Boston University

OpenBU **http://open.bu.edu**

Boston University Theses & Dissertations Boston University Theses & Dissertations

2019

High-performance communication infrastructure design on FPGA-centric clusters

https://hdl.handle.net/2144/38207

Downloaded from DSpace Repository, DSpace Institution's institutional repository

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

HIGH-PERFORMANCE COMMUNICATION INFRASTRUCTURE DESIGN ON FPGA-CENTRIC CLUSTERS

by

CHEN YANG

B.S., Wuhan University, 2012
M.S., University of Florida, 2014

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2019

© 2019 by
CHEN YANG
All rights reserved

Approved by

First Reader

Martin C. Herbordt, Ph.D.
Professor of Electrical and Computer Engineering

Second Reader

Michel Kinsy, Ph.D.
Assistant Professor of Electrical and Computer Engineering

Third Reader

Ajay Joshi, Ph.D.
Associate Professor of Electrical and Computer Engineering

Fourth Reader

Woody Sherman, Ph.D.
Chief Science Officer of Silicon Therapeutics

The beginning of knowledge is the discovery of something we do not understand.

Frank Herbert

Acknowledgments

First and foremost, I want to thank my advisor, Prof. Martin Herbordt. During the five year collaboration with him, he offered tremendous help on my research work. It is a great honor for me to have him as my research mentor. I appreciate his supports and insightful suggestions on both my research and my life.

Secondly, I want to thank my committee members. Professor Michel Kinsy has invested a significant amount of time on my thesis and presentation to help me improve my thesis organization. Professor Ajay Joshi has helped my final defense by pointing out the big picture research questions. Dr. Woody Sherman has offered valuable suggestions on the Molecular Dynamics Simulation project.

Thirdly, I want to take this opportunity to thank my colleagues at CAAD lab, Dr. Jiayi Sheng, Ahmed Sanaullah, Qingiqng Xiong, Tianqi Wang, Rushi Patel, Tong Geng, Chunshu Wu, Anqi Guo. They all offered lots of help to my research projects. I also want to thank my friends who share the same lab with me in the past five years, Hao Chen, Tiansheng Zhang, Boyou Zhou, Yenai Ma, Yijia Zhang, Emre Ates, Zihao Yuan, Prachi Shukla, for their support and encouragement.

Fourthly, I want to thank other professors and staff members at Boston University for helping me making progress. I thank Dr. Tali Moreshet's support when I was his teaching assistant. I thank Prof. Rabia Yazicigil for accepting my offer as my defense committee chair. I thank Prof. Anna Swan and Dr. Christine Ritzkowski for their help through various stages of my Ph.D. career.

Lastly, I want to thank my family for their unconditional and limitless support during the past five years. I thank my dearest wife for always stand by me when I feel stressed. I thank my parents for fully supporting my decision in pursuing my Ph.D. degree. I thank my beloved grandparents for all the concern and love over the past 29 years. I cannot be who I am without my dearest families.

HIGH-PERFORMANCE COMMUNICATION INFRASTRUCTURE DESIGN ON FPGA-CENTRIC CLUSTERS

CHEN YANG

Boston University, College of Engineering, 2019

Major Professor: Martin C. Herbordt, Ph.D.
Professor of Electrical and Computer Engineering

ABSTRACT

FPGA-Centric Clusters (FCCs) with the FPGAs directly linked through their Multi-Gigabit Transceivers (MGTs) have a proven advantage over other commodity architectures for communication bound applications. To date, however, communication infrastructure for such clusters has generally only taken one of two simple approaches: nearest-neighbor-only, which is fast but of limited utility, and processor-based, which is general but slow. The overall problem addressed in this dissertation is the architecture, design, and implementation of communication networks for FCCs. These network designs should take advantage of the decades of design experience in networks for High-Performance Computing (HPC) clusters, but should also account for, and take advantage of, unique characteristics of FCCs, in particular, the configurability of the FPGAs themselves.

This dissertation has seven parts. We begin with in-depth implementations of two model applications, Directional Dark Matter (DM) Detection, and Molecular Dynamics (MD). These implementations expose the necessary characteristics of FCC networks from physical through application layers.

The second is the systematic exploration of communication microarchitecture for FCCs, as has been done previously for HPC clusters and for Networks on Chips (NoCs) on both FPGAs and ASICs. One outcome of this part is to find the properties of FCCs that substantially influence the router design space. Another outcome is to create a selection of candidate routers and generalize it so that it is parameterized by routing algorithm, arbitration policy, number of virtual channels (VCs), and other parameters.

The third part is to use the proposed application-aware framework to evaluate the resulting design space with respect to a number of common communication patterns and packet sizes. The results from this part enable two sets of designs. One is the selection of an optimal router for a given resource budget that accounts for all the workloads. The other is to take advantage of FPGA reconfigurability to select the optimal router accounting for both resource budget and a particular workload.

The fourth part is to evaluate the advantages of this approach of adapting the router design to the application. We find that the optimality of the router design varies significantly with workloads. We observe that compared with the router configuration with the best average performance, application-aware router selection can lead to substantial improvement in performance or reduction in resources required.

The fifth part is application-specific optimizations in which we develop several modules and functional units that can provide specific optimizations for certain types of communication workloads depending on the application it going to serve.

The sixth part explores topology emulation, e.g., when a three-dimensional network is used in the computation of an application that is logically two dimensional. We propose a generalized *fold-and-cut* mechanism that both preserves the locality in logical mapping, while also making use of the extra links provided by our 3D-torus fixture.

The seventh part is a table-based static-scheduled router for applications with a static or persistent communication pattern. The router supports various cases, including unicast, multicast, and reduction. By making routing decisions *a priori*, we can bring better load-balance to network links and reduce congestion.

Contents

1 Introduction **1**

2 Background and Related Work **12**

2.1 Background 12

2.1.1 FPGA Background 13

2.1.2 FPGAs in High Performance Computing 13

2.1.3 Communication Background 15

2.2 Related Work 17

2.2.1 Routing Algorithms 17

2.2.2 Existing HPC Routing Solutions 21

2.2.3 Previous Work on FPGA NoCs 25

2.2.4 Previous Work on FPGA Clusters 28

2.2.5 Previous Work on Model Applications 29

3 Model Applications on Multi-FPGA System **32**

3.1 Directional Dark Matter Detection 32

3.1.1 Hardware Configuration 33

3.1.2 Implementation 36

3.1.3 Evaluation 41

3.1.4 Summary on Directional Dark Matter Data Acquisition System 43

3.2 Molecular Dynamics Simulation 44

3.2.1 MD Background 45

3.2.2 MD System Architecture 48

3.2.3 MD System Implementation 60
3.2.4 MD Performance Evaluation 66
3.2.5 MD Strong Scaling onto Multi-FPGA Platforms 71
3.2.6 MD Summary 74
3.3 Summary 74

4 FPGA-Centric Cluster Platform 76
4.1 Design Choices 76
4.1.1 Direct and Integrated Network 77
4.1.2 Network Topology 77
4.1.3 Standard HPC Cluster vs. FPGA Cloud vs. FPGA-Centric Cluster 77
4.2 Novo-G# Architecture 78
4.3 Novo-G# Link Performance 80
4.3.1 Configurations of MGT 80
4.3.2 MGT Link Protocol 82
4.3.3 Measurement of Link Variance 84
4.4 Cast Study on 3D FFT 85
4.5 FPGA-Centric Cluster (FCC) Router Design Constraints 87
4.5.1 Network is direct 88
4.5.2 Long Inter-node Latency 89
4.5.3 Large Phit and Flit Size 90
4.6 Summary 91

5 FPGA-Centric Cluster Router, Part 1: Dynamic Router Design for Unicast Traffic 92
5.1 Conventional Design: VC-based Wormhole Router 92
5.2 Shared Design Components 96

5.2.1 Packet Formatting 96
5.2.2 Routing Algorithms 98
5.2.3 Switch Arbitration Policies 103
5.2.4 Deadlock & Livelock Avoidance 105
5.3 Proposed Router Architecture 1: VC-based Wormhole-Style Router . 109
5.3.1 General Work Flow 110
5.3.2 Addition of Single Large Input Buffer 111
5.3.3 Optimization on Routing Computation (RC) 114
5.3.4 Optimization on VCs 115
5.3.5 Optimization on User Logic with Direct Injection and Ejection Capability 117
5.3.6 Optimization on Switch 118
5.3.7 Addition of Asynchronous Output FIFO 122
5.3.8 Adaptation: Credit Flit Based Flow Control 123
5.3.9 Summary on Current Design 125
5.4 Proposed Router Architecture 2: Wormhole Router with Advance Flow Control 126
5.4.1 Design 1 Drawbacks 126
5.4.2 True Wormhole Architecture 127
5.4.3 Summary on Current Design 132
5.5 Proposed Router Architecture 3: Virtual Cut-Through Style Router . 132
5.5.1 Virtual Cut-Through Background 133
5.5.2 Overall Architecture 133
5.5.3 Input Buffer Unit 134
5.5.4 Switch Allocation 136
5.5.5 Flow Control 137

5.5.6 Summary 137
5.6 Router Performance Evaluation 138
5.6.1 Routing Performance Metrics 138
5.6.2 Communication Patterns 140
5.6.3 FPGA Resource Usage 141
5.6.4 Experiment Setup 143
5.6.5 Routing Performance under Different Workloads 143
5.7 Summary 150

6 FPGA-Centric Cluster Router, Part 2: Dynamic Router Design for Collective Traffic 152

6.1 Background 152
6.2 Motivation 153
6.3 Multicast Support 154
6.4 Reduction Support 156
6.5 Performance Evaluation 159
6.5.1 Resource Usage 159
6.5.2 Collective Performance 159
6.6 Summary 160

7 FPGA-Centric Cluster Router, Part 3: Application-Aware Framework 161

7.1 Application-Aware Framework 162
7.1.1 Software Based Cycle Accurate Simulator 162
7.1.2 HDL Code Generation 163
7.2 Evaluation of Application-Aware Selection Benefits 164
7.2.1 Searching for Optimal Combinations of Routing Algorithm and Arbitration Policy 165

7.2.2 Locate the "Global Optimal" Design 170
7.2.3 Application-Aware Speedup over Global Optimal Design . . . 173
7.3 Summary of Application-Aware Selection 177

8 FPGA-Centric Cluster Router, Part 4: Statically Scheduled Router Design 179

8.1 Table-based Static-Scheduled Router Architecture 179
8.1.1 Table-based Routing 180
8.1.2 Switch Architecture 181
8.1.3 Routing Algorithm 183
8.2 Performance Evaluation on Static-Scheduled Router 186
8.2.1 Hardware Cost 186
8.2.2 Performance 190
8.3 Summary of Statically-Scheduled Router Design 191

9 FPGA-Centric Cluster Router, Part 5: Topology Emulation 192

9.1 Logical Topology to Physical Topology Mapping 192
9.1.1 Folding Mechanism 193
9.2 Topology Emulation Performance 194
9.2.1 Fold and Cut Performance 194
9.2.2 Link Usage Imbalance 199
9.3 Summary 199

10 Conclusion 201

10.1 Conclusion 201
10.2 Future Work 203
10.2.1 Future Work on Inter-FPGA Link 203
10.2.2 Future Work on Router Design 203
10.2.3 Future Work on Model Applications 204

References 205

Curriculum Vitae 222

List of Tables

3.1 Resource utilization for the FE and BE FPGAs. 41

3.2 MD Design Variations 66

3.3 Full system resource usage. Columns 2-4 are post Place & Route. Columns 5-6 give the number of replications of RL pipeline and LR Grid Mapping units in each design. Column 7 lists running frequency of each design. The last two give the stand-alone performance of RL and LR units. 67

3.4 Performance comparison: the middle column shows time to perform one full iteration (23k dataset); the right column shows throughput with a $2fs$ timestep. 69

3.5 Various testing datasets evaluating the impacts on workload mapping selection 69

4.1 Comparison between Interlaken IP and SerialLite III IP 83

4.2 3D FFT Latency Case Study 87

5.1 Reduction tree configuration and resource usage with regarding to different number of VCs multiplexed on each input port. 121

5.2 Switch resource usage on different FPGA chips. 122

5.3 Workloads to evaluate router performance. For tornado pattern, the YSIZE is the number of nodes on the Y dimension. 141

5.4 Design 1: Baseline Router Resource Usage on Stratix V & Stratix 10 FPGAs 142

5.5 Design 2: Advance Flow Control Router Resource Usage on Stratix V & Stratix 10 FPGAs 142
5.6 Design 3: VCT Router Resource Usage on Stratix V & Stratix 10 FPGAs 143

6.1 Application-Specific Support: Resource Usage on Stratix V & Stratix 10 FPGAs 159

7.1 VC number assigned for different patterns 165
7.2 Optimal Designs Under Different Scenarios 173
7.3 Number of each combination of routing algorithm and switch arbitration policy 174
7.4 Application-aware optimal designs targeting batch and continuous mode under a certain workload 175
7.5 Application Aware Optimal Speedup for **Scenario 1** (Global Optimal Design: Design 1, VC = 2, Combination 2) 176
7.6 Application Aware Optimal Speedup for **Scenario 2** (Global Optimal Design: Design 2, VC = 9, Combination 2) 177
7.7 Application Aware Optimal Speedup for **Scenario 3** (Global Optimal Design: Design 3, Combination 2) 178

