A NOVEL FLEXIBLE ON-CHIP SWITCH ARCHITECTURE FOR RECONFIGURABLE HARDWARE ACCELERATORS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

FATİH YAZICI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

AUGUST 2021

Approval of the thesis:

A NOVEL FLEXIBLE ON-CHIP SWITCH ARCHITECTURE FOR RECONFIGURABLE HARDWARE ACCELERATORS

submitted by **FATİH YAZICI** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** ____________

Prof. Dr. İlkay Ulusoy
Head of Department, **Electrical and Electronics Engineering** ____________

Prof. Dr. Şenan Ece Güran Schmidt
Supervisor, **Electrical and Electronics Engineering, METU** ____________

Examining Committee Members:

Prof. Dr. Gözde Bozdağı Akar
Electrical and Electronics Engineering, METU ____________

Prof. Dr. Şenan Ece Güran Schmidt
Electrical and Electronics Engineering, METU ____________

Prof. Dr. İlkay Ulusoy
Electrical and Electronics Engineering, METU ____________

Prof. Dr. Murat Manguoğlu
Computer Engineering, METU ____________

Prof. Dr. Özcan Öztürk
Computer Engineering, Bilkent University ____________

Date: 13.08.2021

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Fatih Yazıcı

Signature :

ABSTRACT

A NOVEL FLEXIBLE ON-CHIP SWITCH ARCHITECTURE FOR RECONFIGURABLE HARDWARE ACCELERATORS

Yazıcı, Fatih
M.S., Department of Electrical and Electronics Engineering
Supervisor: Prof. Dr. Şenan Ece Güran Schmidt

August 2021, 85 pages

This thesis work proposes ReFlex Switch, a novel, scalable on-chip packet switch architecture, that is designed to interconnect heterogeneous IP cores at high speeds. One target application for ReFlex switch is hardware accelerated cloud computing where the cloud servers feature FPGA cards with reconfigurable regions to implement accelerators demanded by the users. In this setting, the increasing data rates call for line-speed operation of the on-chip switch to maintain scalability. The first requirement of the line rate operation is a fabric arbiter design to achieve maximal throughput while allocating the switching bandwidth as required among contending traffic sources. The second requirement together with the limited on-chip memory resources is the efficient buffer management at the switch inputs. Furthermore, the service needs of the applications change with the realized accelerators on the FPGA card.

ReFlex Switch fulfills these requirements with an on-chip hardware architecture that is flexibly configured according to the specified system parameters. To this end, this thesis proposes Credit ARbiter (CAR), a novel fabric arbiter with Quality of Service

(QoS) support and ReFlex Buffer Management (ReFBM) a novel input buffer organization for buffer allocation to the connected cores according to their traffic demand.

ReFlex Switch is implemented on Xilinx XC7Z100 SoC FPGA at 40 Gbps line rate, with CAR and ReFBM together with legacy arbiters and buffer organizations for resource use and performance comparison. The evaluations demonstrate that ReFlex switch can be flexibly instantiated with different memory parameters, arbiters and memory organizations. Furthermore, CAR and ReFBM achieve desired performance goals and outperform comparable work in the literature.

Keywords: On-chip switch, switch fabric arbitration, dynamic buffer management, hardware accelerated cloud data center, partial reconfiguration

ÖZ

YENİDEN YAPILANDIRILABİLİR DONANIM HIZLANDIRICILAR İÇİN YENİ ESNEK BİR YONGA ÜSTÜ ANAHTAR MİMARİSİ

Yazıcı, Fatih
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Şenan Ece Güran Schmidt

Ağustos 2021 , 85 sayfa

Bu tez çalışması, heterojen IP çekirdeklerini yüksek hızlarda birbirine bağlamak için tasarlanmış yeni, ölçeklenebilir bir yonga üstü paket anahtar mimarisi olan ReFlex Switch'i önermektedir. ReFlex Switch için bir uygulama alanı, kullanıcılar tarafından talep edilen hızlandırıcıları gerçeklemek için yeniden yapılandırılabilir bölgelere sahip olan FPGA kartlarının bulunduğu donanım hızlandırmalı bulut bilişimdir. Bu bağlamda artan veri hızları, ölçeklenebilirliği korumak için yonga üstü anahtarın hat hızında çalışmasını gerektirir. Hat hızı işleminin ilk gereksinimi, anahtarlama bant genişliğini rekabet eden trafik kaynakları arasında gerektiği gibi tahsis ederken maksimum verim elde etmeye yarayan bir çekişme çözümleyici tasarımıdır. Sınırlı yonga üstü bellek kaynakları ile birlikte ikinci gereksinim, anahtar girişlerinde verimli bir arabellek yönetimidir. Ayrıca FPGA kartları üzerinde gerçeklenen hızlandırıcılar ile uygulamaların hizmet ihtiyaçları da değişmektedir.

ReFlex Switch, belirtilen sistem parametrelerine göre esnek bir şekilde konfigüre edilen bir yonga üstü donanım mimarisi ile bu gereksinimleri karşılamaktadır. Bu amaçla bu tez, Hizmet Kalitesi (QoS) desteğine sahip yeni bir çekişme çözümleyici olan Cre-

dit ARbiter'ı (CAR) ve bağlı çekirdeklere trafik taleplerine göre arabellek tahsisi için yeni bir arabellek organizasyonu olan ReFlex Buffer Management'ı (ReFBM) önermektedir.

ReFlex Switch, Xilinx XC7Z100 SoC FPGA üzerinde 40 Gbps hat hızında, CAR ve ReFBM ile birlikte kaynak kullanımı ve performans karşılaştırması için eski çekişme çözümleyici ve arabellek organizasyonlarıyla da gerçeklenmiştir. Sonuçlar, ReFlex Switch'in farklı bellek parametreleri, çekişme çözümleyicileri ve bellek organizasyonları ile esnek bir şekilde uygulanabildiğini göstermektedir. Ayrıca, CAR ve ReFBM, istenen performans hedeflerine ulaşmakta ve literatürdeki karşılaştırılabilir çalışmalardan daha iyi performans göstermektedir.

Anahtar Kelimeler: Yonga-üstü anahtar, anahtar örgüsü çekişmesi, dinamik arabellek yönetimi, donanım hızlandırıcılı bulut veri merkezi, kısmi yeniden yapılandırma

To my family

ACKNOWLEDGMENTS

I would like to thank my supervisor Prof. Dr. Ece Güran Schmidt for her precious support and sincere efforts. This work is made possible by her timely help and guidance as well as countless hours of disciplined hard work together throughout the study, from concept to completion.

This thesis was supported by the Scientific and Research Council of Turkey (TUBITAK) [Project Code 117E667-117E668].

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xi

LIST OF TABLES xiii

LIST OF FIGURES xv

LIST OF ALGORITHMS xvii

LIST OF ABBREVIATIONS xviii

CHAPTERS

1 INTRODUCTION 1

2 BACKGROUND AND PREVIOUS WORK 7

2.1 Packet Switching 7

2.1.1 Preliminaries 8

2.1.2 A Packet Switch with Crossbar Fabric: Buffer Organization and Operation 9

2.1.3 Fabric Arbiters 11

2.2 On-chip Packet Switches 13

2.2.1 Homogeneous On-Chip Switches 14

2.2.2 Heterogeneous On-Chip Switches to Interconnect Hardware Accelerators . . . 17

2.3 Contribution and Placement of the Thesis Work in the Literature . . . 21

3 REFLEX SWITCH . . . 27

3.1 CAR (Credit Arbiter) . . . 29

3.2 ReFBM (ReFlex Buffer Management) . . . 34

3.3 ReFlex Switch Hardware Architecture . . . 36

3.3.1 Pipelined Switching Cycles . . . 36

3.3.2 Buffer Management . . . 39

3.3.3 One-Phase-Per-Cycle Arbitration . . . 42

3.3.4 Configuration and Status Registers (CSR) . . . 44

3.3.5 Parametric ReFlex Switch RTL Generation . . . 49

4 EVALUATION . . . 51

4.1 Performance Evaluation . . . 51

4.2 Hardware Implementation on FPGA . . . 64

5 CONCLUSION AND FUTURE WORK . . . 71

REFERENCES . . . 73

APPENDICES

A CAR EXECUTION EXAMPLE . . . 79

LIST OF TABLES

TABLES

Table 2.1 Summary of Related Previous Work - Homogeneous (part 1) 23

Table 2.2 Summary of Related Previous Work - Homogeneous (part 2) 24

Table 2.3 Summary of Related Previous Work - Heterogeneous (part 1) 25

Table 2.4 Summary of Related Previous Work - Heterogeneous (part 2) 26

Table 3.1 ReFlex Switch Pipeline Stages for Single Flit (indicates last flit) . 37

Table 3.2 ReFlex Switch Pipeline Stages for Multiple Flits (indicates last flit) 38

Table 3.3 Bit Field Widths of a CSR Address 44

Table 3.4 CSR CORE Register Map 45

Table 3.5 CSR FB Register Map 46

Table 3.6 CSR RB Register Map 47

Table 3.7 CSR ARBT Register Map 48

Table 4.1 Bit Field Widths of a ReFlex Switch Input Flit 64

Table 4.2 FPGA Implementation Results (target: 156.25 MHz) of Different Arbiters 66

Table 4.3 FPGA Implementation Results (target: highest) of Different Arbiters 66

Table 4.4 FPGA Implementation Results (target: 156.25 MHz) of Different Input Buffer Management Schemes for Single Port 67

Table 4.5 FPGA Implementation Results (target: highest) of Different Input Buffer Management Schemes for Single Port 67

Table 4.6 FPGA Implementation Results (target: 156.25 MHz) of Different 8-Port ReFlex Switch Configurations 68

Table 4.7 FPGA Implementation Results (target: highest) of Different 8-Port ReFlex Switch Configurations 68

LIST OF FIGURES

FIGURES

Figure 1.1 ReFlex Switch Architecture Example Context 5

Figure 2.1 Crossbar Fabric 9

Figure 2.2 HoL Blocking Problem 10

Figure 2.3 Buffer Organization 11

Figure 3.1 ReFlex Switch - Architecture and Memory Organization (instantiation with $N = 8$ and $M = 16$) 28

Figure 3.2 *Flex Buffer* Allocation for Resource Efficient FIFO Organization 35

Figure 3.3 ReFBM Compact FIFO Organization 39

Figure 3.4 Rotation and Permutation in Combinational Priority Encoders . . 43

Figure 4.1 Simulator Verification with DRR 52

Figure 4.2 CAR Quality of Service with Infinite VOQ 54

Figure 4.3 ReFlex Buffer Management with DRR (uniform traffic) 56

Figure 4.4 ReFlex Buffer Management with DRR (nonuniform traffic) . . . 58

Figure 4.5 Effect of *Flex Buffer* Count with DRR 60

Figure 4.6 ReFlex Buffer Management with CAR (ports) 62

Figure 4.7 ReFlex Buffer Management with CAR (total) 63

Figure 4.8 Synthesized Datapath of an Input Block with ReFlex Buffer Management 70

Figure A.1 CAR Execution Example: Initialization 80

Figure A.2 CAR Execution Example: Request 81

Figure A.3 CAR Execution Example: Grant 82

Figure A.4 CAR Execution Example: Accept 83

Figure A.5 CAR Execution Example: Accept - Update Credits 1 84

Figure A.6 CAR Execution Example: Accept - Update Credits 2 85

LIST OF ALGORITHMS

ALGORITHMS

Algorithm 1 CAR Iteration Pseudocode 32

Algorithm 2 CAR Credit Update Pseudocode 33

LIST OF ABBREVIATIONS

ABBREVIATIONS

2-D	Two Dimensional
ACCLOUD	Accelerated Cloud
ALM	Adaptive Logic Module
ASIC	Application Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
AXI	Advanced Extensible Interface
BRAM	Block RAM
CAR	Credit ArbiteR
CPU	Central Processing Unit
CSR	Configuration and Status Register
DDR	Double Data Rate
DMA	Direct Memory Access
DRAM	Distributed RAM / Dynamic RAM
DRR	Dual Round Robin
DSA	Domain Specific Architecture
DSP	Digital Signal Processor
ECC	Error Correction Code
EOF	End-of-Frame
ER	Elastic Router
FAC	FPGA Accelerator Card
FB	Flex Buffer
FBFLY	Flattened Butterfly

FF	Flip Flop
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GbE	Gigabit Ethernet
GPU	Graphics Processing Unit
HA	Hardware Accelerator
HACDC	Hardware Accelerated Cloud Data Center
HOL	Head-of-Line
IP	Internet Protocol / Intellectual Property
iSLIP	Iterative Round Robin Matching with Slip
KNN	K-Nearest Neighbours
LBNOC	Look Ahead Bypass Network-on-Chip
LFSR	Linear Feedback Shift Register
LUT	Look Up Table
LUTRAM	Look Up Table RAM
MPSOC	Multiprocessor System-on-Chip
NOC	Network-on-Chip
PCIE	Peripheral Component Interconnect Express
OpenCL	Open Computing Language
OS	Operating System
PE	Processing Element
PR	Partial Reconfiguration
QOS	Quality of Service
QSFP+	Quad Small Form-Factor Pluggable Plus
RAM	Random Access Memory
RB	Reassembly Buffer
ReFBM	Reflex Buffer Management

ROM	Read Only Memory
RR	Reconfigurable Region
RTL	Register Transfer Level
SDM	Space Division Multiplexing
SL	Service Level
SOC	System-on-Chip
SODIMM	Small Outline Dual Inline Memory Module
SPICE	Simulation Program with Integrated Circuit Emphasis
SW	Switch / Software
TDM	Time Division Multiplexing
VC	Virtual Channel
VCID	Virtual Channel Identity
UDN	Unidirectional Network-on-Chip Crossbar
VCT	Virtual Cut-Through
vFPGA	Virtual FPGA
VHSIC	Very High Speed Integrated Circuit Program
VHDL	VHSIC Hardware Description Language
VM	Virtual Machine
VOQ	Virtual Output Queue
XBAR	Crossbar

CHAPTER 1

INTRODUCTION

Hardware Accelerated Cloud Data Centers (HACDC) offer hardware accelerators (HA) as computing resources in addition to the conventional resources such as memory, processor (CPU) and disk [1, 2]. To this end, FPGA Accelerator Cards (FAC) can be installed in the servers, or FACs with an SoC processor can be directly connected to the HACDC network. Multiple HAs can be instantiated on partially Reconfigurable Regions (RR) on the same FPGA [3].

Such organization requires high-speed data exchange among FAC components such as accelerators, memory, server processor (through PCIe) and datacenter network. FPGA hardware accelerators come with different size and bandwidth requirements [4, 5, 2, 6]. Furthermore, Quality of Service (QoS) support is necessary both for satisfying the requirements of the applications and enabling bandwidth allocation for the Virtual Machines [7]. The varying range of heterogeneity of the interfaces on the FAC and the non-uniform load distribution among them require a single flexible and reconfigurable on-chip packet switch with sufficient number of ports particularly in the critical path between datacenter network interfaces such as Ethernet and HAs in contrast to the mesh based multiple-router Network on Chip (NoC) Architectures which interconnect similar or identical type generic Processing Elements (PE).

Within the scope of FPGA hardware accelerators, throughput and end-to-end packet latency of data between accelerators, on-chip switches and the datacenter network are the relevant performance metrics whereas logic element, block RAM and power consumption outline the resource usage. Throughput determines the overall flow of data and low latency is desirable to lower the penalty for accessing on-chip and off-chip resources, which in turn improves the individual accelerator performance. Decreas-

ing resource usage of the interconnect architecture enables larger and more integrated accelerators.

Existing solutions such as mesh based NoC architectures focus on providing a multiple-router fast interconnect [8, 9, 10]. This method is suitable when generic processing elements are utilized to perform a variety of acceleration tasks. However, when the accelerators themselves are reconfigurable in runtime, such a fixed architecture will fail to provide optimal bandwidth and latency guarantees.

In this thesis, we propose ReFlex Switch to interconnect the HAs, the SoC processor on the FAC and the onboard DDR memory. ReFlex Switch is a Layer-2 switch architecture and it provides communication between coordinated HAs and with remote servers in the HACDC over Ethernet at a speed of 40 Gbps. It adopts architectural features from high-speed computer network and Network on Chip (NoC) routers. ReFlex Switch runs at line speed for scalability and features the Virtual Output Queue (VOQ) input buffer organization for computer network switches. VOQs are buffers per destination output at each input port different than the per packet buffers in the NoC applications.

This thesis makes the following main contributions:

CAR (Credit ARbiter), a novel on-chip switch arbiter with quality of service support

ReFBM (ReFlex Buffer Management), a flexible dynamic VOQ buffer allocation scheme

ReFlex Switch Architecture, a scalable and flexible on-chip switch that runs at line rate and can be instantiated with different parameters

The data that are received from the heterogeneous IP Cores on the FAC are stored in the respective VOQs. CAR provides service differentiation to these VOQs proportional to their respective specified bandwidths without decreasing the throughput of the fabric that runs at line speed. Our comparative performance evaluations show that CAR can achieve desired bandwidth allocations at the outputs. Furthermore, CAR achieves the same total throughput as well-known arbiters which are designed

for maximum throughput. To the best of our knowledge, CAR is the only fabric arbiter that provides differential service per connection while maintaining the maximal throughput, in the literature for computer networks and heterogeneous on-chip switches.

For on-chip switches, due to the limited amount of on-chip resources, queuing and buffer management present a certain resource-performance trade-off in addition to the choice of arbiter. Amount of block RAM resources dedicated to switch queues should be minimized and utilized as much as possible while also avoiding performance bottleneck issues such as Head of Line blocking. ReFBM provides dynamic and flexible memory allocation to the VOQs according to the arriving packet traffic by adopting an approach similar to a linked-list. The overall scalability is maintained by linking buffer segments that we call Flexible Buffer rather than packet or flit size memory. Our performance evaluations show that ReFBM results in better memory efficiency and hence better throughput compared to fixed size VOQ buffers under the same total amount of memory. To the best of our knowledge, ReFBM is the only on-demand memory VOQ allocation method in the packet switch literature.

ReFlex is a novel on-chip switch architecture that allows for different arbitration strategies at its core through a well-defined arbiter interface. This allows choosing different arbiters from a list to suit different needs, or even completely externalizing the decision for example to implement datacenter level accelerator policies. Performance can be improved with an arbiter that is aware of the application traffic patterns and bandwidth requirements. These parameters are user defined according to the available hardware resources, application specifications and desired performance providing reconfigurable and flexible hardware implementation. We provide detailed hardware synthesis results for Xilinx XC7Z100 SoC that demonstrate the resource consumption, achieved frequency and estimated power consumption for ReFlex Switch with different instantiation parameters. Our results show that CAR and ReFBM can be implemented to operate at 40 Gbps line speed with moderate increase in the utilized hardware resources in comparison to legacy arbiters and fixed size VOQ buffers.

We further develop a cycle accurate, event-triggered simulator in the scope of this thesis that is verified by comparing to RTL simulations. This simulator enables us to achieve the performance figures under different traffic specifications and large number of packets in reasonable run times.

The remainder of this thesis is organized as follows. In Chapter 2, we provide the fundamental preliminaries for packet switching and on-chip packet switches. We introduce the hardware accelerators as the motivating on-chip application for our ReFlex Switch. We present a detailed and comparative study for on-chip packet switch architectures focusing on relevant and recent work. We conclude the chapter by placing the contributions of this thesis work in the literature survey. Chapter 3 presents the three main contributions of this thesis. An algorithmic description of CAR arbiter, the ReFBM buffer management and a detailed hardware implementation of the entire ReFlex architecture are presented. We show the performance evaluation of CAR under different traffic scenarios and in comparison to legacy arbiters followed by the comparative evaluations of the ReFBM with different parameters in 4. Chapter 5 outlines the conclusions of this thesis and the theoretical and practical extensions of the ReFlex Switch that we plan in the future.

Figure 1.1 shows an example context in which the proposed ReFlex Switch resides for a Hardware Accelerated Cloud Datacenter.

