound data flow graph. The nodes (big circles) represent the operations; the small circles on the horizontal lines indicate which registers are used to store the data at that cycle. On the right side of the figure is the RDR architecture with data transfers and their transferring paths. The small circles in the register stations represent the registers. The thick lines are the paths that the data transfers take. Instead of showing the FUs as Figure 2-1, the operations are directly marked on the corresponding clusters for clarity. For instance, the resulting data of operation  $op1$  generated and stored in register 1 in cluster A is then sent to register 4 in cluster C to serve as an operand for  $op4$ .

As mentioned before, the number of required wires (and register pairs) in RDR/MCAS is lower-bounded by the number of maximum possible concurrent data transfers at a cycle. Because global wire is an expensive resource, the luxurious using ways as in dedicated point-to-point interconnects in RDR/MCAS should at best be avoided.

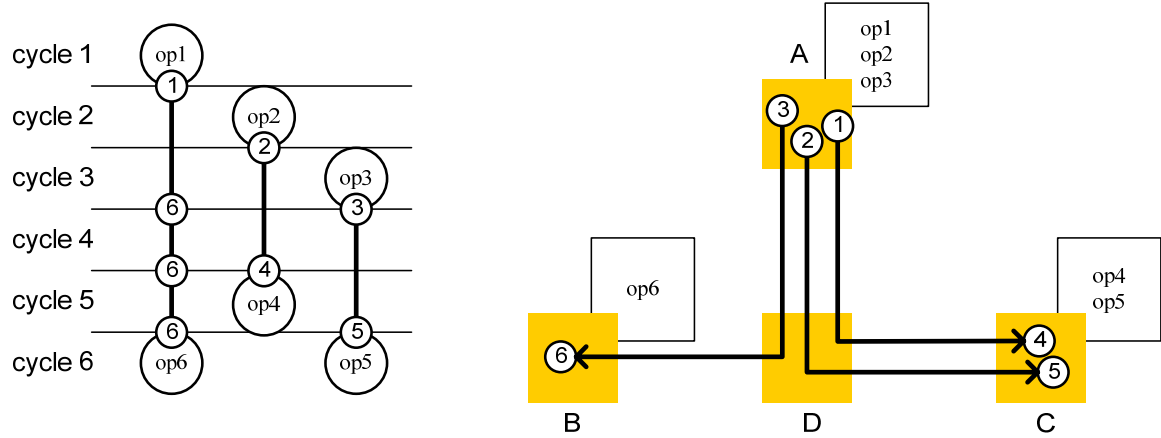


Figure 2-2. Global communication in the RDR architecture.

### 2.2.2 RDR-pipe

An extension named RDR-pipe/MCAS-pipe is proposed to solve the heavy overhead of global wires in the RDR architecture [18]. This extended architecture chops a long wire into segments by inserting extra pipeline registers. Hence a data transfer no longer occupies the whole long wire at a cycle but uses only a segment instead. In other words, at a single cycle, a long dedicated wire between two clusters can be shared by several data transfers as long as these transfers do not use the same segment. In addition, the data transfer scheduling algorithm in MCAS-pipe exploits the possible slacks between the actual transfer latency and the arrival-to-deadline interval to reduce the required registers and wires.

Figure 2-3 shows an example of global communication in RDR-pipe. In this example, two data need to be transferred from cluster A to cluster C and the remaining data from cluster A to cluster B. Different from the RDR architecture, RDR-pipe allows two data transfers with the identical source-destination pair to share the same wires by inserting pipeline registers as intermediate stops.

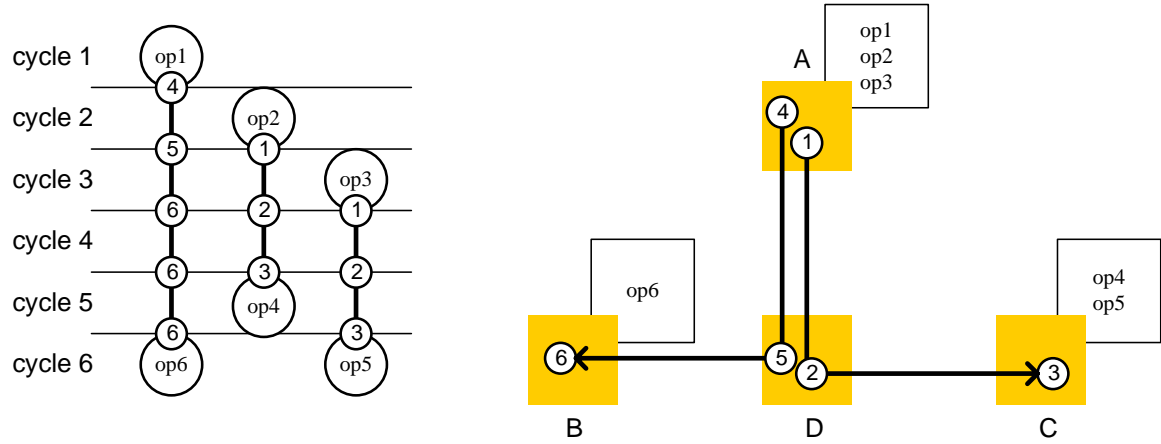


Figure 2-3. Global communication in the RDR-pipe architecture.

### 2.2.3 RDR-GRS

Although the RDR-pipe architecture allows limited sharing of wires, it introduces extra overhead on registers. To further reduce the required interconnect resources, the RDR-GRS architecture enables global sharing of all wires and pipeline registers among all data transfer paths.

The pipeline registers in RDR-pipe just simply forward the current data every cycle, but in the RDR-GRS architecture, all registers residing in the register stations can either function as a normal pipeline register or store the operands for computation of local functional units in the same cluster. Moreover, a register can be pre-scheduled (i.e., controlled by a finite state machine) to send the current data it holds to any register in the neighboring clusters through channels. Thus global wires and registers can both be shared by a set of data transfers with overlapping transfer paths as long as there is no conflict in time. An example of global communication in RDR-GRS is shown in Figure 2-4. All three data transfers in this example go through the channel between cluster A and cluster D so that they can share a single wire in this part of their transfer. The wire between clusters D



and C can also be shared by two data transfers as in the RDR-pipe architecture. Besides, the three data transfers share the same register in cluster A and in cluster D. In such a way, the sharing is longer limited to data transfers with identical source-destination cluster pairs, thus the interconnect resources can potentially be further reduced. In order to fully exploit the global sharing of interconnect resources in RDR-GRS, [8] proposes an Integer Linear Programming (ILP) formulation such that the channel and register allocation problem can be optimally solved.

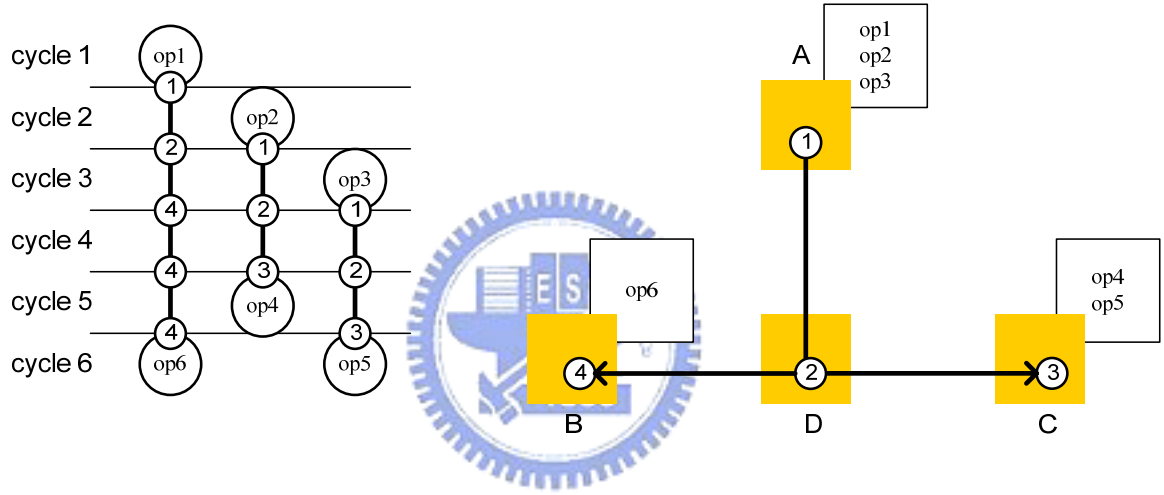


Figure 2-4. Global communication in the RDR-GRS architecture.

## 2.3 Motivational Example

Figure 2-5, 2-6 and 2-7 show an example of the channel and register allocation results on RDR/MCAS, RDR-pipe/MCAS-pipe and RDR-GRS/ILP, respectively. On the left hand side of the figures are the scheduled and bounded data flow graphs and on the right hand side are the architectures with placed operations in the clusters. To fairly compare the three architectures, we define the length of a wire as the distance of two neighboring clusters. As the figure shows, RDR/MCAS needs 12 wires and 10 registers to satisfy all data transfers, while RDR-pipe/MCAS-pipe needs 7 wires and 12 registers. RDR-pipe saves 5 wire

segments compared to RDR at the cost of adding two pipeline registers. It is a reasonable register-wire tradeoff since RDR-pipe enables local sharing of wires. RDR-GRS/ILP, however, needs only 4 wires and 7 registers. By supporting global sharing, RDR-GRS is able to save registers and wires simultaneously.

This example clearly demonstrates the potential of reducing interconnect resources in the RDR-GRS architecture. Nevertheless, though ILP can solve the allocation optimally in terms of the number of wires and that of registers, it is generally known to suffer from long runtime. Thus the ILP approach is not suitable to be applied on practical problems. In addition, since the RDR architecture is proposed for multicycle communication in the near future, it is important that the synthesis flow can deal with large-scale problems. Therefore, developing a good and efficient channel and register allocation algorithm to reduce demanded interconnect resources is necessary.