8.1 Logic elements utilization of RPM router and OCR router on 8×8×8 torus network 189
8.2 Memory consumption of routing tables (including multicast tables and reduction tables) of OCR algorithm on 4×4×4 torus network 189
8.3 Requirements of worst-case buffer size (depth) of online routing and offline routing for these three synthetic patterns, injection rate here is 1 packet per node per cycle 189

List of Figures

2·1 Direct network topology: (a) 2-ary 4-cube (hyper-cube); (b) 3-ary 2-cube (torus) (Ni and McKinley, 1993) 16

3·1 Schematic of a WIMP-induced fluorine recoil in a TPC with strip readout. The WIMP (blue line) collides with a fluorine nucleus (black dot) to create a fluorine recoil (pink arrow). The resulting track of charge drifts to the x-y readout plane (orthogonal, electrically isolated strips), forming a 2D projection of the track (orange dashed arrow). The track position and geometry are found by spatial coincidence between the strips. In this example, strips marked with red ovals at their ends will receive a significant charge signal. 34

3·2 Top-level architecture of our design, showing the eight FE boards that each provide 125:1 multiplexing of detector channels onto a single digital MGT link running at 2.5 Gbps. The BE board handles the 16 Gbps data stream and writes triggered events to disk via UDP on gigabit ethernet (GigE). 35

3·3 Design block diagram. 37

3·4 Data organization of a FE packet (a) and in DRAM (b). 38

3·5 Finite State Machine of the event trigger. 39

3·6 Voltage threshold trigger results. (a) & (c) are the pre-loaded sampled data. (b) & (d) are the triggered data as read out from the BE BRAM module implemented for this test. 42

3·7 Integral trigger results. (a) preloaded low-energy sampled data, (b) integral filtering of the input data, and (c) triggered data. At bottom is the input data filtered with a 512-point integral window. 43
3·8 MD End-to-End System Overview. Details of each section is covered in following figures. 49
3·9 RL Evaluation Architecture Overview 50
3·10 Simulation space about particle P. Its *cell neighborhood* is shown in non-gray; cell edge size is the cutoff radius (circle). After application of N3L, we only need to consider half of the neighborcells (blue) plus the homecell (red). 51
3·11 Cell to RAM Mapping Schemes: (a) Mem 1: all cells mapped onto a single memory module; (b) Mem 2: each cell occupies an individual memory module. 53
3·12 Workload mapping onto force pipelines: (a) all pipelines work on the same reference particle; (b) all pipelines work on the same homecell, but with different reference particles; (c) each pipeline works on a different homecell. 54
3·13 LR Evaluation Architecture Overview 56
3·14 Particle to grid flow: (a) Initial particle position data; (b) Particle to 1D interpolation for each dimension using basis functions; (c) Mapping 1D interpolation results to a 4x4x4 3D grid; (d) Final 64 grid points to 16 independent memory banks 57
3·15 Bonded Force Evaluation Architecture 58
3·16 (a) Double buffer mechanism inside position and velocity cache; (b) Force cache with accumulator. 61
3·17 Force evaluation with first order interpolation. 63

3·18 One instance of the particle to grid conversion equation: The unit is replicated 12 times. Four instances represent the four basis equations for each dimension X, Y, and Z. 64
3·19 Motion Update Pipeline 65
3·20 Motion Update Pipeline 65
3·21 Performance with Different Datasets: (a) Number of RL pipelines that can map onto a single FPGA; (b) RL simulation performance, normalized to Design 1 for each dataset. 70
3·22 Energy Waveform 71
3·23 2D analog of cell to node mapping scheme: (a) single cell mapping to single node; (b) multiple cells mapping to single node; (c) single cell mapping to multiple nodes 72
3·24 3D FFT on Mult-FPGAs 73

4·1 FPGA-based HPC system models: (a) stardard HPC cluster; (b) FPGA cloud; (c) FPGA-Centric Cluster 78
4·2 Novo-G# node architecture detailing the 3D torus network stack. Services provided to the user through RTL or third-party IP are also shown. 81
4·3 Stratix V Transceiver Architecture (Altera, 2014c) 82
4·4 Variations of end-to-end latency over time for four lanes caged inside a single QSFP connector 85
4·5 (a) Latency distribution comparison among four Multi-Gigabit Transceivers (MGTs) (b) Latency distribution comparison between two Altera Stratix V boards 86

5·1 Classic VC-based Wormhole Router Architecture 94
5·2 An exmaple showing how VC-based flow control solves the blocking issue: (a) Non-VC wormhole switching; (b) VC-based wormhole switching. 95

5·3 Packet Formatting Example: The exact bit field may vary based on application requirements. For the shown example, the constraints are: Network Size 8×8×8; Max Supported Packet Size: 64; Max Packet ID: 1024; Priority Value range: $1 \sim 2^{15}$; Misc Routing Info (for O1TURN and RLB): 3-bit; (a) Head Flit; (b) Body Flit; (c) Tail Flit; (d) Single Flit; (e) Credit Flit. 98
5·4 Dividing VCs into two classes and applying a dateline: (a) On positive link class type starts with 0; (b) On negative link class type starts with 1. 106
5·5 VC devided into two classes: (a) VC exclusively used for a certain class; (b) Overlapping VCs from both classes to rebalance buffer usage 108
5·6 Forbidden Turns: (a) forbid Y- to X turn; (b) forbid Z- to X and Y turn. 109
5·7 Proposed FCC VC-based Wormhole Router Architecture 110
5·8 Smart Input Buffer 112
5·9 3-bit pseudo-random binary sequence generator. The 3-bit PRBS's output is XOR'd with the M-bit PRBS's result to create a 3-bit output. Even though a single PRBS sequence can achieve similar results, when there are only 3 bits, the generated number is easy to get correlated with the number of cycles. 114
5·10 Implementation of connection between VCs and switch (assume M ports and N VCs per port): (a) fully connection: VCs direct connect to switch with individual switching for each VC (MN to M^2N); (b) fully mux: Mux VCs attached to the same physical link and grant no more than single input for each output port (M to M); (c) intermediate: VCs direction connect to switch while grant access to no more than a single input for each output (MN to M). 119

5·11 Reduction-Tree-Based Switch 121
5·12 Proposed Wormhole Router with Advance Flow Control 128
5·13 Proposed Virtual Cut-Through Router Architecture 134
5·14 Design 3 Input Buffer Unit Architecture 135
5·15 VC number impact on batch latency when packet size is small. The lower the value is, the better. 145
5·16 VC number impact on batch latency when packet size is large. The lower the value is, the better. 146
5·17 VC number impact on throughput when packet injection speed is fast. The high the value is, the better. 148
5·18 VC number impact on throughput when packet injection speed is slow. The high the value is, the better. 149

6·1 Collective operations: (a) Unicast-based multicast; (b) Tree-based multicast; (c) Unicast-based reduction; (d) Tree-based reduction 153
6·2 Multicast packet path: (a) slicing the cube into multiple YZ planes; (b) inside each YZ plane, grant priority to Y direction, then turn to Z at last 155
6·3 Architecture of Multicast Unit 156
6·4 Reduction packet path: slicing the cube into multiple XY planes; inside each XY plane, grant priority to X direction, then turn to Y, with Z being the last direction. 157
6·5 Architecture of Reduction Unit 158
6·6 Collective Performance 160

7·1 Interface Between two Classes inside Cycle-Accurate Simulator 163

7·2 Batch lantency performance with regarding to 15 combinations of routing algorithm and arbitration policy (small pattern). The lower the value is, the better. 167
7·3 Batch latency performance with regarding to 15 combinations of routing algorithm and arbitration policy (large pattern). The lower the value is, the better. 168
7·4 Throughput performance with regarding to 15 combinations of routing algorithm and arbitration policy (slow injection). The higher the value is, the better. 169
7·5 Throughput performance with regarding to 15 combinations of routing algorithm and arbitration policy (fast injection). The higher the value is, the better. 171
7·6 The aggregated performance number for all six patterns under different scenario and workload. 173

8·1 Node Table Routing Example 180
8·2 Packet Format for Static-Scheduled Routing 181
8·3 Router pipeline: (a) classic four-stage VC-based flow control; (b) proposed seven-staged pipeline 182
8·4 Switch architecture: (a) The switch is connected by seven input and seven output handlers. (b) The input handler has four stages: input buffer consumption, routing table lookup, multicast table lookup, and virtual channel allocation. (c) The output handler has three stages: switch allocation, reduction table lookup, and reduction table write-back. 183

8·5 (a) and (c) is the routing decision made by RPM, (b) and (d) is the expected better routing decision. In (c) and (d), the north and south links are more congested than the west and east links 184
8·6 Pseudo code for proposed offline collective routing algorithm on 3D tours space 186
8·7 Partition evaluation example of OCR algorithm. (a) partition along yz plane, (b) partition along xz plane, (c) partition along xy plane . . 187
8·8 8 regions on a 2D plane, Region 0, 2, 4 and 6 are called corner regions. Region 1, 3, 5 and 7 are called side regions. 187
8·9 part of pseudo code of 2D OCR algorithm for region 0, north link and east link 188
8·10 batched experiments with three typical benchmarks (all-to-all, nearest neighbor, and bit rotation) for two kinds of network size: $4 \times 4 \times 4$ and $8 \times 8 \times 8$. 191
8·11 Average latency of multicast packets in $4 \times 4 \times 4$ network 191

9·1 Step 1: Fold 193
9·2 Step 2: Cut 193
9·3 Step 3: Z-Fold 194
9·4 2D to 3D Mapping Performance: All-to-All Batch Latency 195
9·5 2D to 3D Mapping Performance: All-to-All Throughput 196
9·6 2D to 3D Mapping Performance: Square Nearest Neighbor Batch Latency 197
9·7 2D to 3D Mapping Performance: Square Nearest Neighbor Throughput 198
9·8 Link usage under different fold and cut configurations 199

List of Abbreviations

ACK	....	Acknowledge
AFC	....	Advanced Flow Control
ALM	....	Adaptive Logic Module
API	....	Application Programming Interface
ASIC	....	Application-Specific Integrated Circuits
BRAM	....	Block Random Access Memory
CC	....	Clock Cycle
CCAR	....	Credit Count Adaptive Routing
COTS	....	Commercial Off-The-Shelf
DFHR	....	Dihydrofolate Reductase
DOR	....	Dimensional Order Routing
DSP	....	Digital Signal Processor
FCC	....	FPGA-Centric Cluster
FFT	....	Fast Fourier Transform
FIFO	....	First-In First-Out
Fmax	....	Maximum Operating Frequency
FPGA	....	Field-Programmable Gate Array
FSM	....	Finite State Machine
GPU	....	Graphics Processing Unit
HDL	....	Hardware Description Language
HLS	....	High-Level Synthesis
HOL	....	Head-Of-Line (Blocking)
HPC	....	High-Performance Computing
IC	....	Integrated Circuit
IP	....	Intellectual Property
LJ	....	Lennard-Jones
LR	....	Long-Range
LT	....	Link Transfer
MD	....	Molecular Dynamics
MGT	....	Multi-Gigabit Transceiver
ML	....	Machine Learning
MPP	....	Massive Parallel Processing
MSB	....	Most Significant Bit
N3L	....	Newton's 3rd Law

NIC .... Network Interface Controller
NoC .... Network-on-Chip
O1TURN .... Orthogonal One-turn Routing
OSI .... Open System Interconnection
PBC .... Periodic Boundary Conditions
PCS .... Physical Coding Sublayer
PHY .... Physical Layer of the OSI Model
PMA .... Physical Medium Attachment
PME .... Particle Mesh Ewald
PRBS .... Pseudo-random Binary Sequence
RAM .... Random Access Memory
RC .... Routing Computation
RL .... Range-Limited
RLB .... Randomized Load Balance Routing
ROMM .... Randomized, Oblivious, Multi-phase, Minimal
RTL .... Register Transfer Level
SoC .... System-on-Chip
SP .... Scalable POWERparallel
TCP .... Transmission Control Protocol
TOR .... Top-Of-Rack
TPC .... Time Projection Chamber
VC .... Virtual Channel
VCT .... Virtual Cut-Through
VOQ .... Virtual Output Queue
YARC .... Yet Another Router Chip

Chapter 1

Introduction

Over the last few decades, High-Performance Computing (HPC) has firmly established itself as being central to the advancement of Science and Engineering, joining theory and experiment to be the third pillar of their support. HPC growth has been aided dramatically by performance improvements in communication networks. However, from a raw processing viewpoint, the advance of peak performance has depended, first, on ever faster/denser CPUs, and then, in the last 10-15 years, just on increasing density alone. Specialized architectures, especially those with a higher proportion of compute units, have often helped. But inasmuch as semiconductor technology now faces fundamental physical limits, two new approaches become central to achieving higher net performance: *configurability* and *integration*. Configurability enables hardware to map to the application and vice versa. Integration enables system components that have generally been single function – e.g., a network to transport data - to have additional functionality - e.g., also to operate on that data. In current technology, configurability and integration are jointly manifest in Field Programmable Gate Arrays (FPGAs).

The use of FPGAs in HPC is motivated first by their inherent low power and high computational efficiency. The Novo-G# reconfigurable supercomputer at the University of Florida has a peak performance of 20 Peta-ops (32-bit integer) while drawing less than 15KW of power and requiring no extra cooling infrastructure. And, while floating-point (FP) performance is somewhat lower than that of GPU-based systems,

this is changing in the new generation FPGAs with hard FP cores; the Intel Stratix-10 offers 10 TFlops of peak performance, albeit single precision. The Stratix-10 also uses the same high-bandwidth memory interface technology as the next generation GPUs and so closes the gap there as well. IBMs CAPI interface (Stuecheli et al., 2015), and other emerging interfaces such as CCIX, enable CPUs and FPGAs to share an address space. And FPGAs can generally be configured to use nearly all of their resources leading to high utilization rarely achieved by other computing technologies. The second for use of FPGAs in HPC is that they are at their heart communication switches, having scores of Multi-Gigabit Transceivers (each 100 Mb/s in the next generation), a capability allowing for direct processor-network data transfer, switching, programmable communication, and *integration of communication and computation*.