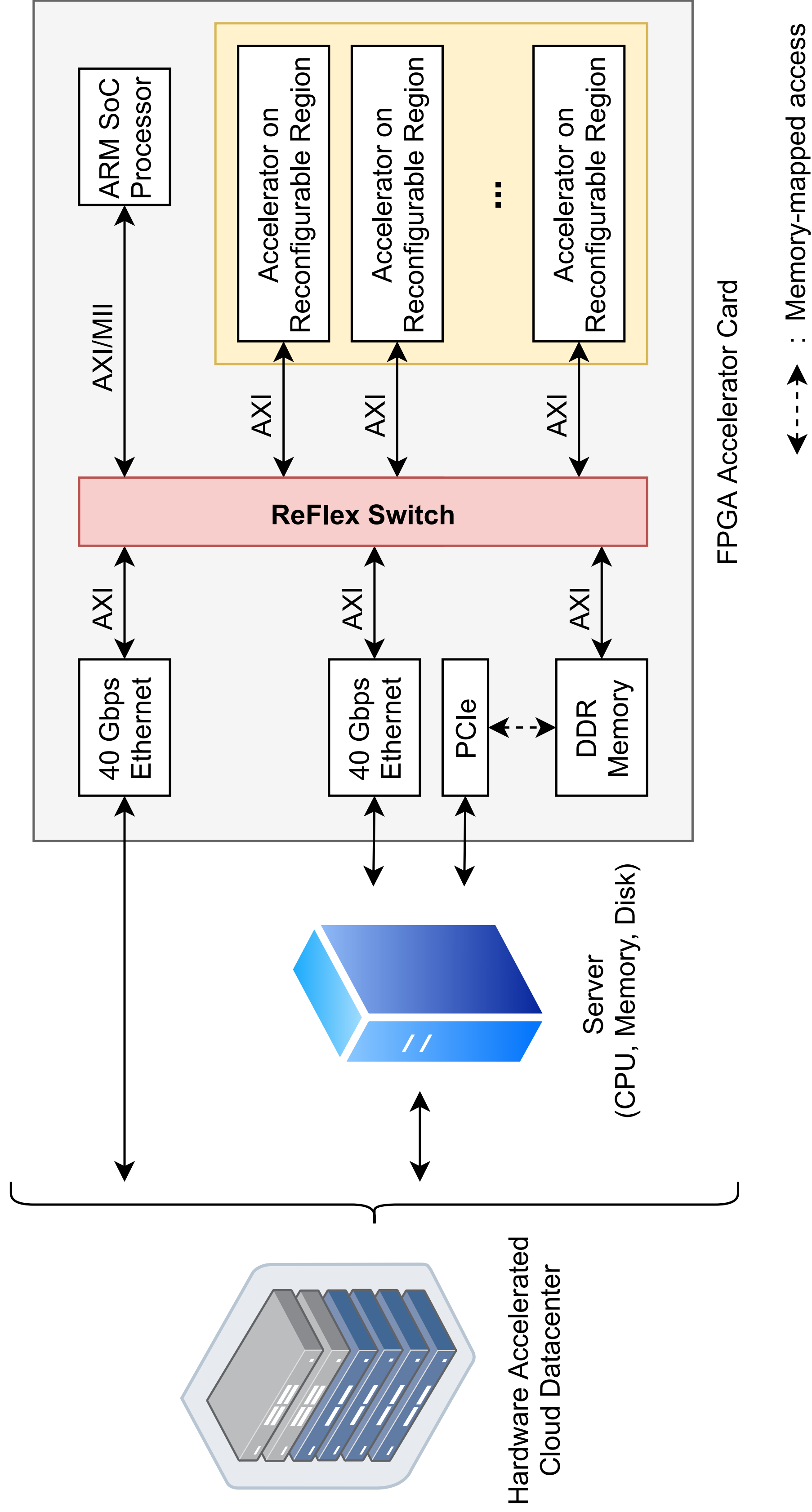


Figure 1.1: ReFlex Switch Architecture Example Context

CHAPTER 2

BACKGROUND AND PREVIOUS WORK

In this chapter, we provide an overview and preliminaries for packet switching buffer organization and operation in computer networks. In this context, we present the fundamental and relevant previous work for fabric arbiters in computer network switches. Next, we focus on the previous work for on-chip packet switches. To this end, we introduce the legacy *homogeneous* on-chip switches designed for multiprocessor systems. Finally we focus on the *heterogeneous* on-chip switches that are proposed for the recent System on Chip (SoC) applications with *hardware accelerators*.

2.1 Packet Switching

A distributed computer application consists of multiple *interconnected* end-systems that exchange messages to accomplish a certain task. Computer networking architectures including wide area computer networks and the Internet, embedded real-time computer networks or networks for cloud computing facilitate such message exchange at different formats and speeds. The dominating paradigm for computer networking is *packet switching* where the application data is broken into chunks and each chunk is packed with control information into a *packet*. These packets travel between the source and destination through communication links and packet switches. A *packet switch* receives a packet on one of its incoming ports and forwards that packet on one of its outgoing communication ports [11, 12]. The decision for the output port is according to a pre-configured forwarding table and the interconnection among the multiple input/output ports is established by a *switch fabric*. The routers that forward network layer IP packets in core networks and link-layer switches in

access networks are examples for packet switching devices.

2.1.1 Preliminaries

The legacy packet switch architecture for computer networks employs an NxN switch fabric that connects the set of inputs $I = \{0 \ldots N-1\}$ to the set of outputs $O = \{0 \ldots N-1\}$. Without loss of generality, one can assume that all input ports and output ports operate at the same line rate of C. The packets are segmented into fixed size units for synchronous hardware operation. The fixed size units are stored in the buffers at the input ports until they are switched. Output ports have buffers for the reassembly of the packets.

The switch fabric can interconnect the input and output ports using *time-division* by multiplexing the fixed size data units from different inputs and forwarding them through a data path connecting all inputs and outputs. Typical time-division switching structures are the shared-memory and the bus. The main advantages of this architecture are its low complexity and inherent support of broadcast and multi-cast. However, the full interconnection requires that the internal communication bandwidth is equal to $N \times C$, which is the aggregated forwarding bandwidth of all the input ports. This requirement limits the scalability of the switch in terms of N and C.

Space-division switch fabric is a solution to this scalability problem which consists of multiple data paths between the input and the output ports. Typical space division fabrics are crossbars as seen in Fig. 2.1 and multistage interconnection networks which arrange multiple such crossbar fabrics in regular topologies.

A space division fabric that runs at the line rate of C, can forward N fixed size units concurrently between input-output ports. Note that, the input-output pairs must be one-to-one connected. These one-to-one matches between i, j, $i \in I, j \in O$ are decided by an *Arbiter*. If there are more than one input ports with data to switch to a given output port, only one input port can be connected. The remaining input ports must buffer the data until it is switched. Such switch architecture is a *Pure Input Queuing Switch*.

It is possible to decrease the amount of data buffered at the input ports and the switch-

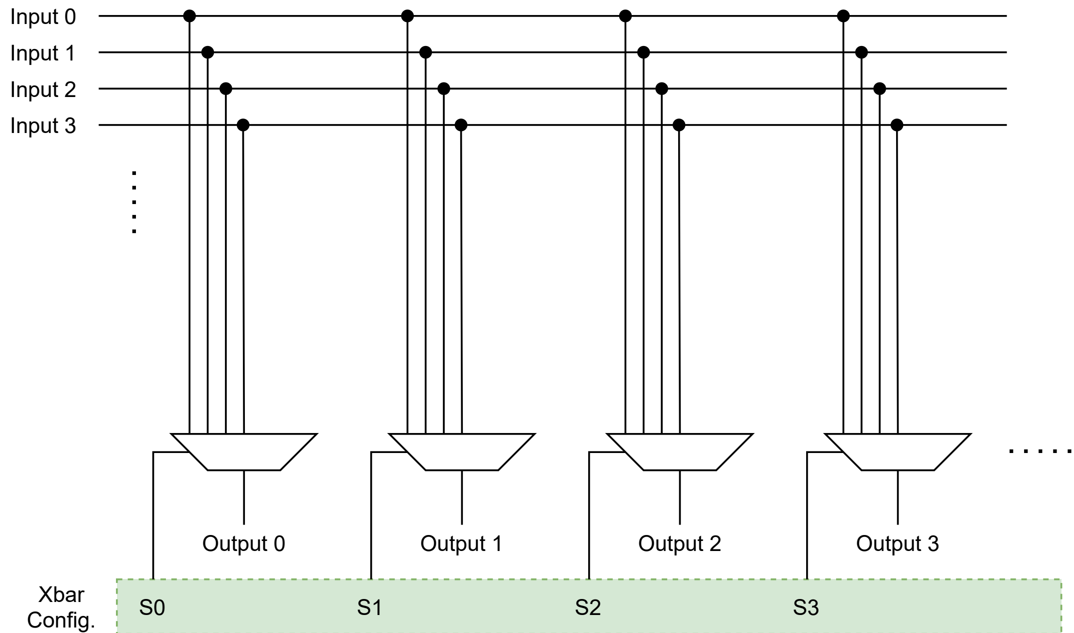


Figure 2.1: Crossbar Fabric

ing latency of the packets by running the fabric faster than the line rate. A space division fabric that runs at a speed of S C can switch S fixed size data units to an output port within the time to switch one data unit at the line rate. This speed-up of S trades off amount of buffering at the input and packet latency with cost and complexity of the hardware. Furthermore, additional buffers are necessary at the output ports. Selecting $1 < S < N$ leads to a *Combined Input/Output Queuing* switch with buffers at input and output ports. If the fabric operates at $S = N$ all buffers are at the output ports and the switch is called a *Pure Output Queuing Switch*.

We focus on scalable switching in this work hence we consider *crossbar switch fabrics that run at line speed*.

2.1.2 A Packet Switch with Crossbar Fabric: Buffer Organization and Operation

The fixed size data units for wide area computer network switches and routers are called *cells* after the ATM networks [12]. The segmentation of the packets into cells depend on the packet arrival times and packet lengths. Similarly, the reassembly of

the cells into packets depend on the time the packet is completed. A segmentation module in each input port and a reassembly module in each output port execute the respective function.

As a simple and basic design, it is possible to organize the input cell buffers with one FIFO queue per input port. This buffer organization faces the so called *head-of-line (HoL)* blocking problem. An example can be seen in Figure 2.2. Here, $i = 1$ and $j = 2$ cannot be connected because the Arbiter decided to connect $i = 3$ and $j = 2$. To this end, the Head of Line cell at input 1 is blocking the next cell that is destined to output $j = 1$. There are no other cells for $j = 1$, consequently it stays idle although there is a cell queued for it at input $i = 1$.

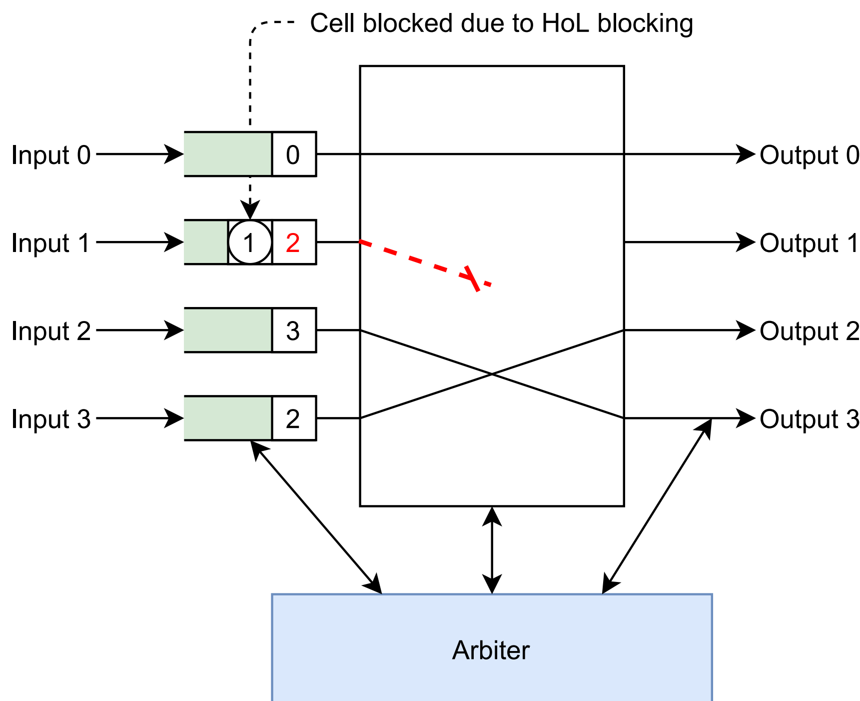


Figure 2.2: HoL Blocking Problem

The buffers at a given input port can be organized as dedicated separate queues that are called *Virtual Output Queues (VOQs)*. $VOQ_{i;j}$ at input i stores the cells to switch to an output port j to eliminate the head of line (HoL) blocking problem. VOQs statically partition the input buffer which might lead to the inefficient use of buffer space under non-uniform and fluctuating traffic among the destination ports [12]. A computer network switch with a crossbar fabric that operates at the line rate with

VOQ buffers is shown in Figure 2.3.

The synchronous switch operation with fixed size data units repeats *switching cycles*. Each switching cycle consists of *Arbitration*, *Configuration* and *Data* transfer stages which are executed in sequence. Accordingly, the arbiter decides the fabric configuration that connects the selected i, j pairs. Subsequently, the multiplexer select signals $S_0, \ldots, S_{N-1}$ in the crossbar fabric as shown in Figure 2.1 are generated. Finally, the fixed size units are read out from VOQ$_{i,j}$ at input i and written to the buffer at output port j.

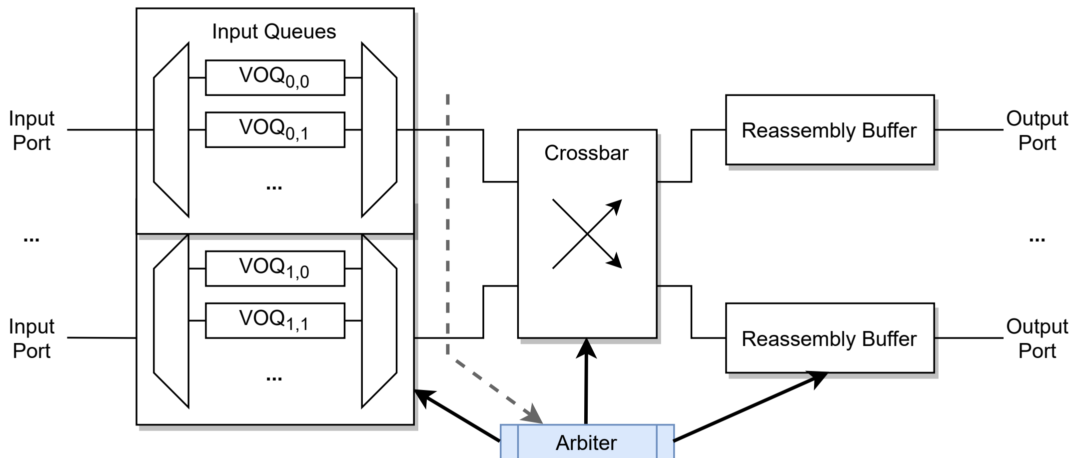


Figure 2.3: Buffer Organization

2.1.3 Fabric Arbiters

The fabric arbiter design affects the performance of the entire switch. To this end, first, the arbiter is required to provide high *throughput* by matching the largest number of i, j pairs. The arbiter runs at line speed every switching cycle so its *implementation complexity* must be low. Furthermore, the *latency* of the packets in the switch should be low and the VOQs should get *fair switching service*. The most basic requirement for this fairness is preventing the starvation of any VOQ. In addition, the application might require *switching service differentiation* among VOQs for Quality of Service (QoS) support.

The switch throughput is increased by matching the largest number of i, j pairs. A *maximal match* between inputs and outputs can be iteratively achieved when it is

no longer possible to match a new i, j pair without changing the existing matching [12]. We call an arbiter a *work conserving arbiter* if all i, j in a maximal match are switched.

[13, 14, 15, 16] propose arbiters that work with VOQs for computer network switches. [13, 14] maximize the throughput by exchanging control signals between inputs and outputs to iteratively compute a maximal one-to-one match. The prioritized variation of [13] implements $P \times N$ VOQs for P priority levels which increases the complexity and leads to a more fractured and less utilized memory organization. A weighted variation of [13] for grant issuing is briefly mentioned without elaboration or implementation. [16] allocates different bandwidths to connections by a weighted input-output mapping. However, [16] is not *work-conserving* and does not exceed the allocated bandwidth even if excess bandwidth is available. [17] proposes a weighted fabric scheduler targeted for optical switches with 100s of ports. The arbiter is N-stage pipelined which increases the latency. The arbitration is by a fixed length repeating schedule that is defined by a *frame*. The frame duration determines the bandwidth allocation granularity. The arbitration complexity increases with the number of ports N and the frame duration. No performance evaluation is reported.

[15] is a single iteration algorithm that improves the packet latency by considering the VOQ sizes without any service differentiation. They rank all VOQs at each input according to their queue sizes, where the longest VOQ has the highest priority in the arbitration. The performance evaluation is entirely simulation for a 64 port switch and shows that their algorithm achieves smaller latencies compared to [13] and other selected algorithms.

2.2 On-chip Packet Switches

Increasing computational requirements of applications such as multimedia or computational intelligence together with the advances in silicon technology resulted in the integration of an entire system onto the same silicon die that is called System-on-Chip (SoC). SoCs can incorporate cores including general-purpose fully programmable processors, co-processors, DSPs, dedicated hardware accelerators, memory blocks and I/O blocks. These cores exchange information by a packet-switched micro-network of interconnects, that is known as Network-an-Chip (NoC) architecture. NoCs can feature more than one switch that forward the packets of the cores on a multi-hop path [18]. The fixed size data unit in an NoC is called a *flit* which is switched or transmitted in one clock cycle.

There are two main approaches for packet switching in NoCs. The first one is *store and forward* where each switch fully receives the packet before forwarding to the next switch on the route. The second approach is *cut-through* switching which enables the switch start forwarding the header of the packet to the next switch after the required processing is completed while still receiving the packet data from the input port. *Virtual cut-through* (VCT) switching applies flow control at the packet level and the input buffer sizes are big enough to store entire packets which lose the arbitration. *Wormhole* switching applies the flow control at the flit level and it is possible that the packet is buffered in multiple switches on its route. Wormhole switching requires smaller buffer sizes than VCT switching, but its saturated throughput and worst-case latency is affected by the number of switches on the route and the traffic load whereas VCT outperforms wormhole particularly under heavy load [19].

Large number of homogeneous cores constitute a Multiprocessor system on a chip (MPSoC) for parallel and distributed applications such as neural network computations or multimedia processing. Such cores are called Processing Elements (PEs). On-chip switch architectures for MPSoC feature regular tile-based topologies [20, 21]. We call such switches as *Homogeneous On-Chip Switch*.

SoC applications that feature heterogeneous cores including hardware accelerators require on-chip switch architectures that support core-specific interfaces and data rates.

We call such switches as *Heterogeneous On-chip Switch*. Specific NoC Switches have similar designs and functionalities to legacy computer network switches. On the one hand, they can exploit the on-chip advantages as high speed interfaces and small latencies. On the other hand, they are bound by the on-chip resource constraints. The focus of this thesis is Heterogeneous On-chip Switches that are designed for interconnecting hardware accelerators to different interfaces with different bandwidth requirements.

2.2.1 Homogeneous On-Chip Switches

In such architectures each PE is connected to a dedicated on-chip switch. The packet is sent from source PE (PE_s) to the destination PE (PE_d). PE_s and PE_d are connected to switches SW_s and SW_d respectively. A *flow* is a flow of flits of *a single packet*. The routing between SW_s and SW_d is decided per packet. The homogeneous NoC switches are architecturally similar to the computer network switches. However, they are interconnected in a mesh topology with a small number of ports. As a typical configuration, each on-chip switch has $P = 5$ ports (N,E,S,W) and local PE [22] on a 2-D mesh.

Multiplexing the flits of different packets on the physical channels of the NoC is done by Virtual Channels (VC). A VC serves as an index to the NoC packet switching state with a VCID. There are distinct flit buffers at the NoC switch inputs per VC. The routing of the packet from SW_s to SW_d is done when its *head flit* is processed. Then VCs are allocated by connecting an input VC to an output VC in each switch as the head-flit progresses. To this end, a concatenation of VCID's for all flits of the packet is achieved. The flits are stored at the dedicated VC buffers at switch inputs on the route if the output port is not available. The *tail flit* de-allocates the VCs (VCIDs and buffers). We note that, the VCs are dynamically allocated and released *for each packet* different than the static, port-based VOQ organization [23].

The allocation of VCs to packets can be executed with different algorithms. A fundamental work by [24] proposes a canonical on chip router architecture for VC allocation. In this model, there are p_i input and p_o output ports with $p_i \; v$ number of input VCs and $p_o \; v$ number of output VCs. Each input VC can be connected to one of

p_o v output VCs according to the routing and VC availability which is selected by a p_o $v : 1$ arbiter. Then each output VC can be connected by one of p_i v input VCs selected by a p_i $v : 1$ arbiter. The legacy buffer organization is a fixed number of k flits per VC buffer. To this end, each input has k v flit buffers.

[22, 10, 25] are works on homoneous NoC switches that provide high-granularity access to avoid HoL Blocking rather than simply exploiting the output port information. [22] implements the NoC buffers as registers and assigns at most one packet to each VC as in the legacy implementation. Assuming small buffer space, they provide very fine grained individual flit access control to k v flit buffers. This buffer space can be organized in a range between v VCs (when each VC occupies the maximum of k flits) and v k VCs (when each VC occupies the minimum of 1 flit). This increases the input arbiter complexity by a factor of k despite decreasing the output arbiter complexity by v compared to [24]. The performance demonstration is by a cycle accurate simulator for a total 80 flit-buffer with 128 bit flits which is a small total buffer capacity. All simulations are performed in a 64-node (8x8) mesh network. Each switch has 5 physical ports including the PE port.

[10, 25] propose an architecture where they allow multiple packets that can be destined to different output ports to be stored in the same VC buffer resulting in HoL blocking problem per VC buffer. They introduce a data structure implemented in registers to keep pointers to certain packets in the VC buffer. If the HoL packet in a VC buffer cannot be switched another packet can be located in the same VC buffer by using the information in the registers. The register content must be updated on packet arrivals and departures. A four-port switch implementation is presented with a software simulator and there is no hardware implementation. 2-D mesh and flattened butterfly (FBFly) topologies with 64 nodes are considered. For the mesh, every terminal connects with one switch, and every switch is connected to four terminals for the FBFly. There are 4 VCs in every input port in the mesh and 16 VCs in the FBFly. Every VC has a buffer size of 8 flits in the mesh and 4 flits in the FBFly. Flit size is 64 bits.

[26] propose a three-stage Clos switch with speed-up of 2 to implement a Unidirectional NoC Crossbar (UDN) store and forward fabric. [27] employ [13] for the

fabric scheduling of Mesh NoC Switches. They implement the proposed architecture on Virtex 5 XC5VTX240T, FF1759,-2 device. A 4-port switch is implemented with 6197 LUTs, 4112 Registers at 255 MHz.

[28] suggests asynchronous on-chip switch structures that prioritize lines using static priority arbiters (SPA). The implementation is in the form of circuit design. Flit sizes change between 8 to 128 bits. The minimal router data cycle for an 8 bit flit is 4.5 ns. 5-port implementation is carried out. The operation has four stages. The first stage groups the incoming packets within each service level (SL) and maps them to VCs. Second stage is arbitration within the service level. Third stage is arbitration among service levels. The fourth stage forwards the selected packets to output.