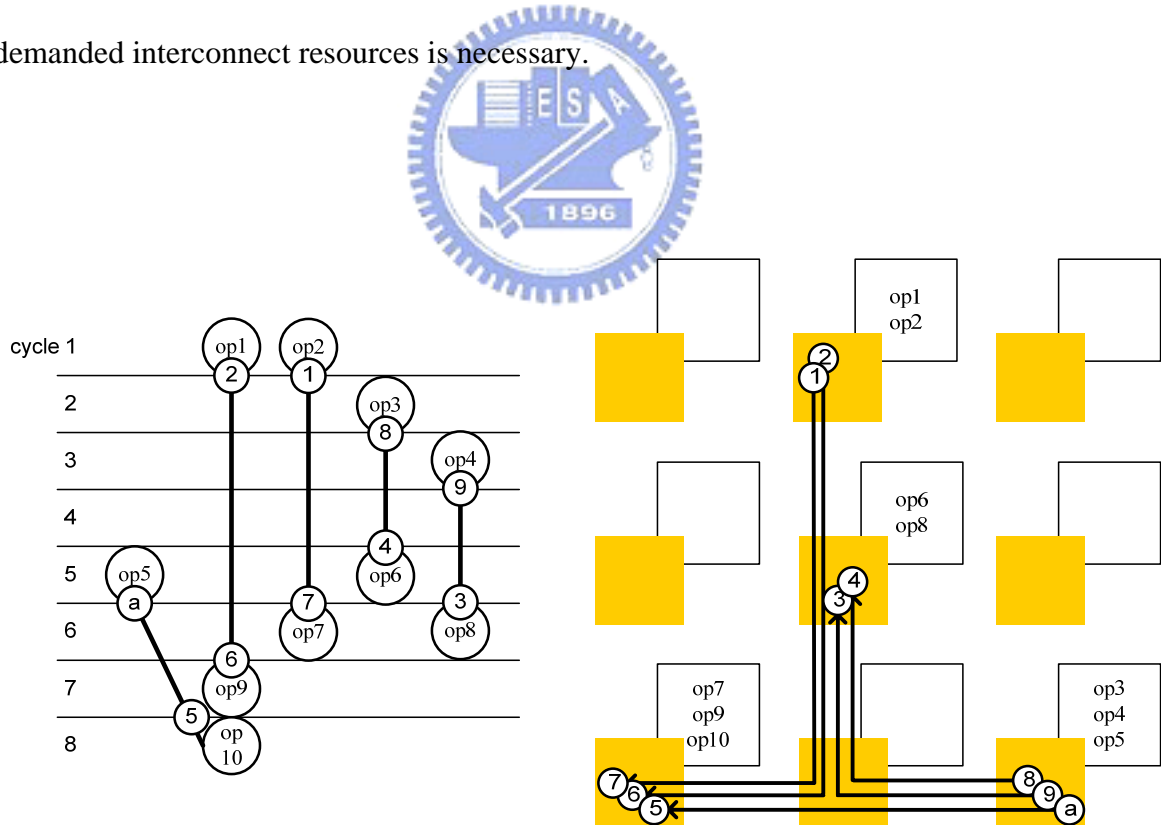


Figure 2-5. The demand of interconnect resources on RDR/MCAS.

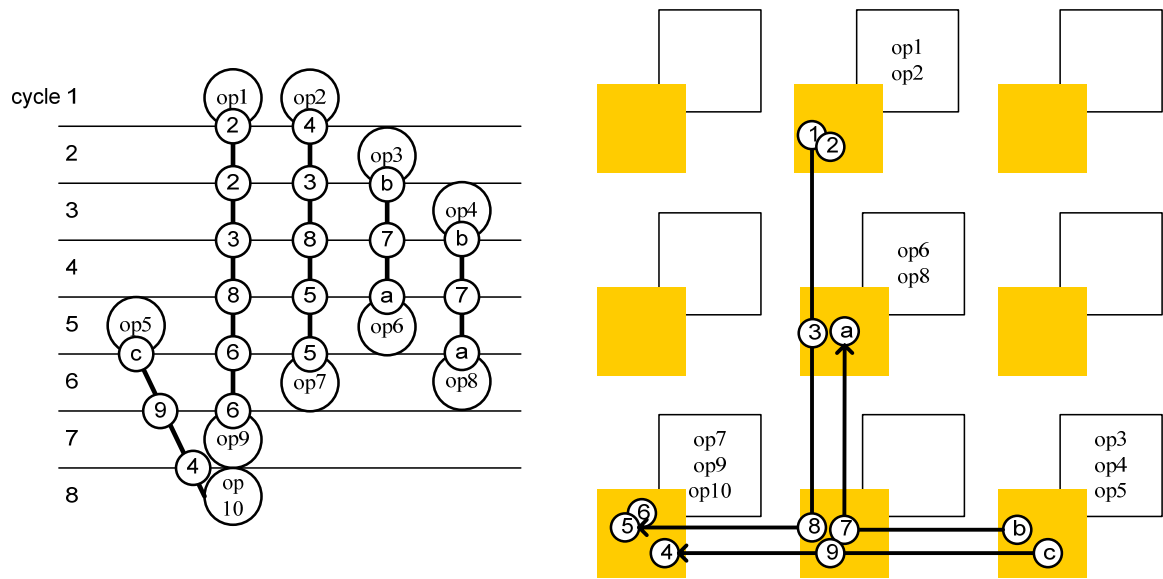


Figure 2-6. The demand of interconnect resources on RDR-pipe/MCAS-pipe.

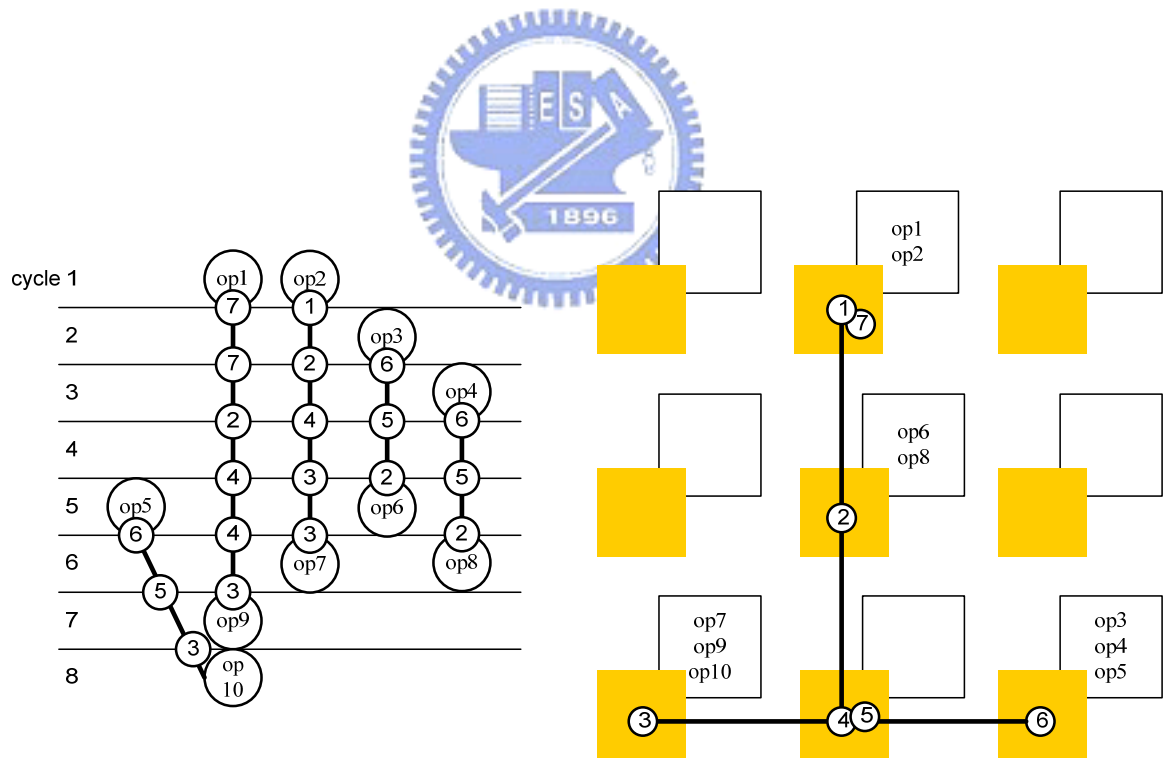


Figure 2-7. The demand of interconnect resources on RDR-GRS/ILP.

# Chapter 3

## Proposed Algorithm

In this chapter, we formulate the channel and register allocation problem as a transfer scheduling problem. Next, we present the central ideas of our algorithm and then the two major parts of it, the concentration phase and the iterative rescheduling scheme. The pseudo code of our algorithm is given at the end of the chapter.

### 3.1 Problem Formulation

Before presenting the transfer scheduling problem, we define two required input data, the architecture specification and the data transfers set, which is extracted from the scheduled, bound DFG and the placed FU with topology information

#### 3.1.1 Architecture Specification

The target architecture is the RDR-GRS architecture with an  $M \times N$  cluster array. The architecture contains a set of clusters  $c_1$  to  $c_{MN}$ , a set of register stations  $rs_1$  to  $rs_{MN}$  and a set of uni-directional channels. Figure 3-1 shows the  $2 \times 3$  cluster array architecture. The neighborhood of a cluster is defined as the four clusters above, below, left and right, respectively. Similarly, the neighborhood of a register station is defined as the four register stations in the neighborhood of the current cluster. The uni-directional channels, being composed of wires, connect neighboring register stations. A wire can transfer data from a register to its neighboring register station within a cycle. Hence a global data transfer is

composed of a series of data transfers between two neighboring register stations through the wires in the channels. Also, as stated before, the registers residing in a register station can serve either as an intermediate stop for global data transfer or as an operand register for computation of local FUs.

Note that though the target architecture contains uni-directional channels, our algorithm can be applied on architectures with uni-directional or bi-directional channels. For simplicity, the examples/figures using in the remainder of the chapter (starting from Section 3.2) uses bi-directional channels.

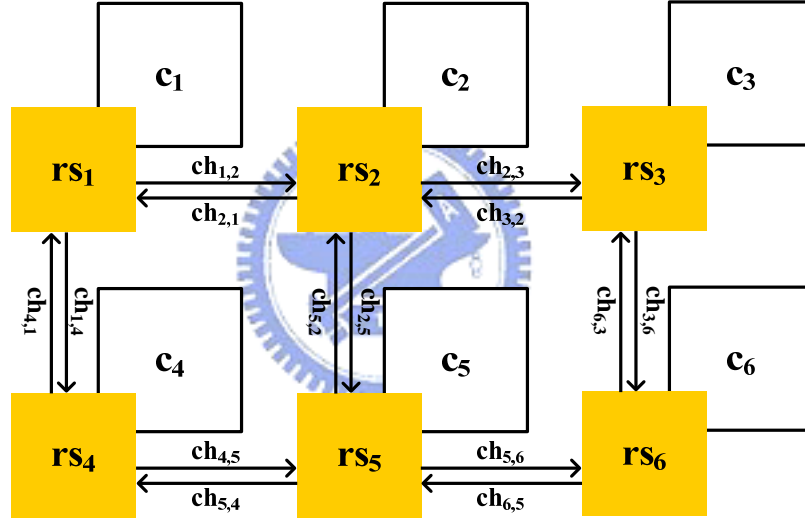


Figure 3-1. Specification of the 2x3 RDR-GRS architecture.

### 3.1.2 Data Transfer Set

For the given scheduled, bound DFG, FU placement, and architecture specification, we define a *data transfer set*  $Tr = \{tr_1, tr_2, \dots\}$ . Below are the parameters of a data transfer  $tr_i$ .

$source(tr_i)$ : at which register station the data to be transfer is generated.

- $dest(tr_i)$ : to which register station the data should be transferred.
- $ready(tr_i)$ : at which cycle the data is generated and so is ready to transfer.
- $deadline(tr_i)$ : before or at which cycle the data should arrive at its destination.
- $short\_dist(tr_i)$ : the number of cycles needed to transfer a data from source ( $tr_i$ ) to  $dest(tr_i)$  if taking the shortest possible path. It is decided by the physical distance of source ( $tr_i$ ) and  $dest(tr_i)$  according to the architecture specification.
- $slack(tr_i)$ : the number of cycles that the data is allowed to be delayed (staying in the register station) if taking the shortest possible path. It is calculated as  $(deadline(tr_i) - ready(tr_i)) - short\_dist(tr_i)$

An example of a data transfer is shown in Figure 3-2. The data of  $tr_1$  is the result of operation 4 and is ready to be transferred at cycle 4 at register station  $rs_5$ . The destination register of  $tr_1$  is  $rs_1$ , and the arriving deadline is cycle 7. From the given architecture specification, the shortest distance of  $rs_1$  and  $rs_5$  is 2, so it takes at least two cycles to complete the data transfer. However, the allowable number of cycles for  $tr_1$  is  $7 - 4 = 3$ . Thus the available slack is 1.