For many years the gap in speed between processor and network has been growing exponentially. While some of this gap is attributable to the contrasting economics of microprocessors and HPC networks, in the long term this ever-increasing spread remains inevitable. Three ways to mitigate the problem, other than routing more connections, are as follows: (i) make the connection to the network as close to the compute logic as possible, preferably on the same chip; (ii) use the network efficiently, such as by minimizing congestion and thus queueing delay; and (iii) overlap computation and communication (e.g., by performing computation in the network).

All of these ways of mitigating the network/processor performance gap are supported in HPC clusters where FPGAs are the central component (FPGA-centric clusters or FCCs). In FCCs, the FPGAs are interconnected directly through their MGTs in what is often called a secondary or *back-end* network; nodes generally also have the other components expected in HPC clusters such as CPUs, memory, NIC (for a primary or *front-end* network), and possibly additional accelerators such as a GPUs. In particular, (i) within the FPGA, data from application logic can be transferred

directly to the communication FIFOs with zero cycle delay; (ii) when the FPGA is used as a router it supports complex and application-aware communication schemes - this is a central part of this dissertation; and (iii) FCCs improve on standard communication/computation overlap by enabling in-flight data transformations, i.e., the on-FPGA router can be enhanced to perform computations on routed data.

Using FPGAs as the central compute components in large-scale systems has drawn substantial attention in the past few years driven by communication-heavy computation tasks such as Page Rank, Oil and Gas Exploration, Bioinformatics, Molecular Dynamics (MD) simulations, and Machine Learning using Convolutional Neural Networks (CNNs). In the meantime, several major events occurring in industry have consolidated the use of FPGAs for HPC. Intel acquired Altera with the belief that "one third of the data center market could be using FPGAs by 2020"; Microsoft launched Project Catapult to boost the performance of the Bing search engine and to enable real-time Artificial Intelligence (AI) by "augmenting CPUs with FPGAs" and has since deployed millions of FPGAs in their data centers; Amazon's AWS launched the new F1 instance with FPGAs making them easy and inexpensive to use; and Alibaba's Acceleration-as-a-Service introduced another FPGA-enhanced solution for AI inference and other fields that require intense computation.

A further advantage of FCCs, the exploration of which is a contribution of this thesis, is that they enable the communication infrastructure to be tailored to the workload. HPC applications like MD simulation, image processing, and Machine Learning (ML), all use different communication patterns. On the one hand, finding a *globally optimal* router design to fit all applications is unrealistic. On the other hand, different routers generally share similar functions such as routing computation, switching, and input/output buffers. It is known that HDL designs generally require a long development period and have high requirements on developers' experience so that

optimal performance can be reached. In order to fully utilize the reconfigurability of the FPGA-Clusters, while minimizing the gap between general user and FPGA's high-performance capability, what is needed is a router infrastructure with customizable units to fit various applications.

The problem we address in this dissertation is that to date communication infrastructure for FCCs has generally taken one of two approaches: nearest neighbor, which is fast but has limited utility, and processor-based, which is general, but relatively slow. What is needed is for the communication microarchitecture of these systems to be systematically explored, as has been done for HPC clusters and NoCs, both FPGAs (Huan and DeHon, 2012; Kinsy et al., 2013b; Papamichael and Hoe, 2015; Kapre and Gray, 2017) and ASICs (Fallin et al., 2011; Stanford Concurrent VLSI Architecture Group, 2014). The microarchitecture of communication routers has long been a fundamental concern in computer engineering, with many basic principles having reached textbook status over 20 years ago (Duato et al., 1997; Dally and Towles, 2004). Even so, every new technology, change of scale, or workload, reopens this domain for further examination. Such is the current status with respect to FPGA-Clusters for general HPC workloads.

The primary task of this thesis is the systematic design space exploration of communication infrastructure targeting FCCs. The thesis itself is that this exploration results in finding network designs that are uniquely preferred for FCCs in that they have substantially better area-performance than the baseline alternatives.

There has been a vast amount of previous work on router designs targeting various platforms, especially for NoC and ASIC switching chips. Those designs serve as a good starting point for our router design. Among those, the virtual channel (VC) based wormhole switch is the *de facto* standard that has been adopted by both FPGA

NoC (Dally and Towles, 2001; Fick et al., 2009; Huan and DeHon, 2012; Kinsy et al., 2013b; Papamichael and Hoe, 2015) and ASIC switches (Dally and Song, 1987; Kessler and Schwarzmeier, 1993; Stunkel et al., 1994; Scott, 1996). Since the underlying hardware configuration is quite different between NoC and FCC (details of which is presented in Chapter 4), modifications must be made moving the design onto FCCs. These differences result in quite different designs.

The general approach we take is as follows: we start from the conventional wormhole switch optimized for NoCs, and extract the major design components; then, based on the difference of working environment between NoCs and FCCs, we make modifications accordingly; after that, we insert various parameters into our router design for design space exploration; following which, we perform experiments on the router using different types of workloads, for each trying to locate the optimal design parameters for each pattern. Finding the optimal design for the wormhole switch is itself not the end of our task: we look beyond and try different switch alternative like virtual cut-through, and repeat the same steps from beginning. Lastly, based on different application requirements, we add special modules to accelerate certain types of workloads.

We find that there is no "one-for-all" router design that can provide the best-case performance under all circumstances. On one hand, in contrast with ASIC designs, FPGAs feature reconfigurability, which makes application-aware adaptation possible. But on the other hand, the overhead for redesigning a complex router from scratch for each application is huge. And it takes quite a long time to traverse all the possible designs using conventional HDL simulations. To address this, we propose an application-aware framework, which features a software-based cycle-accurate simulator and an HDL generator script. For any given application, the user can extract the communication pattern and feed it into the simulator. The simulator then provides

a routing performance estimation with regards to different design parameters. After this the user can select the parameters that lead to the best performance and use the HDL generator to compose the HDL router design for the FPGA.

We have made a number of contributions which we organize by network layer.

1. **Physical Layer: inter-FPGA link exploration.** We have examined the inter-FPGA link performance on two aspects: using different link-layer protocols and measuring link performance in different levels. Our results show a ~100 ns level link delay. The impact of the result is, due to the relatively longer link delay comparing with that of indirect network routers or Networks-on-Chip (NoC), the conventional wisdom on router design may no longer be directly applied onto the communication infrastructure design. We therefore propose multiple different designs in this thesis to locate the one suits FCC the best.

2. **Network Layer: router architecture design and application-aware framework.** Our router framework has support for two routing approaches: dynamic (online) routing and static-scheduled (offline) routing, which can be used for different applications. Within the dynamic router design, we introduce three different architectures with two wormhole-style routers and a virtual cut-through router. For each one of the three dynamic routers, we add support for five different router algorithms, as well as three switch arbitration policies. While on the static-scheduled router side, we use the table-based routing approach with a new offline routing algorithm (OCR) dedicated for collective operations such as multicast and reduction, which improve the network performance.

   Another contribution on the network layer is the introduction of an application-aware framework. FPGAs feature reconfigurability, which makes application-specific router feasible. But the time cost for redesign a router for each ap-

plication is big. Thus, we introduce a software-based cycle-accurate simulator. With this, for any given communication pattern, we can profile and estimate performance based on various design combinations from our design space, and select the design with the best performance. Then a generation script is used to generate the HDL design that is ready to run on FPGAs. Compared with a "one-for-all" fixed design, our application-aware router can provide performance improvement ranging from 2 % to 110 % with a similar level of resource consumption.

3. **Application Layer: application-specific optimization.** To better serve special needs from certain applications, we added support for tree-based multicast and reduction on top of our dynamic router. Unlike the static-scheduled collective support, we design this module with minimal impact on general non-collective network traffic and generate the dependency graph during runtime. Our simulation shows a best-case improvement of 75 % over the unicast-based broadcast.

   In practice, the overhead for modifying the cluster wiring just to match the logical topology for different applications is unacceptable, especially for clusters with thousands of FPGAs. We include a "Fold-and-Cut" mapping scheme in our application-aware framework that converts a 2D logical topology onto our 3D-torus fixture and performs node coordinate conversion in the process. The post-mapping network is grant freedom in using the extra links provided by the 3D fixture.

4. **Application acceleration on multi-FPGA systems.** One major feature of our router infrastructure is application-aware optimization. Thus, having a good model application is crucial in our design. In this thesis, we look at two applications that require multi-FPGA clusters.

The first one we look at is Directional Dark Matter (DM) detection, which is an experiment on detecting the evidence of an interaction between a dark matter particle and a target nucleus in the laboratory. The system requires in-line processing of data sampled from 1,000 sensor channel at a rate of 1MHz. The hardware infrastructure including eight front-end (FE) FPGAs which performance Analog-to-Digital Conversion (ADC) on 125 sensors, as well as a single back-end (BE) FPGA which collects data from the FE FPGAs via Multi-Gigabit Transceivers (MGTs) and performance pre-processing and storage. Our experiments show good recovery quality and meet our design needs, which sufficiently demonstrates FPGAs communication capability.

The second application we investigate is Molecular Dynamics (MD) simulation. MD is the core of computational chemistry and comprises a large fraction of all supercomputing cycles. It includes a 2D iterative process of force evaluation and motion integration. FPGAs acceleration of MD has been explored for many years. The first generation of complete FPGA/MD systems accelerated only the range limited (RL) force and used CPUs for the rest of the computation. While performance was sometimes competitive, high cost and lack of availability of FPGA systems meant that they were never in production use. In the last few years, however, it has been shown that FPGA clusters can have performance approaching that of ASIC clusters for the Long Range force computation (LR), the part of MD that is most difficult to scale. With the advance of technology, nowadays high-end FPGAs hosts a large number of computation and communication units. In our work, we implement a single-chip end-to-end FPGA simulation system which supports on-chip data storage, force evaluation, and motion update. Our single-chip implementation achieves a comparable performance with regarding the latest-GPU running a production

MD package. We also project the scaling of our single-chip implementation onto FCCs, which serves as a good use case for our router framework.

The rest of the dissertation is organized as follows.

In Chapter 2, we present background information on FPGAs and communication infrastructure. After that, we list a selection of related work on various topics covered in this thesis, including HPC with FPGAs, routing algorithms, current HPC routing solutions, router micro-architectures optimized for NoCs, existing FPGA-clusters and their configurations, and previous work on two model applications covered in this thesis.

In Chapter 3, we cover two model applications that are suitable for acceleration on multi-FPGA systems: data acquisition system for directional DM detection, and MD simulation. We introduce our implementation details on those applications, followed by results and performance evaluation. With these two applications we demonstrate FPGA communication and computation capabilities on real-life applications. We thus justify and motivate building FCCs.

In Chapter 4, we introduce the hardware platform on which we conduct our experiments. We next analyze the difference in hardware configuration compared with other use cases; this differences have a major influence on the router infrastructure design. A contribution here involves the measurement of the inter-FPGA link performance variance. In order to estimate the impact on applications from link variance, we conduct a case study on the 3D FFT. The evaluation results have proved FCC's support for complex multi-hop communication tasks and show that the link variance has little impact on the application. This has the important result of determining the feasibility of static-scheduled routing.

Chapter 5 presents three unicast-based router micro-architectures featuring different flow control mechanisms and resource usage. The three router designs, coupled

with various routing algorithms, different switch arbitration policies, and multiple design parameters, form a design space which can be explored. Our evaluation shows that under different routing patterns, workloads, and evaluation metrics, the optimal design can vary significantly. Given that FPGAs are reconfigurable, this motivates the need for an application-aware optimal router design selection mechanism. The designs covered in this chapter also serves as the basis for other designs presented in the following chapters.

In Chapter 6, we showcase the dynamic router designs optimized for collective operations. These demonstrate a router architecture for supporting multicast and reduction workloads that has minimal impact on general non-collective traffic (e.g., by obeying the same set of deadlock avoidance rules). Results show that the multicast support provides good improvement over unicast-based solutions.

In Chapter 7, we present an implementation of the application-aware framework and demonstrate the benefits of application-aware selection over a fixed *one-for-all* router design. The proposed framework provides optimal routing performance/area on FCCs: compared with a fixed baseline router design, users can either get much better routing performance by using a limited amount of extra resources, or consume far fewer resources with a similar level of performance.

Chapter 8 covers a table-based static-scheduled router design optimized for FCCs, as well as a new offline collective routing algorithm that takes advantage of the knowledge of communication patterns to load-balance network links and reduce congestion. Unlike our previously introduced collective support (in Chapter 6), which supports a fixed pattern, this new design supports arbitrary multicast and reduction patterns. The experiments show that this offline routing solution has significantly better performance and lower hardware cost than a state-of-the-art online routing solution.

In Chapter 9, we present preliminary work on topology emulations on FCCs.

We propose a general *fold-and-cut* mechanism that supports mapping from a 2D logical topology to 3D torus hardware with arbitrary sizes. Results show different optimal fold and cut configurations for different workloads, which further expand the application-aware design space. Another fact we notice is the imbalanced workload on different links, which provides a chance to further optimize our router design by reallocating the buffer resources to links that are heavily used.