[19] proposes switching between wormhole and virtual cut-through switching schemes at run-time. The RTL model of the switch was synthesized by using Xilinx Vivado Design Suite for Virtex-7 XC7VX485 FPGA device. The maximum frequency was 108MHz. The QoS mode of the switch defines fixed prioritization of certain packet types. Implementation is for a 4x4 switch with RTL modeling in VHDL. The network is 8x8 mesh network. The packet size is a maximum of 10 flits and the buffer is one packet long. The flit size is 35 bits. The total input buffer is 16 flits for 4 VCs. The resource consumption us 2899 Flipflops (%0.47), 3729 LUTs (%1.23), 352 Memory LUTs (%0.27). The throughput is 0.087 flits/cycle.

[8] proposes a memory access organization for efficient BRAM use and parallel VC and switch allocation for decreasing the latency. For small input buffer sizes DRAMs, for large input buffer sizes the BRAMs are used instead of the DRAMs to improve the performance. The architecture is strictly for homogenous on-chip switches and flit-level credit based VC flow control is adopted. The architecture consists of two stage allocation with virtual channel allocator and switch allocator similar to [22]. The arbiter can work in roundrobin and strict priority modes. The implementation is carried out on Xilinx Zynq 7000 ZC702SoC. The evaluation is done for 4x4 and 5x5 mesh topologies. Packet length is 4 flits. Buffer depth is 16 flits. For 256 bit flits 4095 Register RAMs, 176 LUTRAMs and 4 BRAMs are used for the FIFO buffer. For 128 bit flits, 2048 Register RAM's, 88 LUTRAMs and 2 BRAMs are used. For 16 VCs, 323 LUTs, 208 FFs are used. These numbers scale linearly by the flit size. For the

entire network with 5x5 mesh, 4 VCs per port, buffer depth of 4 and flit width of 64 bits, 90% of the LUTs, 23% of FFs, 75% of BRAMs are utilized. The throughput saturates at less than 40%. LBNoC NoC architecture operates at 205 MHz.

[29] dynamically changes the VC allocation datapath to reduce the delay of the arbiter if the number of active VC allocation requests is less than the total number of input VCs. Accordingly, the reduced datapath delay can be used to increase the frequency or decrease the power consumption. The implementation is by software simulation. They also implement the switch RTL model to extract its correspondent SPICE netlists using Synopsys Design Compiler. Detailed performance and power consumption values are calculated based on transistor level simulations. They assume 128 bit flits and 4 flit packets. The simulation is for an 8x8 mesh with 4 VCs per port and 4 flits per VC buffer.

2.2.2 Heterogeneous On-Chip Switches to Interconnect Hardware Accelerators

The switch architecture developed in this thesis is for interconnecting hardware accelerators implemented on reconfigurable regions on a single FPGA. We envision that these accelerators are offered as computation service by a cloud service provider. To this end, we first introduce architecture and operation of hardware accelerators to provide insight for our target applications.

The defining rules of hardware architectures for computing have been Moore's Law [30] and Dennard's Scaling [31] for decades. These rules are gradually losing their relevance as in 2018 there was roughly a 15-fold gap between Moore's prediction and the current capability. Furthermore, the prediction of constant power per mm^2 of silicon stated in Dennard scaling does not hold anymore since 2012.

[32] propose Domain-specific architectures (DSA) which are "tailored to a specific problem domain and offer significant performance (and efficiency) gains for that domain" to continue improving performance and energy efficiency. DSAs accelerate a component of an application and achieve performance improvement when compared to executing the entire application on a general-purpose CPU. To this end, they are

often called *accelerators* and they include graphics processing units (GPUs), neural network processors used for deep learning or IP Cores on FPGA's. GPUs' performance gain is very significant for intensive Floating Point calculations. FPGAs provide significant performance gain over CPU when using operands with custom data widths, for combinational logic problems, finite state machines and parallel MapReduce problems [33]. Machine learning and video processing problems, where FPGAs are competitive on energy efficiency compared to CPUs and GPUs employ Integer Arithmetic, Hamming Distance, KNN voting, dot product, scalar multiplication, vector addition, and Binarized 2D Convolution compute kernels [6]. Furthermore, sort and search computations can be accelerated by parallel networks that can be efficiently realized on FPGA [34]. FPGA accelerator realization and interfacing with the CPU is streamlined. One possible workflow for Xilinx platforms is the OpenCL computing model, where one or more compute kernels can be implemented on the FPGA. The data to be processed is first transferred from the host memory to the global FPGA DDR memory. Then, the CPU triggers the compute kernels on the FPGA to start processing. The kernel reads the data from the device memory, processes it, and writes the results back to the device memory. If there are more than one kernels they can exchange data. Then, the results are transferred from the FPGA global memory to the host memory and are available for the CPU [35, 34].

Hardware Accelerated Cloud Data Centers (HACDC) offer hardware accelerators (HA) as computing resources in addition to memory, processor (CPU) and disk [36, 2]. To this end, FPGA Accelerator Cards (FAC) can be installed in the servers or FACs with an SoC processor can be directly connected to the HACDC network. Multiple HAs can be instantiated on partially Reconfigurable Regions (RR) on the same FPGA [3, 37, 38]. Such organization requires high-speed data exchange among FAC components and *Quality of Service (QoS)* support for both satisfying the requirements of the applications and enabling bandwidth allocation for the Virtual Machines [7]. This data exchange is provided by a custom design on-chip packet switch with heterogeneous interfaces in [39, 38, 2, 40, 37, 41, 42].

[39] is an early work which proposes a packet switch realized on a chip with four PCIe Gen2 ports, each of which has eight lanes reaching 4 Gbytes/sec data rates to interconnect accelerators on different nodes. The switch is realized on Altera's Stratix

IV GX FPGA, which can provide four PCIe Gen2 x8 modules.

[38] calls the RRs virtual FPGAs (vFPGAs). A vFPGA can be configured with a compatible partial bitstream to implement a particular accelerator and more than one accelerators on a RR is possible. All vFPGAs have an AXI4-Stream interface to the PCIe core, an AXI4-Stream interface to external DRAM and two other stream interfaces to connect to adjacent vFPGAs. Every stream interface to/from a vFPGA is integrated with the rest of the FPGA fabric through AXI4-Stream based asynchronous FIFOs, which enable the logic within the vFPGAs to run at a different clock frequency from the interface. These physical interfaces are managed by a switch that is implemented in the FPGA static logic and it is not reconfigured during FPGA uptime. This switch realizes a round robin arbitration among vFPGAs and gives them fair access to the FPGA board. vFPGAs can also be prioritised with higher bandwidth if application needs are different. The switch also implements independent DMA controllers to manage data transfer between each vFPGA and the server. The whole system is implemented on Xilinx VC709 FPGA board containing a XC7VX690T FPGA, supporting PCIe Gen 3×8, and hosting 8GB of on-board memory. The hardware logic for PCIe and DRAM communication and reconfiguration management consumes about 7% of the FPGA area and is implemented statically.

[2] propose the Elastic Router (ER) to support intra-FPGA communication between accelerators implemented on the RRs on the same FPGA and inter-FPGA communication between RRs running on other FPGAs. The interfaces of the ER are PCIe DMA, RR (Accelerator), DRAM, and Remote (to custom Lightweight Transport Layer connecting to 40 Gbps Ethernet Interface). The ER is an input buffered switch which supports multiple virtual channels (VCs). The ER employs credit-based flow control, one credit per flit, that allows a pool of credits to be shared among multiple VCs. The whole design is implemented on Altera Stratix V D5, with 172.6K ALMs of programmable logic. The FPGA has one 4 GB DDR3-1600 DRAM channel, two independent PCIe Gen 3x8 connections for an aggregate total of 16 GB/s in each direction between the CPU and FPGA, and two independent 40 Gb Ethernet interfaces with standard QSFP+ connectors. ER uses 3449 ALMs which is 2% of the area and runs at 156 MHz.

[40] describes the on-chip switch for the accelerators on FPGA as a combination of input and output module. The input module receives the Ethernet packets sent to the FPGA. These packets can come from the outside cloud servers from the Ethernet interface of the FAC or from other FPGA kernels. The packets also carry a two-byte address for the specific destination kernel. The whole design is implemented on a Xilinx Virtex 7 XC7VX690TFFG-1157 FPGA (433200 LUTs, 866400 Flip Flops, 1470 BRAM Tiles, 36Kb each). The implementation board has two 8GB ECC-SODIMM for memory speeds up to 1333MT/s and Dual SFP+ cages for high speed optical communication including 10 Gigabit Ethernet. The FPGA hypervisor module for virtualization has a 1 GbE core which actually limits the throughput of the system to 1 Gbps. The input module consists of an Input bridge that converts the Ethernet frame to an AXI Stream packet and an input demultiplexer that directs the data to the correct FPGA kernel. The input Bridge uses 87 LUTs (0.02%), 170 FlipFlops (0.019%) and 2 BRAMs (1.36%). The Input Multiplexer has 16 outputs. It uses 82 LUTs (0.019%) and 124 FlipFlops (0.014%). The output module receives data from the user kernels and the input demultiplexer. The output module consists of Ethernet FIFO controller, Packet formatters which convert the data to a proper Ethernet frame with the correct MAC address and an output switch. Output module switch has 16 inputs. It uses 517 LUTs (0.119%) and 138 FFs (0.016%). The packet formatters each use 230 LUTs (0.053%), 252 Flipflops (0.029%) and 2 BRAMs (1.36%) for each kernel (accelerator). The Ethernet FIFO controller uses 26 LUTs (0.006%), 12 FlipFlops (0.014%) and 2 BRAMs (1.36%). The system runs at 125 MHz.

[37] proposes an OS view of the abstractions between the CPU and the FPGA components that are proposed as vPGAs similar to [38]. The interconnection among vFPGAs, the custom partial reconfiguration controller, CPU (over PCIe), DRAM and Network Interface is provided by crossbar switches. However no implementation details are provided for these switches.

In our previous work [41, 42], we propose a packet switch architecture that we call ACCLOUD-SWITCH (ACcelerated CLOUD Switch) that is implemented on FPGA in the scope of our proposed architecture for offering hardware accelerator as a cloud service [3]. ACCLOUD-SWITCH interconnects the hardware modules on the FAC including accelerators realized on RRs, the SoC processor, DRAM and 40Gbps Eth-

ernet with the cloud server over PCIe. ACCLOUD-SWITCH features an 8x8 crossbar fabric that works at line speed of 40Gbps. It has fixed size VOQs and a FIFO buffer infront of the VOQs to serve as an extension of a VOQ that can be full for better memory utilization. The fabric arbiter provides the VOQs access to the switch fabric that is proportional to bandwidth allocation for QoS support. The flit size is 256 bits. 1% of the packets are 40 Bytes and the remaining packets are 1500 Bytes. Total memory is 640 flits. ACCLOUD-SWITCH is implemented for Xilinx XC7Z100 SoC running on single 156.25 MHz clock using Xilinx Vivado 2016.4. The selected SoC has available 277400 LUTs, 554800 FFs and 755 BRAMs (with 36 Kb capacity). The resource consumption is approximately 24k LUTs, 45k FFs and 224 BRAMs.

2.3 Contribution and Placement of the Thesis Work in the Literature

This thesis work proposes the ReFlex Switch which is a heterogeneous on-chip architecture that interconnects a number of modules with different high-speed network interfaces. ReFlex Switch is designed to work at line speed to scale with the increasing data rates.

The main contributions of this thesis are; a novel packet switch fabric arbiter (Credit based Arbiter-CAR) with QoS support, a novel input buffer management for an on-chip packet switch (ReFBM) and a very detailed description for the hardware architecture of the entire proposed on-chip packet switch (Reflex Switch). Reflex Switch can be flexibly instantiated on FPGA with desired buffer and arbiter configurations that are specified in terms of parameters. The performance evaluations of ReFlex switch are carried out on a cycle accurate simulator that is developed in the scope of this thesis and verified for timing accuracy with an RTL simulator.

CAR provides fabric access to VOQs proportional to their respective specified bandwidths without decreasing throughput different than [16]. CAR arbitration is not a fixed schedule different than [17] and takes place in every switching cycle according to the flits stored in the VOQs. [28],[29],[8] are homogeneous on-chip switches

which apply per-packet flow control in the form of Virtual Circuits (VC) with very small buffer sizes. They all apply static priority assignment to the VCs.

To the best of our knowledge, CAR is the only work conserving arbiter that provides differential service per connection in the literature for computer networks and heterogeneous on-chip switches.

ReFBM provides dynamic and flexible memory allocation to the VOQs according to the arriving packet traffic. [22] proposes allocating different VC buffer sizes per packet in the expense of increased arbiter complexity and for very small buffer sizes.

To the best of our knowledge, ReFBM is the only on-demand memory VOQ allocation method in the packet switch literature.

Reflex Switch architecture is a heterogeneous on-chip packet switch architecture with configurable hardware design parameters. We present all hardware design details of Reflex Switch in this thesis.

[39, 38, 2, 40, 37] focus on the system design which incorporate an on-chip switch however they do not provide the hardware design details. [2] states that the system parameters are configurable however there is no description of the method.

To the best of our knowledge, Reflex Switch is the only parametric configurable on-chip packet heterogeneous packet switch architecture that is specified in all detail.

Mentioned previous studies are summarized in Tables 2.1, 2.2, 2.3 and 2.4.

Table 2.1: Summary of Related Previous Work - Homogeneous (part 1)

Source, Year	Goal	QoS Support	Flit Size (bits)	Buffer Size (*) and Allocation	Switch Parameters	Platform
[22], 2006	Efficient memory use	None	128	80 flits, per packet VC	8x8 mesh, 5 ports per switch	Software simulator
[28], 2009	QoS	Static Priority	8-128	-, per packet VC	5 port switch	ASIC simulator
[10, 25], 2018	Efficient memory use	None	64	32 flits, multi-packet VC	8x8 mesh, 4 ports per switch	Software simulator

(*) Buffer size per port.

Resource usage, frequency and throughput were not reported in these studies.

Table 2.2: Summary of Related Previous Work - Homogeneous (part 2)

Source, Year	Goal	QoS Support	Flit Size (bits)	Buffer Size (*) and Allocation	Switch Parameters	Platform	Resource Usage	Freq.	Tput.
[19], 2019	Power-throughput trade-off	Static Priority	35	16 flits, per packet VC	8x8 mesh, 4 ports per switch	Xilinx Virtex-7 XC7VX485	2899 FF (0.47%), 3729 LUT (1.23%), 352 LUTRAM (0.27%)	108 MHz	0.087 flits / cycle
[29], 2019	Power-latency tradeoff	None	128	16 flits	8x8 mesh, 4 ports per switch	Software simulator			
[8], 2020	Efficient memory use	Static priority	64-256, 64 reported	16 flits, per packet VC. DFF for small input buffer sizes, BRAM for large	5x5 mesh, 4 VCs per port, buffer depth 4	Xilinx Zynq 7000 ZC702SoC	90% LUT, 23% FF, 75% BRAM	205 MHz	<40%

Freq.: Frequency, Tput.: Throughput.

(*) Buffer size per port.

Table 2.3: Summary of Related Previous Work - Heterogeneous (part 1)

Source, Year	Goal	QoS Support	Platform	Resource Usage	Freq.	Tput.
[38], 2015	Interconnect acc. on RRs, DRAM, CPU (PCIe)	Round robin or prioritization for acc. to access DRAM or PCIe	Xilinx VC709 FPGA board with XC7VX690T+8xPCIe	7% including reconfiguration controller		
[2], 2016	Interconnect acc. on RRs, DRAM, PCIe DMA, 40 Gbps eth.	None	Altera Stratix V D5 with 172.6K ALMs. 4GB DDR3-1600 DRAM, 2 x PCIe	3449 ALMs (2% of area)	156 MHz	
[40], 2017	Interconnect eth. and acc.	None	XC7VX690TFFG - 1157 FPGA, 433200 LUT, 866400 FF, 1470 x 36kb BRAM	66k LUT, 81k FF, 264 x 36kb BRAM (for 16 streams)	125 MHz	1 Gbps

Freq.: Frequency, Tput.: Throughput, eth.: ethernet, acc.: accelerator.

Flit size, buffer size, allocation and switch parameters were not reported in these studies.

Table 2.4: Summary of Related Previous Work - Heterogeneous (part 2)

Source, Year	Goal	QoS Support	Flit Size (bits)	Buffer Size (*) and Allocation	Switch Parameters	Resource Usage	Freq.	Tput.
ACCLOUD-SWITCH, [41], 2020	Interconnect acc. realized on RRs, SoC CPU, DRAM, 40 Gbps eth., PCIe.	Arbitration proportional to allocated bandwidth.	256	80 flits, fixed size VOQs	8x8 crossbar	24k LUT (8.7%), 45k FF (8.1%), 224 x 36kb BRAM (30%)	156.25 MHz	40 Gbps
ReFlex Switch, [43], 2021	Interconnect acc. realized on RRs, SoC CPU, DRAM, 40 Gbps eth., PCIe.	Arbitration proportional to allocated bandwidth.	256	64 flits, 16 FB ReFBM	8x8 crossbar	17k LUT (6.1%), 6k FF (1.1%), 72 x 36kb BRAM (9.5%) (8 ports, CAR+16FBs)	156.25 MHz	40 Gbps

Freq.: Frequency, Tput.: Throughput, eth.: ethernet, acc.: accelerator.

(*) Buffer size per port.

Platform common to both of these studies is: Xilinx XC7Z100 SoC, 277400 LUT, 554800 FF, 755 x 36kb BRAM

ACCLOUD-SWITCH [41] and ReFlex Switch [43] are both part of this thesis work.

CHAPTER 3

REFLEX SWITCH

ReFlex Switch has an $N \times N$ crossbar fabric with VOQ buffer organization. We present a detailed architecture diagram of ReFlex Switch as shown in Figure 3.1. Without loss of generality, we assume that all input ports, output ports and the fabric operate at the line rate of C. To this end, ReFlex Switch is an input buffered switch with a fabric speed of 1 and the output ports only have reassembly buffers (RBs). A *connection* $c_{i,j}$ is the flow of flits between input port $i \in I = \{0 \dots N-1\}$ and output port $j \in O = \{0 \dots N-1\}$. The flits of $c_{i,j}$ are buffered in $VOQ_{i,j}$ at input i until they get access to the switch fabric.

In this work, we present a new QoS Arbiter, the Credit Arbiter (CAR) and a new VOQ buffer management, ReFlex Buffer Management (ReFBM) that can be implemented in the ReFlex Switch.

The synchronous operation of the ReFlex Switch with fixed size data units repeats *switching cycles*. Each switching cycle consists of *Arbitration*, *Configuration* and *Data* transfer stages which are executed in sequence. The Arbiter block decides the fabric configuration that connects i, j pairs.

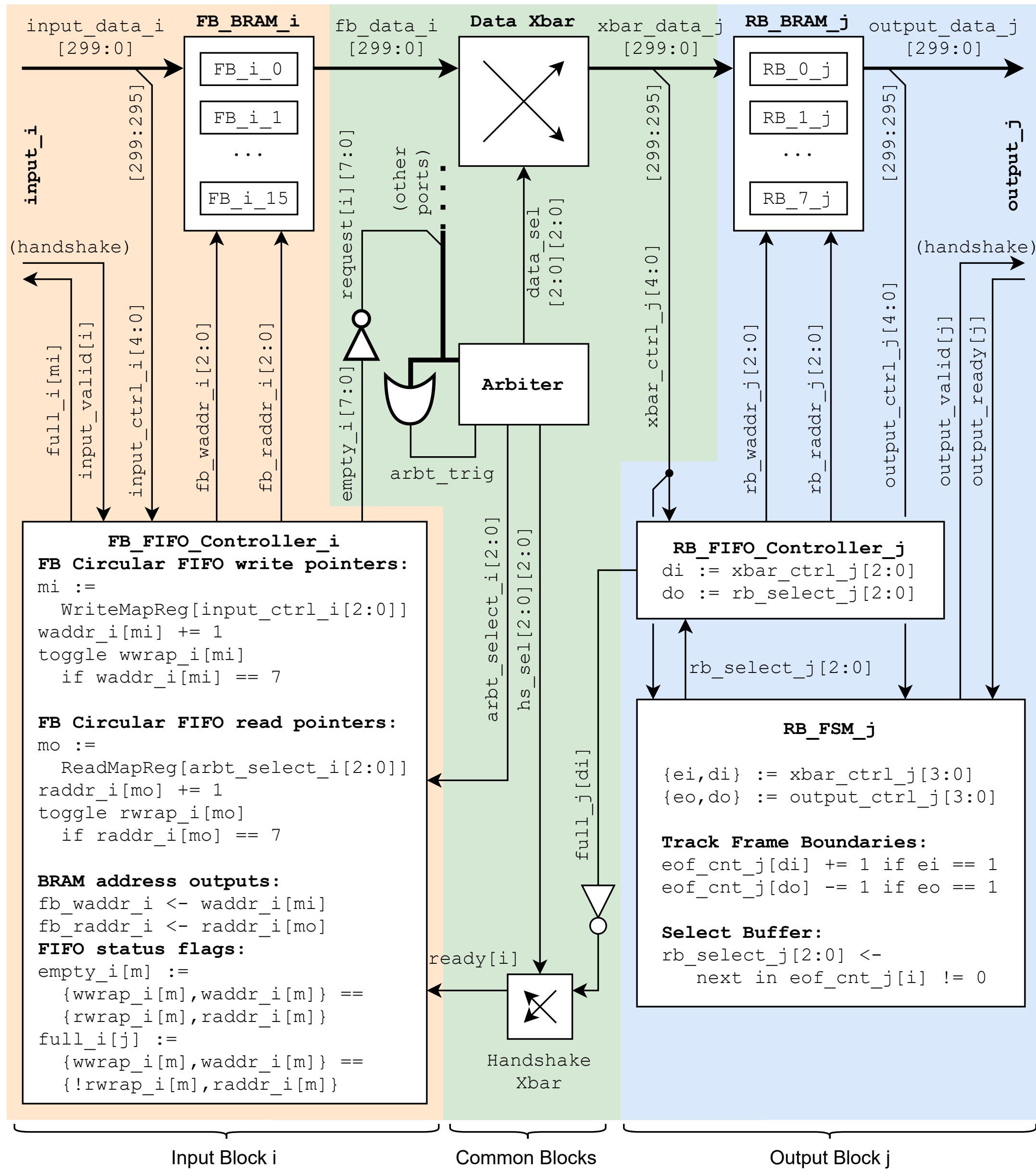


Figure 3.1: ReFlex Switch - Architecture and Memory Organization (instantiation with $N = 8$ and $M = 16$)

3.1 CAR (Credit Arbiter)

We assume that each connection has an *allocated minimum rate* ${}_{i,j}$. If there is no connection between a given input-output pair $p \in I, q \in O$, then ${}_{p,q} = 0$. Here we note that CAR can support a Best Effort connection between a given input-output pair p, q, by defining a small allocated rate ${}_{p,q} = {}_{be}$. We denote the set of outputs for input i with ${}_{i,j} > 0$ as O_i and the set of inputs for output j with ${}_{i,j} > 0$ as I_j.