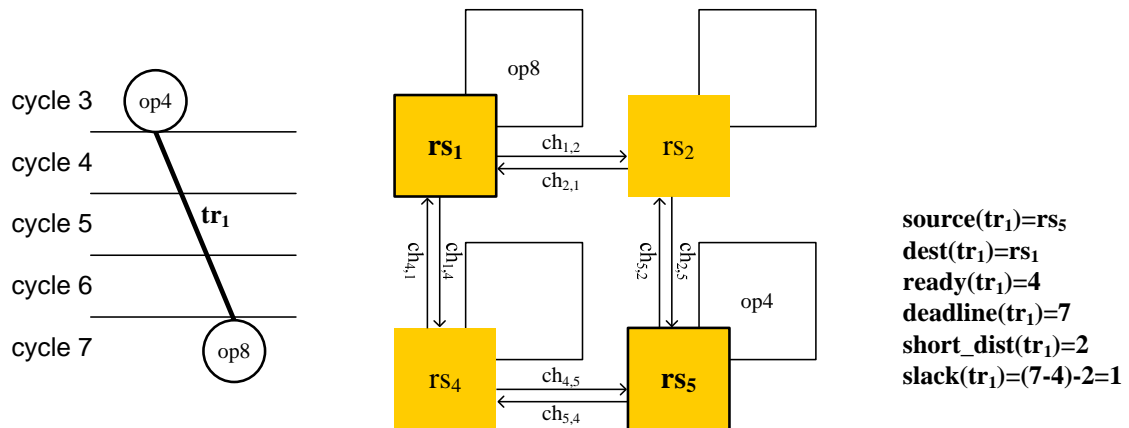


Figure 3-2. A data transfer and the associated parameters.

### 3.1.3 Transfer Scheduling

We formulate the channel and register allocation problem as a transfer scheduling problem that is to find a feasible schedule for all data transfers such that the deadlines of arrivals are met. A transfer scheduling algorithm takes a set of data transfers and the corresponding architecture specification and decides the behavior of every data transfer  $tr_i$  at each cycle during the lifetime of the transfer (from  $ready(tr_i)$  to  $deadline(tr_i)$ ).

Transfer scheduling can be further divided into two phases from the perspective of space and time: *transfer path scheduling* and *transfer time scheduling*. Transfer path scheduling explores possible routing paths for the data transfers on the target architecture. For example, in Figure 3-3, the data transfer  $tr_1$  should be routed from  $rs_5$  to  $rs_1$ . One possible path is to go through channel  $ch_{5,2}$  and  $ch_{2,1}$ , which is shown in the figure by solid lines. The other path uses  $ch_{5,4}$  and  $ch_{4,1}$ , shown in dotted lines. The scheduling algorithm chooses between these two paths for  $tr_1$  according to its objective.

Transfer time scheduling, on the other hand, explores the possible timing schedules along the routing path for the data transfers by utilizing their slacks. In other words, a transfer time scheduling algorithm determines at which cycles the data stay in the register without transferring through channels. Take  $tr_1$  in Figure 3-3 for example, assume that  $tr_1$  is scheduled to take path1 (the solid lines), using  $ch_{5,2}$  and  $ch_{2,1}$ . Then, the timing schedule of  $tr_1$  can be one of the three schedules in Figure 3-4. Each rectangle in the figure represents the status of a data transfer at a cycle: transferring, holding or arrived at destination. Since  $tr_1$  has one slack, we can choose one of cycle 4, 5 and 6 to be a non-transferring cycle at which the register simply holds the data without forwarding. Note that if a data transfer can only be routed in a single path and has no slacks, it has no room for path or time scheduling. This type of data transfer is called *fixed data transfer*. We consider that the fixed data transfers are pre-scheduled in the only way they can be.

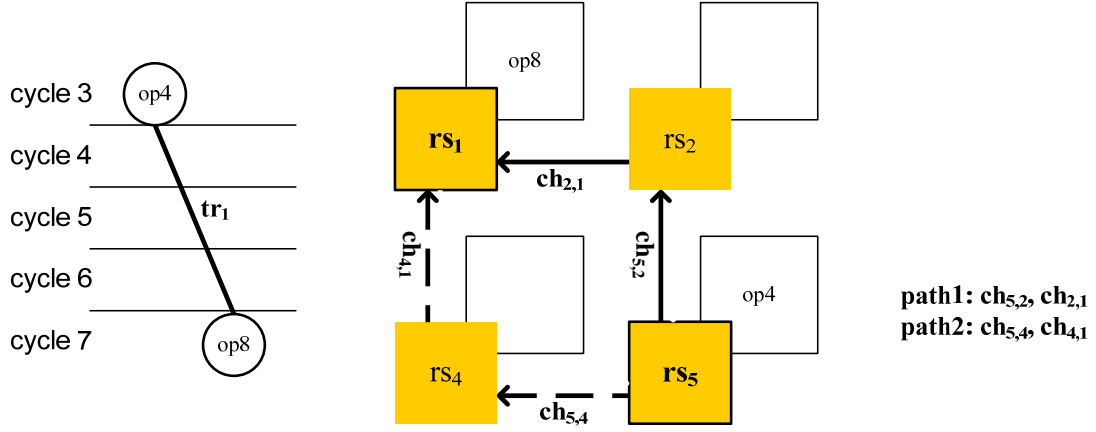


Figure 3-3. Transfer path scheduling.

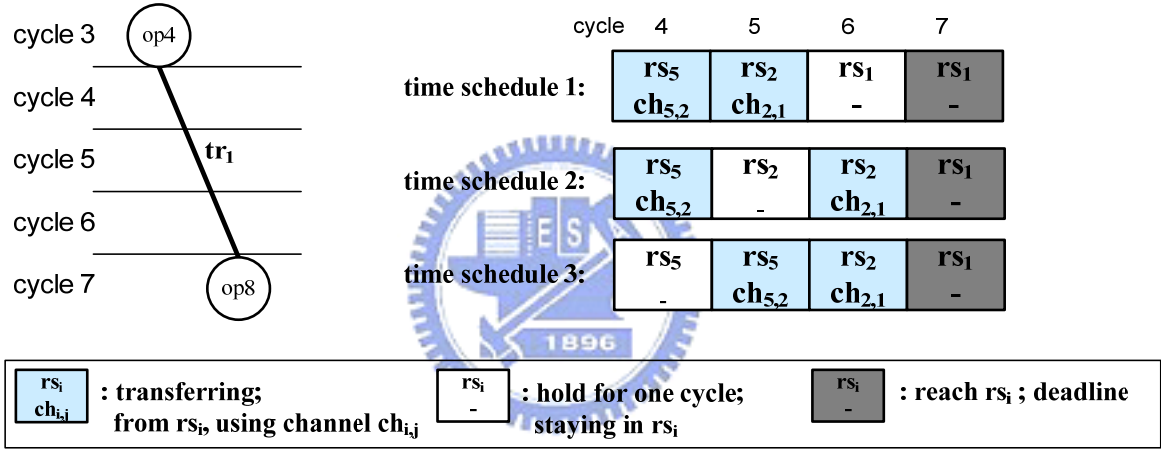


Figure 3-4. Transfer time scheduling.

In this thesis, our transfer scheduling algorithm aims at minimizing the required number of wires by maximizing the sharing of wires among data transfers. We focus on reducing wires instead of registers because global wire is an expensive resource and is much scarcer than registers. This is also why RDR-pipe inserts pipeline registers to save global wires. Though we do not explicitly take the number of registers into consideration in our algorithm, the nature of register sharing in RDR-GRS should be able to lower down the register usage. In fact, experimental results in Chapter 4 show that our algorithm indeed keeps the number of registers roughly the same as that in RDR.



### 3.2 How to Increase the Probability of Wire Sharing

Transfer path scheduling, at first glance, seems very similar to traditional Field Programmable Gate Array (FPGA) routing and grid-based global routing because of the regularity present in the RDR-GRS architecture. However, the new dimension of time introduced in time scheduling increases the complexity of transfer scheduling and makes it much more different than any traditional routing problem. Several questions therefore arise as wire sharing is allowed:

- a.) *Should we perform transfer path scheduling and time scheduling subsequently or simultaneously?*

Though it may be a good option to consider path and time scheduling at the same time for achieving optimality, it consists of high complexity and is also uneasy to start with. A more reasonable way is to first perform path scheduling for a transfer, then decide the timing schedule of it based on the channels it goes through and the status of these channels. Note that the transfer cannot adapt its timing schedule to share wires with other routed transfers until it is routed and the transfers using the same channels with it are clear.

- b.) *Which routing pattern is preferred in path scheduling? Figure 3-5(a) or (b)?*

Figure 3-5(a) and (b) show possible routing results of data transfers in Figure 3-5(c) in distributing and concentrating ways respectively. In traditional routings, one major goal of the global routers is to distribute the transfers to minimize possible congestions which will be exposed after detailed routing. Many congestion estimation algorithms have also been developed to predict congestion earlier in the routing such that it can be avoided. Hence, from the perspective of the traditional routers, Figure 3-5(a) is absolutely a better choice than (b).

However, from the perspective of wire sharing, is (a) still most likely to outperform (b)? Consider that a transfer time scheduler is applied to both figure (a) and (b). First, if all the transfers using the same channel can share a single wire, including the channels with heavy traffic, the number of required wires in (b) is clearly fewer than that in (a). Of course, the chance that any number of data transfers in a channel can share a single wire may not be so high. But even if this is the case, more data transfers using the same channel indicate a higher probability that some of these data transfers can actually share a wire. On the other hand, it is highly doubtful that the router in (a) is clever enough to distribute the transfers such that the few data transfers using same channel just happen to be able to share a wire. Therefore, opposite of the traditional routing problem, concentrating is much more preferable and useful for transfer scheduling in RDR-GRS to increase the probability of wire sharing.

c.) *Which one is more worthwhile of exploring? The timing slacks or the detouring paths?*

The shortest path of a data transfer lies within the rectangle bounded by the source and destination register stations of the transfer, usually known as the bounding box. The detouring path uses the timing slacks so that it stretches outside the bounding box and takes extra cycles to complete the transfer. In traditional routing, detouring is commonly used to prevent the transfer from passing through congested areas. Nevertheless, as stated before, wire sharing favors the concentrating routing scheme so that detouring is not necessarily required. Moreover, with longer paths, the data transfer goes through more channels and thus it may potentially lead to a more required number of wires. On the contrary, when only the shortest paths are allowed, the slacks can be reserved for the transfer time scheduling, which arranges the timing schedules of the

transfer to resolve conflicts and achieve better wire sharing.