Chapter 10 summarizes the entire dissertation and discusses possible future works.

Chapter 2

Background and Related Work

In this chapter, we present background and related work for this dissertation. We start by introducing a few existing problems in the High-Performance Computing (HPC) community and explain how those can be addressed by using reconfigurable computing devices (FPGAs). Then we introduce a selection of background information on FPGA and communication infrastructure. Lastly, we list a selection of previous works that are related to different levels of designs introduced in this thesis.

2.1 Background

HPC remains a critical aspect of nearly all branches of science and engineering. However, it is still facing challenges. To obtain better compute efficiency, a large number of accelerators are used to boost FLOPs/chip-area. But huge computation power comes with huge power consumption. Nowadays high-end system sinks 10 megawatts, and continuation to 100 megawatts is unacceptable. Besides, the use of different types of accelerators makes performance/portability farther away than ever before. Perhaps most importantly, as computing power increases, data movement dominates many applications. Given the above-mentioned challenges, FPGAs could provide a good mechanism to address all those.

2.1.1 FPGA Background

FPGAs feature a large number of configurable logic gates plus a few thousand ASICs like Block RAMs, DSP units, and high-speed communication ports. On top of that, FPGAs also have abundant configurable connections. With a configuration file, we can get a highly efficient application-specific processor. Traditionally the configuration file only defined by Hardware Description Language (HDL), but with the introduction of High-Level Synthesis (HLS) (Xilinx, 2018) and OpenCL support for FPGAs (Altera, 2015a), the programmability of FPGA is catching up.

Nowadays, high-end FPGAs are equipped with 10 s to 100 s of Multi-Gigabit Transceivers (MGTs), which enables direct application-level inter-chip communication with high bandwidth and low latency (Altera, 2014c; Intel, 2019; Xilinx, 2009). In many applications, the MGTs are used for communication with the hosting processor or with a network switch. But MGTs also provide a good opportunity for having a large amount of FPGAs directly connect together and forming an FPGA-Centric Cluster (FCC). The FCCs can make use of the high-bandwidth, low-latency communication link to close coupling FPGAs computation and communication capability, thus to better serve the HPC applications' needs.

2.1.2 FPGAs in High Performance Computing

Prior work in using FPGAs for High Performance Computing includes overviews (Herbordt et al., 2007b; Herbordt et al., 2008a; VanCourt and Herbordt, 2009), general solutions to programmability and performance (Herbordt and VanCourt, 2005; VanCourt and Herbordt, 2005a; VanCourt and Herbordt, 2006a; VanCourt and Herbordt, 2006c), programmability and performance using a commercial tool chain (Yang et al., 2017; Sanaullah and Herbordt, 2017; Sanaullah and Herbordt, 2018b; Sanaullah and Herbordt, 2018c; Sanaullah and Herbordt, 2018a; Sanaullah et al., 2018a), FPGA

system design and architecture (Pascoe et al., 2010; Khan and Herbordt, 2012; Sheng et al., 2015b; George et al., 2016; Sheng et al., 2016b; Sheng et al., 2016a; Sheng et al., 2017; Sheng et al., 2018b; Sheng et al., 2018a), FPGAs used with middleware (Xiong et al., 2018b; Xiong et al., 2018a; Xiong et al., 2019; Stern et al., 2017; Stern et al., 2018), and case studies involving applications.

Molecular Dynamics studies include surveys (Chiu et al., 2008; Chiu and Herbordt, 2010b; Herbordt, 2013; Khan et al., 2013) integration (Gu et al., 2006c), datapath optimization (Gu et al., 2006a; Gu et al., 2006b; Gu et al., 2008), handling neighbor lists (Chiu and Herbordt, 2009; Chiu and Herbordt, 2010a; Chiu et al., 2011), particle mapping (Sanaullah et al., 2016a; Sanaullah et al., 2016b), the long range force using multigrid (Gu and Herbordt, 2007b), the 3D FFT (Humphries et al., 2014; Sheng et al., 2014), the bonded force (Xiong and Herbordt, 2017) and complete FPGA integration (Yang et al., 2019b; Yang et al., 2019a).

Other HPC applications include Discrete Molecular Dynamics (Model and Herbordt, 2007; Herbordt et al., 2008b), Molecular Docking (VanCourt et al., 2004a; VanCourt and Herbordt, 2005b; VanCourt and Herbordt, 2006b; Sukhwani and Herbordt, 2008; Sukhwani and Herbordt, 2010), Microarray Analysis (VanCourt et al., 2003; VanCourt et al., 2004b), Adaptive Mesh Refinement, (Wang et al., 2019b; Wang et al., 2019a), and Machine Learning (Sanaullah et al., 2018b; Geng et al., 2018b; Geng et al., 2018a; Geng et al., 2019b; Geng et al., 2019a).

Bioinformatics work includes studies of dynamic programming based algorithms (VanCourt and Herbordt, 2004; VanCourt and Herbordt, 2007), heuristic sequence alignment such as BLAST (Herbordt et al., 2006; Herbordt et al., 2007a; Park et al., 2009; Park et al., 2010; Mahram and Herbordt, 2010; Mahram and Herbordt, 2012a; Mahram and Herbordt, 2016), multiple sequence alignment (Mahram and Herbordt, 2012b), and other string matching applications (Conti et al., 2004).

2.1.3 Communication Background

Interconnect Classification

Based on the wiring among the compute nodes, we can divide the interconnect into two categories: *Direct network* and *Indirect network*. Inside a direct network, each node has a point-to-point connection to some number of neighboring nodes (Ni and McKinley, 1993), which provides good scalability. While inside an indirect network, the compute nodes are attached at the edge of the network with a clear boundary between compute nodes and network nodes.

The network can also be classified as *Discrete network* or *Integrated Network* based on the location of the compute unit and communication unit. When both units are located on a single node, it is called an integrated node. When the network nodes are physically separate from the compute node, it is defined as a discrete network (Dershowitz et al., 1998).

Since the FCCs are designed to minimize communication overhead and close coupling the computation and communication, the FCC network falls into the category of direct and integrated.

Direct Network Topologies

There are many popular network typologies proposed for HPC clusters: for indirect networks, the widely used types including Fat-tree (Leiserson, 1985), Butterfly (Malkhi et al., 2002), Dragonfly (Kim et al., 2008). While for direction networks Mesh and Torus, which fall into the category of k-ary n-cubes (Dally, 1990), are widely used because their regular topologies simplify the routing (Ni and McKinley, 1993). In a k-ary n-cube, n is the dimension number, k is the number of nodes in each dimension. A few examples is shown in Figure 2·1.

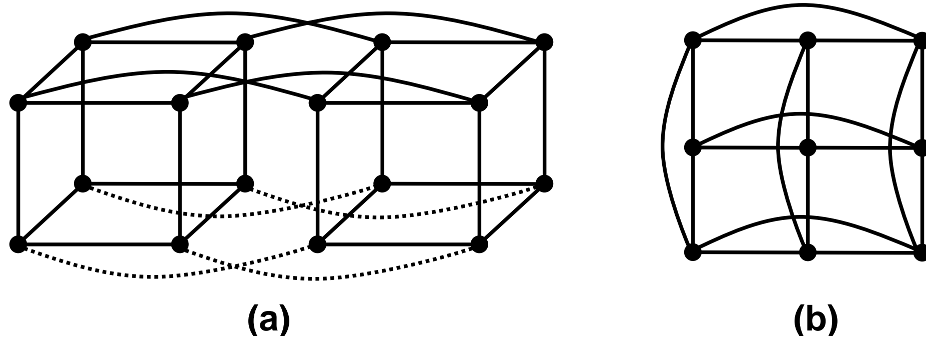

Figure 2·1: Direct network topology: (a) 2-ary 4-cube (hyper-cube); (b) 3-ary 2-cube (torus) (Ni and McKinley, 1993)

Switching Techniques

Based on the way data id transferred from source to destination, we can classify switching techniques into *Circuit Switching* and *Packet Switching*. With circuit switching, the connections between source and destination are set up before data transmission. When the transmission starts, the path is held by the source and destination pair and data flows continuously through the connection (Porter et al., 2013). Circuit switching can guarantee communication quality through dedicated bandwidth. However, there are situations when no data is transferred on the link, thus lead to resource waste. While in packet switching, the data is divided into smaller units named "packet". Each packet carries the destination information, but the intermediate path is determined at each intermediate hop. With packet switching, the network resource is shared by all requests. However, when multiple packets are contesting for the same resource, only one a single packet can win the arbitration. As a result, the transmission quality for other packets is influenced.

In packet switching, there are various flow control methods, including *Store-and-Forward*, *Viral Cut-Throughput (VCT)*, and *Wormhole*. With store-and-forward, each node waits for the entire packet to arrive, then forwards the packet to the next one (Lam, 1976). When packet size is large, the waiting time on each intermediate

node is long, thus resulting in long latency. To overcome such overhead, VCT is proposed, with the packet being forward to the next node as soon as the first few frames of the packet is received and resources on the next node are available. Thus, this design eliminates the waiting time for the entire packet on each intermediate node (Kermani and Kleinrock, 1979). However, VCT still requires that each node provides enough buffer space for the entire packet. In case of blocking occurs, the intermediate node must be able to hold the entire packet in place. Both store-and-forward and VCT require a large buffer space. However, there are cases when only a tight resource budget is allocated for routers. In (Dally and Seitz, 1986), Dally and Seitz first propose the wormhole switching method: with which the packet is broken down into smaller units, *flits*, and the flow control is performed based on that. When the head flit is blocked at one intermediate node, the flits belong to the same packet is buffered in place. As a result, the wormhole router does not need large buffers. We further introduce the details of this method in Chapter 5.

2.2 Related Work

2.2.1 Routing Algorithms

Routing algorithms can be classified based on different implementation characteristics. Based on locations where routing decisions are made, we can classify them as *Source Routing* or *Distributed Routing* (Ni and McKinley, 1993). For source routing, the path a packet is going to traverse is generated on the source node and carried in the packet header. Depending on the length of the pre-determined path, the routing information can occupy a various number of bits inside the packet header, thus increasing communication overhead. For distributed routing, the routing decisions are made on each intermediate node through which packets are going to traverse. Distributed routing solutions reduce the packet overhead at the cost of adding routing

computation logic on each node.

Routing algorithms can also be classified as *Oblivious Routing* or *Adaptive Routing*, based on the path selection methods (Mohapatra, 1998). With oblivious routing, the path is fixed given a source and destination address. On the contrary, adaptive routing algorithms select among multiple available paths based on the network workload. In general, oblivious routing algorithms have a more straightforward evaluation process, thus requires fewer resources on routing computation. However, under some circumstances, oblivious routing leads to the unbalanced workload on different links, and as a result, lead to bad performance.

Routing algorithms can be further classified based on the distance the packets need to traverse. In *Minimal Routing*, the packets take one of the shortest paths between source and destination node. With *Non-Minimal Routing*, as indicated by the name, can take an arbitrary path provided in the network. Even at the cost of traversing the longer distance, non-minimal routing provides an escape route when multiple packets contesting the same link, thus alleviate the number of packets need to be buffered on an individual node, or reduce the chances of packet dropping.

Dimensional Order Routing (DOR)

Designed for a large scale MIMD machine, DOR was first proposed in (Sullivan and Bashkow, 1977). DOR is also referred to as XYZ routing. Following dimensional order, all the packets must start routing along X dimension. They are not allowed to enter Y dimension until their current position has the same X coordinates as their destinations. Similarly, packets are not authorized to enter Z dimension until their current position has the same X and Y coordinates as their destinations. The XYZ routing is an oblivious routing algorithm. Its rules can be easily implemented in hardware with little cost. There is a lot of large-scale parallel systems using it because of its simplicity on hardware implementation.

Valiant Routing (VAL)

VAL was proposed in (Valiant, 1982). It is a two-phase, randomized routing in which a single random node is used as an intermediate destination. VAL is not necessarily a minimal routing algorithm since the randomly selected node can be anywhere in the network. The algorithm first routes the packet to that intermediate node following XYZ dimensional order. After that, the packet moves towards the destination, again following the dimensional order. Comparing with DOR, VAL introduces a level of randomization to avoid the congestion on a single dimension when the network load is heavy. However, the non-minimal path can potentially worsen the network congestion status.

Randomized, Oblivious, Multi-phase, Minimal Routing (ROMM)

ROMM was first introduced in (Nesson and Johnsson, 1995). It resembles VAL in a way that it randomly selects intermediate nodes and let the packet move through those nodes before reaching the destination. However, it obeys minimal routing by only selecting intermediate nodes from those that are in between the source and destination. ROMM also support multiple phases, which means multiple intermediate nodes can be selected. Thus, ROMM adds more randomization into the routing paths.

Orthogonal One-turn Routing (O1TURN)

O1TURN is another minimal routing scheme (Seo et al., 2005). The authors claim that O1TURN can achieve near-optimal worst-case throughput. Similar to DOR, O1TURN still have a dimensional order, but it randomly selects among different orders. Taking 3D-torus for example, O1TURN can issue a different dimensional order among the six possible combinations: XYZ, XZY, YXZ, YZX, ZXY, ZYX. Thus, O1TURN also brings some level of randomization into the network.

Path-Based, Randomized, Oblivious, Minimal Routing (PROM)

PROM (Cho et al., 2009) can be viewed as a generalization of both ROMM and O1TURN. But unlike ROMM, which can only have a fixed number of intermediate nodes, randomization is happening at each intermediate node. Between the destination and the intermediate node, PROM can select at most 3 potential output directions on each node in a 3D topology. PROM selects one of the outputs with a programmable possibility, thus achieves randomization on each node.