We assume that there is a connection admission procedure that guarantees the following:

$$\forall i \in I, \quad \sum_{j \in O_i} {}_{i,j} \quad C, \tag{3.1.1}$$

$$\forall j \in O, \quad \sum_{i \in I_j} {}_{i,j} \quad C. \tag{3.1.2}$$

We propose a new work-conserving switch fabric arbiter that we call CreditArbiteR (CAR) which provides *proportional bandwidth allocation* for each input i and output j by *switching service differentiation*. CAR works in parallel for input/output ports. To this end, it is scalable and compatible with the architecture in Figure 3.1.

CAR is a three step arbitration similar to [13]. This three-step arbitration is repeated until it is a maximal match where it is not possible to match new i, j pairs, or a preset number of iterations, $MaxIters$, is reached to maximize the throughput. The i, j pairs that are *matched* are marked. Following the fabric configuration the fixed size flits of the matched connections $c_{i,j}$ are switched completing the switching cycle.

The arbitration control information is exchanged using shared registers $Request$, $Grant$ and $Accept$ between the inputs and outputs. In the first step, each input i with a non-empty $VOQ_{i,j}$ sets $Request(i, j) \quad 1$. In the second step, for each output j, the arbiter selects an input i with $Request(i, j) = 1$ to match with j and sets $Grant(i, j) \quad 1$. In the third step, for each output i, the arbiter selects an output j with $Grant(i, j) = 1$ and sets $Accept(i) \quad j$.

CAR achieves service differentiation by the prioritization of the granted inputs and accepted outputs. To this end, input i has the highest grant priority for output j for consecutive $G_{i,j}$ switch cycles and output j has the highest accept priority for consecutive $A_{i,j}$ cycles. We call $G_{i,j}$ and $A_{i,j}$ *grant credit* and *accept credit* for connection $c_{i,j}$ and determine them according to (3.1.3) and (3.1.4) to maintain the proportional service differentiation among connections. Here, note that it is possible to set different bandwidth shares to connections at the input and output ports by independently selecting $G_{i,j}$ and $A_{i,j}$ which is not in the scope of this study.

$$\frac{G_{i,j}}{\sum_{i \in I_j} G_{i,j}} = \frac{{}_{i,j}}{\sum_{i \in I_j} {}_{i,j}}; \tag{3.1.3}$$

$$\frac{A_{i,j}}{\sum_{j \in O_i} A_{i,j}} = \frac{{}_{i,j}}{\sum_{j \in O_i} {}_{i,j}}; \tag{3.1.4}$$

We present the operation of CAR in Algorithm 1. $p_G(j)$ and $p_A(i)$ are grant and accept pointers for output $j \in O$ and input $i \in I$. At a given switching cycle, $c_{i,p_A(i)}$ is the highest priority connection for input i and $c_{p_G(j),j}$ is the highest priority connection for output j. CAR tracks the switching service received by inputs to output j and the switching service received by outputs from input i with the *grant credit counter* $k_G(j)$ and *accept credit counter* $k_A(i)$. To this end, $k_G(j)$ and $k_A(i)$ are initialized with $G_{p_G(j),j}$ and $G_{i,p_A(i)}$ respectively and are decremented with every data transfer for $c_{i,p_A(i)}$ and $c_{p_G(j),j}$.

Different than [16], CAR is a work conserving algorithm. If the highest priority input does not have a flit for output j ($Request(p_G(j), j) = 0$) then an alternative input which has a flit for j is granted. Similarly if $Accept(p_A(i)) = 0$ then an alternative output that has a grant for i is accepted. These alternative inputs and outputs are selected by the Shuffle function irrespective of their credits. Here, Shuffle(S) represents a pseudo random permutation of the elements of a given set S and guarantees that the arbiter checks all alternative ports in a non-repeating, random order. Such alternative port selection is more likely under light loads where the contention for the fabric is low. As our results show in Section 4 all $c_{i,j}$ achieve ${}_{i,j}$ with a 100% throughput.

Such alternative port selection is unlikely at high loads.

When $k_G(j)$ decrements to 0, $p_G(j) \leftarrow i_{next}$ and $k_G(j) \leftarrow G_{i_{next},j}$ as presented in Algorithm 2. Similarly, when $k_A(i)$ decrements to 0, $p_A(i) \leftarrow j_{next}$ and $k_A(i) \leftarrow A_{i,j_{next}}$. CAR determines i_{next}, j_{next} with the $\mathsf{Random}(a, b)$ function which represents a pseudo-random integer in the closed range $[a, b]$. To this end, a long term fair allocation of the switching capacity among the connections is achieved.

We note that the $\mathsf{Random}(a, b)$ and $\mathsf{Shuffle}(S)$ operations need to be executed in a single cycle. $\mathsf{Random}(a, b)$ can be realized with a sufficiently sized pseudo-random number generator, such as an LFSR (Linear Feedback Shift Register). The default LFSR we use is 16-bit Fibonacci LFSR with feedback polynomial $x^{16} + x^{14} + x^{13} + x^{11} + 1$. The output space is equally divided to $N - 1$ partitions.

We adapt a loopless iterative permutation generation [44] to implement $\mathsf{Shuffle}(S)$. To this end, we keep $O(n \log n)$ state variables where $n = N - 1$ is the number of elements in the set S. These state variables hold the current permutation with the elements of S, the same permutation in terms of the index in factoradic terms [45], the next permutation and the direction where each element will move in the next iteration. This way, the permutation generation is achieved in single cycle with reasonable logic and space cost.

An example execution of the CAR algorithm can be seen in the Appendix A.

```
Initialization:
p_G(j) ← 0, k_G(j) ← G_{0,j}
p_A(i) ← 0, k_A(i) ← A_{i,0}
for t ← 0 to MaxIters do
    Request Phase:
    Request(i, j) ← 0
    for (i, j), i, j ∈ 0 ... (N − 1) do
        if i and j not matched and i has flit for j then
            Request(i, j) ← 1
    Grant Phase:
    Grant(i, j) ← 0
    for j ∈ 0 ... (N − 1) do
        for i' ∈ [0, Shuffle(1 ... (N − 1))] do
            i := (i' + p_G(j)) mod N
            if Request(i, j) = 1 then
                Grant(i, j) ← 1
                break
    Accept Phase:
    Accept(i) ← EMPTY
    for i ∈ 0 ... (N − 1) do
        for j' ∈ [0, Shuffle(1 ... (N − 1))] do
            j := (j' + p_A(i)) mod N
            if Grant(i, j) = 1 then
                Accept(i) ← j
                mark i and j as matched
                update credits for i, j
    if no new connections then
        break
Connect accepted connections: i → Accept(i)
```

Algorithm 1: CAR Iteration Pseudocode

Update credits for $i; j$**:**

if $k_G(j) > 1$ **then**

  $k_G(j) \quad k_G(j) \quad 1$

else

  $i_{next} := (i + \text{Random}(1; N \quad 1))$ **mod** N

  $p_G(j) \quad i_{next}$

  $k_G(j) \quad G_{i_{next};j}$

if $k_A(i) > 1$ **then**

  $k_A(i) \quad k_A(i) \quad 1;$

else

  $j_{next} := (j + \text{Random}(1; N \quad 1))$ **mod** N

  $p_A(i) \quad j_{next}$

  $k_A(i) \quad A_{i;j_{next}}$

Algorithm 2: CAR Credit Update Pseudocode

3.2 ReFBM (ReFlex Buffer Management)

The buffer memory is allocated in F-flit *buffer segments* that we call *Flex Buffer* (FB). Let the entire buffer memory of an input port i be B flits. Then, one can choose an FB size of $F = B/M$ flits where $M = N + D$ with $D \geq 0$. We index the FBs at port i with $FB_{i,m}$, where $m = 0, \dots, M - 1$. At any given time, each $VOQ_{i,j}$ consists of $VOQ_{i,j,k}$ FBs that are organized as a linked list, with $1 \leq k \leq D + 1$ FBs and there are $0 \leq Q_i \leq D$ free FBs that can be dynamically allocated to any VOQ. Here, we would like to note that the buffer organization is the legacy VOQ if $D = 0$ allocating each VOQ a fixed number of B/N flits.

Figure 3.2 shows the organization of buffers in detail. The indices of the head and tail FBs for each $VOQ_{i,j}$ are stored in Write Map and Read Map registers respectively. ReFlex Buffer Manager at input i (RBM_i) writes the arriving flit with destination j in the tail FB of $VOQ_{i,j}$. If the tail FB of $VOQ_{i,j}$ is full and $Q_i > 0$, then RBM_i allocates the memory and links a free $FB_{i,m}$ to $VOQ_{i,j}$ as the new tail FB, updates the Write Map Register and $Q_i \leftarrow Q_i - 1$. RBM_i reads out the next flit selected by the arbiter to switch from the head FB of $VOQ_{i,j}$. If $VOQ_{i,j,k} > 1$ and the head FB becomes empty, then RBM_i frees the head FB, updates the Read Map Register and $Q_i \leftarrow Q_i + 1$. Note that if $VOQ_{i,j,k} = 1$ the empty FB stays allocated to $VOQ_{i,j}$. In Figure 3.2, the empty and full flags are also illustrated. These signals are critical for bus handshaking and for providing the arbiter stages with current requests status.

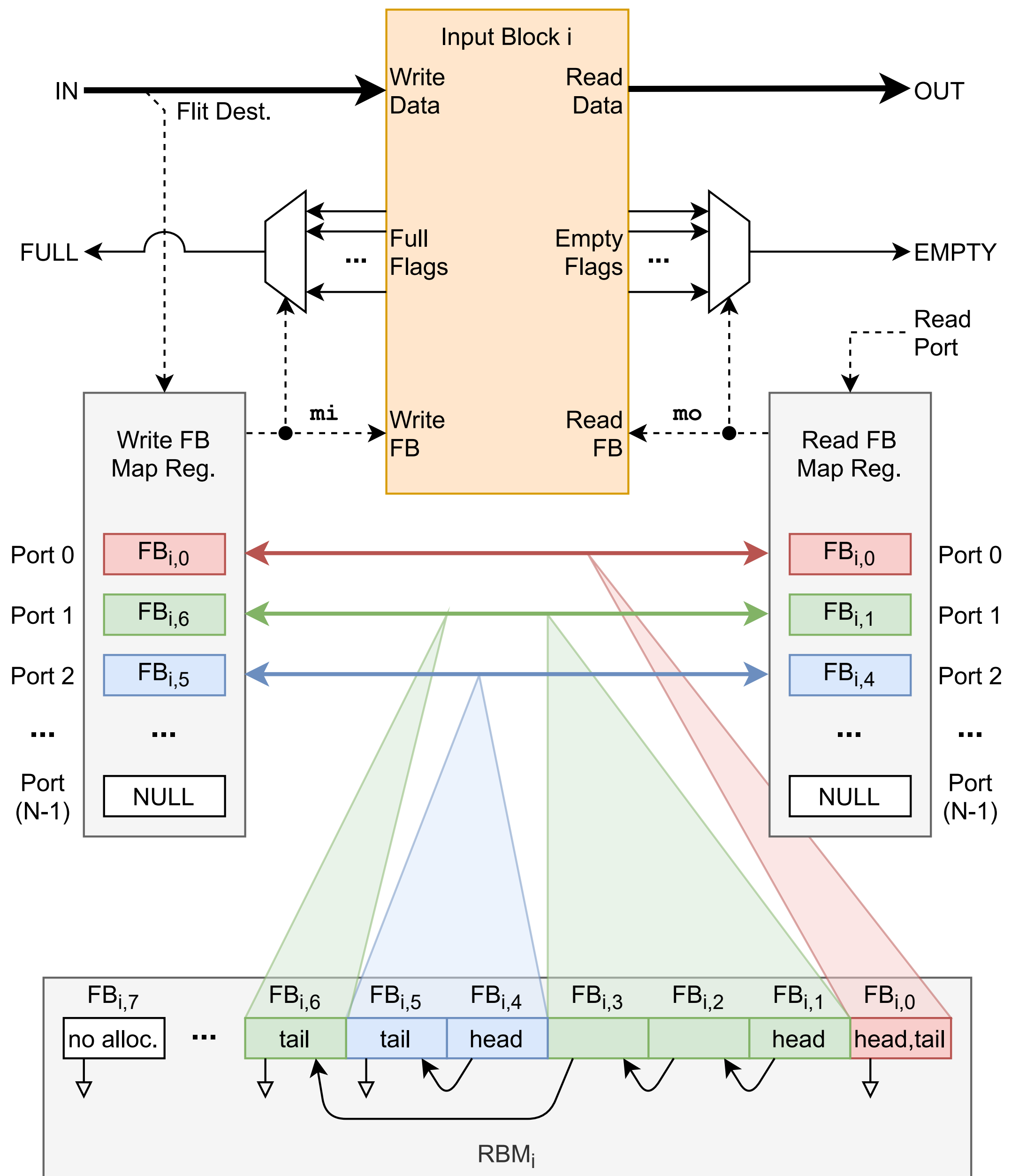


Figure 3.2: *Flex Buffer* Allocation for Resource Efficient FIFO Organization

3.3 ReFlex Switch Hardware Architecture

The detailed hardware architecture of ReFlex Switch is presented in Figure 3.1. This presented architecture is flexible. The automatic generation of synthesis-ready RTL code for the ReFlex switch IP core is detailed in Section 3.3.5.

The Arbiter Block in Figure 3.1 can be CAR or can be selected among other VOQ switch arbiters which exchange control signals between inputs and outputs including [13, 14, 16]. The pipelined switching cycles of ReFlex Switch is explained in Section 3.3.1. The Arbiter decision consists of the new crossbar forward and reverse mappings and two bit vectors enabling the bound ports. The forward mapping is from inputs to outputs and it is used for transferring the data. The reverse mapping is from outputs to inputs and enables transfer of handshake signals back from the targeted buffers. We exploit the on-chip implementation of ReFlex Switch and merge the request collection and result writeback latencies into the corresponding iterations first and last cycles in the pipeline respectively. This reduces each iteration to two clock cycles, and the operation is detailed in Section 3.3.3.

The VOQ organization is by ReFBM which can be configured to the legacy fixed size VOQ arrangement by setting $D = 0$. To this end, Fig. 3.1 shows an instantiation of the ReFlex Switch with $N = 8$ and $M = 16$. The implementation of input and output side buffers are explained in Section 3.3.2.

Section 3.3.4 demonstrates the Configuration and Status Registers implemented in ReFlex Switch.

3.3.1 Pipelined Switching Cycles

We implement the ReFlex Switch with a seven stage pipeline as shown in Table 3.1 and Table 3.2. In the tables, an example execution of ReFlex Switch pipeline stages is demonstrated for flits of connection $c_{i;j}$: $f_{i;j;1}$, $f_{i;j;2}$, $f_{i;j;3}$, $f_{i;j;4}$ and $f_{i;j;5}$. Here, indicates the last flit of a packet. There is a status flag $s_{i;j}$ that indicates if $VOQ_{i;j}$ is non-empty. We assume that $VOQ_{i;j}$ is empty before Cycle 1 and $s_{i;j} = 0$. Each iteration $t = 1$ $MaxIters$ of the arbiter has two cycles shown by $A_{t;1}$ and $A_{t;2}$.

Table 3.1: ReFlex Switch Pipeline Stages for Single Flit (indicates last flit)

Cycle	1	2	3	4	5	6	7
Input	$f_{i,j,1}$						
Queue	,!	$s_{i,j} \leftarrow 1$					
Arbiter	,!	$A_{1,1}$	$A_{1,2}$				
Config.			,!	update			
XBAR				$f_{i,j,1}$			
RB				,!	R_1	R_2	
Output						,!	$f_{i,j,1}$

For the single flit case demonstrated in Table 3.1, in Cycle 1, $f_{i,j,1}$ is received at input i. In Cycle 2, $f_{i,j,1}$ is inserted in $VOQ_{i,j}$ and $s_{i,j} \leftarrow 1$. The Arbiter operation is initiated by taking the boolean OR of $s_{i,j}\ \forall i \in I, j \in O$. When initiated, the Arbiter stage executes $A_{1,1}$ in Cycle 2 and $A_{1,2}$ in Cycle 3. The decision result is stored in pipeline registers at the end of Arbiter Stage. In Cycle 4, crossbar ("XBAR") configuration is updated by transferring the arbiter decision from pipeline registers to the crossbar registers.

In Cycle 5, $f_{i,j,1}$ appears on the RB input and initiates RB state machine which tracks first and last flits of packets. This operation goes through two states in two cycles, R_1 in Cycle 6, and R_2 in Cycle 7, to output the first flit of a packet and make it available on the output. Note that for this flit, the arbiter took only a single iteration (2 phases, 2 clock cycles). In this case, it takes 7 clock cycles for a flit to traverse through the empty and idle switch.

For the multiple flit case demonstrated in Table 3.2, a new sequence of four flits is presented to the switch pipeline. In Cycle 1, $f_{i,j,2}$ is inserted in $VOQ_{i,j}$ and $s_{i,j} \leftarrow 1$. When initiated, the Arbiter stage executes $A_{1,1}$. For the sake of this example, arbitration runs for three iterations (each with 2 phases, total 6 clock cycles). Similar to the previous run, RB receives the flits but in this case it waits in state R_1 until the last flit is observed. Then, RB state R_2 is executed and the tracked three flits are

available on the output from the Cycle 13 to 16.

Depending on the previous configuration and RB states, up to a two cycle dependency related stalls may occur in updating the configuration. These stalls occur because of the handshake signalling required between VOQ and RB buffers. Such a stall would delay the RB states by two cycles, i.e. R_1 in Cycle 9 of Table 3.2 would be shifted to Cycle 11. Consequently, the flits on the output would appear delayed by two cycles. In normal operation, after the cold start is finished, one flit can be transferred each and every cycle with the match decisions of the arbiter.

Table 3.2: ReFlex Switch Pipeline Stages for Multiple Flits (indicates last flit)

Cycle	1	2	3	4	5	6	7	8
Input	$f_{i;j;2}$	$f_{i;j;3}$	$f_{i;j;4}$	$f_{i;j;5}$				
Queue	,!	$s_{i;j}$ 1						
Arbiter	,!	$A_{1;1}$	$A_{1;2}$	$A_{2;1}$	$A_{2;2}$	$A_{3;1}$	$A_{3;2}$	
Config.							,!	update
XBAR								$f_{i;j;2}$
RB								,!
Output								
(cont'd)	9	10	11	12	13	14	15	16
Input								
Queue								
Arbiter								
Config.								
XBAR	$f_{i;j;3}$	$f_{i;j;4}$	$f_{i;j;5}$					
RB	R_1	R_1	R_1	R_2				
Output				,!	$f_{i;j;2}$	$f_{i;j;3}$	$f_{i;j;4}$	$f_{i;j;5}$

3.3.2 Buffer Management

The buffers at input port i and output port j are implemented in `FB_BRAM_i` and `RB_BRAM_j` as seen in Figure 3.1. The input and output side multiplexers for FIFOs incur a logic and routing cost proportional to the flit width in bits. All of the buffer components work at line rate resulting in only one FB/RB to write and one FB/RB to read at a time. This allows partitioning `FB_BRAM_i` and `RB_BRAM_j` into equal-sized M FBs and N RBs as logically separated circular FIFO buffers respectively as shown in Figure 3.3. `FB_FIFO_Controller_i` and `RB_FIFO_Controller_j` manage these buffers by arrays of pointers: `waddr_i`/`waddr_j` for the current write addresses and `raddr_i`/`raddr_j` for the current read addresses. To this end, our FIFO implementation decreases the hardware complexity and enables writes and reads in a single cycle without additional delay at line rate.