Based on the above concepts and ideas, we propose an algorithm with a *concentration-oriented path scheduler* and a *channel-based time scheduler*. The concentration-oriented path scheduler routes one transfer at a time by choosing the shortest path with the highest accumulated *sharing scores* of the channels. The channel-based time scheduler is responsible for resolving the data transfers in a channel and determines how to adjust the timing schedules of the transfers and which transfers to rip up. Our proposed algorithm is partitioned into two parts: 1) the *concentration phase* that calls the path scheduler to route all data transfers; 2) the *iterative rescheduling scheme* that interchangeably invokes the time scheduler and the path scheduler until all transfers are well scheduled both in time and path.

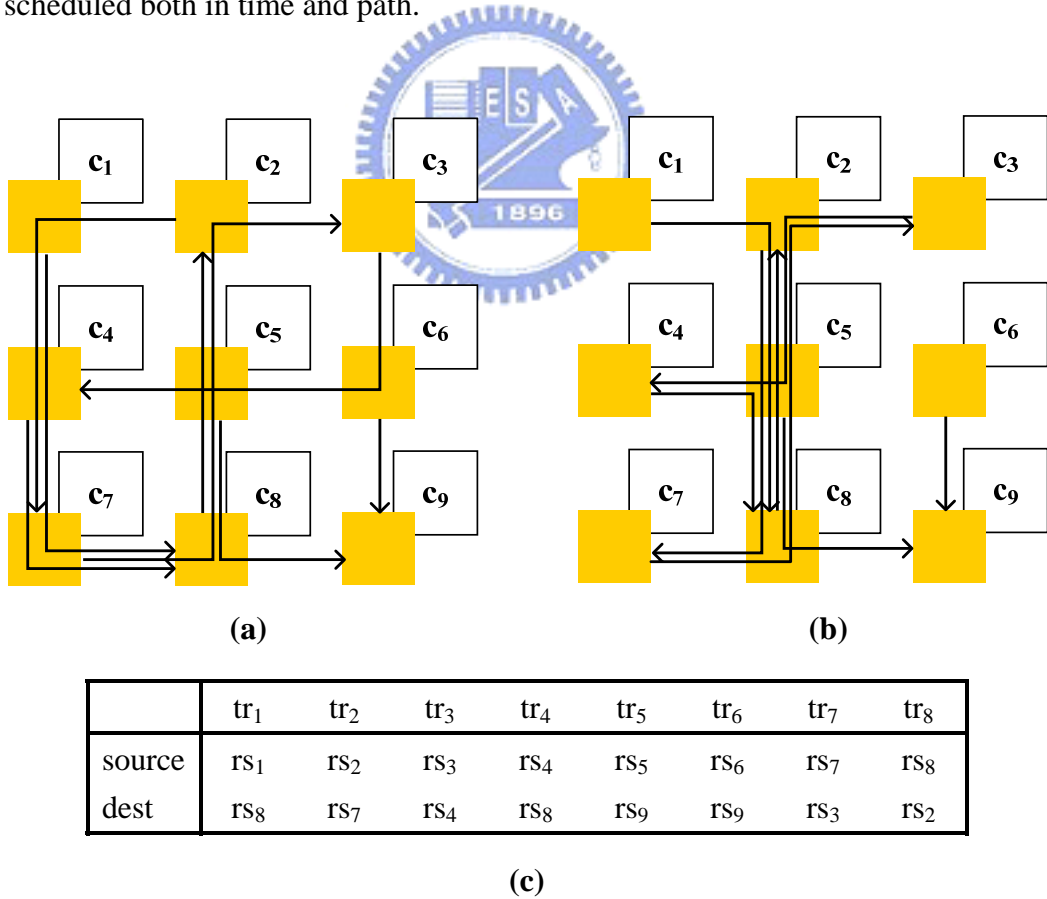


Figure 3-5. (a) Data transfers routed in a distributing way and (b) a concentrating way; (c) the sources and destinations of the data transfers.

### 3.3 Concentration Phase

The objective of the concentration phase is to produce a good initial solution for the subsequent iterative rescheduling scheme. In this phase, the sharing scores of the channels are added and then all data transfers are routed once by the concentration-oriented path scheduler.

In the beginning, every channel is assigned a sharing score that is set to zero. Next, for every data transfer, the sharing scores of the channels within the bounding box of it are incremented by 1. Figure 3-6 (a) shows the adding sharing scores according to transfer  $tr_1$  in Figure 3-5. The shaded area represents the bounding box of  $tr_1$ , and the thick lines with incremented scores are the channels that  $tr_1$  might use. Figure 3-6 (b) shows the final scores after completing adding the sharing scores. The sharing score represents the maximum number of data transfers that can utilize this channel. It also indicates the potential of wire sharing in this channel.

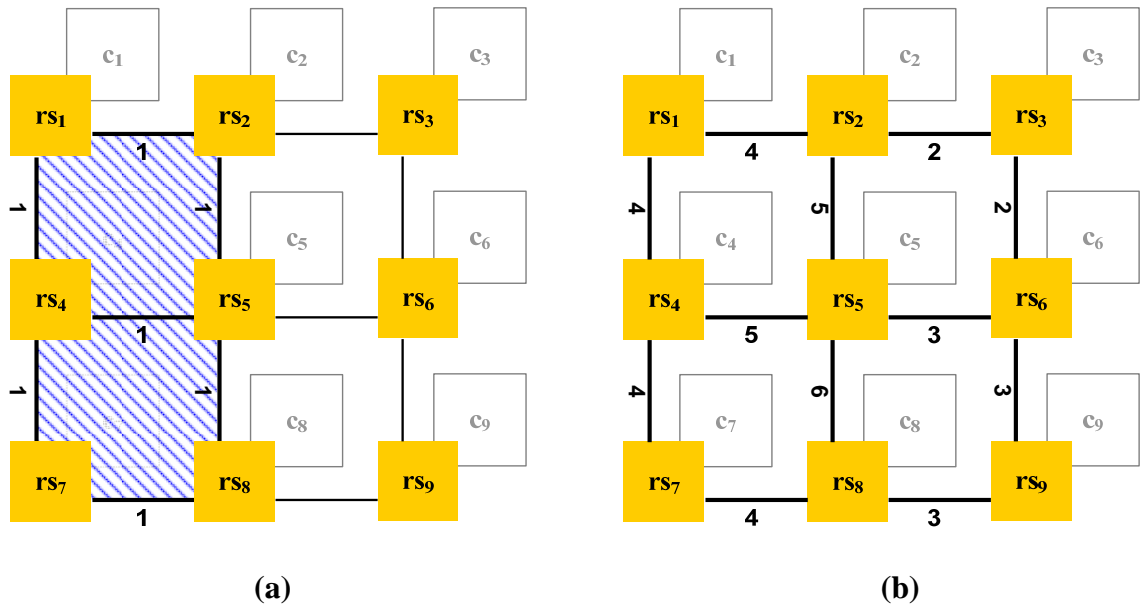


Figure 3-6. (a) The adding sharing scores according to  $tr_1$ ; (b) the final sharing scores.

The core of the concentration-oriented path scheduler is a monotonic router which finds the path of highest accumulated sharing score for a data transfer by dynamic programming. Briefly speaking, the router starts at the source register station and updates the accumulated scores monotonically toward the destination register station. Afterwards, the path with the highest score can be traced back from the destination register station by looking up the direction recorded during the updating process. A detailed algorithm can be found in [19]. In fact, the monotonic router is proposed in [19] as part of a global router to distribute the transfers, which is exactly the opposite of what we intend to do. Therefore, we transform the scores so that the router can match our needs. There are two advantages of a monotonic router: its capability of finding the exact path of highest score and the low time complexity compared to maze routing. Maze routing enumerates all paths and is thus time-consuming. However, by dynamic programming, it takes merely  $O(mn)$  for the path scheduler to route a data transfer with a bounding box of size  $m \times n$ . Other routing methods such as pattern routing (Z-shaped or L shaped) though possesses lower complexity, cannot fulfill the needs of concentrating the data transfers due to its limited available paths.

The path scheduler first uses the monotonic router to find the path with the highest score. Take transfer  $tr_1$  in Figure 3-6 as an example,  $tr_1$  has two paths that both exhibit the highest score:  $ch_{1,2}$ ,  $ch_{2,5}$ , (or  $ch_{1,4}$ ,  $ch_{4,5}$ ),  $ch_{5,8}$ ; the associated accumulated sharing score is calculated as  $4+5+6=15$ . Then, the path scheduler assigns a *tight timing schedule* to the transfers being routed. Here the timing schedule refers to the schedules that leave all the slacks in the end; i.e., the slacks are not exploited but simply spent at the destination register station. An example is the time schedule 1 shown in Figure 3-4. Note that these routed data transfers are still regarded unscheduled in terms of timing. The tight timing schedule is simply pre-allocated to facilitate the channel-based time scheduling to be detailed in the next section. Therefore in the rest of the chapter, we simply call this kind of data transfers as the transfers unscheduled in time. After all data transfers are processed by

the concentration-oriented path scheduler, we obtain an initial concentrated solution suitable for the exploration of timing schedules in order to maximize wire sharing.

### 3.4 Iterative Rescheduling Scheme

There are four major components in the iterative rescheduling scheme as shown in Figure 3-7. Tr\_queue is a priority queue which stores the data transfers unscheduled in time, i.e. those which are not processed by the time scheduler yet; these data transfers in Tr\_queue are sorted by their slacks in a decreasing order. The channel-based time scheduler assigns the timing schedules for the data transfers within a single channel. The reroute procedure contains a ripup function to rip up the routed paths and the concentration-oriented path scheduler introduced in the last section for rerouting the ripped up transfers. The channel control mechanism is responsible for modifying the sharing scores of the channels and updates the channel capacities (the allowable number of wires of the channels).

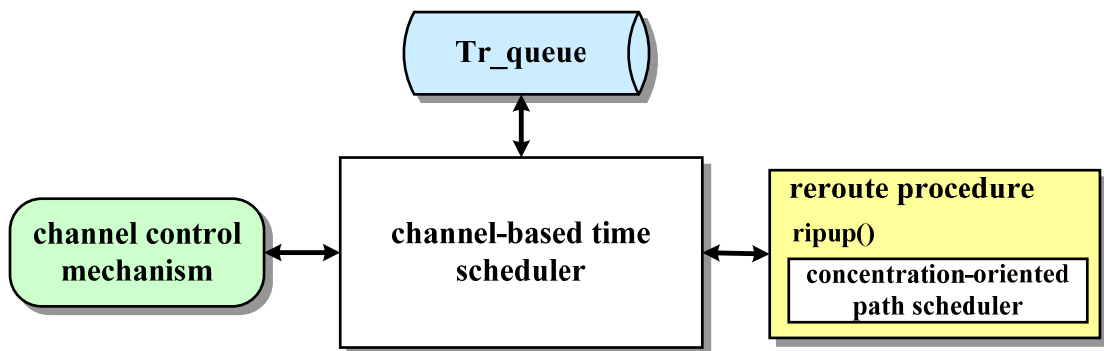


Figure 3-7. Major components of iterative rescheduling scheme.