Randomized Load Balance Routing (RLB)

RLB (Singh et al., 2002) is a non-minimal routing algorithm. Inside a torus network with bidirectional links, RLB aims to balance the link load within any given rings. Unlike minimal routing algorithms, which forces each packet to take the shortest path along each dimension, in RLB, packets have a probability of taking either direction on a given a ring. To avoid generating too much overhead, the chances for taking the longer paths is proportional to the difference in path length between the two directions. RLB only performs selection once to avoid livelock.

Adaptive Routing Algorithms

There is a vast amount of literature on different adaptive algorithms. Given the information of free and allowed output ports, an output direction could be chosen randomly (Feng and Shin, 1997) or based on the remaining hop count in each dimension (Badr and Podar, 1989). An algorithm called NOTURN tries to avoid turns by following a dimension until it is either exhausted or blocked (Glass and Ni, 1994). These three algorithms are unaware of congestion information of neighboring nodes. Therefore, they are inherently unable to deal with the imbalance issue. There are other algorithms that gather the local/global congestion information before making routing decisions. (Dally and Aoki, 1993) counts the numbers of idle VCs on neigh-

boring nodes and route packet to the direction with the most significant number of VCs. If we want to implement this algorithm in our network, we need extra logic to monitor the idle VC numbers. Moreover, additional bandwidth is consumed to transfer this information to neighboring nodes. (Singh et al., 2003) counts the available slots at the output buffers to determine routing decisions. In (Kim et al., 2005), the Credit Count Adaptive Routing Algorithm (CCAR) is introduced for NoC routers with 2D mesh or torus topology. On each node, the routing computation unit first selects the possible outgoing directions each packet can take during a “pre-selection” process. Then it compares the available credits from each selected downstream node from the first stage to find the one with maximum credits and set is as the output direction.

Deflection Routing Routing Algorithms

The deflection routing algorithm was first introduced in (Baran, 1964), which “misroute” those packets that are not winning the arbitration. As a result, it saves buffer space. The deflected packets are moving further away from the destination and can be route back at a later cycle. To avoid livelock, in (Moscibroda and Mutlu, 2009), the authors use the oldest first mechanism: if a packet is in the network for too long, it obtains a higher priority on each node. Thus, the packets that are deflected too many times can eventually reach the destination. Deflection routing requires fewer buffer resources. However, the misrouting cost can be enormous, especially when the link delay is long.

2.2.2 Existing HPC Routing Solutions

According to (Top500, 2019), the communication solution now used in large supercomputers falls into the following categories: Gigabit internet, Infiniband, Omini-Path, and Custom Interconnects. In this subsection, we cover the existing routing

solutions on HPC clusters.

Gigabit Ethernet

10 Gigabit Ethernet is the latest Ethernet standard that transforms data frames at a rate of 10 Gbps. A selection of technical details is introduced below.

Physical Link: 10G Ethernet cable encloses 4 differential pairs (8 wires) forming 4 individual lanes. The basic unit transferred on the physical link is an octet referred to as "symbol", which serves as the basic unit composing a frame. The frame is transmitted with the most-significant octet (byte) first; within each octet, however, the least-significant bit is transmitted first. 10G Ethernet can support various size packets with an average size of 1500 Byte.

Flow Control: Ethernet transfer packets in a handshaking manner: every time receiver received a packet, it sends an ACK packet back, acknowledging the arrival of the packet. Then the sender moves on and sends the next packet. Ethernet realize flow control on both *Data Link* layer and *Transport* layer. On the data link layer, if the receiver's buffer is almost fully consumed, a *Pause On* frame is broadcast to all the hosts that connecting to it. The requester stops sending new packets until a *Pause Off* frame is received, or a countdown timer expires. On the other layer, Transport Layer, Transmission Control Protocol (TCP) employs a *Sliding Window* mechanism (Cerf and Kahn, 1974). Generated by the receiver, the window information is incorporated inside an ACK packet, which specifies the maximum number of packets on the fly. The sender can send up to the window size amount of packets before it receives an ACK packet. Based on the network status, the receiver can adjust the window size and attach a new window size in the ACK packet.

InfiniBand

The InfiniBand (IB) is another network communication standard widely used in HPC environment. Comparing with Ethernet, it features higher throughput and lower latency (Pfister, 2001). Similar to Ethernet, IB is configured as an indirect network with each server and storage system connected to a dedicated switch. Over the years there are various versions of IB introduced. The latest version, HDR, featuring a 600 Gbps bandwidth under the configuration of the 12× link with 600 ns latency (Mellanox, 2018). A selection of technical details of IB is introduced below.

Physical Link: IB defines the link speeds at the physical layer, and can be configured as 1×, 4×, 12× (Mellanox, 2003). Each link is a four-wire serial differential connection (two wires in each direction) that provide a full-duplex connection. IB transmits data in packets of up to 4 KB. There are two types of packets: management and data packets. The first one is used for link configuration and maintenance, while the later one is used for data payload.

Credit Flow Control: IB uses credit-based flow control mechanism (Infiniband, 2015). On the *Message Level*, a transmission is initialized by the sender by sending a transmission request to the receiver when it has enough credits. The credit information is carried inside the ACK message from the responder to the requester. Once acknowledged by the receiver, the sender starts sending the whole message. On the *Link Level*, IB utilizes "absolute" credit-based flow control. Unlike traditional credit-based flow control that provides incremental updated to the credits, IB specifies a "credit limit", which marks the total amount of data that is authorized to go through since the link setup. Every time a packet send out, the credit number is decreased by one. To avoid inconsistencies in credit count on both sending and receiving side due to transmission error, the sender periodically sends out the total number of packets it sends out, which is used on the receiver to re-synchronize state.

Intel Omnipath

The Intel Omnipath is designed for the integration of CPU and memory components to enable low latency and high-bandwidth communication within datacenters. It provides 126 Gbps bandwidth under the configuration of 16× PCIe lanes, with a latency of 100∼110 ns inside the switch (Birrittella et al., 2015).

Link Transfer (LT) layer: Each packet is divided into equal-sized 64-bit Flow Control Digits (FLITs). All LT packets share the same size of 16 FLITs. There are four types of FLITs: head, body, control, and command. Among which, head and body FLITs are general data flits. Control FLITs are used to orchestrate the retransmission protocol. Command FLITs are used to transmit flow control credits. The flow control is credit-based, which is similar to what is introduced in (Dally, 1992). Omnipath supports up to 31 Virtual Lanes (VLs), each with dedicated storage space, as well as a shared resource pool that can be dynamically reallocated.

Cray Interconnects

Cary has developed a variety of HPC networks over the years. The T3D (Kessler and Schwarzmeier, 1993) is Cray's first generation of Massive Parallel Processing (MPP) machine. It is configured under a bidirectional, 3D-torus network with a link width of 24-bit running at 150 MHz. Wormhole switching is used in T3D: it relies on the ACK signal to return the downstream Virtual Channel (VC) status through a dedicated link to achieve flow control (Scott and Thorson, 1994). The next generation, T3E, introduced adaptive routing on top of T3D (Scott, 1996). On top of the original VC-based wormhole router, T3E assigns one of the VCs as an adaptive VC. When flits enter the adaptive VC, it not necessaries following the dimensional order and turn to any other possible outgoing direction as long as it is in line with the minimal path. The Yet Another Router Chip (YARC) is an ASIC developed for Cray BlackWidow

scalable vector multiprocessor (Scott et al., 2006). It features a high-radix clos network with 64 18.75 Gb/s bidirectional ports on each chip. Starting from YARC, the routing solutions inside cray supercomputers are using an indirect network. Later on, Cray introduced two ASICs, Seastar (Brightwell et al., 2006) and Gemini (Cray, 2010), to deliver high performance on MPI applications and filesystem traffic, with the difference being Gemini can provide more network interface controllers (NICs) to connect to network nodes. In (Alverson et al., 2012), Cray introduced Aries, a System-on-Chip (SoC) with four NICs with Dragonfly topology. Most recently, they propose a new interconnect, Slingshot, with a better flow control mechanism to avoid congestion (Cray, 2019). They introduce a "global adaptive" routing mechanism along with a new congestion control method which can hold packets from injection into the network on the node that is further away from the congestion site.

IBM Custom Interconnects

The Scalable POWERparallel (SP) is a series of supercomputers from IBM. They are designed to have thousands of microprocessor-based nodes. The processing nodes are non-switching devices, including processors, I/O nodes, and gateways. To support the data exchange among those units, the Vulcan Switch Chip is introduced in (Stunkel et al., 1994; Stunkel, C.B., et al., 1995). The chip featuring 8 receivers and 8 transmit modules with a centralized queue in the middle of the chip to buffer those packets that are not winning the switch arbitration. The Vulcan switch chip used a buffered wormhole routing method, which makes use of the large centralized queue to holding the flits from the entire packet even the head flit is blocked on the node.

2.2.3 Previous Work on FPGA NoCs

There is a vast amount of literature on FPGA-based NoC router design. In this subsection, we list a few widely known ones.

CMU CONNECT

The CONNECT is an NoC generator that is specifically tuned for FPGAs (Papamichael and Hoe, 2012; Papamichael and Hoe, 2015). Their design is based on the classic VC-based wormhole router. CONNECT supports two router micro-architectures: simple input buffering and VC-based, along with other design parameters like number of VCs and different buffer allocation mechanisms. They are the first to discuss the difference in design philosophy between NoCs designed for ASICs and FPGAs. They believe that FPGA NoC router should have a single-stage pipeline to reduce the routing latency. They also suggest routers should be tightly coupled with more wires between adjacent routers with a dedicated link for flow control signals. The design philosophy of CONNECT rather is to achieve minimal resource utilization while maintaining a moderate operating frequency.

Split-Merge

In (Huan and DeHon, 2012), the authors propose a pipelined NoC router named Split-Merge PS. which also adopts the VC-based wormhole router design. This design takes a different approach comparing with CONNECT, by introducing multiple pipeline stages to the router pipeline to improve the operating frequency. The authors implement two sets of buffers: an input buffer and an output buffer. The "split" occurs in routing computation stage which sends to different output queues based on outgoing direction; while the "merge" happens when the output arbiter select among the output queues from each input queues. Split-Merge is reported to run at 200∼300 MHz depending on different configurations, while in comparison, CONNECT can only run at around 100 MHz. The cost of this implementation, on the other hand, is higher resource utilization.

BSORM

In (Kinsy et al., 2009), the authors first propose a static-scheduled routing solution, BSORM, on NoCs. This design is optimized for point-to-point communication pattern to improve network throughput given rough estimates of flow bandwidth. On top of that, the authors introduce a turn model and static VC allocation to achieve deadlock-free. The authors prove that if the communication pattern is known as a priori, static-scheduled routing can provide better performance over dynamic routing (XY and YX routing) solutions.

Heracles

In (Kinsy et al., 2013a), the authors propose the Heracles framework that generates an entire multicore processor with an NoC inside. It supports two types of oblivious routing methods: off-line generated table-based routing and dynamic routing based on DOR routing algorithm. VC-based wormhole switching is also used in this design.

Hoplite

In (Kapre and Gray, 2015; Kapre and Gray, 2017), the authors introduced Hoplite and HopliteRT NoC, which targeting the unidirectional 2D-torus topology. The design goal is minimizing resource usage. Unlike buffered NoC routers, this design featuring a buffer-less deflective routing method. When multiple packets are contesting for the same output directions, the packet that failed the arbitration process is misrouted to a node that is further away from its destination instead of buffering it in place. Following this design idea, Hoplite router is simplified to a few muxes with no storage units, at the cost of poor routing performance. To reduce the deflection rate, in (Garg et al., 2019), an optimized design, HopliteBuf, is introduced. It adds a small buffer to hold the deflected packets temporarily. The main idea behind Hoplite is to trade performance for less resource usage.

2.2.4 Previous Work on FPGA Clusters

There have been many attempts to design multi-FPGA systems, with different topologies and optimized for different applications. In this subsection, we present a selection of well-known ones.

Reconfigurable Computing Cluster (RCC)

The reconfigurable computing cluster (RCC) project is a multi-FPGA cluster developed at UNC Charlotte by the Sass Group (Sass, R. et al., 2007). All the FPGAs are directly connected with MGT links forming a 3D torus network. Besides that, they developed a configurable network abstraction layer called AIREN to provide friendly and efficient on-chip and off-chip network interfaces (Schmidt et al., 2009; Schmidt et al., 2012). Sass, et al. also designed specialized hardware cores to offload MPI collective operation from software (Gao et al., 2009).

Toronto Molecular Dynamics Machine (TMD)

The Toronto Molecular Dynamics machine (TMD) was a multi-FPGA system with MGT interconnections (Patel et al., 2006) developed at the University of Toronto. A major application was Molecular Dynamic simulations. Arun et al. instantiated soft processors on each FPGA and implemented a specialized MPI library called TMD-MPI to handle inter-FPGA communication (Peng et al., 2014). While soft cores have the potential for generality, they can cause high latency on critical paths.

Zedwulf

Zedwulf is a 32-node multi-FPGA SoC. The inter-FPGA communication is implemented by connecting all the 32 FPGAs via an Ethernet switch (Moorthy and Kapre, 2015). Although it is a good practice at multi-FPGA SoC implementation, the

Ethernet-based interconnection introduces much higher latency overhead than MGT interconnection.