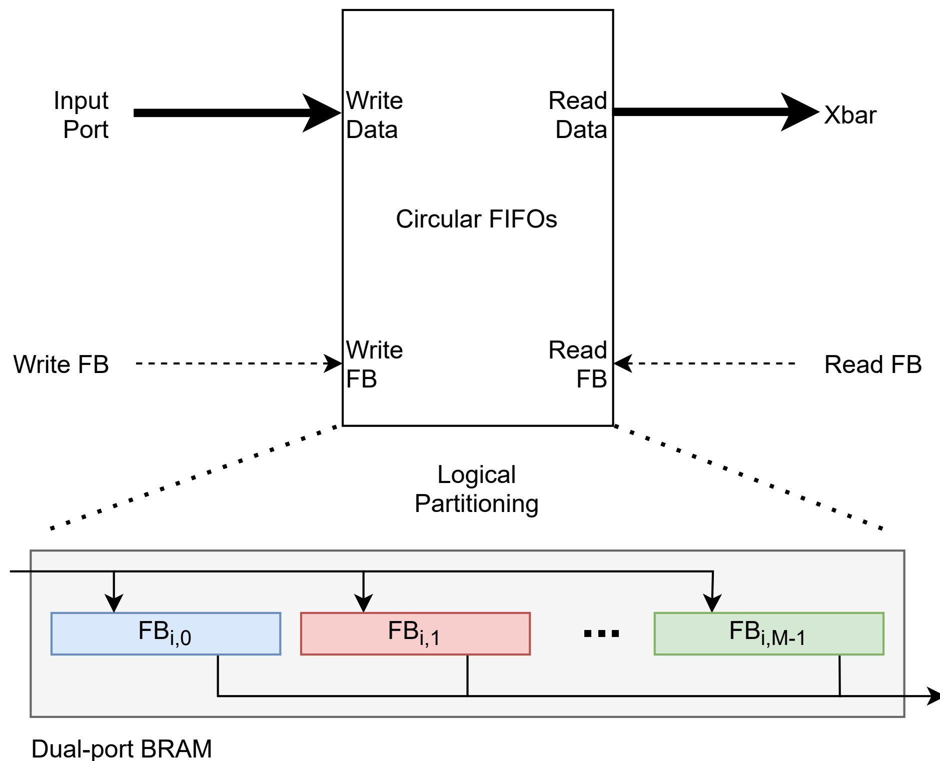


Figure 3.3: ReFBM Compact FIFO Organization

The logic for each FB_FIFO_Controller_i is illustrated in detail in Figure 3.1. RB_FIFO_Controller_j works analogously and its logic is only briefly presented. The switch output port numbers for write and read that come from input_ctrl_i and arbt_select_i are mapped to mi and mo through Write Map Reg. and Read Map Reg. given in Figure 3.2. Here, mi, mo $\in [0, M-1]$. waddr_i[mi] is the address in BRAM where the next flit that came to input i and is destined to output j should be written. Similarly, raddr_i[mo] indicates the address in BRAM where the next flit that is stored in input i's buffers and is destined to output j should be read. waddr_i[mi] and raddr_i[mo] is advanced by one when a write and read happens respectively. When a pointer reaches the boundary address of its associated FB, the pointer wraps around to the start address of its FB as per the circular FIFO procedure. Two arrays of boolean flags are stored to track the wrap-arounds: wwrap_i for write pointer wraps and rwrap_i for read pointer wraps. These flags also help determine the FIFO status flags and lengths (number of flits stored in an FB). For example, for all $m = 0, \ldots, M-1$, waddr_i[m]=raddr_i[m] indicates that FB_i_m is either *empty* or *full*. In this case, if the corresponding wrap status flags wwrap_i[m] and rwrap_i[m] are equal, then the FIFO is empty. If they are not equal, then the FIFO is full.

The linked list organization illustrated in Figure 3.2 is implemented for RBM_i by keeping 5 registers within the scope of Input Block i in Figure 3.1 (omitted for brevity). These registers are WriteMapReg, ReadMapReg, alloc_used, alloc_next_valid and alloc_next_fb.

WriteMapReg and ReadMapReg are logically shown in Figure 3.2. They are the arrays of FB indices that hold the tail and head FBs for write and read operations respectively. alloc_used and alloc_next_valid are arrays of boolean flags. alloc_used indicates whether an FB is currently in use (allocated) or not. alloc_next_fb is an array of FB indices that holds the index of next FB in the linked list structure. alloc_next_valid indicates whether the information stored in alloc_next_fb is in a valid state, i.e. the FB has a next FB or not.

By definition, an AXI4-Stream transaction happens only when both valid and ready signals are set [46]. AXI4-Stream handshake logic in the input and output interfaces require the ready and valid status of input and output side FIFOs respectively. For the interface input_i, the valid signal input_valid[i] comes from the upstream AXI4-Stream interface. The ready signal going to this interface should only be set if the targeted FB_i_mi is *not full*. The selected FB_i_mi is determined by input_ctrl_i as explained previously, and the full_i[mi] signal is exported from FB_FIFO_Controller_i to serve as the ready signal of this interface. For the interface output_j, the ready signal output_ready[j] comes from the downstream AXI4-Stream interface. The valid signal output_valid[j] going to this interface should only be set if the corresponding RB_i_j is *not empty*. The selected RB_i_j is determined by the RB_FSM_j. For the control of transactions through the crossbar between input and output side memories, a similar handshake routine is realized by a 1-bit wide Handshake Xbar that operates in reverse of the data flow direction for a given arbitration decision.

Figure 3.1 also shows the operation of RB_FSM_j. RB_i_j are connected to the downstream AXI4-Stream port. The count of fully reassembled packets residing in RB_i_j, $ctr_{i,j}$, are stored in registers eof_cnt_j. When a flit with *Last* bit set arrives and departs at RB_i_j, $ctr_{i,j}$ $ctr_{i,j}+1$ and $ctr_{i,j}$ $ctr_{i,j}$ 1 respectively. We apply a round robin scheme for RB_i_j with $ctr_{i,j} > 0$, RB_i_j to select the packet to transmit out of port *j* and generate output_valid[j]. Optimization of the RB management is out of the scope of this study.

3.3.3 One-Phase-Per-Cycle Arbitration

The pipeline operation as described in 3.3.1 shows how the three iterations of arbitration are performed in two cycles. In short, the first phase, that is the request collection phase, is executed during the last phase of the previous iteration thanks to pipelined operation.

The request bitmap register is combinationally generated from the boolean inversion of empty flags. This way, the requests are available for use in the grant phase without wasting a clock cycle in $Request(i; j)$ for CAR arbiter.

In the grant phase, the collected requests are arbitrated using the prioritization information stored in $p_G(j)$. Functionally, the request bitmap is scanned staring from the $p_G(j)$-th port until a set bit is found, which is shown as a *for*-loop in the Algorithm 1. In RTL, this is done by using a priority encoder. However, the conventional priority encoder always prioritizes a fixed input. For arbitration, we need the input that is indicated by the grant pointer to be selected first if its request bit is set. If not, then the other ports should be checked in a shuffled order as needed by CAR.

To this end, the inputs to the priority encoder are permuted after being rotated, and then the permutation is reverted after priority encoder's selection and reverse rotation. Note that this permutation excludes the first input so as to always prioritize $p_G(j)$ first. If $p_G(j)$'s request bit is not set, then the rest are checked in a different permuted order every iteration. These modifications around the priority encoder are shown in the Figure 3.4. Similar priority encoders are used also for the accept phase. The rotation amount comes from $p_A(i)$. In both cases, the new permutation is read from the permutation generation module common to the arbiter which is discussed in Section 3.1.

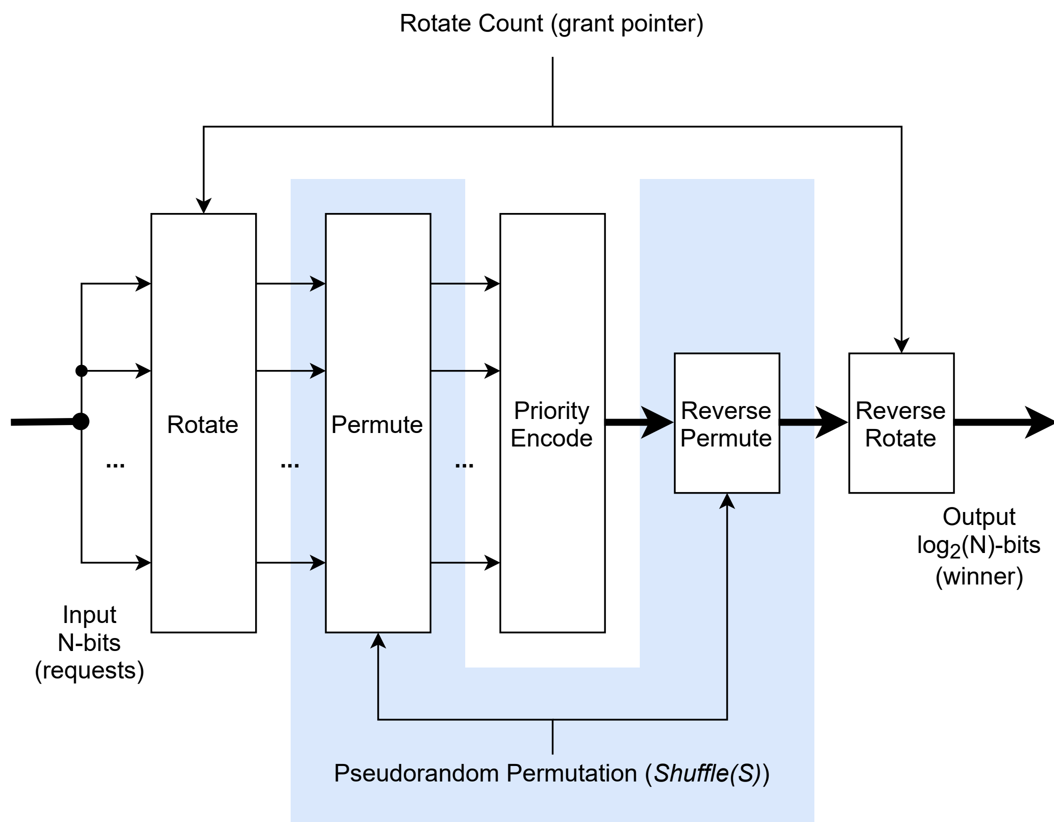


Figure 3.4: Rotation and Permutation in Combinational Priority Encoders

3.3.4 Configuration and Status Registers (CSR)

Configuration and Status registers allow setting option bits and retrieving status flag or counter values in runtime. For ReFlex Switch, the Configuration and Staus Registers are exposed through an AXI4-Lite interface. This approach is common for IP cores in FPGA development [47].

The ReFlex Switch consists of multiple submodules as shown in the Figure 3.1. The CSR AXI4-Lite interface is common to the top level module. This common interface is shared to submodules using a custom decoder that performs read and write bus transactions by the address bits of the host request. The address space is divided among the submodules for modularity. The arbiter module exposes its own AXI4-Lite CSR interface to the ReFlex Switch AXI4-Lite decoder. This way, the selected arbiter can be configured without changing the rest of the code.

Data channel width of AXI4-Lite is 32-bits. The address channel width is 16-bits. As the AXI4-Lite bus is byte addressable [46], the lower two bits of the addresses are not used. There are four slaves on this bus: CORE, FB, RB, ARBT. These stand for the top level integration layer, VOQs or *Flex Buffers*, Reassembly Buffers and the Arbiter respectively. The higher 2-bits of the addresses are used to select the targeted slave. The bit fields of a CSR address is shown in Table 3.3.

Table 3.3: Bit Field Widths of a CSR Address

15	14	13-2	1	0
s1	s0	r11 r10 . . . r0	0	0
s1s0: slave number. r11-r0: register number in slave. Byte offset 00 is reserved.				

Here, s1s0 determines the slave that the request will be forwarded to: 00 for CORE, 01 for FB, 10 for RB and 11 for ARBT. The byte index bits of the address is reserved and overriden with 00 and byte addressing is not supported as it is not used for ReFlex Switch CSRs. The 12-bit middle part of the addresses, r11 r10 . . . r0, is

the register number inside a given slave. Slaves interpret this part of the address differently based on the registers they expose in this address space.

Tables 3.4, 3.5, 3.6 and 3.7 show the register placement for ReFlex Switch CSR slaves.

Table 3.4: CSR CORE Register Map

15 14	13-2	1 0	Register Description
00	000000000000	00	HWREVR Hardware revision register. This register returns the hardware RTL code revision for debugging and SoC host side software compatibility. Read-only.
00	000000000001	00	HWCONFR Hardware configuration register. This register is reserved for retrieving ReFlex Switch parameters and configuration (port count, buffer organization, arbiter type etc.) for SoC host side software compatibility. Read-only.
other			Reserved registers. Returns 0 when read. Writes have no effect.

Table 3.5: CSR FB Register Map

15 14	13-2	1 0	Register Description
01	0000000aaaaa	00	FBEMPTYR *Flex Buffer* empty register. This register returns the empty flag status bits for VOQs or *Flex Buffers*. j-th bit of the returned value is FB_i _j 's empty flag where $i = (aaaaa)_2$. Read-only.
01	0000001aaaaa	00	FBFULLR *Flex Buffer* full register. This register returns the full flag status bits for VOQs or *Flex Buffers*. j-th bit of the returned value is FB_i _j 's full flag where $i = (aaaaa)_2$. Read-only.
01	01aaaaabbbbb	00	FBLENGTHR *Flex Buffer* length register. This register returns the FIFO length for VOQs or *Flex Buffers*. The returned value is FB_i _j 's length in flits where $i = (aaaaa)_2$ and $j = (bbbbb)_2$. Read-only.
other			Reserved registers. Returns 0 when read. Writes have no effect.

Table 3.6: CSR RB Register Map

15 14	13-2	1 0	Register Description
10	0000000aaaaa	00	RBEMPTYR Reassembly Buffer empty register. This register returns the empty flag status bits for Reassembly Buffers. j-th bit of the returned value is RB_i _j 's empty flag where $i = (aaaaa)_2$. Read-only.
10	0000001aaaaa	00	RBFULLR Reassembly Buffer full register. This register returns the full flag status bits for Reassembly Buffers. j-th bit of the returned value is RB_i _j 's full flag where $i = (aaaaa)_2$. Read-only.
10	01aaaaabbbbb	00	RBLENGTHR Reassembly Buffer length register. This register returns the buffer length for Reassembly Buffers. The returned value is RB_i _j 's length in flits where $i = (aaaaa)_2$ and $j = (bbbbb)_2$. Read-only.
other			Reserved registers. Returns 0 when read. Writes have no effect.

Table 3.7: CSR ARBT Register Map

15 14	13-2	1 0	Register Description
11	000000000000	00	ARBTREVR Arbiter hardware revision register. This register returns the arbiter hardware RTL code revision for debugging and SoC host side software compatibility. Read-only.
11	000000000001	00	ARBTTYPER Arbiter type register. This register returns the arbiter type used in code generation for SoC host side software compatibility. Return value is 0x1 for DRR, 0x2 for CAR, other values are reserved. Read-only.
11	01aaaaabbbbb	00	CARGRANTR CAR-specific grant credit register. Initialized to 0. Writes set the value of $G_{i;j}$ and reads get the value of $G_{i;j}$ where $i = (aaaaa)_2$ and $j = (bbbbb)_2$. Read-only.
11	10aaaaabbbbb	00	CARACCEPTR CAR-specific accept credit register. Initialized to 0. Writes set the value of $A_{i;j}$ and reads get the value of $A_{i;j}$ where $i = (aaaaa)_2$ and $j = (bbbbb)_2$. Read-only.
other			Reserved registers. Returns 0 when read. Writes have no effect.

3.3.5 Parametric ReFlex Switch RTL Generation

The synthesis-ready RTL code for the ReFlex Switch IP core is automatically generated with a Python script which takes a number of architectural parameters as input. These parameters are; the number of ports N, the flit size, the bus control signal widths, arbiter maximum iteration count ($MaxIters$), buffer depths, VOQ organization, the fabric arbiter, and output directory organization. ReFlex Switch RTL with code generation script can be obtained from [48].

Parametric generation of the ReFlex Switch wrapper module makes the architecture compatible with Xilinx custom IP packaging guidelines [49], which allows easier integration. The ReFlex Switch code consists of the following Verilog files:

- acc_sw_wrapper_PxP_D.v: automatically generated ReFlex Switch wrapper. Defines the input-output AXI4-Stream data and AXI4-Lite CSR interfaces. P: port count. D: flit size.
- acc_sw_core.v: Top-level module integrating ReFlex Switch submodules.
- acc_sw_axi4lite_decoder.v: CSR interface decoder.
- acc_sw_arbiter_drr.v: DRR-type arbiter.
- acc_sw_arbiter_car.v: CAR-type arbiter.
- acc_sw_lfsr.v: Pseudorandom number generation for Random(a;b) (CAR only)
- acc_sw_segmented_fifo.v: VOQ controller and memory (segmented compact circular FIFO).
- acc_sw_reflex_fb.v: ReFlex *Flex Buffer* controller.
- acc_sw_xbar.v: Crossbar.
- acc_sw_ra_buffer.v: Reassembly Buffer.
- acc_sw_shuffled_pri_enc.v: Priority encoder with Shu e(S) support.

The ReFlex Switch wrapper code generation script parameters can be seen in Listing 3.1.

Listing 3.1: ReFlex Switch RTL Generator Parameters

```
$ python3 acc_sw_core_gen.py --help
usage: acc_sw_core_gen.py [-h]
    [--port-count PORT_COUNT] [--addr-width ADDR_WIDTH]
    [--stream-width STREAM_WIDTH]
    [--module-name MODULE_NAME] [--output-dir OUTPUT_DIR]
    [--reduce-keep] [--max-iters MAX_ITERS]
    [--voq-depth VOQ_DEPTH] [--rb-depth RB_DEPTH]
    [--xbar-registered] [--use-reflex-fb]
    [--arbt-type ARBT_TYPE]
optional arguments:
  -h, --help    show this help message and exit
  --port-count PORT_COUNT
        switch input/output port count
  --stream-width STREAM_WIDTH
        AXI4-Stream bus data width
  --addr-width ADDR_WIDTH
        AXI4-Stream bus id/dest width
  --module-name MODULE_NAME
        name of the generated module
  --output-dir OUTPUT_DIR
        output folder
  --reduce-keep
        enable encoding/decoding of AXI keep bits
  --max-iters MAX_ITERS
        switch arbiter maximum iteration count
  --voq-depth VOQ_DEPTH
        Virtual Output Queue maximum depth
  --rb-depth RB_DEPTH
        Reassembly Buffer maximum depth
  --use-reflex-fb
        enable use of FlexBuffers instead of pure VOQ
  --arbt-type ARBT_TYPE
        arbiter type to use (options: DRR CAR)
```

CHAPTER 4

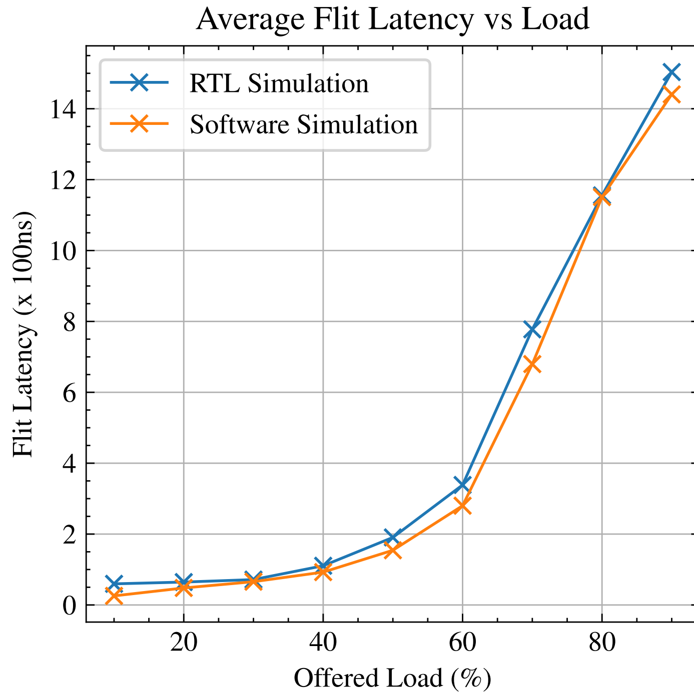
EVALUATION

4.1 Performance Evaluation

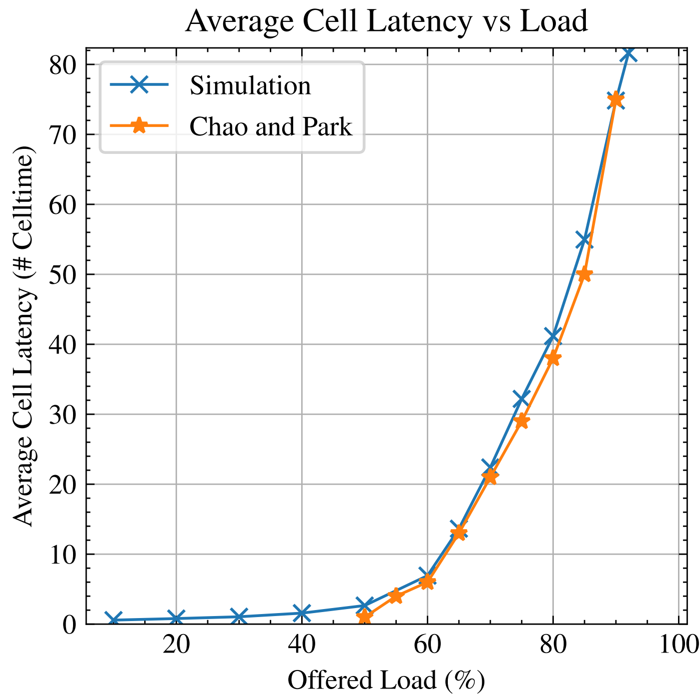
The performance of the ReFlex Switch is evaluated using an event based and cycle accurate software simulator that we developed in C++. We use cycle accurate software simulation because it is faster to run and easier to generate different workloads in comparison to RTL simulation. The software simulator codes can be obtained from [50]. All software simulations use 200k packets and ignore the first 20k for statistics.

We verified our software simulator by comparing software and RTL simulation results, and observed that we get similar latency measurements with respect to the offered line load as seen in Figure 4.1 (a). In Figure 4.1 (b), we further functionally verified our simulator by repeating the DRR experiment with uniform load on the 16x16 switch presented by Chao and Park in [51] and observed that we get very similar latency measurements with respect to the offered line load.