The iterative rescheduling scheme consists of two levels of loops. Before entering the loops, all data transfers are pushed in to the priority queue  $Tr\_queue$ , and all channel capacities are set to one. At the outer loop, a data transfer  $tr_c$  is popped from  $Tr\_queue$  during each iteration. Then, in the inner loop, the channel-based time scheduler resolves the channels along the path of  $tr_c$  from  $source(tr_c)$  to  $dest(tr_c)$ . Figure 3-8 shows the resolving order of the channels when  $tr_c=tr_1$ . What the time scheduler tries to do is to assign the data transfers using the channel to the available time slots of wires limited by the channel capacity. If, however, some of the transfers cannot be well scheduled in time under the current channel capacity, these transfers are added to the ripup list so that they are ripped up later. Then the list is passed to the concentration-oriented path scheduler to reschedule the path for the ripped up transfers. At this point, the transfers in the list are scheduled in path but not in time, so they are pushed back to  $Tr\_queue$ . Also, if  $tr_c$  is in the ripup list, the time scheduler stops to examine the channels in the rest of the path, and a new iteration with a new  $tr_c$  from  $Tr\_queue$  starts. The iterative scheme terminates when  $Tr\_queue$  is empty, i.e., all the transfers are well scheduled in both time and path. Figure 3-9 gives the partial pseudo code of the two-level loops of the iterative scheme.

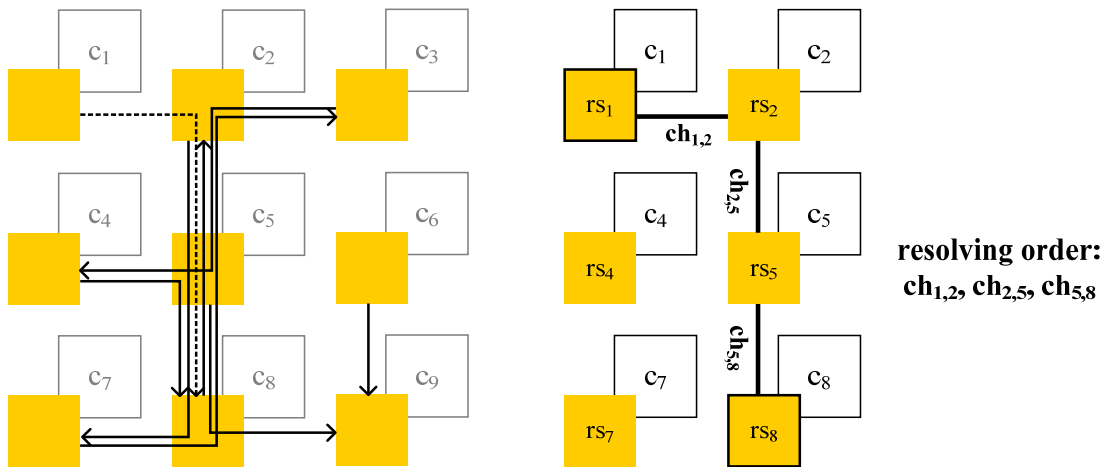


Figure 3-8. Channel resolving order of  $tr_1$ .

```

// Tr: a given set of data transfers

Tr_queue.push(Tr);
while(Tr_queue is not empty){
    trc = Tr_queue.pop();
    for(every channel  $ch_{i,j}$  in the scheduled path of  $tr_c$  from source( $tr_c$ ) to
    dest( $tr_c$ )){
        ripup_list = channel-based_time_scheduler( $ch_{i,j}$ );
        ripup(ripup_list);
        for(every  $tr_i$  in ripup_list)
            concentration-oriented_path_scheduler( $tr_i$ );
        Tr_queue.push(ripup_list);
        if( $tr_c$  is in ripup_list)
            break;
        ripup_list.clear();
    }
}

```

Figure 3-9. Partial pseudo code of iterative rescheduling scheme.

Note that during the iterations, the channel control mechanism dynamically modifies the sharing scores based on the status of the channels, so the concentration-oriented path scheduler does not always route the same path for a data transfer. Moreover, the channel control mechanism monitors the usage of channel and increases the channel capacities if necessary. The details of channel control mechanism are given in the following sections, after the channel-based time scheduler and the reroute procedure.



### 3.4.1 Channel-Based Time Scheduler

For a channel  $ch_{m,n}$  in the path of  $tr_c$ , the channel-based time scheduler tries to fit the data transfers into the  $capacity(ch_{m,n})$  wires by adjusting the timing schedules of the transfers. For example, if the  $capacity(ch_{m,n})$  is two, the number of available timing slots is that of the total cycle count of the DFG times two; there are two time slots for every cycle. Of course, every data transfer can only be scheduled to a limited set of all available timing slots in the channel depending on their routed path and slacks. Thus contention happens when two or more data transfers demand the same time slot of a wire. In this section, we explain what the feasible time slots are for a data transfer and how to determine at which cycle the data transfers use  $ch_{m,n}$ .

We classified the data transfers using channel  $ch_{m,n}$  into two types. The current data transfer of the iteration,  $tr_c$ , belongs to the first type; other transfers, whether scheduled in time or not, belong to the second type. The first type of the data transfers,  $tr_c$ , is allowed to be scheduled to a time slot only if it affects just the timing schedules of the channels that are successors of  $ch_{m,n}$  in the routed path. Remember that the initial timing schedules for all routed data transfers are tight. Also, the time scheduler resolves the channels used by  $tr_c$  one at a time starting from its source to destination register station. Therefore, as long as the timing adjustment in  $ch_{m,n}$  has no impact on the following channels along the path of  $tr_c$ , no previously scheduled cycles will be ruined. Therefore the feasible set of cycles for  $tr_c$  to use  $tr_{m,n}$  is defined as

$F_{i,m,n} = \{cycle_r, \dots, cycle_r + 1, \dots, cycle_r + slack(tr_i)\}$ , where  $cycle_r$  is the original scheduled cycle for  $tr_i$ .

Take the data transfer  $tr_1$  shown by the dotted line in Figure 3-8 as an example. Assume  $tr_c = tr_1$  and the channel-based time scheduler is resolving  $ch_{2,5}$ , that is,  $ch_{m,n} = ch_{2,5}$ . Figure

3-10 (a) shows the initial tight timing schedule of  $tr_1$ . The current time slot is cycle two in the tight schedule, however,  $tr_1$  can also be scheduled to use  $ch_{2,5}$  at cycle 3, 4 or 5 without affecting the usage of  $ch_{1,2}$  due to the 3 slacks it has. Hence the feasible set of cycles for  $tr_1$  to use  $ch_{2,5}$  is  $\{2, 3, 4, 5\}$ . Figure 3-10 shows an example of  $tr_1$  being scheduled at cycle 2; only channel  $ch_{5,8}$  needs to change because of the adjustment in  $ch_{2,5}$ .

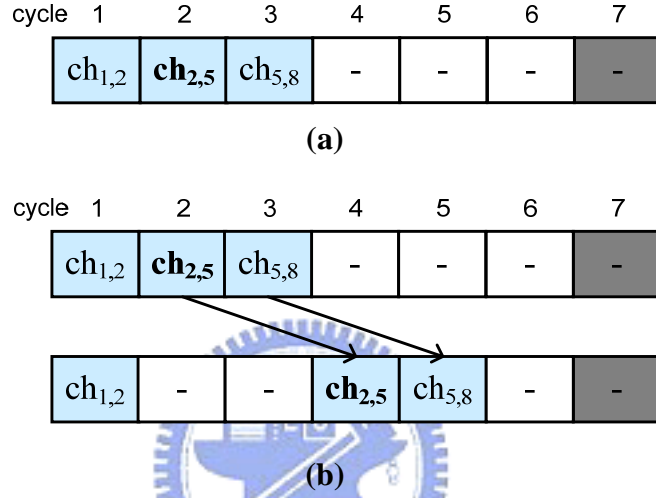


Figure 3-10. (a) Tight timing schedules of  $tr_1$ ; (b) before and after timing adjustment.

For the second type of data transfers, since some of them are already scheduled in time, their feasible set of time slots is subject to the constraint that the timing adjustment will not affect the scheduled cycles of other channels they use. We define *for\_slack* and *back\_slack* as the number of non-transferring cycles right before and after the current scheduled cycles respectively. The feasible set of cycles for scheduling data transfer  $tr_j$  at  $ch_{m,n}$  is

$$F_{i,m,n} = \{cycle_r - for\_slack(tr_i, ch_{m,n}), \dots, cycle_r, \dots, cycle_r + back\_slack(tr_i, ch_{m,n})\},$$

where  $cycle_r$  is the original scheduled cycle for  $tr_i$ .

The data transfer can be rescheduled to use the channel at one of the cycles in this set without affecting any other transferring cycles of  $tr_j$ . Using the same example as of Figure

3-8 and 3-10, this time assume  $tr_3$  is a data transfer already scheduled in time. Figure 3-11 shows the timing schedule of  $tr_3$ . There are two other cycles that  $tr_3$  can use, cycle 2 and cycle 4. None of the other transferring cycles will be affected by this local time rescheduling.

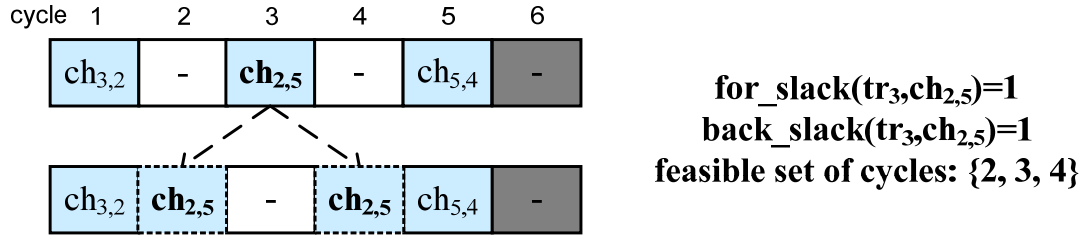


Figure 3-11. Feasible set of cycles for  $tr_3$  to use  $ch_{2,5}$ .

According to the feasible cycles of the data transfers, the time scheduler can then assign the existing time slots to the data transfers. First, the scheduler seeks for the timing slots where there is no contention, then assigns the slots to the corresponding transfers. Here the condition of no contention happens when the available time slots for a specific cycle are fewer than the candidate data transfers which could be scheduled at this cycle. Any new assignment being made may invoke decrement of candidates of some other cycles. Hence the scheduler keeps scanning the cycles until no new assignment can be made. If all data transfers are scheduled in this step, the time scheduler stops. Otherwise, it enters to the next step of resolving contentions based on the *weights* of the data transfers. The weight of a data transfer is defined as

$$w(tr_c) = \alpha \times \#reroute(tr_c) - \beta \times slack(tr_c)$$

$$w(tr_i) = \alpha \times \#reroute(tr_i) - \beta \times (for\_slack(tr_i, ch_{m,n}) + back\_slack(tr_i, ch_{m,n})), \quad \text{where}$$

$tr_i \neq tr_c$  and  $\alpha, \beta$  are given positive constants.

The underlying concept of the weighting function is 1) the more times the data transfer is

rerouted, the higher the weight is; 2) the more choices (feasible cycles) the data transfer has, the lower the weight is. Hence, the scheduler sorts the data transfers by their weights in the decreasing order such that the transfer with higher weight can be scheduled earlier. Then each data transfer  $tr_i$  is scheduled to one of its feasible cycles at which the fewest transfers have already been scheduled. Due to limited timing slots, some of the data transfers may fail to fit in under the current channel capacity. These data transfers are recorded on the ripup list and are sent to the reroute procedure.