Catapult

The most significant FPGA application project of the last decade is Microsofts Catapult project, where they have integrated FPGAs into their data center with 1,632 nodes (Putnam, A. et al., 2014). On each node, an FPGA daughter card is plugged on the back-plane with direct connections to the neighboring node via MGTs, organized in a 6×8 2D torus network within each rack. The FPGAs are working in a pipeline manner with limited routing supports, which fits the ranking service requirements for the Bing search engine. In their second generation, Catapult V2, the direct connection between the FPGAs are removed due to the high maintenance cost and limitations on direction FPGA communications (FPGA can only directly communicate with its neighbors reside in the same rack) (A. M. Caulfield et al., 2016). They placed the FPGA between the NoC and CPU as a network side "bump-in-the-wire". Inter-FPGA communications are now achieved by Top-Of-Rack (TOR) and multiple levels of commodity routers. Their focus of the system is on scalability over the inter-FPGA communication performance.

2.2.5 Previous Work on Model Applications

In this thesis, we introduce two model applications targeting multi-FPGA systems: Direction Dark Matter (DM) Detection and Molecular Dynamics (MD) simulations. The DM detection project aims to design a data acquisition system that samples data from thousands of senors and processes those at line rate. Due to physical limitations, it needs multiple Front-End (FE) FPGAs with each handling a subset of channels. To locate the event of interest, it needs to process the data from all the channels together, thus requires a single Back-End (BE) system to collect the data from FE

in-time. The MGTs on FPGAs provide an excellent opportunity for achieving that.

MD is a well-know computational intensive application, but it is hard to achieve strong scaling on multi-FPGA systems. To bring the data close to computational units, MD usually distribute the particle data onto multiple chips. However, during the force evaluation process, it requires frequent data exchange with other FPGAs, which have high requirements on communication performance.

Since we examine those two applications in the later chapters, we list a selection of related work on those two applications.

Previous Work on Directions Dark Matter Detection

ASICs and FPGAs have been used in dark matter detection before. For example, the NEWAGE directional dark matter experiment uses an Amplifier-Shaper-Discriminator (ASD) ASIC for data collection and trigger application which can be used in a micro pixel chamber (Orito et al., 2004). However, that chip records Time-Over-Threshold (TOT) across four channels instead of recording the entire waveform. The MIMAC directional dark matter experiment created a custom ASIC to sample and trigger on 1024 channels at a rate of 50 MHz, again, recording only TOT, not the full waveform (Richer et al., 2011). More recently, the LUX non-directional dark matter experiment has developed an FPGA-based trigger system to monitor the signal on 122 Photomultiplier Tubes (PMT). The FPGA implements digital filtering and an event trigger based on the analog sum of eight PMT signals, and a COTS waveform digitizer records waveforms (Akerib et al., 2016). That system has 10× fewer detector channels and does not implement channel-by-channel triggering.

Previous Work of Molecular Dynamics Simulation on FPGAs

FPGAs have been explored as possible MD accelerators for many years (Azizi et al., 2004; Hamada and Nakasato, 2005; Scrofano and Prasanna, 2006; Kindratenko and

Pointer, 2006; Alam et al., 2007; Chiu and Herbordt, 2010a; Cong et al., 2016). The first generation of complete FPGA/MD systems accelerated only the Range Limited (RL) force and used CPUs for the rest of the computation. While performance was sometimes competitive, high cost and lack of availability of FPGA systems meant that they were never in production use. In the last few years, however, it has been shown that FPGA clusters can have performance approaching that of ASIC clusters for the Long Range (LR) force computation (Lawande et al., 2016; Sheng et al., 2017), the part of MD that is most difficult to scale.

BU CAAD lab has been focusing on advancing the state-of-the-art of MD on FPGAs consistently for more than a decade. (Gu et al., 2006a) implemented MD on a Xilinx Virtex-II board, which demonstrated a speedup up to 88. (Gu et al., 2006c; Gu et al., 2006b) integrated FPGA accelerated non-bonded forces calculation into ProtoMol MD code (Matthey, 2004). In (Gu and Herbordt, 2007a), multi-grid computation, which is an essential part of long-range force calculation, was implemented on FPGAs with a speed of 5× to 7×. (Chiu and Herbordt, 2009) proposed new methods of filtering and mapping particles onto different pipelines. (Chiu and Herbordt, 2010a) systematically examined the design space of the short-range force pipelines on FPGAs. (Khan and Herbordt, 2011) presented an event-based decomposition to scale discrete molecular dynamics simulation. (Khan and Herbordt, 2012) studied the communication requirements for MD on FPGA clusters. (Herbordt, 2013) researched the approaches of the architecture/algorithm co-design of MD processors.

Chapter 3

Model Applications on Multi-FPGA System

Featuring scores of high-speed I/Os and millions of logic elements, in theory, FPGA should have both excellent computation and communication capabilities. In order to demonstrate how good FPGA(s) can perform in accelerating real-life applications, we introduce two model applications: Direction Dark Matter (DM) Detection and Molecular Dynamics (MD) Simulation. In the first section, we introduce the design of a data acquisition system for directional dark matter detection, which includes communication between eight Front-End (FE) FPGAs and a single Back-End (BE) FPGA. In the later section, we describe the implementation of the central HPC application, MD simulation, on a single FPGA to showcase FPGA's computational capability. This design serves as the basic unit whereby strong scaling may be achieved with FCCs.

3.1 Directional Dark Matter Detection

Astrophysical observations reveal that dark matter accounts for $\sim 80\%$ of the matter in the universe (Ade et al., 2016). There is a world-wide program underway to detect the evidence of an interaction between a dark matter particle and a target nucleus in the laboratory. For Weakly Interacting Massive Particles (WIMPs – a favored dark matter candidate (Bertone et al., 2005)), the interaction is an elastic collision that creates a recoiling nucleus. Directional dark matter detection seeks to reconstruct the

angular distribution of dark matter particles traveling through the laboratory, which provides a smoking-gun signature of WIMP dark matter (Spergel, 1988). A directional detector with high spatial resolution has the potential to increase the sensitivity per unit volume by over two orders of magnitude, but requires the development of a high-channel-count, high-speed readout system. In this section, we describe a multi-FPGA system to handle a 16 Gbps data stream from $\mathbf{10^3}$ independent detector channels sampled at 1 MHz.

A promising, mature technology for WIMP detection is the low-pressure-gas Time Projection Chamber (TPC) (Marx and Nygren, 1978) shown in Figure 3·1, in which a nuclear recoil generates a track of ionization that is drifted to a readout plane using a uniform electric field. To reconstruct 3D tracks in a TPC, one needs fine spatial granularity ($\sim 200\,\mu$m) over large areas ($1\,\mathrm{m}^2$). A challenge, then, is how to read out the charge signal from such a detector that has $\sim 10^4$ independent channels. At 10^3–10^4 channels, the data processing requirements of MiNI-3D fall in an intermediate regime: too substantial for commercially available equipment, yet not large enough to justify dedicated ASIC development.

In this work, we introduce an ASIC- and FPGA-based charge readout system for a prototype directional dark matter detector. Beyond dark matter detection, this work is of a broad interest in experimental particle physics because it provides high spatial resolution in a large-volume detector.

3.1.1 Hardware Configuration

The detector system consists of three major parts (see Figure 3·2): the TPC, including 1,000 orthogonal sensing strips with 500 strips in the x and 500 strips in the y directions; the FE electronics for signal conditioning and digitization; and a single FPGA-based BE that handles the real-time data stream, applies trigger conditions, and saves interesting events to a computer for off-line analysis.

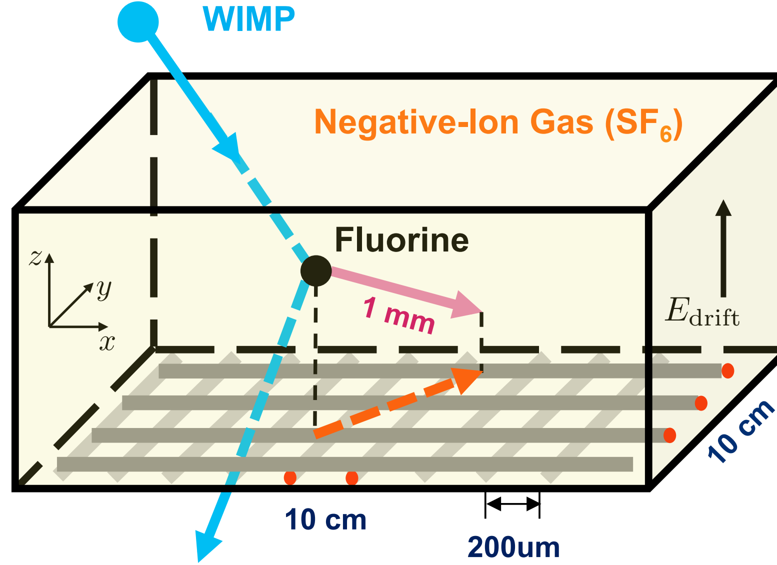

Figure 3·1: Schematic of a WIMP-induced fluorine recoil in a TPC with strip readout. The WIMP (blue line) collides with a fluorine nucleus (black dot) to create a fluorine recoil (pink arrow). The resulting track of charge drifts to the x-y readout plane (orthogonal, electrically isolated strips), forming a 2D projection of the track (orange dashed arrow). The track position and geometry are found by spatial coincidence between the strips. In this example, strips marked with red ovals at their ends will receive a significant charge signal.

FE: signal digitization

There are a total of 8 FE boards measures the charge signal on each of the 10^3 micromegas strips. Each FE board provides analog signal conditioning and digitization and outputs a single serialized stream of the digital data. One FE board contains 8 analog and 8 ADC ASICs and a single Cyclone IV EP4CGX50DF27C7N FPGA that handles the digital stream from the digitizer ASICs. The FPGA serializes the digital data for transmission on a single transceiver line. The result is a digital stream of data from up to 128 detector channels sampled simultaneously at up to 2 MHz. Our detector has 1,000 readout channels, and so we assign a single FE board to 125 adjacent channels (4 FE boards for x and 4 FE boards for y), and select a 1 MHz

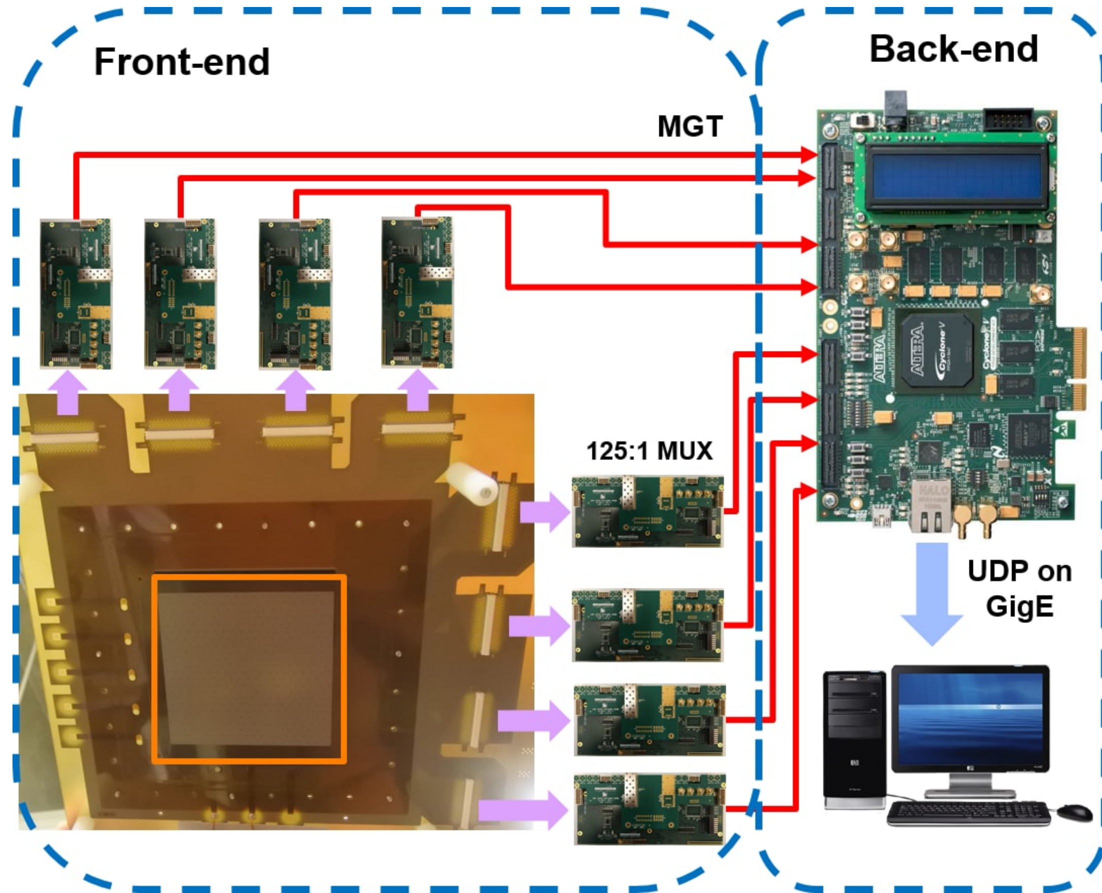

Figure 3·2: Top-level architecture of our design, showing the eight FE boards that each provide 125:1 multiplexing of detector channels onto a single digital MGT link running at 2.5 Gbps. The BE board handles the 16 Gbps data stream and writes triggered events to disk via UDP on gigabit ethernet (GigE).

sampling rate. Each FE board operates independently, sending its data stream to a single back-end at a data generation rate of 2 Gbps.