Next, we implement CAR as defined in Algorithms 1 and 2 in our simulator with an 8x8 switch with 40 Gbps line rate on all ports. In both CAR experiments, experiment 1 and 2, 1% of the packets are 40 Bytes and the remaining packets are 1500 Bytes as we assume that Hardware Accelerators (HAs) will produce streams of data. Buffers are infinite in length for both CAR experiments. With our first experiment we demonstrate the service differentiation capability of CAR with unlimited VOQ sizes and without ReFBM. The destinations are uniformly distributed. We set 4 different service levels; $_{0;j} = {}_{1;j} = 8$ Gbps, $_{2;j} = {}_{3;j} = 6$ Gbps, $_{4;j} = {}_{5;j} = 4$ Gbps, $_{6;j} = {}_{7;j} = 2$ Gbps. To this end, The respective CAR configuration is $8j; G_{i;j} = [8;8;6;6;4;4;2;2]$ and $8i; A_{i;j} = [8;8;6;6;4;4;2;2]$.



(a) By RTL Simulation

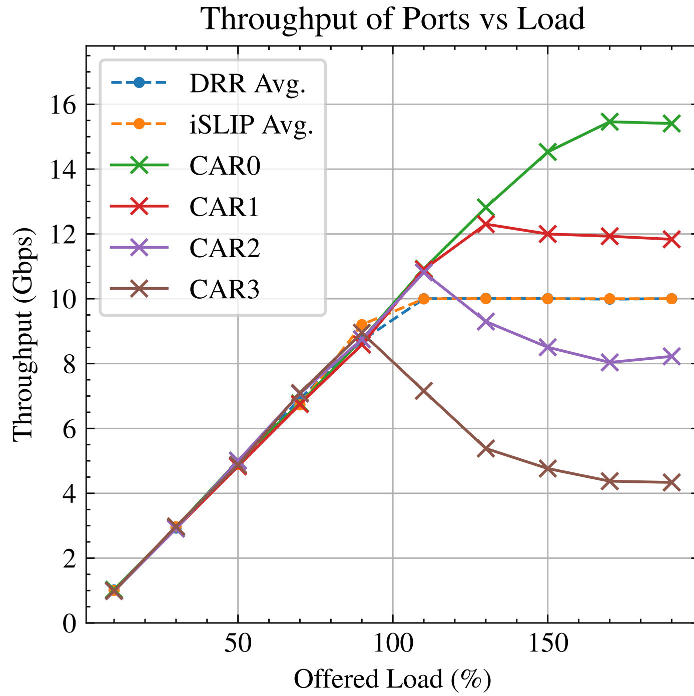


(b) By Chao and Park's results

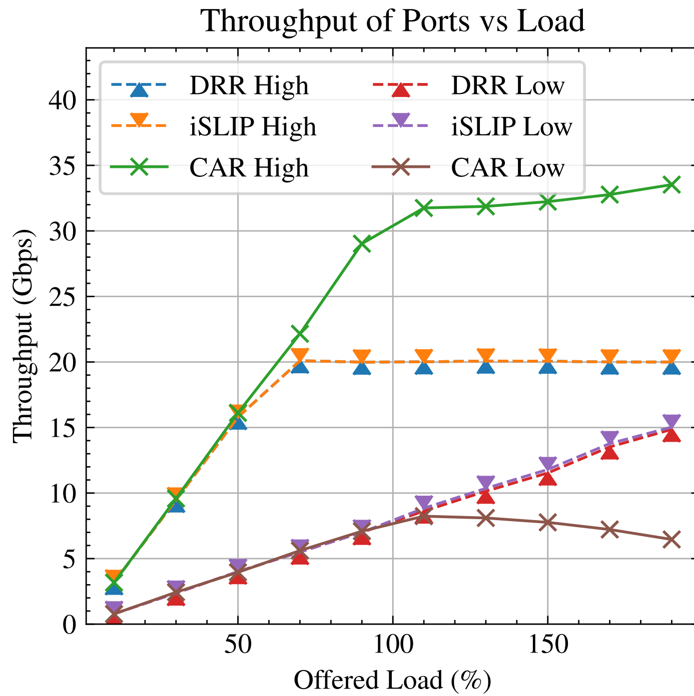
Figure 4.1: Simulator Verification with DRR

Figure 4.2 (a) shows the throughput of an output port j with CAR, DRR [14] and iSLIP [13] under uniformly distributed destinations. All three arbiters work with $MaxIters = 3$ as we observe that CAR achieves convergence with an average number of iterations of 2.74 at 90% load. Total throughput for each service level is indicated by CAR0, CAR1, CAR2 and CAR3 respectively. The average throughput for each service level is the same under DRR and indicated with DRR Avg. First of all, we observe that CAR, DRR and iSLIP are work conserving as the throughput is the same as the offered load until 100% offered load, and after that the total throughput is very close to 40 Gbps. The second observation is that CAR achieves bandwidth allocation values of 15.4, 11.8, 8.3, 4.3 Gbps (a total of 39.8 Gbps) throughput respectively which are close to the desired values under 190% overload, where all service levels get an equal throughput of 10 Gbps under DRR and iSLIP. At 90%, achieved latencies are 525, 646, 838 and 1133 ns for CAR0, CAR1, CAR2, CAR3, 806 ns for DRR Avg. and 797 ns for iSLIP Avg.

In the second experiment, we evaluate two service levels with $_{i,0\ 3} = 2$ Gbps and $_{i,4\ 7} = 8$ Gbps where the average throughput for each service level is indicated by CAR High and CAR Low respectively in Figure 4.2 (b). Here, we evaluate a hot-spot destination scenario where all source ports contend to reach four destination ports. Destination ports are selected from a probability distribution where the four of the outputs are *hot-spots* which receive 20% of the total traffic each. Hence, each hot-spot port has 4 times the load of a non-hot-spot port. The respective CAR credit configuration is $\forall j, G_{i,j} = [2,2,2,2,8,8,8,8]$ and $\forall i, A_{i,j} = [2,2,2,2,8,8,8,8]$. The simulation results show that the matching efficiency of arbitration is affected from the non-uniform traffic, observed by the limited throughput obtained with DRR and iSLIP. By configuring service levels proportional to expected traffic pattern via $G_{i,j}$ and $A_{i,j}$, the throughput is improved. In this experiment, at 190% overload, achieved total throughputs for different service levels are 33.5 and 6.46 Gbps for CAR High and CAR Low, 19.9 and 14.8 Gbps for DRR High and DRR Low, and 20.0 and 14.9 Gbps for iSLIP High and iSLIP Low. At 90%, achieved latencies are 1741, 1888, 518 and 522 ns for CAR High, CAR Low, DRR Low and iSLIP Low respectively. Latency of DRR and iSLIP High flits diverge to a large value due to saturation as a result of decreased matching efficiency.



(a) CAR Bandwidth Allocation (uniform traffic)

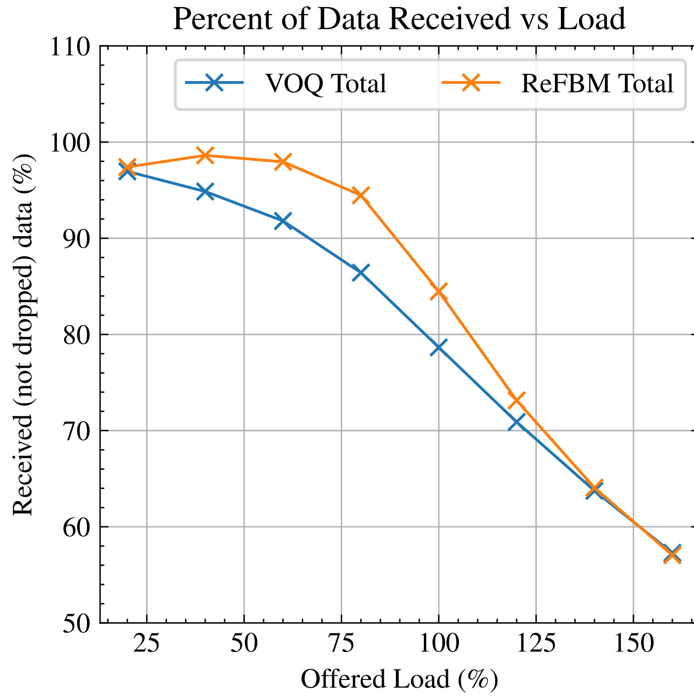


(b) CAR Bandwidth Allocation (nonuniform traffic)

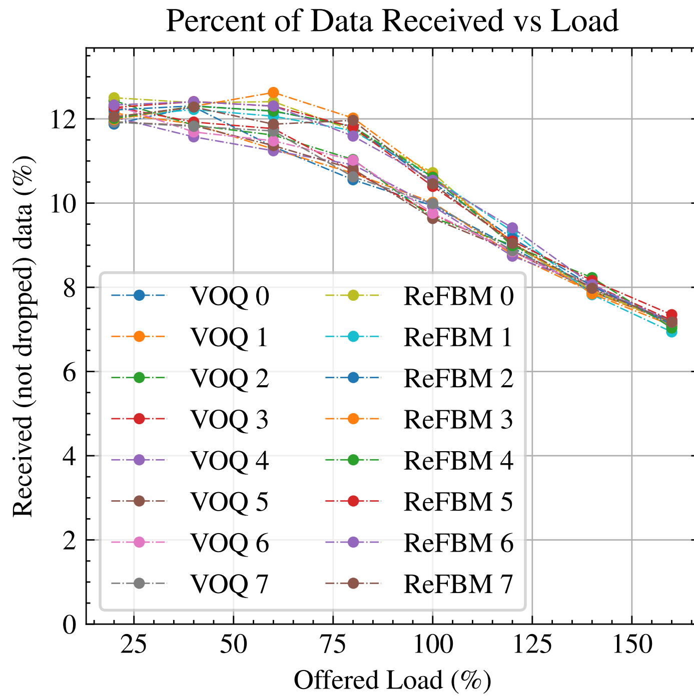
Figure 4.2: CAR Quality of Service with Infinite VOQ

We performed the ReFBM experiments with DRR as arbiter to isolate the effect of arbiter on the memory efficiency. In experiments 3, 4 and 5, the queue lengths are limited. Under this condition, if a flit comes to input port i which targets the output port j and $VOQ_{i;j}$ is full (the tail FB is full and $Q_i = 0$), the flit is dropped. The performance metric used in evaluation is the percent of data received on the output, i.e. flits that are not dropped. This % flit throughput metric demonstrates the effect of buffer allocation under limited queue lengths. In all ReFBM experiments, 1% of the packets are 40 Bytes and the remaining packets are 1500 Bytes as we assume that HAs will produce streams of data.

In the third experiment, we evaluate an 8-port switch with ReFBM and 16 FBs per input port compared to fixed size VOQ buffer organization under uniform traffic load. All input ports contend equally to send data to all output ports. The arbiter is DRR. FBs are 32 flit deep and VOQs are 64 flit deep. Thus, the amount of BRAM memory used by both implementations are the same. Figure 4.3 (a) shows the average total percent of data received by an output port. The ReFBM case is shown by ReFBM Total and the VOQ case is shown by VOQ Total. At very low loads such as 20%, both schemes perform similarly: 96.8% for VOQ and 97.1% for ReFBM. Thanks to the flexible buffer allocation, ReFBM performs better than VOQ with same amount of memory in most cases. At 80%, ReFBM allows 95.2% of the data to be received whereas VOQ only achieves 86.5%. Due to the uniform load traffic pattern, both buffer allocation schemes perform similarly under 140% overload and further. Figure 4.3 (b) shows the distribution of data received between different source ports. All ports are served equally without any starvation.



(a) ReFBM Performance - Total (uniform traffic)



(b) ReFBM Performance - Each Port (uniform traffic)

Figure 4.3: ReFlex Buffer Management with DRR (uniform traffic)

In the fourth experiment, we evaluate an 8-port switch with ReFBM and 16 FBs per input port compared to fixed size VOQ buffer organization under nonuniform traffic load. Again, we evaluate a hot-spot destination scenario where all source ports contend to reach four destination ports. Destination ports are selected from a probability distribution where the four of the outputs are *hot-spots* which receive 20% of the total traffic each. Hence, each hot-spot port has 4 times the load of a non-hot-spot port. The arbiter is DRR. FBs are 32 flits deep and VOQs are 64 flits deep. Thus, the amount of BRAM memory used by both implementations are the same. Figure 4.3 shows the average total percent of data received by output ports. ReFBM cases are shown by ReFBM Total for total, ReFBM High for hot-spot ports and ReFBM Low for non-hot-spot ports. Similarly, the VOQ cases are shown by VOQ Total, VOQ High and VOQ Low. At very low loads such as 20%, both schemes perform similarly: 96.6% for VOQ and 98.2% for ReFBM. Upto 60%, non-hot-spot ports get equal service under both buffer allocation schemes around 20%. Upto 80%, ReFBM allocates more FBs to hot-spot destinations as expected and achieves better performance. At 60%, ReFBM High achieves 77.3% while VOQ High is limited to 69.8%. However, at 80%, ReFBM Total and VOQ Total lines cross around 78.0%. After this point, ReFBM's total performance decreases slightly in comparison to pure VOQ scheme due to the starvation of non-hot-spot ports represented with ReFBM Low. Although ReFBM High and VOQ High stay similar afterwards, ReFBM Low shows that non-hot-spot ports suffer from the starvation that is caused by hot-spot ports allocating most of the free FBs under continued loading.

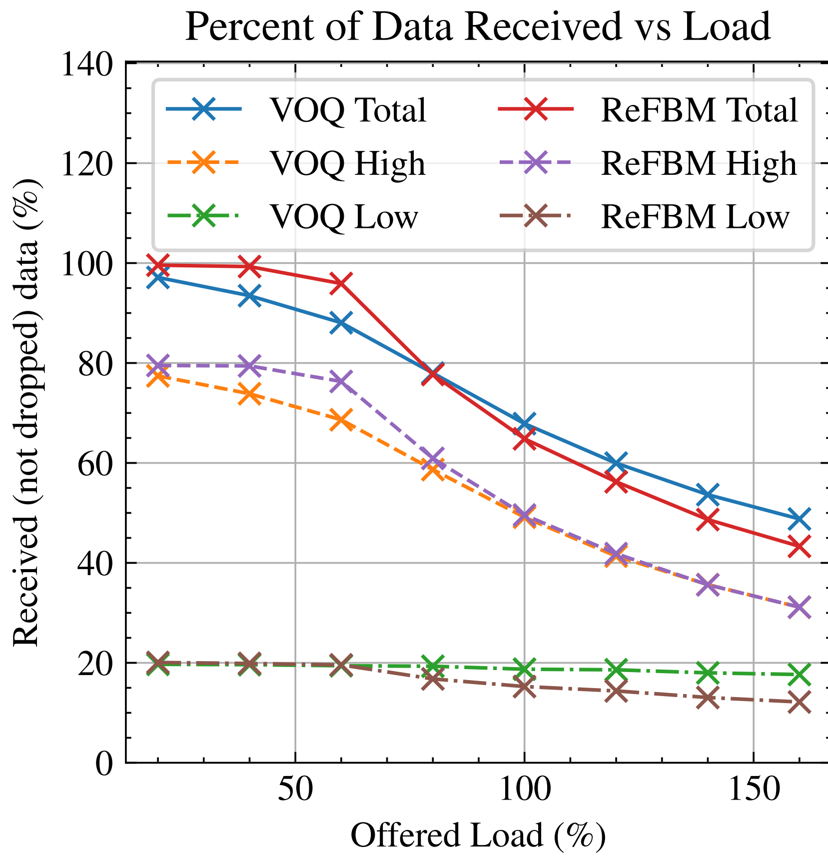
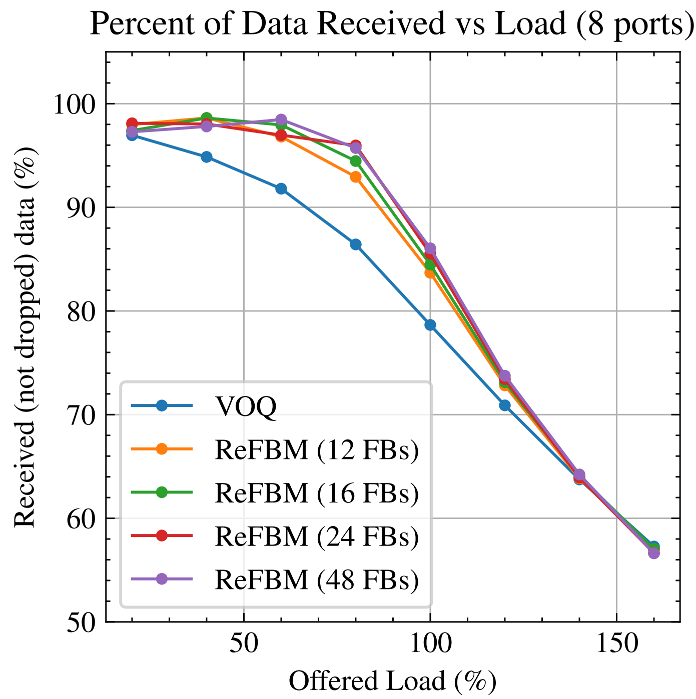
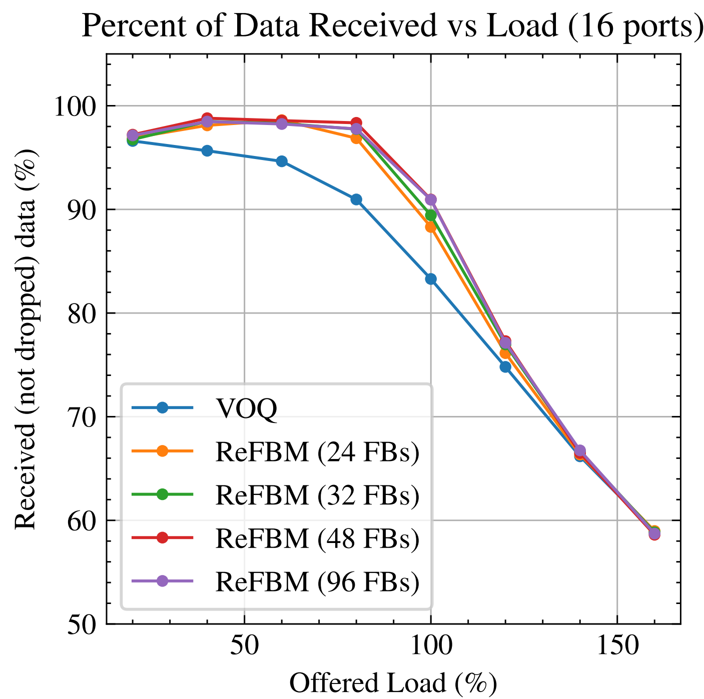


Figure 4.4: ReFlex Buffer Management with DRR (nonuniform traffic)

In the fifth experiment, we show the effect of the number of FBs per input port on the switch performance using the same amount of memory in BRAMs. In Figure 4.5 (a), we perform the experiment for an 8-port switch under uniform traffic. VOQs are 64 flits deep and FBs are 42, 32, 21, 16 flits deep for 12, 16, 24, 32 FBs. Thus, all scenarios use the same amount of memory in BRAMs. The performance metric is average total percent of data received by an output port. We see that using more but smaller FBs (keeping the total memory same) gives diminishing returns after $M = 2N$. For low loads at 20% and high overloads beyond 140%, all ReFBM configurations and the VOQ perform similarly. At 80%, the measured performances are 86.4% for VOQ and 93.1%, 94.2%, 96.1%, 96.0% for 12, 16, 24, 48 FBs per input port respectively. In Figure 4.5 (b), we obtain similar results for a 16-bit switch under uniform traffic. Here, VOQs are 64 flits deep and FBs are 42, 32, 21, 16 flits for 24, 32, 48 and 96 FBs cases in order. Thus, all scenarios use the same amount of memory in BRAMs. At 80%, VOQ scheme performs with 90.9% of data received (not dropped) where 24, 32, 48 and 96 FBs per input port achieve 97.2%, 98.0%, 98.3%, 98.0% respectively.



(a) Different Number of FBs (uniform traffic, 8 ports)



(b) Different Number of FBs (uniform traffic, 16 ports)

Figure 4.5: Effect of *Flex Buffer* Count with DRR

In the sixth experiment, we demonstrate the combined performance of CAR with ReFBM against CAR with VOQ under uniform traffic and finite buffers. We set four service levels for CAR similar to the first experiment: CAR+ReFBM/VOQ-0, CAR+ReFBM/VOQ-1, CAR+ReFBM/VOQ-2 and CAR+ReFBM/VOQ-3 with $_{0;j}$ = $_{1;j}$ = 8 Gbps, $_{2;j}$ = $_{3;j}$ = 6 Gbps, $_{4;j}$ = $_{5;j}$ = 4 Gbps, $_{6;j}$ = $_{7;j}$ = 2 Gbps respectively. To this end, The respective CAR configuration is $8j; G_{i;j} = [8;8;6;6;4;4;2;2]$ and $8i;j; A_{i;j} = 2$. Different than the first experiment, the queue lengths are limited. ReFBM has 16 FBs and FBs are 32 flit deep. VOQs are 64 flit deep. Figure 4.6 shows the throughput of an output port j under CAR with ReFBM and VOQ. The total throughput for each service level under CAR is indicated by CAR+ReFBM/VOQ-0, CAR+ReFBM/VOQ-1, CAR+ReFBM/VOQ-2 and CAR+ReFBM/VOQ-3 respectively. Here, we observe that although the finite buffer lengths disturb the proportional bandwidth allocation, CAR still achieves switching service differentiation with both ReFBM and VOQ. Furthermore, ReFBM improves proportional allocation of bandwidths between service levels. At 190% overload, CAR+ReFBM achieves 16.1, 14.0, 6.8 and 2.9 Gbps (a total of 39.8 Gbps) whereas CAR+VOQ achieves 13.0, 11.6, 9.5, 5.7 Gbps (a total of 39.8 Gbps). We also note that at 130% overload, the throughput of CAR+ReFBM-1 service level slightly exceeds the CAR+ReFBM-0 as a result of and interaction between CAR and ReFBM. Figure 4.7 (a) and (b) shows the total and flit % throughput of a port respectively. Both ReFBM and VOQ schemes achieve similar throughputs below 70% and above 150%. At 90%, CAR+ReFBM gets 35.9 Gbps total and 99.4% flit % throughput while CAR+VOQ gets 32.2 Gbps total and 89.4% flit % throughput.