Let's go through an example to see more clearly how the time scheduler functions. Figure 3-12 lists the data transfers that uses channel  $ch_{2,5}$  and their associated feasible cycles. Assume the  $capacity(ch_{2,5}) = 1$  and  $w(tr_2) > w(tr_3) > w(tr_7)$ . Because there is only one candidate at cycle 5,  $tr_1$  is scheduled to cycle 1. The same applies to cycle 7, so the only time slot of cycle 7 is assigned to  $tr_8$ . Transfer  $tr_2$  has the highest weight so it is scheduled to cycle 4. Then  $tr_3$  is scheduled to cycle 2. However,  $tr_7$  cannot fit in the channel because the only time slot of cycle 4 has been taken by  $tr_2$ . Therefore,  $tr_7$  is recorded on the rippup list. To sum up, after the timing scheduling, four transfers share a single wire while the remaining transfer  $tr_7$  enters the reroute procedure.

<b>data transfer</b>	<b><math>tr_1</math></b>	<b><math>tr_2</math></b>	<b><math>tr_3</math></b>	<b><math>tr_7</math></b>	<b><math>tr_8</math></b>
<b>feasible cycles</b>	2,3,4,5	4	2,3,4	4	7

Figure 3-12. Feasible cycles of data transfers at  $ch_{2,5}$ .

### 3.4.2 Reroute Procedure

After resolving a channel  $ch_{m,n}$ , the reroute procedure rips up the data transfers on the ripup list, and sends the list to the concentration-oriented path scheduler to be rerouted. To avoid the transfers from using the  $ch_{m,n}$  again, a large negative integer is added to the sharing score of  $ch_{m,n}$ . Under this condition, the path scheduler tends not to use channel  $ch_{m,n}$  unless it is necessary or no better path can be found. Figure 3-13 shows the sharing scores and the path of  $tr_7$  before and after being rerouted.

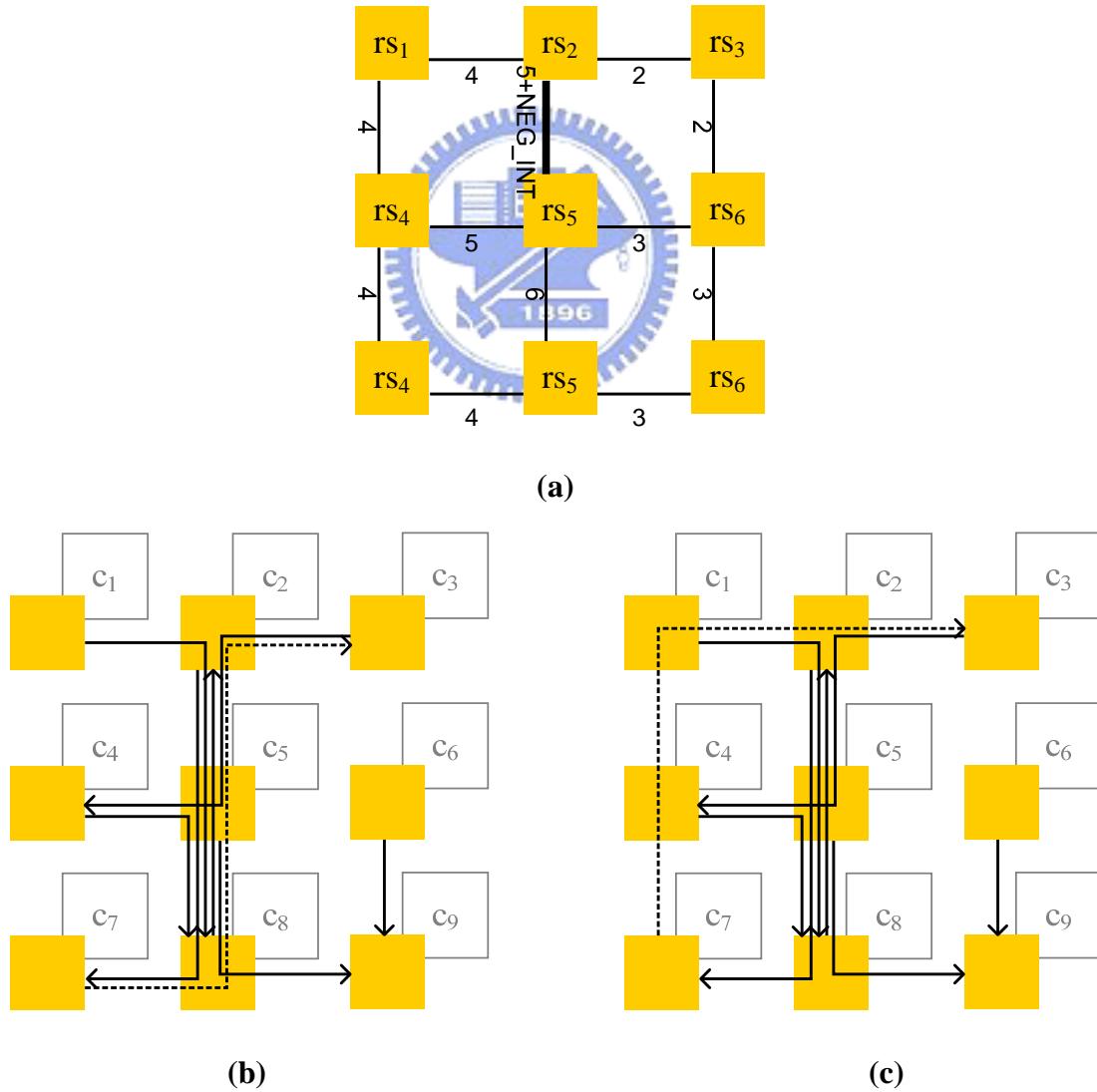


Figure 3-13. (a) The updated sharing score; (b) before and (c) after the rerouting.

It is possible that the sharing score of  $ch_{m,n}$  will be reset to normal under certain conditions which will be specified in the next section. However, the data transfers that have not yet been scheduled in time, except for  $tr_c$ , are exempted from ripping up and rerouting. Since those transfers are still in  $Tr\_queue$  and will be popped out and scheduled later, there is no need to reroute them at this time.

### 3.4.3 Channel Control Mechanism

The channel control mechanism is responsible for two tasks: 1) locking and unlocking a channel; 2) increasing channel capacity. Locking a channel is performed by adding a large negative integer to the sharing score of that channel as described in the reroute procedure. When the time scheduler resolves a channel and the resultant ripup list is not empty, indicating that this channel is currently overused, the channel should be locked. Once the channel is locked, the time scheduler naturally tends to not use the channel in the following iterations. On the other hand, if the channel is resolved again, and no data transfers need to be ripped up, the channel is no longer overused and the sharing score of it should be reset to normal, i.e., the channel is unlocked.

To perform the second task, a counter is used to record the total number of transfers on the ripup list from the beginning of the iterative rescheduling scheme. If the counter exceeds the given threshold  $\gamma$ , this indicates that the channel is very popular and a new wire needs to be added to expand the channel capacity for accommodating more data transfers. In addition, because the fixed data transfers are unmovable, the required number of wires of those transfers set the lowest bound of the channel capacity. Below is the definition of channel capacity.

$capacity(ch_{m,n}) = \max(\# ripup(ch_{m,n}) / \gamma, mwire(ch_{m,n}), 1)$ , where  $\# ripup(ch_{m,n})$  is the sum of the data transfers on the ripup list of all iterations in channel  $ch_{m,n}$  and  $\gamma$  is a given constant and  $mwire(ch_{m,n})$  is the minimum required wires of channel  $ch_{m,n}$  due to the fixed data transfers.

The monotonically increasing channel capacity guarantees the data transfers will gradually settle down and the whole rescheduling process terminates. Moreover, remember  $Tr\_queue$  sorts the data transfers in the decreasing order by their slacks. The data transfers with more slacks are scheduled earlier in the rescheduling process and thus have more opportunities to be ripped up, rerouted and also rescheduled in time. By contrast, the less flexible data transfers are usually processed after the channel capacity has increased. Therefore, the iterative rescheduling scheme manages to achieve explorations of time and space while also reserving room for the less flexible data transfers during the rescheduling iterations.



### 3.5 Pseudo Code of Our Algorithm

```
Transfer_Scheduling ( Tr, arch_spec){  
  
    // Concentration Phase  
    for (every data transfer  $tr_i$  in Tr)  
        add_sharing_score( $tr_i$ );  
    for (every data transfer  $tr_i$  in Tr)  
        concentration-oriented_path_scheduler( $tr_i$ );  
  
    // Iterative Rescheduling Scheme  
    Tr_queue.push(Tr);  
    while (Tr_queue is not empty){  
         $tr_c$  = Tr_queue.pop();  
        for (every channel  $ch_{i,j}$  in the scheduled path of  $tr_c$  from source( $tr_c$ ) to dest( $tr_c$ )){  
            capacity( $ch_{i,j}$ )= max(#ripup( $ch_{i,j}$ )/C, required_#wire( $ch_{i,j}$ ), 1);  
            ripup_list = channel-based_time_scheduler( $ch_{i,j}$ );  
            if (ripup_list is not empty)  
                ripup(ripup_list);  
                for(every  $tr_i$  in ripup_list)  
                    concentration-oriented_path_scheduler( $tr_i$ );  
                lock( $ch_{i,j}$ )  
            else  
                unlock( $ch_{i,j}$ )  
            Tr_queue.push(ripup_list);  
            if ( $tr_c$  is in ripup_list)  
                break;  
            ripup_list.clear();  
        }  
    }
```

Figure 3-14. Pseudo code of our transfer scheduling algorithm.



# Chapter 4

## Experimental Results

We have implemented our algorithm in C++/Linux environment on a workstation with a Xeon 3.2GHz CPU and 2GB RAM. The target RDR-GRS architecture is an MxN cluster array with uni-directional channels connecting neighboring register stations.

Section 4.1 describes the pre-works of our algorithm. 4.2 and 4.3 demonstrate the experimental results on the DFGs from Mediabench set and large synthetic DFGs, respectively. 4.4 discusses the observation and facts about our algorithm.

### 4.1 Experiment Setup

The inputs to our proposed algorithms are architecture-specific topology information, scheduled and bound data flow graphs, and the functional unit placements. To obtain the latter two, we need to generate data flow graphs and then send them to a simple high-level synthesis flow.

#### 4.1.1 DFG Generation

We extract seven pure data flow graphs from applications in MediaBench [20] by the flow shown in the Figure 4.1. With SUIF compiler infrastructure [21] as front end and Machine SUIF [22] as back end, the original C codes in the benchmark suite are converted to the virtual machine codes through a few steps. Then, the internal representations (IR) of the virtual machine codes are extracted to produce the desired data flow graphs.