BE: digital data processing and storage

The BE must receive the digital data stream from eight FE boards (total data generation rate of 16 Gbps), store the data in a circular buffer, apply triggering conditions, and save triggered data to a PC for off-line analysis. We use a COTS Altera Cyclone V GT FPGA development board for the BE. The board features a Cyclone V

5CGTFD9E5F35C7 FPGA, 2 HSMC connectors that expose 8 transceiver ports, and 512 MB of DDR3 SDRAM with a ×64 soft memory controller.

The FE-BE data link is based on MGT. An MGT is a serial link that provides low latency, high bandwidth, and low energy cost for FPGA-to-FPGA connections (Altera, 2014a; Altera, 2015b; Sheng et al., 2015b). Upon arrival at the BE board, data goes through three stages. First, a pattern detector looks for triggering patterns. Next, the serial data is aligned by time and channel number using time-stamp data attached to each data packet by the FE board. Because the block RAM (BRAM) size on the FPGA is not large enough to buffer pre-trigger data from all detector channels, the aligned data is written to off-chip DRAM for temporary storage. Finally, triggered data is transferred from DRAM to a PC via UDP on Gigabit Ethernet for off-line analysis.

3.1.2 Implementation

As shown in Figure 3·3, our design consists of the following parts: front-end data packet generators to organize data on each FE board; send and receive transceiver controllers for FE-BE data exchange; a trigger for event detection; and DRAM control logic for data buffering and replacement.

FE Data Packet Generation

The ADC ASIC has 12-bit precision, while the transceiver encrypts and serializes 16-bit data each time. We use the extra 4 bits for in-packet indexing. In our current configuration, we simply cycle the index for the 125-channel payload. In this way, data loss during transmission can be easily detected. In addition to the payload, we add three extra 16-bit words for alignment across different packets: a starting word, an ending word, and a sampling time-stamp (see Figure 3·4(a)).

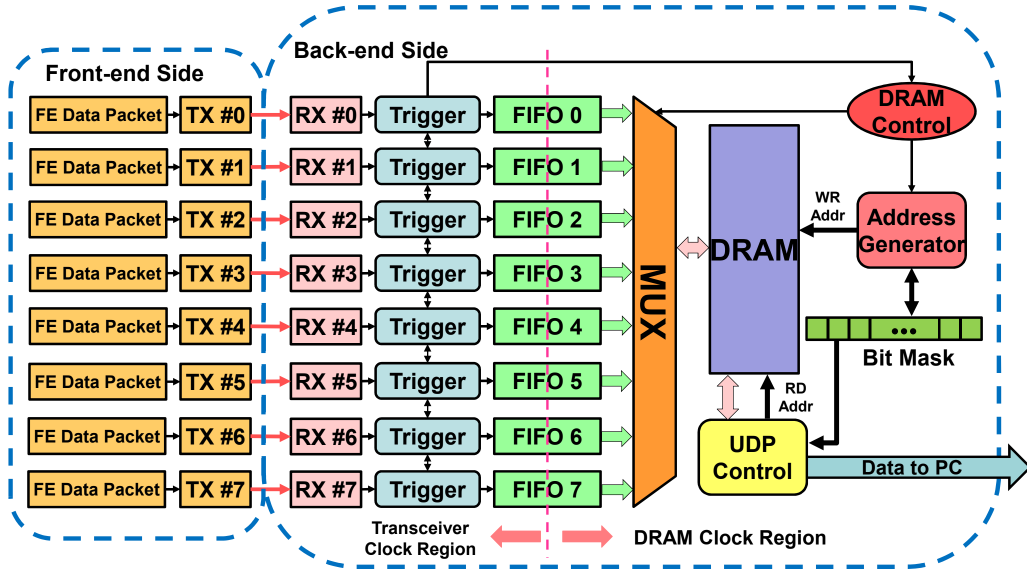

Figure 3·3: Design block diagram.

Transceiver controller

Altera FPGAs support a wide range of protocols and datarate standards (Altera, 2014a; Altera, 2015b), though the supported protocols vary by FPGA model. As the sending and receiving sides in our design use different FPGAs, we use a customized communication protocol for our transceivers.

On the FE, we instantiate an ALTGX IP core with a single TX channel and 8b/10b encoding.A single sample from the FE comprises a 128×16-bit data packet. For 1 MHz sampling, the data generation rate on each FE is 2 Gbps; we, therefore, chose a transceiver rate of 2.5 Gbps.

On the BE, we chose the Native PHY IP, which exposes all low-level MGT control and status signals. An alignment pattern is set in the IP for channel synchronization. We implement eight single-channel Native PHY IP transceivers, one for each FE.

To synchronize the eight FE boards, a system reset signal is generated by a button push on the BE. The signal is propagated to the FE boards via GPIO in a daisy chain. On reset, the FE transceivers prepare the synchronization pattern and the BE

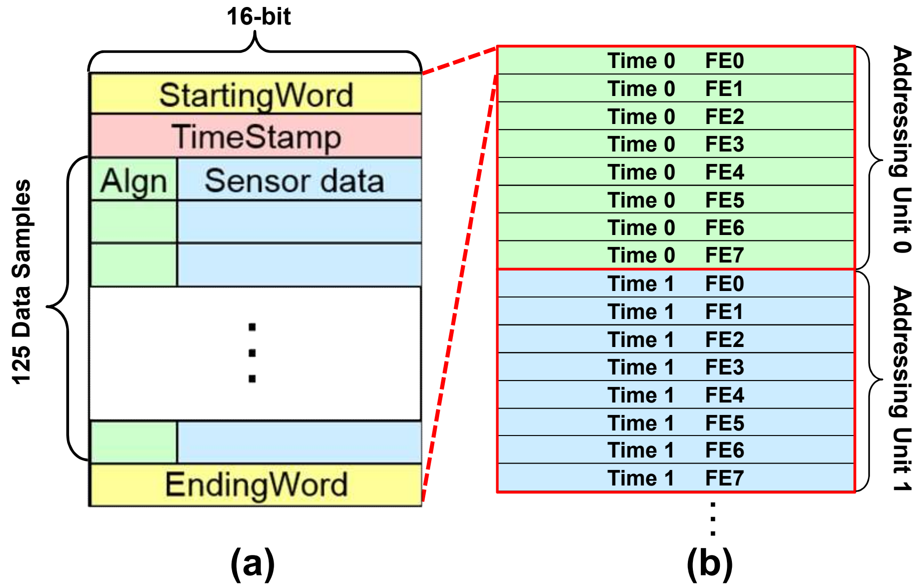

Figure 3·4: Data organization of a FE packet (a) and in DRAM (b).

transceiver wait for the sync pattern to arrive. After the reset clears, all FE boards will repeatedly send out their alignment words for 1 ms. We have confirmed that this scheme synchronizes all transceiver channels.

Event Trigger

A design requirement is to record waveforms to disk from channels that satisfy the trigger, as well as on neighboring channels that may not generate a trigger, but that may contain some low level of signal charge that can be recovered with off-line analysis. A neighboring channel is not necessarily digitized by the same FE board. A BE event trigger is implemented on each of the eight receiving transceiver output ports to handle this.

Figure 3·5 shows the event trigger finite state machine. When in reset mode, the trigger controller resets all control signals and loads the threshold value. The controller stays in *Initial State* during that time. After the reset clears, the event trigger enters the *Non-triggering State* and processes each 16-bit data output from

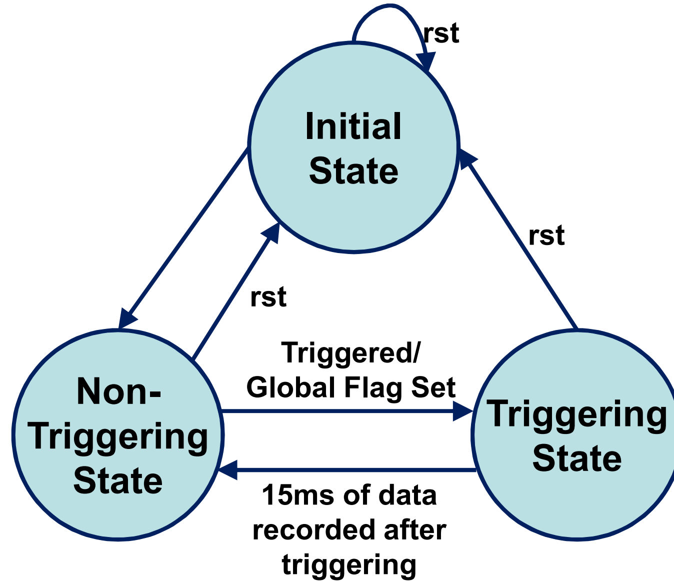

Figure 3·5: Finite State Machine of the event trigger.

the receiving transceiver. The detector first looks for the starting word of a packet and then records the time-stamp from the following word. The 12 LSB of the payload is then sent to the comparator. Upon triggering, the detector sets a global flag along with the trigger time-stamp, which is checked by all channels every cycle. In this way, all 10^3 detector channels enter the triggered state at the same time. In the *Triggering State*, the BE records 15,000 packets (15 ms at 1 MHz) before returning to the non-triggered state. The BE also sends the trigger time-stamp to the DRAM controller to ensure the preservation of 5 ms of pre-trigger data.

We implement two different triggering schemes. Under the first scheme, the detector is triggered when the voltage rises above a fixed threshold on a single channel. Another trigger scheme is to takes into account the integral of the waveform, thus to eliminate the influence of pulse shape due to different recoil track geometry. In our design, we choose an integral window of 5 samples and sum up the sample value in a reduction tree manner.

DRAM Controller

To buffer 5 ms of pre-trigger data for 10^3 channels at 1 MHz sampling takes about 10 MB, which is larger than the on-FPGA BRAM capacity. However, the BE Cyclone V development board has four $\times 16$ SDRAM chips, which provide 512 MB of storage. The total data generation rate from the FE is 16 Gbps. At 300 MHz, the theoretical read and write bandwidth is 38.4 Gbps (Altera, 2014b), fast enough to write all received data directly to DRAM.

Writes to DRAM are organized by sample time and aligned by the FE board number-of-origin (Figure 3·4(b)). We use the FE packet as the DRAM read and write unit. The 512 MB DRAM space is pre-allocated into 2^{21} slots, each slot holds eight 256-bit words to fit a single FE packet. When an entire FE data packet is written into the related FIFO and a ready signal is received, the DRAM controller enables the address generator to send a starting address based on the time-stamp and board number pointing to the pre-allocated space for that packet.

A 1-bit address mask is assigned to each of the above units. If data is determined to be useful by the event trigger, then the related mask bit is set so all the data sampled at the same time will be kept. When the address generator reaches the end of the DRAM address space, the new incoming data set is again assigned to address unit 0. Before that address is assigned, the mask bit is examined. If the bit is set, then the controller looks for the next unset mask bit and uses that related address as the starting point for the following data sets. In this way, we can keep the useful triggering data while fully reusing the non-triggering part.

We record 20 ms of data for each trigger (5 ms pre-trigger and 15 ms post-trigger), or 16 Gbps$\times$20 ms $=$ 0.32 Gbit of data per trigger (assuming all channels are saved). Our DRAM capacity is 4 Gbit, so we can store data from up to 12 triggering events.

Table 3.1: Resource utilization for the FE and BE FPGAs.

FE	ALUTs	Registers	Memory Bits	TX Channel	PLL
Usage	6300	3753	33492	1	3
%	13%	NA	1%	13%	38%

BE	ALUTs	Registers	Memory Bits	RX Channel	PLL
Usage	6936	9334	499232	8	1
%	6%	NA	4%	67%	5%

3.1.3 Evaluation

We evaluate three aspects of our design. First, we present the FPGA resource utilization. Then, we present event trigger evaluation results for several sample waveforms using the two threshold and integral trigger schemes.

Resource Usage

The resource utilization of the FE and BE FPGAs is shown in Table 3.1. The FE utilization is the same for all FE boards as the design is common to all. The BE utilization includes all modules required to interface with 8 FE boards.

Event Trigger Evaluation

We pre-load simulated data onto the two FE board FPGAs. These data represent low-energy and high-energy events. Each set of sample data contains 2000 data points stored in BRAM. We load one set of data onto each of the FE boards and read out one sample every microsecond for transmission to the BE via the MGT. On receipt at the BE, the data passes through the event trigger and is then written to DRAM. Here, we add extra logic to also store the data into 2 special BRAM modules dedicated for this test, one for each FE board, upon the trigger. Using the data sample ID (essentially a counter for incoming samples) that initiates the trigger, we locate the pre- and post-trigger data in BRAM. Once the full 2000-sample waveform from each

FE board has been transmitted and processed, we read out data from BRAM using the *In Memory Content Editor* tool in the Altera Quartus II development suite.