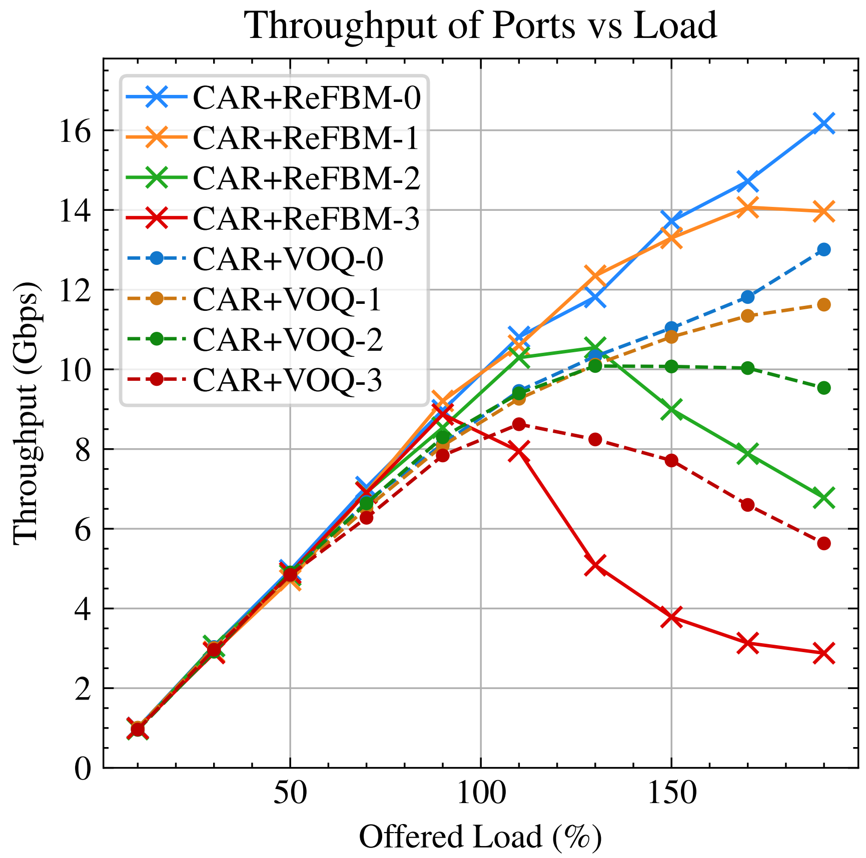
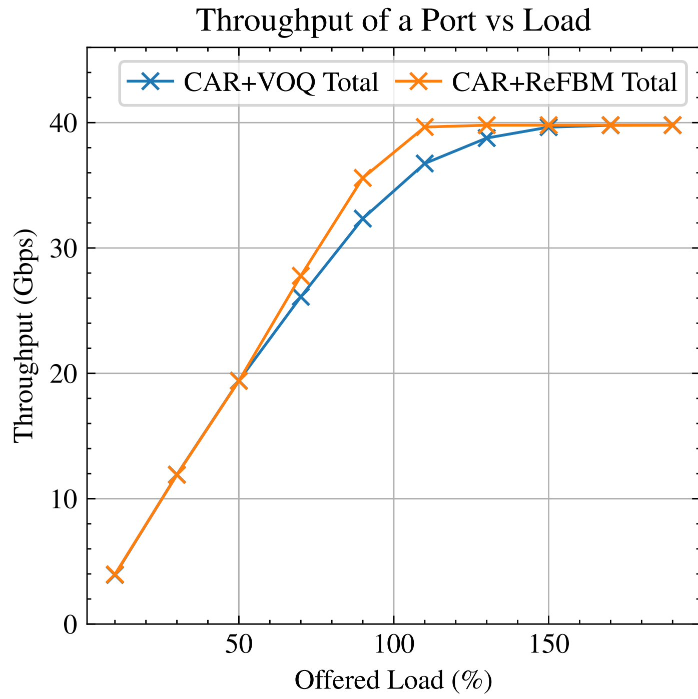
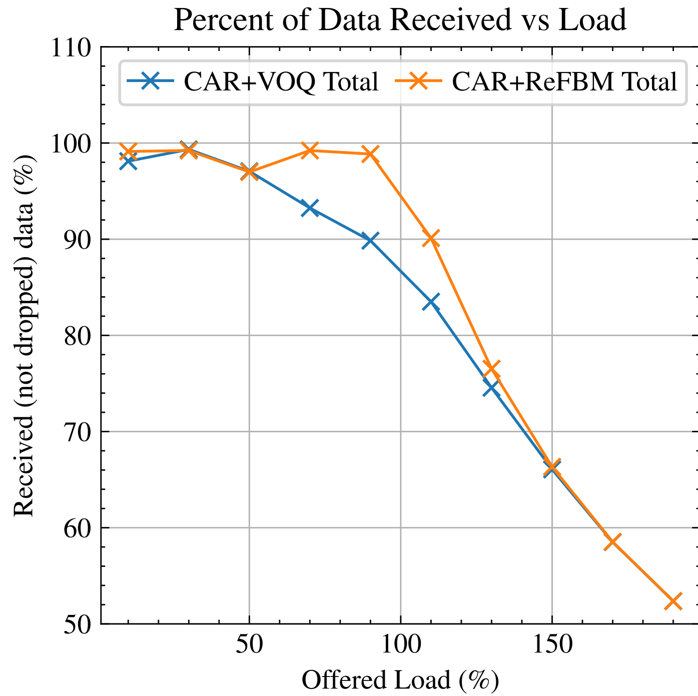


Figure 4.6: ReFlex Buffer Management with CAR (ports)



(a) CAR + ReFBM Throughput in Gbps with CAR (uniform traffic)



(b) CAR + ReFBM Flit % Throughput with CAR (uniform traffic)

Figure 4.7: ReFlex Buffer Management with CAR (total)

Table 4.1: Bit Field Widths of a ReFlex Switch Input Flit

1	1	3	1	3	3	32	256
Valid	Last	Dest.	Error	Tdest	Tid	Tkeep	Tdata
Control			*Payload*				

8-port ReFlex Switch is implemented for Xilinx XC7Z100 SoC [52] running on single 156.25 MHz clock using Xilinx Vivado 2019.2. The selected SoC has available 277400 LUTs, 554800 FFs and 755 BRAMs. On-chip BRAM modules are 72 bits wide, 512 elements deep, and thus have 36kb capacity each. Listing 4.1 shows the synthesis constraints file used for implementation.

BRAMs with depth of 512 and width of 300 bits are used. The allocation of bitfields for the flit structure is given in Table 4.1 for RTL generation parameters introduced in Section 3.3.5 given as `PORT_COUNT=8`, `STREAM_WIDTH=256` and `ADDR_WIDTH=3`. Xilinx Vivado synthesizer uses 4.5 BRAMs in parallel to span the 300-bits wide flits using on-chip BRAM modules. The half-BRAM optimization is achieved by the synthesizer thanks to Xilinx 7-Series memory architecture [53]. Ports 0, 1, 2, and 3 of the ReFlex Switch are connected to four reconfigurable regions. Other four ports are connected to SoC processor, DDR memory and two 40 Gbps ethernet interfaces [54] in order. All output ports have RA buffers with a depth of 64 flits, using 4.5 BRAMs per port for the 8-port ReFlex Switch.

Listing 4.1: Xilinx Vivado Synthesis Constraints

```
### Define the top level system clock of the design
# 40 Gbps requires 156.25MHz clock
#  => 6.4ns clock period
create_clock -period 6.400 -name clk [get_ports clk]

### Timing exceptions
# di and do are test suite input and output signals
#  => exempt path between these signals and the pins
set_false_path -from [get_ports di]
set_false_path -from * -to [get_ports do]

### I/O Constraints
# standard I/O for XC7Z100
#  => clock rules are handled by 'create_clock'
set_property IOSTANDARD LVCMOS18 [get_ports clk]
set_property IOSTANDARD LVCMOS18 [get_ports di]
set_property IOSTANDARD LVCMOS18 [get_ports do]
```

Resource usages for CAR and DRR [14] for comparison are given in Table 4.2. In this table, the synthesis target frequency was the operating frequency of 156.25 MHz. Here, the maximum frequency is the highest achievable operating frequency of the resulting implementation, and it is provided only for reference. It can be seen that the proposed CAR method can be implemented on FPGA with less than three times the resource consumption of a similar DRR alternative. 16-port instantiations of the DRR and CAR are implemented for scalability comparison. 24- and 32-port CAR implementations are given only to demonstrate the resource usage complexity. LUT usage scales with $O(N^2\lceil\log N\rceil)$ as they are utilized to generate $N\lceil\log N\rceil$ outputs from N^2 inputs. FF usage scales with $O(N)$. For design speed performance comparison, 8- and 16-port designs are also synthesized towards the highest possible frequency. These results are given in the Table 4.3. In all cases DRR implementation achieves slightly higher maximum clock frequency than the CAR implementation.

Table 4.2: FPGA Implementation Results (target: 156.25 MHz) of Different Arbiters

Arbiter	LUT	FF	Max. Freq.
DRR 8 ports	829 (0.3%)	201 (0.04%)	219 MHz
CAR 8 ports	2149 (0.7%)	401 (0.07%)	191 MHz
DRR 16 ports	3427 (1.2%)	589 (0.10%)	161 MHz
CAR 16 ports	9778 (3.5 %)	1067 (0.19%)	160 MHz
CAR 24 ports	36k (13%)	2151 (0.38%)	-
CAR 32 ports	56k (20%)	3085 (0.56%)	-

Table 4.3: FPGA Implementation Results (target: highest) of Different Arbiters

Arbiter	LUT	FF	Max. Freq.
DRR 8 ports	1050 (0.4%)	201 (0.04%)	293 MHz
CAR 8 ports	2617 (0.9%)	401 (0.07%)	232 MHz
DRR 16 ports	4130 (1.5%)	589 (0.11%)	186 MHz
CAR 16 ports	11k (3.9%)	1067 (0.19%)	162 MHz

Resource usages of ReFBM and VOQ for 8-ports are given in Table 4.4. In this table, the synthesis target frequency was the operating frequency of 156.25 MHz. Here, the maximum frequency is the highest achievable operating frequency of the resulting implementation, and it is provided only for reference. Individual FB depths are adjusted in each case for equal total memory usage between VOQ and ReFBM implementations. On each port, 8 FIFOs of 64 flits deep are used for VOQs and 42, 32, 21 flits deep FIFOs are used for 12, 16, 24 FBs. Resource consumption scales approximately $O(\log M)$ where M is the FB count for a fixed port count and total memory usage. For design speed performance comparison, same designs are also synthesized towards the highest possible frequency. These results are given in the Table 4.5. Note that the differences in maximum frequencies are due to the modules being synthesized in isolation without any handshake or status flag signalling. Only the minimum necessary connections for data input and output are made, so the synthesizer strips out the unused fields and the memory usage is reduced to 4 BRAMs for a single port.

Table 4.4: FPGA Implementation Results (target: 156.25 MHz) of Different Input Buffer Management Schemes for Single Port

Input Buffer Type	LUT	FF	BRAM	Max. Freq.
VOQ 8 ports	403 (0.1%)	369 (0.06%)	4 (0.5%)	256 MHz
ReFBM 12 FBs	903 (0.3%)	609 (0.11%)	4 (0.5%)	203 MHz
ReFBM 16 FBs	955 (0.3%)	673 (0.12%)	4 (0.5%)	216 MHz
ReFBM 24 FBs	1295 (0.5%)	889 (0.16%)	4 (0.5%)	190 MHz

Table 4.5: FPGA Implementation Results (target: highest) of Different Input Buffer Management Schemes for Single Port

Input Buffer Type	LUT	FF	BRAM	Max. Freq.
VOQ 8 ports	448 (0.2%)	369 (0.06%)	4 (0.5%)	429 MHz
ReFBM 12 FBs	997 (0.4%)	609 (0.11%)	4 (0.5%)	262 MHz
ReFBM 16 FBs	1106 (0.4%)	673 (0.12%)	4 (0.5%)	325 MHz
ReFBM 24 FBs	1602 (0.6%)	889 (0.16%)	4 (0.5%)	251 MHz

Lastly, Table 4.6 shows the overall implementation results of two comparable configurations of the ReFlex Switch for 8 ports. In this table, the synthesis target frequency was the operating frequency of 156.25 MHz. Here, the maximum frequency is the highest achievable operating frequency of the resulting implementation, and it is provided only for reference. Similar to previously mentioned cases, the total memory usage is adjusted to be the same between VOQ and ReFBM. In this table ReFBM refers to 8 port ReFBM with 16 FBs. For design speed performance comparison, same designs are also synthesized towards the highest possible frequency. These results are given in the Table 4.7. The maximum frequencies for the complete switch setup are similar for all schemes. 300 bit flits span 4.5 BRAMs in width as explained previously. 8 VOQs of 64 flits deep on each port requires 512 flits in depth. RBs also have similar memory configuration. 16 FBs of 32 flits deep on each port also requires 512 flits in depth. BRAM modules used in our design can hold 512 elements, so the total memory usage is $8 \quad 4.5(VOQs=FBs) + 8 \quad 4.5(RBs) = 72$ BRAMs.

Table 4.6: FPGA Implementation Results (target: 156.25 MHz) of Different 8-Port ReFlex Switch Configurations

Configuration	LUT	FF	BRAM	Max. Freq.	Power
VOQ + DRR	12k (4.3%)	3k (0.6%)	72 (9.5%)	163 MHz	0.595 W
VOQ + CAR	13k (4.6%)	3k (0.6%)	72 (9.5%)	165 MHz	0.640 W
ReFBM + DRR	16k (5.7%)	5k (0.9%)	72 (9.5%)	163 MHz	0.698 W
ReFBM + CAR	17k (6.1%)	6k (1.0%)	72 (9.5%)	164 MHz	0.708 W

Table 4.7: FPGA Implementation Results (target: highest) of Different 8-Port ReFlex Switch Configurations

Configuration	LUT	FF	BRAM	Max. Freq.	Power
VOQ + DRR	12k (4.3%)	3k (0.6%)	72 (9.5%)	175 MHz	0.599 W
VOQ + CAR	13k (4.6%)	3k (0.6%)	72 (9.5%)	164 MHz	0.687 W
ReFBM + DRR	16k (5.8%)	5k (0.9%)	72 (9.5%)	173 MHz	0.747 W
ReFBM + CAR	17k (6.1%)	6k (1.0%)	72 (9.5%)	167 MHz	0.792 W

Figure 4.8 shows the synthesis output RTL visualization of an input block with ReFlex Buffer Management enabled. The visualization is obtained with Intel Quartus Prime software version 19.1. Block diagram is colored to indicate the functionality provided by each block. This figure visually demonstrates the overhead RBM_i introduces for an input block.

These results demonstrate that although the proposed methods do not prohibitively increase resource usage, the difference is not negligible and needs to be considered when configuring the ReFlex Switch. The flexibility of ReFlex Switch allows for optimizing towards performance or resource usage.

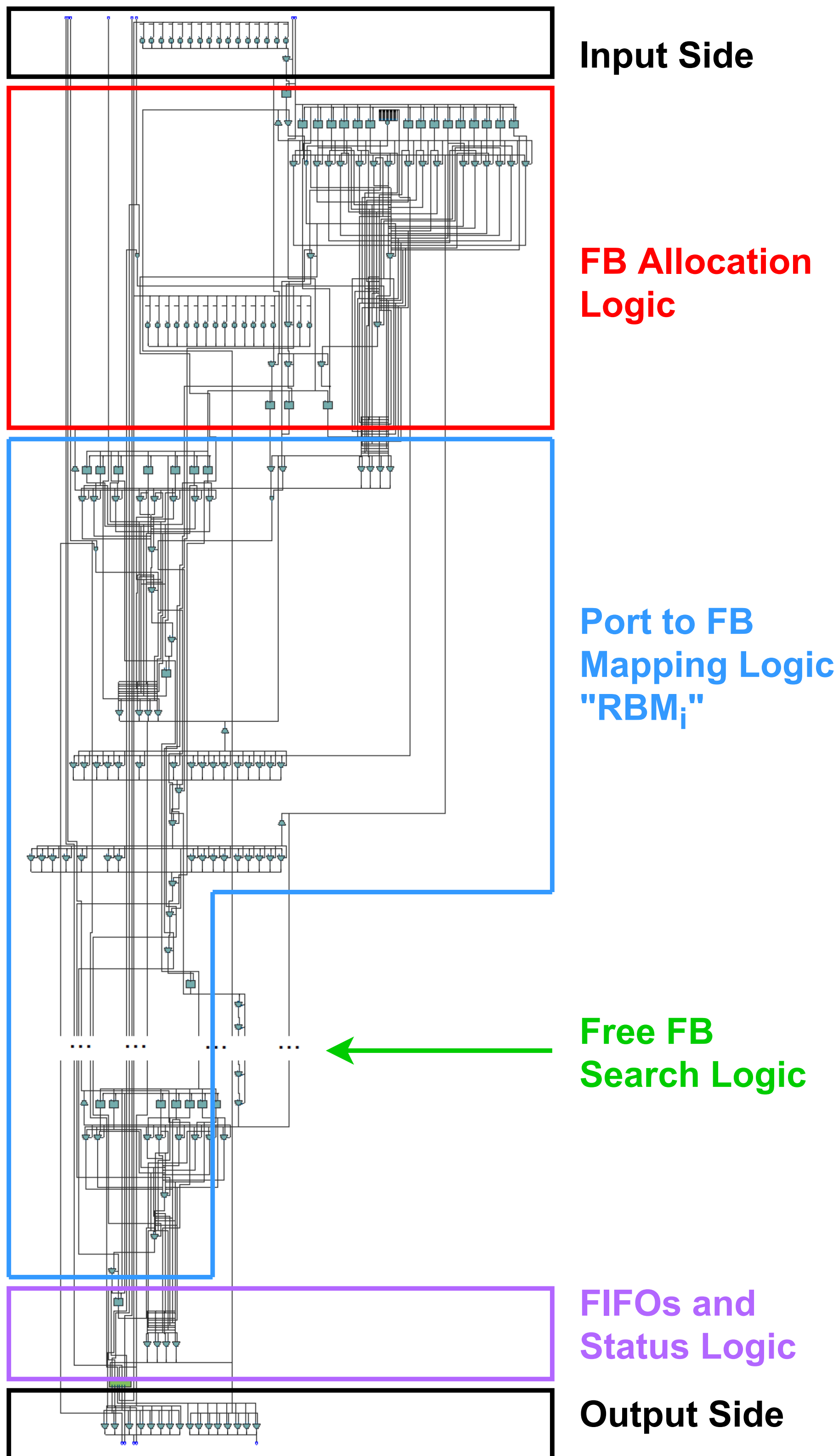


Figure 4.8: Synthesized Datapath of an Input Block with ReFlex Buffer Management

CHAPTER 5

CONCLUSION AND FUTURE WORK

This thesis study proposes, implements and evaluates a novel, scalable on-chip packet switch architecture that we call ReFlex Switch. ReFlex switch has Virtual Output Queue (VOQ) input buffer organization and runs at line speed for scalability to increasing data rates. The target application for the ReFlex switch is hardware accelerated cloud servers which feature custom IP cores interconnected with the CPU and memory with different types of network interfaces and service requirements. To this end, ReFlex Switch incorporates a novel switch fabric arbiter that we call Credit ARbiter (CAR) and a new VOQ buffer management method that we call Reflex Buffer Management (ReFBM). CAR provides service differentiation to the VOQs without affecting the throughput to allocate the available bandwidth according to the requirements of the specific IP core. ReFBM allocates the limited available on-chip memory among the VOQs by the demand of the arriving traffic. To this end, it provides more efficient buffer use compared to fixed size VOQ buffers. Our results show that CAR fulfills its goal by achieving the same throughput as legacy, throughput maximal arbiters with the capability of allocating desired bandwidth shares to the IP Cores. ReFBM achieves better throughput compared to fixed size VOQ organization under the same amount of memory as it allocates unused memory to the VOQs with more input traffic. ReFlex Switch is a complete parametrizable on-chip switch hardware architecture that offers legacy arbiters and fixed size VOQ implementations as selectable options in addition to CAR and ReFBM. Our results show that CAR and ReFBM can be implemented together achieving 40 Gbps throughput with moderate increase in hardware resources.

Our future work has both theoretical and practical studies. In the scope of the theoret-

ical extensions, we plan to add a traffic monitoring module to ReFlex Switch that can dynamically adjust the weights of CAR. One application to benefit from this extension would be connections with short term hot spot or burst behaviour. When the monitoring module detects such traffic, the weights of such connections can be increased to switch them to the output without allocating all the bandwidth at the input queues. In the same framework, the ReFBM can be extended by incorporating thresholds to the allocated FB's to individual VOQs to prevent a certain VOQ dominating the memory unfairly. The practical extension of this work includes testing the Reflex Switch implementation on FPGA in an experimental hardware accelerated cloud server in the laboratory setting with real network traffic and accelerator implementations.