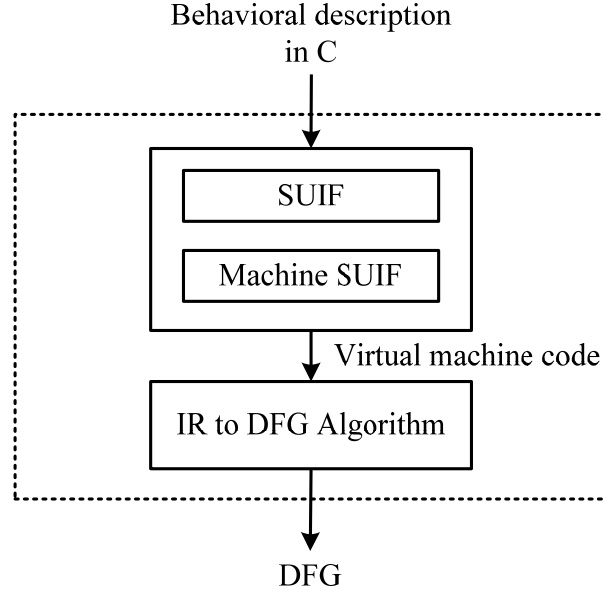


Figure 4-1. DFG generation

#### 4.1.2 Scheduled and Bound DFG

To schedule and map a data flow graph to the RDR-based architecture, a simplified high-level synthesis algorithm, performing scheduling, functional unit binding, placement and rescheduling in order, is applied. Figure 4-2 gives the complete synthesis flow, including DFG generation. Note that this synthesis flow aims only to generate the required inputs to our work and does not address any performance issues (e.g. cycle time or latency). For more detailed and performance-driven synthesis flow for RDR-based architectures, please refer to MCAS [7].

The resource constraints in Figure 4-2 state the available types of functional units and their area, the operations they perform, and the corresponding computation time. In this experiment, we adopt two types of functional units, a single-stage ALU and a two-stage multiplier. First, the force-directed scheduling algorithm [23] schedules the operations in the data flow graph to minimize the required computation resource, i.e., the functional units, under the timing constraint of minimal cycle count. The minimal cycle count is

obtained by using the ASAP schedule for all operations in the data flow graph. Next, an approximate max-clique based binding algorithm [24] assigns each operation in the data flow graph to a specific functional unit. Then, with the given RDR-based architecture specification, the functional units are randomly placed in the clusters. Here the architecture specification describes the topology as an  $M \times N$  cluster array and records the area of each cluster. Random placement should be reasonably fair for comparing different channel and register allocation algorithms since it does not favor any level of interconnect resource sharing. Finally, the rescheduling algorithm reschedules the operations under the determined placement to achieve better cycle count. As shown in Figure 4-2, we obtain the final scheduled and bound DFG with functional unit placement.

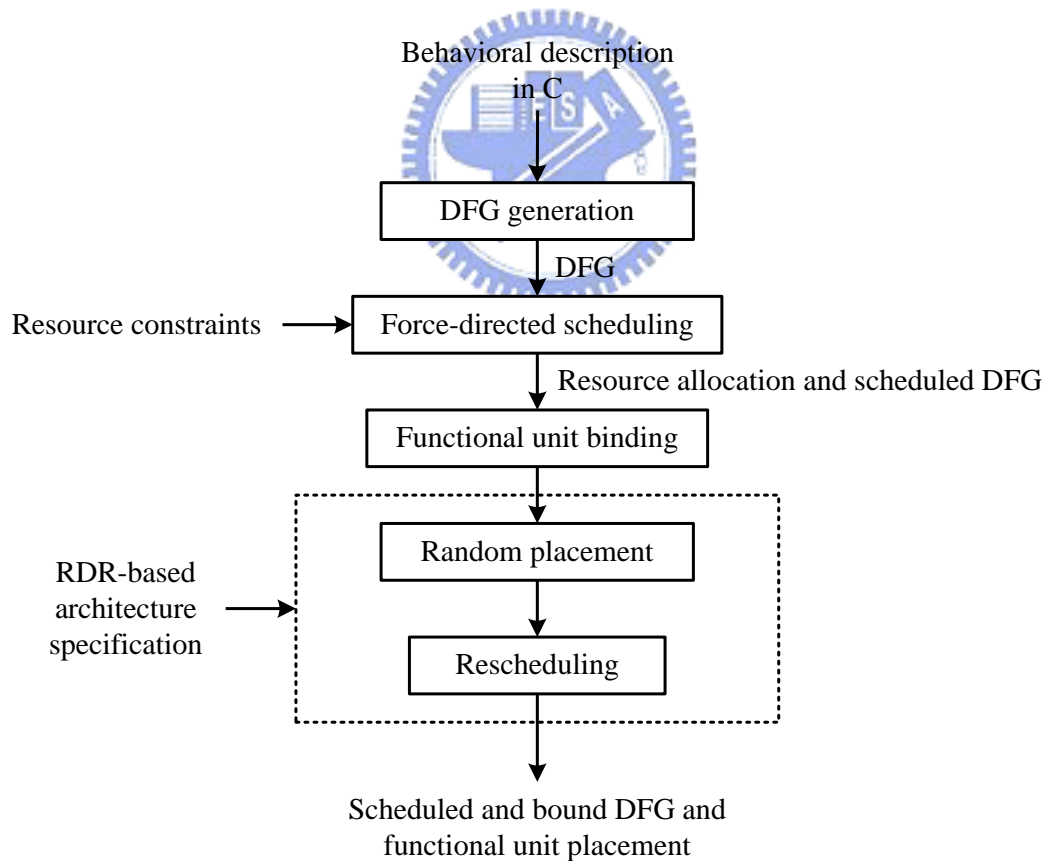


Figure 4-2. Flow of simplified synthesis

## 4.2 Part I: MediaBench

As mentioned earlier in this chapter, seven data flow graphs are extracted from applications in MediaBench. They are scheduled and bound with the 3x3 RDR-based architecture through the synthesis flow. These data flow graphs, named DFG1 to DFG7, are listed in Table 1. The second column in Table 1 denotes the number of nodes in the data flow graph; the third column specifies the required number of functional units; the fourth column denotes the total cycle count of the scheduled data flow graph; and the fifth column lists the number of fixed and active data transfers, respectively. The fixed data transfer has no slacks and only one path for schedule, so they are regarded as unmovable, pre-scheduled transfers when our algorithm starts. On the other hand, the schedules of active transfers can either be exploited in time, space, or both.

Table 1. Scheduled and Bound DFGs on 3x3 Architecture

DFG	#node	FU resources		cycle count	#data transfer	
		#ALU	#MUL		#fixed	#active
DFG1	101	9	4	27	54	38
DFG2	66	5	2	25	39	18
DFG3	196	18	8	19	108	64
DFG4	100	9	4	20	63	29
DFG5	58	9	0	14	29	20
DFG6	119	7	5	33	91	38
DFG7	140	11	6	27	86	38

We also implement three additional methods to solve channel and register allocation problems in RDR, RDR-pipe and RDR-GRS. The first method conforms to the point-to-point inter-cluster communication model of RDR, and each data transfer is assigned a dedicated interconnect throughout the transfer cycles. The second method is targeted to support RDR-pipe by inserting pipeline registers along the interconnection and exploiting the slacks for channel sharing of data transfers with identical source-destination cluster pair. The third method uses ILP formulation and solves the problem by an ILP solver, lpsolve 5.5.0.0 [25], to provide optimal solution for RDR-GRS. The objective function in this method is set to  $5*(\#wire)+1*(\#reg)$ .

The above three methods and our algorithms are applied to the 7 data flow graphs listed in Table 1, and the results are reported in Table 2. The second, third, fourth and the fifth columns are demanded interconnect resources (number of wires and registers) of RDR, RDR-pipe, RDR-GRS/ILP and RDR-GRS/Ours, respectively. In some cases, RDR-GRS/ILP fails to produce solutions within 12 hours, so the results of those cases are left unfilled. Moreover, in Table 3, we normalized the results in Table 2 to the demanded interconnect resources of RDR so that the pros and cons of these methods are more clearly presented. As expected, RDR-pipe reduces nearly 20% of wires by introducing 50% extra pipeline registers. RDR-GRS/ILP, though provides the optimal solution in both the number of wires and registers, runs out of time frequently. However, our algorithm provides averagely 54.2% reduction on wires, which is close to the optimal solution. Since we focus on minimizing the global wires, the number of registers of our algorithm is not as good as that of RDR-GRS/ILP. Nonetheless, our algorithm still uses less registers than RDR, and far less than RDR-pipe.

Table 2. Interconnect Resources Demand of DFGs

DFG	RDR		RDR-pipe		RDR-GRS/ILP		RDR-GRS/Ours	
	#wire	#reg	#wire	#reg	#wire	#reg	#wire	#reg
DFG1	79	57	71	87	29	27	37	47
DFG2	64	35	48	56	28	24	33	38
DFG3	179	104	144	166	-	-	80	88
DFG4	93	62	78	96	-	-	46	55
DFG5	60	42	49	64	-	-	37	48
DFG6	74	56	57	75	25	27	33	41
DFG7	113	73	82	108	-	-	25	56

Table 3. Interconnect Resources Demand of DFGs Normalized to RDR

DFG	RDR-pipe		RDR-GRS/ILP		RDR-GRS/Ours	
	#wire	#reg	#wire	#reg	#wire	#reg
DFG1	0.899	1.526	0.367	0.474	0.468	0.825
DFG2	0.750	1.600	0.438	0.686	0.516	1.086
DFG3	0.804	1.596	-	-	0.447	0.846
DFG4	0.839	1.548	-	-	0.495	0.887
DFG5	0.817	1.524	-	-	0.617	1.143
DFG6	0.770	1.339	0.338	0.482	0.446	0.732
DFG7	0.726	1.479	-	-	0.221	0.767
Avg.	0.801	1.516	0.381	0.547	0.458	0.898

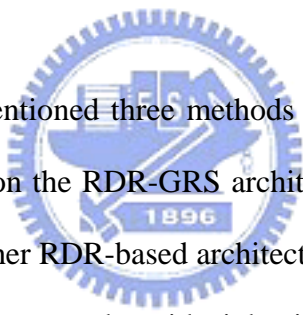
### 4.3 Part II: Synthetic Large DFGs

When the size of architecture grows and the scale of the associated transfer path scheduling problem escalates, the probability of global channel sharing in RDR-GRS should increase. Therefore it is predictable that RDR-GRS can spare numerous channels (registers) than RDR (RDR-pipe) under this condition. Nonetheless, so far no previous research has reported the demands of interconnect resources of different RDR-based architectures in large-scale problems. In this section, we want to use larger data flow graphs to 1) verify the advantage of RDR-GRS over the other two architectures; 2) demonstrates the capability of our algorithm in reducing the interconnect resources.

We generate a set of data flow graphs from the applications in MediaBench and then either duplicate a graph one or more times and attach the duplicates to the original one or simply combine two or more data flow graphs. These manually-created synthetic data flow graphs are synthesized on a 10x10 RDR architecture. Due to the large size of the synthetic DFGs and the large scale of the target architecture, the restriction of resource (FU) minimization in force-directed scheduling during synthesis is relaxed so that the overall synthesis time is kept within a reasonable range. Table 4 shows the information of six scheduled and bound synthetic data flow graphs, named Syn1 to Syn6. Because of the larger architecture, the percentage of active data transfer in all data transfers are obviously increases compared to DFGs in Table 1. Hence, the final quality of transfer scheduling should be less determined by the pre-scheduled fixed data transfers, but more depend on the scheduling algorithm. Notice that the number of required functional units for every data flow graph is controlled in the range of 90 to 100. When these functional units are placed on the 10x10 architecture, i.e., 100 clusters, almost every cluster contains one functional unit.