REFERENCES

[1] "Amazon ec2 instance types – amazon web services (aws)." https://aws.amazon.com/ec2/instance-types/, Amazon, Accessed: 2020-08-20.

[2] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, *et al.*, "A cloud-scale acceleration architecture," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 7, IEEE Press, 2016.

[3] A. Yazar, A. Erol, and E. G. Schmidt, "Accloud (accelerated cloud): A novel fpga-accelerated cloud archictecture," in *2018 26th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4, IEEE, 2018.

[4] L. M. Al Qassem, T. Stouraitis, E. Damiani, and I. A. M. Elfadel, "Fpgaaas: A survey of infrastructures and systems," *IEEE Transactions on Services Computing*, 2020.

[5] X. Wang, Y. Niu, F. Liu, and Z. Xu, "When fpga meets cloud: A first look at performance," *IEEE Transactions on Cloud Computing*, 2020.

[6] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, *et al.*, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 269–278, 2018.

[7] N. U. Ekici, K. W. Schmidt, A. Yazar, and E. G. Schmidt, "Resource allocation for minimized power consumption in hardware accelerated clouds," in *2019 28th International Conference on Computer Communication and Networks (ICCN)*, pp. 1–8, IEEE, 2019.

[8] K. Parane, P. P. B. M, and B. Talawar, "Lbnoc: Design of low-latency router ar-

chitecture with lookahead bypass for network-on-chip using fpga," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 25, Jan. 2020.

[9] M. Oveis-Gharan and G. N. Khan, "Reconfigurable on-chip interconnection networks for high performance embedded soc design," *Journal of Systems Architecture*, vol. 106, p. 101711, 2020.

[10] C. Li, D. Dong, Z. Lu, and X. Liao, "Rob-router: A reorder buffer enabled low latency network-on-chip router," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 9, pp. 2090–2104, 2018.

[11] J. Kurose and K. Ross, *Computer Networking: A Top-Down Approach*. Pearson, 8 ed., 2021.

[12] H. J. Chao and B. Liu, *High performance switches and routers*. John Wiley & Sons, 2007.

[13] N. McKeown, "The islip scheduling algorithm for input-queued switches," *IEEE/ACM transactions on networking*, no. 2, pp. 188–201, 1999.

[14] J. Chao, "Saturn: a terabit packet switch using dual round robin," *IEEE Communications Magazine*, vol. 38, no. 12, pp. 78–84, 2000.

[15] B. Hu, F. Fan, K. L. Yeung, and S. Jamin, "Highest rank first: A new class of single-iteration scheduling algorithms for input-queued switches," *IEEE Access*, vol. 6, pp. 11046–11062, 2018.

[16] D. Stiliadis and A. Varma, "Providing bandwidth guarantees in an input-buffered crossbar switch," in *Proceedings of INFOCOM'95*, vol. 3, pp. 960–968, IEEE, 1995.

[17] A. Smiljanic, "Flexible bandwidth allocation in high-capacity packet switches," *IEEE/ACM Transactions on Networking*, vol. 10, no. 2, pp. 287–293, 2002.

[18] D. Bertozzi and L. Benini, "Xpipes: a network-on-chip architecture for gigascale systems-on-chip," *IEEE Circuits and Systems Magazine*, vol. 4, no. 2, pp. 18–31, 2004.

[19] H. K. Nguyen and X.-T. Tran, "A novel reconfigurable router for qos guarantees in real-time noc-based mpsocs," *Journal of Systems Architecture*, vol. 100, p. 101664, 2019.

[20] M. B. Taylor, W. Lee, J. E. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. R. Johnson, J. S. Kim, J. Psota, *et al.*, "Tiled multicore processors," in *Multicore Processors and Systems*, pp. 1–33, Springer, 2009.

[21] A. Olofsson, "Epiphany-v: A 1024 processor 64-bit risc system-on-chip," *arXiv preprint arXiv:1610.01832*, 2016.

[22] C. A. Nicopoulos, D. Park, J. Kim, N. Vijaykrishnan, M. S. Yousif, and C. R. Das, "Vichar: A dynamic virtual channel regulator for network-on-chip routers," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pp. 333–346, IEEE, 2006.

[23] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004.

[24] L.-S. Peh and W. J. Dally, "A delay model and speculative architecture for pipelined routers," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pp. 255–266, IEEE, 2001.

[25] C. Li, D. Dong, X. Liao, J. Wu, and F. Lei, "Rob-router: Low latency network-on-chip router microarchitecture using reorder buffer," in *2016 IEEE 24th Annual Symposium on High-Performance Interconnects (HOTI)*, pp. 68–75, IEEE, 2016.

[26] F. Hassen and L. Mhamdi, "A multi-stage packet-switch based on noc fabrics for data center networks," in *2015 IEEE Globecom Workshops (GC Wkshps)*, pp. 1–6, IEEE, 2015.

[27] T. Karadeniz, L. Mhamdi, K. Goossens, and J. Garcia-Luna-Aceves, "Hardware design and implementation of a network-on-chip based load balancing switch fabric," in *2012 International Conference on Reconfigurable Computing and FPGAs*, pp. 1–7, IEEE, 2012.

[28] R. R. Dobkin, R. Ginosar, and A. Kolodny, "Qnoc asynchronous router," *Integration*, vol. 42, no. 2, pp. 103–115, 2009.

[29] A. Mirhosseini, M. Sadrosadati, F. Aghamohammadi, M. Modarressi, and H. Sarbazi-Azad, "Baran: Bimodal adaptive reconfigurable-allocator network-on-chip," *ACM Transactions on Parallel Computing (TOPC)*, vol. 5, no. 3, pp. 1–29, 2019.

[30] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[31] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[32] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[33] M. Vestias and H. Neto, "Trends of cpu, gpu and fpga for high-performance computing," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–6, IEEE, 2014.

[34] M. Owaida, G. Alonso, L. Fogliarini, A. Hock-Koon, and P.-E. Melet, "Lowering the latency of data processing pipelines through fpga based hardware acceleration," *Proceedings of the VLDB Endowment*, vol. 13, no. 1, pp. 71–85, 2019.

[35] A. Tırlıoğlu, O. B. Demir, A. Yazar, and E. G. Schmidt, "Hardware accelerators for cloud computing: Features and implementation," in *2021 29th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4, 2021.

[36] "Amazon ec2 f1 instances." https://aws.amazon.com/ec2/instance-types/f1/, Amazon, Accessed: 2020-08-20.

[37] D. Korolija, T. Roscoe, and G. Alonso, "Do *f*OS*g* abstractions make sense on fpgas?," in *14th fUSENIXg Symposium on Operating Systems Design and Implementation (fOSDIg 20)*, pp. 991–1010, 2020.

[38] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized fpga accelerators for efficient cloud computing," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 430–435, IEEE, 2015.

[39] T. Hanawa, Y. Kodama, T. Boku, and M. Sato, "Interconnection network for tightly coupled accelerators architecture," in *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, pp. 79–82, IEEE, 2013.

[40] N. Tarafdar, T. Lin, E. Fukuda, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "Enabling flexible network fpga clusters in a heterogeneous cloud data center," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 237–246, 2017.

[41] F. Yazıcı, A. S. Yıldız, A. Yazar, and E. G. Schmidt, "A novel scalable on-chip switch architecture with quality of service support for hardware accelerated cloud data centers," in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pp. 1–4, IEEE, 2020.

[42] F. Yazıcı, A. S. Yıldız, A. Yazar, and E. G. Schmidt, "An on-chip switch architecture for hardware accelerated cloud computing systems," in *2020 28th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4, 2020.

[43] F. Yazıcı and E. Güran Schmidt, "Reflex switch: A flexible on-chip switch architecture for reconfigurable hardware accelerators," *IEEE Transactions on Parallel and Distributed Systems*, vol. To be submitted, 2021.

[44] T. Takaoka, "Multi-level loop-less algorithm for multi-set permutations," *arXiv preprint arXiv:1502.06062*, 2015.

[45] D. E. Knuth, *The art of computer programming Volume 2: seminumerical algorithms*. Addison-Wesley, 1998.

[46] "Axi reference guide." https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf, Accessed: 2021-07-25.

[47] "Axi4-lite ip interface (ipif)." https://www.xilinx.com/support/documentation/ip_documentation/axi_lite_ipif/v3_0/pg155-axi-lite-ipif.pdf, Accessed: 2021-07-25.

[48] "Reflex switch and python script for the parametric rtl code generation." http://accloud.eee.metu.edu.tr/_dw/OEDup/reflex-switch-rel.zip.

[49] "Creating and packaging custom ip." https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug1118-vivado-creating-packaging-custom-ip.pdf, Accessed: 2021-07-25.

[50] "Cycle accurate simulator." http://accloud.eee.metu.edu.tr/_dw/OEDup/simswitch-rel.zip.

[51] H. J. Chao and J.-S. Park, "Centralized contention resolution schemes for a large-capacity optical atm switch," in *1998 IEEE ATM Workshop Proceedings.'Meeting the Challenges of Deploying the Global Broadband Network Infrastructure'(Cat. No. 98EX164)*, pp. 11–16, IEEE, 1998.

[52] "Zynq-7000 SoC data sheet: Overview." https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf, Xilinx, Accessed: 2021-07-25.

[53] "7 series fpgas memory resources." https://www.xilinx.com/support/ documentation/user_guides/ug473_7Series_Memory_Resources.pdf, Accessed: 2021-08-06.

[54] "40Gbps Ethernet solution." HITEK Systems, http://hiteksys.com/pdf/40G-Ethernet-Verification-Report.pdf, Accessed: 2021-07-25.

APPENDIX A

CAR EXECUTION EXAMPLE

Figures A.1, A.2, A.3, A.4, A.5 and A.6 demonstrate a single iteration execution example for the CAR algorithm introduced in Section 3.1. The example follows from the Algorithm 1 and Algorithm 2.

Figure A.1 shows the initialization step of the algorithm. The credit assignment is shown on the upper right side. The numbers are selected for simplicity for the sake of example.

Figure A.2 shows the request collection phase where the input port 2 has a flit destined to output port 0.

Figure A.3 shows the corresponding grant signal generated on the output port 1 using the CAR algorithm.

Figure A.4 shows the corresponding accept signal generated on the input port 1 using the CAR algorithm. Grant and Accept steps also illustrate the application of Shu e(S).

Figures A.5 and A.6 show the credit update in accordance with the algorithm. These also illustrate the application of Random(a; b).

Phase: Init

$G_{i,j} = A_{i,j} =$

(for simplicity)

	j=0	j=1	j=2
i=0	***1***	***2***	***3***
i=1	***2***	***3***	***2***
i=2	***3***	***2***	***1***

Initialization:

$p_G(j) \leftarrow 0, k_G(j) \leftarrow G_{0,j}$

$p_A(i) \leftarrow 0, k_A(i) \leftarrow A_{i,0}$

p_A (0); k_A (0)

0;
1

p_A (1); k_A (1)

0;
2

p_A (2); k_A (2)

0;
3

p_G (0); k_G (0)

0;
1

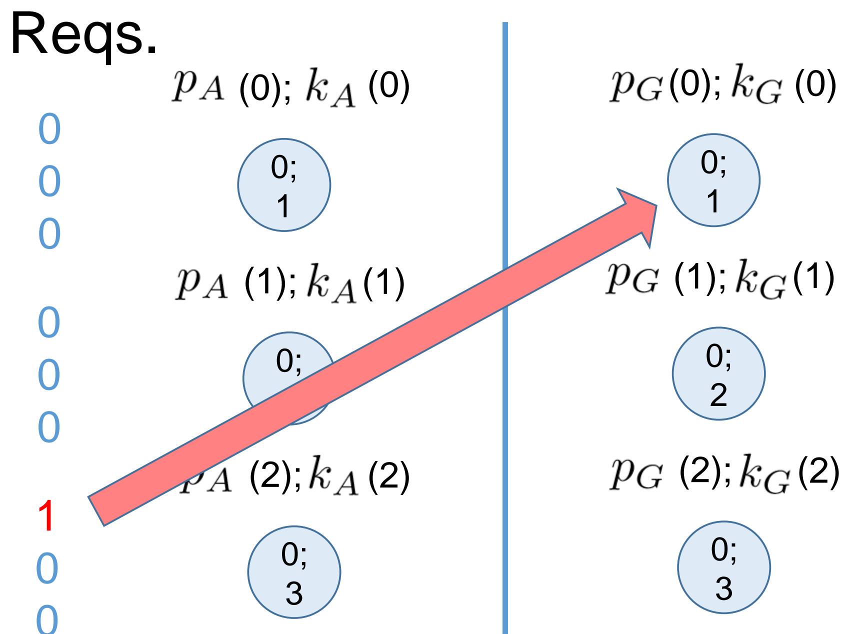
p_G (1); k_G (1)

0;
2

p_G (2); k_G (2)

0;
3

Figure A.1: CAR Execution Example: Initialization



$G_{i,j}$ = $A_{i,j}$ =

	j=0	j=1	j=2
i=0	***1***	***2***	***3***
i=1	***2***	***3***	***1***
i=2	***3***	***2***	***1***

Request Phase:

Request$(i, j) \leftarrow 0$

for (i, j), $i, j \in 0 \ldots (N-1)$ **do**

  if i *and* j *not matched and* i *has flit for* j **then**

    Request$(i, j) \leftarrow 1$

Figure A.2: CAR Execution Example: Request

Phase: Grant

$G_{i,j} = A_{i,j} =$

	j=0	j=1	j=2
i=0	***1***	***2***	***3***
i=1	***2***	***3***	***1***
i=2	***3***	***2***	***1***

p_A (0); k_A (0)

0;
1

p_A (1); k_A(1)

0;
2

p_A (2); k_A (2)

0;
3

Reqs.

p_G(0); k_G (0)

0;
1

p_G (1); k_G (1)

0;
2

p_G (2); k_G(2)

0;
3

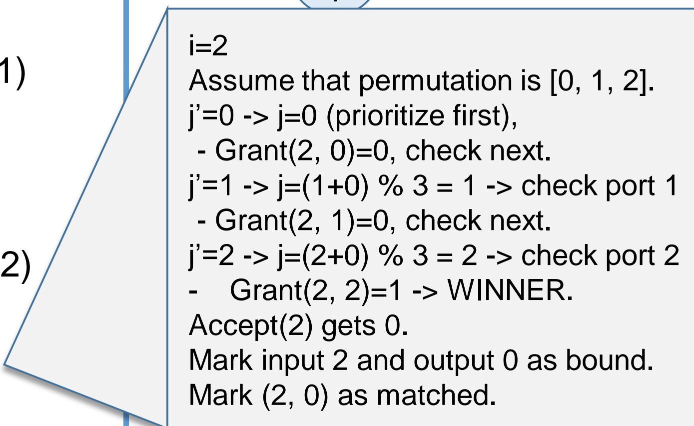
0
0
1

j=0
i’=0 -> i=0
Request(0, 0)=0.

Grant Phase:
$Grant(i, j) \leftarrow 0$
for $j \in 0 \ldots (N-1)$ **do**
  for $i' \in [0, \text{Shuffle}(1 \ldots (N-1))]$ **do**
    $i := (i' + p_G(j))$ **mod** N
    if $Request(i, j) = 1$ **then**
      $Grant(i, j) \leftarrow 1$
      break

Assume Shuffle([1,2]) is currently [2,1].
Then, permutation is [0, 2, 1]
i’=2 -> i=(2+0) % 3 = 2 -> Check port 2.
Request(2, 0)=1 -> WINNER. Reply grant.

Figure A.3: CAR Execution Example: Grant



$G_{i,j} = A_{i,j} =$

	j=0	j=1	j=2
i=0	***1***	***2***	***3***
i=1	***2***	***3***	***1***
i=2	***3***	***2***	***1***

Accept Phase:

$Accept(i) \leftarrow$ EMPTY

for $i \in 0 \ldots (N-1)$ **do**

  for $j' \in [0, \text{Shuffle}(1 \ldots (N-1))]$ **do**

    $j := (j' + p_A(i))$ **mod** N

    if $Grant(i, j) = 1$ **then**

      $Accept(i) \leftarrow j$

      mark i and j as matched

      update credits for i, j

Figure A.4: CAR Execution Example: Accept

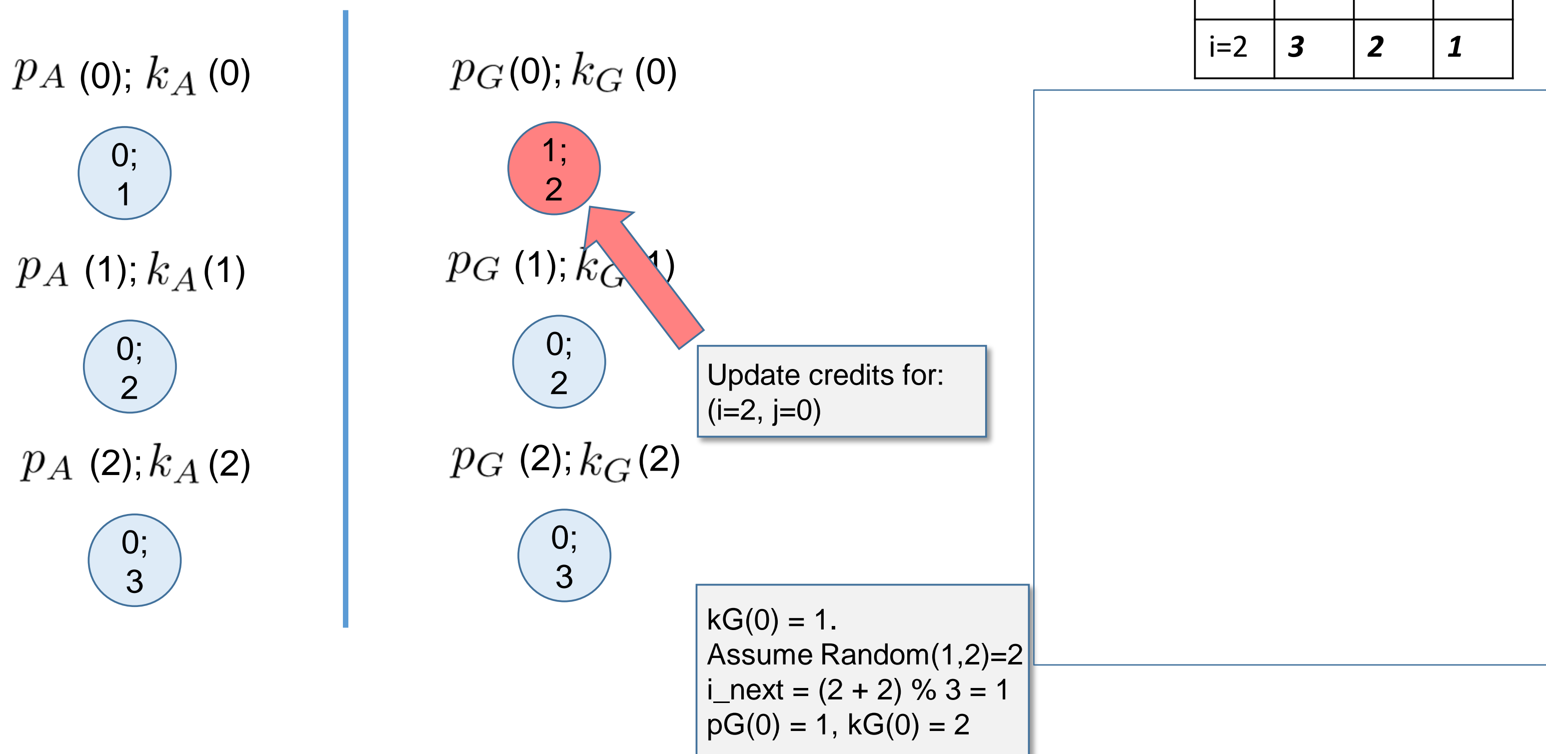


Figure A.5: CAR Execution Example: Accept - Update Credits 1

Phase: Accept – update credits

$G_{i,j} = A_{i,j} =$

	j=0	j=1	j=2
i=0	***1***	***2***	***3***
i=1	***2***	***3***	***1***
i=2	***3***	***2***	***1***

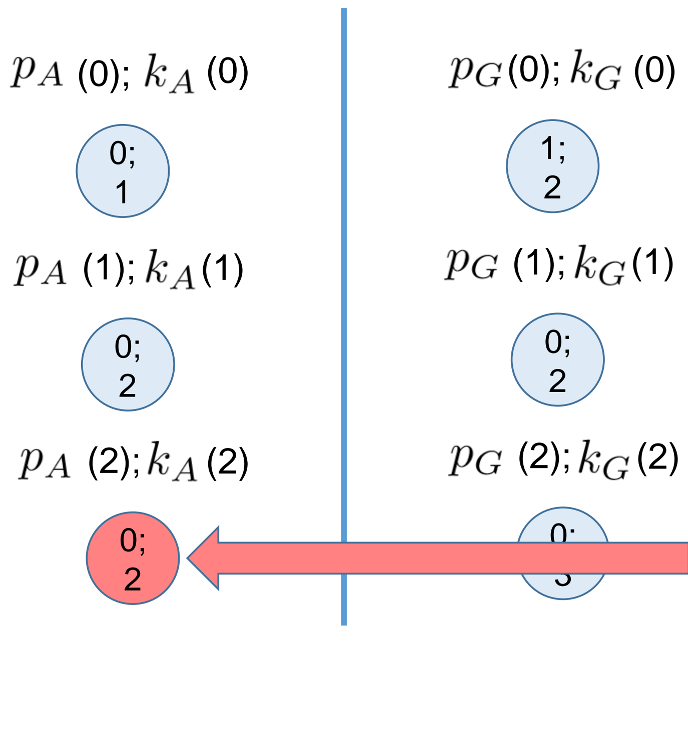


Figure A.6: CAR Execution Example: Accept - Update Credits 2