Table 4. Scheduled and Bound Synthetic DFGs on 10x10 Architecture

DFG	#node	FU resources		cycle count	#data transfer	
		#ALU	#MUL		#fixed	#active
Syn1	567	73	20	88	100	421
Syn2	634	66	32	251	212	801
Syn3	538	64	31	80	124	350
Syn4	497	79	19	84	147	469
Syn5	566	74	20	89	98	675
Syn6	369	78	18	63	131	144



In addition to the aforementioned three methods and our algorithm, another method called RAND is implemented on the RDR-GRS architecture to compare the advantage of global resource sharing over other RDR-based architectures. RAND randomly assigns each data transfer to one of the shortest paths with tight timing schedule. Table 5 reports the demanded interconnect resources of these methods applied on the synthetic data flow graphs. Again, the results normalized to RDR are listed in Table 6. As seen in the table, RDR-pipe reduces only on average 8% number of wires while dramatically increasing the number of registers to almost 500%. We consider that the minor improvement of the number of wires is resulted from the decrease in sharable data transfers. Remember that each cluster in the architecture contains at most one functional unit, so that the percentage of data transfers with the identical source-destination cluster pairs may not be high. Though the reduction of wires may be improved as the number of sharable data transfers increases (e.g. number of functional units per island increases), it implies that the effectiveness of RDR-pipe on saving interconnect resources heavily depends on the



characteristics of inputs due to the limited local scope of sharing. In some cases, like the results in Table 6, it might not be worth introducing an astonishing number of registers to reduce merely a few percents of wires. GRS/RAND, however, uses only 32% of wires and 112.7% compared to RDR. The results validate that RDR-GRS is a great platform for interconnect resource sharing. Our algorithm further explores the potential of global sharing and reduces the number of required wires to 18.3%; the number of required registers is also decreased to 91.8% compared to RDR. Moreover, Table 5 also shows that our algorithm can efficiently produce the results in an extremely short runtime. Hence it is promising to apply our algorithm on larger DFG/architecture. It also indicate the possibility of integrate our algorithm into a FU placer such that the placer can be aware of the potential wire sharing at earlier stage. The interconnect resources could be further reduced by finding a better FU placement.

Table 5. Interconnect Resources Demand of Synthetic DFGs

DFG	RDR		RDR-pipe		GRS/RAND		RDR-GRS/Ours		
	#wire	#reg	#wire	#reg	#wire	#reg	#wire	#reg	runtime (sec)
Syn1	3005	558	2945	3049	844	543	501	457	1.63
Syn2	3896	947	2967	3375	996	915	462	736	2.97
Syn3	2673	460	2584	2667	752	426	484	381	1.41
Syn4	3261	694	3049	3256	847	654	382	524	0.54
Syn5	4542	945	4334	4586	1243	926	638	714	3.07
Syn6	1115	211	1026	1067	634	416	418	332	0.72

Table 6. Interconnect Resources Demand of Synthetic DFGs Normalized to RDR

DFG	RDR-pipe		GRS/RAND		RDR-GRS/Ours	
	#wire	#reg	#wire	#reg	#wire	#reg
Syn1	0.980	5.464	0.281	0.973	0.167	0.819
Syn2	0.762	3.564	0.256	0.966	0.119	0.777
Syn3	0.967	5.798	0.281	0.926	0.181	0.828
Syn4	0.935	4.692	0.260	0.942	0.117	0.755
Syn5	0.954	4.853	0.274	0.980	0.140	0.756
Syn6	0.920	5.057	0.569	1.972	0.375	1.573
Avg.	0.920	4.905	0.320	1.127	0.183	0.918

To rule out the inherent advantage of RDR-GRS architecture and show the effectiveness of our algorithm, we compare our algorithm with the RAND method directly. Table 7 lists the demanded interconnect resources of our algorithm after being normalized to those of RAND. The table shows that our algorithm can averagely reduce 44.5% of wires and 18.2% of registers in comparison with RAND. Also, the fact that our algorithm uses fewer registers in all six cases indicates that increasing channel sharing might at the same time increase register sharing to some extent.

In summary, RDR-GRS outperforms RDR and RDR-pipe in saving interconnect resources, and the advantage of RDR-GRS becomes clearer as the problem size increases. Moreover, our algorithm successfully explores the potential of resource sharing in all six cases of large synthetic DFGs and achieves 81.7% reduction of wires and 8.2% reduction of registers compared to RDR.

Table 7. Interconnect Resources Demand of Synthetic DFGs Normalized to RAND

DFG	RDR-GRS/Ours	
	#wire	#reg
Syn1	0.594	0.842
Syn2	0.464	0.804
Syn3	0.644	0.894
Syn4	0.451	0.801
Syn5	0.513	0.771
Syn6	0.659	0.798
Avg.	0.554	0.818



# Chapter 5

## Conclusions and Future Work

Based on the RDR-GRS architecture, we formulate the channel and register allocation problem and propose a transfer scheduling algorithm to maximize global wire sharing. We present an iterative rescheduling scheme that interchangeably uses the concentration-oriented path scheduler and the channel-based time scheduler to concentrate the paths of the transfers and resolve the contention by exploiting their timing slacks. We also set the channel control mechanism to monitor the usage of the channels and to increase the channel capacities only if necessary. The experimental results demonstrate that the RDR-GRS architecture has the inherent advantage of saving interconnect resources compared to other RDR-based architectures, and the performance gap between RDR-GRS and other architectures widens when the designs become larger. However, though the ILP formulation is capable of providing the desirable optimality, it has been proven to run out of steam fast. On the contrary, our approach can be applied on large synthetic DFGs and successfully reduce on average 81.7% wires and 8.2% registers compared to RDR. Therefore, the proposed approach is capable of efficiently scheduling the data transfers to minimize the demand of interconnect resources under the RDR-GRS architecture for large-scale designs.

To further reduce the demand of interconnect resources, one possible enhancement of the proposed algorithm is to minimize the registers during post-processing by rescheduling some data transfers provided that the channel usage is not affected. The shortest path constraint could also be relaxed at this stage to allow detouring paths in rescheduling as long as the channel or register sharing is bettered. Since the post-processing only seeks for

improvement over the previous solution, it should be able to keep the complexity of detouring within an acceptable range. Another possible direction of the future work is to integrate the proposed channel and register allocation algorithm into an FU placer. With the feedback on the interconnect resources from our allocation algorithm, the placer should be promising in producing less resource-consuming placement. Last but not least, the global resource sharing in RDR-GRS is not achieved at no cost. In fact, additional multiplexing logic is introduced to control data forwarding on this architecture. Therefore, even our algorithm can greatly reduce interconnect resources, a complete experiment includes final area and timing needs to be performed to evaluate the effectiveness of the RDR-GRS architecture and the synthesis system (including our algorithm) on top of it.



# References

- [1] International Technology Roadmap for Semiconductor. Semiconductor Industry Association, 1999.
- [2] Y. Mori, V. Moshnyaga, H. Onodera, and K. Tamaru, "A performance-driven macro-block placer for architectural evaluation of ASIC designs," in *Proceedings of Annual IEEE International ASIC Conference and Exhibit*, pp. 233-236, Sep. 1995.
- [3] V. Moshnyaga and K. Tamaru, "A placement driven methodology for high-level synthesis of sub-micron ASIC's," in *Proceedings of International Symposium on Circuits and Systems*, vol. 4, pp. 572-575, May 1996.
- [4] P. Prabhakaran and P. Banerjee, "Parallel algorithms for simultaneous scheduling, binding and floorplanning in high-level synthesis," in *Proceedings of International Symposium on Circuits and Systems*, vol. 6, pp. 372-376, May 1998.
- [5] D. Kim, J. Jung, S. Lee, J. Jeon, and K. Choi, "Behavior-to-placed RTL synthesis with performance-driven placement," in *Proceedings of International Conference on Computer Aided Design*, pp. 320-325, Nov. 2001.
- [6] J. Jeon, D. Kim, D. Shin, and K. Choi, "High-level synthesis under multi-cycle interconnect delay," in *Proceedings of Asia and South Pacific Design Automation Conference*, pp. 662-667, Jan. 2001.
- [7] J. Cong, Y. Fan, G. Han, X. Yang, and Z. Zhang, "Architecture and synthesis for on-chip multicycle communication," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 4, pp. 550-564, Apr. 2004.
- [8] W.-S. Huang, Optimal channel and register allocation in architectural level synthesis using ILP," Master's Thesis, NCTU, June 2006.
- [9] J. Cong, L. He, K. Y. Khoo, C. K. Koh, and Z. Pan, "Interconnect Design for Deep

- Submicron ICs,” in *Proceedings of International Conference on Computer-Aided Design*, pp. 478-485, Nov. 1997.
- [10] P. Chong and R. K. Brayton, “Characterization of feasible retimings,” in *Proceedings of International Workshop Logic Synthesis*, pp. 1-6, Jun. 2001.
- [11] J. Cong and S. K. Lim, “Physical planning with retiming,” in *Proceedings of International Conference Computer-Aided Design*, pp. 2-7, Nov. 2000.
- [12] J. Cong and X. Yuan, “Multilevel global placement with retiming,” in *Proceedings of Design Automation Conference*, pp. 208-213, Jun. 2003.
- [13] D. P. Singh and S. D. Brown, “Integrated retiming and placement for field programmable gate arrays,” in *Proceedings of International Symposium on Field Programmable Gate Arrays*, pp. 67-76, Feb. 2002.
- [14] L. Scheffer, “Methodologies and tools for pipelined on-chip interconnect,” in *Proceedings of International Conference on Computer Design*, pp. 152-157, Sept. 2002.
- [15] D. M. Chapiro, “Globally-asynchronous locally-synchronous systems,” *Ph.D. dissertation, Stanford Univ., Stanford, CA*, 1984.
- [16] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, “Theory of Latency-Insensitive Design,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059-1076, Sept. 2001.
- [17] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, “A Methodology for ‘Correct-by-Construction’ Latency Insensitive Design,” in *Proceedings of International Conference Computer-Aided Design*, pp. 309-315, Nov. 1999.
- [18] J. Cong, Y. Fan, and Z. Zhang, “Architecture-level synthesis for automatic interconnect pipelining,” in *Proceedings of Design Automation Conference*, pp. 602-607, Jun. 2004.
- [19] M. Pan and C. Chu, “FastRoute 2.0: A High-quality and Efficient Global Router,” in

*Proceedings of Asia and South Pacific Design Automation Conference*, pp. 250-255, Jan. 2007.

- [20] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, pp. 330-335, Dec. 1997.
- [21] SUIF 2 Compiler System. [Online]. Available: <http://suif.stanford.edu/suif/suif2/>
- [22] M. Smith and G. Holloway, "An introduction to machine suif and its portable libraries for analysis and optimization," in *Division of Engineering and Applied Sciences, Harvard University*, 2002.
- [23] P. Paulin and J. Knight, "Force-directed scheduling for behavioral synthesis of ASICs," *IEEE Transactions on Computer-Aided Design*, vol. 8, pp. 661-679, Jun. 1989.
- [24] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.
- [25] lp solve open source LP and MILP solver. [Online]. Available: [http://groups.yahoo.com/group/lp\\_solve/](http://groups.yahoo.com/group/lp_solve/)