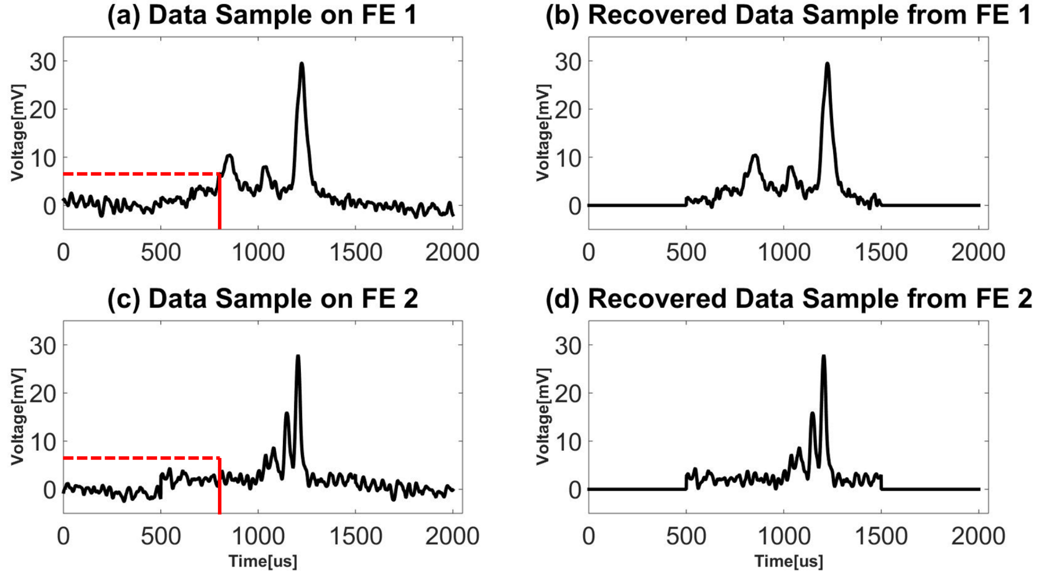

Figure 3·6: Voltage threshold trigger results. (a) & (c) are the pre-loaded sampled data. (b) & (d) are the triggered data as read out from the BE BRAM module implemented for this test.

We first test the voltage threshold trigger with the threshold set to 7 mV. Due to the fact that our sample data has only 2000 data points (2 ms of sampled data), we set the pre- and post-trigger time windows to 0.3 and 0.7 ms, respectively. Samples outside of this window are set to zero. Figure 3·6 presents the results of this test. We see that the event is triggered when data from FE 1 reaches 7 mV at $t = 790\,\mu$s. Both FE 1 and FE 2 are read out over the same period of time, even though data from FE 2 did not initiate the trigger. Thus we confirm that our demonstration system meets the requirements of keeping pre- and post-trigger data from adjacent channels when one channel triggered.

We also evaluate the integral trigger design. We use a low-energy set of waveforms for this test and replace the threshold trigger in the previous test with the integral one. We set the integral threshold value to 10 mV μs. The original and integrated

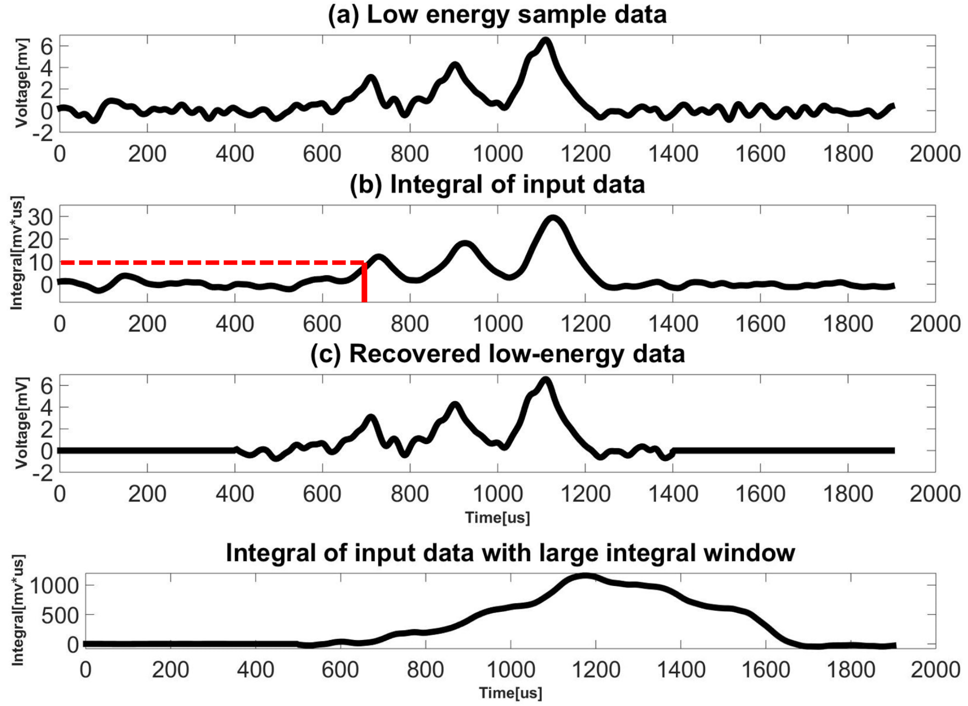

Figure 3·7: Integral trigger results. (a) preloaded low-energy sampled data, (b) integral filtering of the input data, and (c) triggered data. At bottom is the input data filtered with a 512-point integral window.

signals, as well as the triggered data, are shown in Figure 3·7. We see that this narrow integral filter essentially serves as a low-pass filter, and suppresses the high-frequency noise.

Because the integral window size is small ($N_{samples} = 5$) compared to the widths of the peaks (about 512 samples total), the peaks are not integrated. By expanding the integral window size, it would be possible to integrate the signal under all three peaks. The bottom plot in Figure 3·7 shows a simulated integral waveform with $N_{samples} = 512$.

3.1.4 Summary on Directional Dark Matter Data Acquisition System

In this section, we describe an FPGA-based data acquisition system for directional dark matter detection. The design features eight FE ASIC + FPGA boards for data

collection, and a single digital BE FPGA board for data pre-processing and temporary storage. Using the budget friend low-end FPGAs, our final system has the capability of processing 10^3 detector channels, each sampled at 1 MHz.

This project sufficiently demonstrates the communication capability of FPGAs. Even with low-end FPGAs with a limited number of resources, our BE FPGA successfully collect data from 8 FE FPGAs in parallel and achieves in-line processing and data storage at an aggregate data rate of 18 Gbps.

Equipped with hundreds of MGTs and offers higher data rate ($\sim$20 Gbps per channel) (Intel, 2018a), high-end FPGAs nowadays will further unlock the communication capability of FPGAs. This provides a good opportunity in achieving strong scaling for HPC applications accelerated by FPGAs.

3.2 Molecular Dynamics Simulation

Molecular Dynamics (MD) acceleration on FPGA(s) was much studied from 2004-2010. Due to limited chip resources of that era, and the inherent variety and complexity of tasks comprising Molecular Dynamics simulations, those FPGA accelerators relied on the host or embedded processors to organize and pre-process input and output data. This introduced long latency for data movement between simulation iterations and, as technology advanced, drastically limited performance. Current generation FPGAs are equipped not only with abundant on-chip resources, but also have hardware support for floating-point operations; these advances provide an opportunity for creating self-contained MD simulation systems on a single device.

Previous work, however, has consisted of either proof-of-concept implementations of components, usually the range-limited force; full systems, but with much of the work shared by the host CPU; or prototype demonstrations, e.g., using OpenCL, that neither implement a whole system nor have competitive performance. In this section,

we present what we believe to be the first full-scale FPGA-based simulation engine, and show that its performance is competitive with a GPU (running Amber in an industrial production environment).

3.2.1 MD Background

MD Basics

MD alternates between force calculation and motion update. The forces computed depend on the system being simulated and may include bonded terms, pairwise bond, angle, and dihedral; and non-bonded terms, van der Waals and Coulomb (Haile, 1997):

$$\mathbf{F^{total}} = F^{bond} + F^{angle} + F^{dihedral} + F^{non-bonded} \tag{3.1}$$

The *Bonded Force* terms involve small numbers of particles per bond, but the computations themselves can be complex; interactions can be expressed as follows: bond (Equation (3.2)), angle (Equations (3.3) and (3.4)), and dihedral(Equations (3.5) and (3.6)), respectively (from Equation (3.1) (NAMD, 2017)).

$$\mathbf{F_i^{bond}} = -2k(r_{ij} - r_0)\vec{e_{ij}} \tag{3.2}$$

$\vec{e_{ij}}$ is the unit vector from one item to another, r_{ij} the distance between the two particles, k the spring constant, and r_0 the equilibrium distance;

$$\mathbf{F_i^{angle}} = -\frac{2k_\theta(\theta - \theta_0)}{r_{ij}} \cdot \frac{\vec{e_{ij}}cos(\theta) - \vec{e_{kj}}}{sin(\theta)} + f_{ub} \tag{3.3}$$

$$\mathbf{f_{ub}} = -2k_{ub}(r_{ik} - r_{ub})\vec{e_{ik}} \tag{3.4}$$

$\vec{e_{ij}}, \vec{e_{kj}}, \vec{e_{ik}}$ are the unit vectors from one item to another, θ the angle between vectors $\vec{e_{ij}}$ and $\vec{e_{kj}}$, θ_0 the equilibrium angle, k_θ the angle constant, k_{ub} the UreyBradley constant, and r_{ub} the equilibrium distance;

$$\mathbf{F_i^{dihedral}} = -\nabla\frac{U_d}{\vec{r}} \tag{3.5}$$

$$\mathbf{U_d} = \begin{cases} k(1 + cos(n\psi + \phi)) & n > 0, \\ k(\psi - \phi)^2 & n = 0. \end{cases} \tag{3.6}$$

n is the periodicity, ψ the angle between the (i, j, k)-plane and the (j, k, l)-plane, ϕ the phase shift angle, and k the force constant.

The *Non-bonded Force* uses 98% of FLOPS and includes Lennard-Jones (LJ) and Coulombic terms. For particle i, these can be:

$$\mathbf{F_i^{LJ}} = \sum_{j \neq i} \frac{\epsilon_{ab}}{\sigma_{ab}^2} \left\{ 48 \left(\frac{\sigma_{ab}}{|r_{ji}|} \right)^{14} - 24 \left(\frac{\sigma_{ab}}{|r_{ji}|} \right)^{8} \right\} \vec{\mathbf{r_{ji}}} \tag{3.7}$$

$$\mathbf{F_i^{C}} = \frac{q_i}{4\pi} \sum_{j \neq i} \frac{1}{\epsilon_{ab}} \left\{ \frac{1}{|r_{ji}|} \right\}^3 \vec{\mathbf{r_{ji}}} \tag{3.8}$$

where the ϵ_{ab} (unit: kJ or $kcal$) and σ_{ab} (unit: meter) are parameters related to the types of particles.

The LJ term decays quickly with distance, thus a *cutoff radius,* r_c*,* is applied: the LJ force is zero beyond it. The Coulombic term does not decaying as fast; but this term can be divided into two parts, fast decaying within r_c and slowly developing beyond it. Consequently, we approximate the LJ force and the fast decaying part of the Coulombic force as the *Range-Limited (RL) force*, and the other part of the Coulombic force force as the *Long-Range (LR) force.* RL is the more computationally intensive (90% of flops) and is calculated as:

$$\frac{\mathbf{F}_{ji}^{RL}}{\mathbf{r_{ji}}} = A_{ab} r_{ji}^{-14} + B_{ab} r_{ji}^{-8} + QQ_{ab} r_{ji}^{-3} \tag{3.9}$$

where $A_{ab} = 48\epsilon_{ab}\sigma_{ab}^{12}$, $B_{ab} =$ -$24\epsilon_{ab}\sigma_{ab}^{6}$, $QQ_{ab} = \frac{q_a q_b}{4\pi\epsilon_{ab}}$.

The LR force is calculated by solving the Poisson Equation for the given charge distribution.

$$\mathbf{F_i^{LR}} = \sum_{j \neq i} \frac{q_j}{|r_{ji}|} \vec{\mathbf{r_{ji}}} \tag{3.10}$$

$$\rho_g = \sum_p Q_p \phi(|x_g - x_p|) \phi(|y_g - y_p|) \phi(|z_g - z_p|) \tag{3.11}$$

LR is often calculated with a grid-based map of the smoothing function converted from continuous space to a discrete grid coordinate system (Young et al., 2009). Each particle is interpolated to grid points by applying a third-order basis function for charge density calculation. Grid points obtain their charge densities from neighboring particles within a range of two grid points in each direction. There, grid electrostatics are converted into the Fourier domain, evaluated using the Green's function, then converting back through an inverse FFT.

Force Evaluation Optimizations

RL uses the cutoff to reduce the $\mathcal{O}(N^2)$ complexity: forces on each *reference particle* are computed only for *neighbor particles* within r_c. The first approximation is the widely used partitioning of the simulation space into equal-sized cells with a size related to r_c. The particles can be indexed using *cell-lists* (Brown et al., 2011): for any reference particle and a cell length of r_c, only neighbor particles in the 26 *neighboring cells* need to be evaluated. Another optimization is Newton's 3rd Law (N3L): since the force only needs to be computed once per pair, only a fraction of the neighboring cells need to be referenced. Most of the particles, however, are still outside the cutoff radius. In CPU implementations this can be handled by periodically creating neighbor lists. In FPGAs, the preferred method is to do this on-the-fly (Chiu and Herbordt, 2010a) through *filtering*.

Boundary Conditions

To constrain particles movement inside a fixed size bounding box, we apply *Periodic Boundary Conditions (PBC)*. When evaluating particles reside in boundary cells, we imagine a fictional space next to the boundary cell that is an exact copy of our simulated space.

Motion Integration

After the forces have been computed and aggregated, the change of position and velocity of each particle is computed. One popular motion integrator is the Verlet algorithm (Grubmüller et al., 1991). Since we are using short simulation timestep (2 femtoseconds), we can use simple integration equations such as symplectic Euler:

$$\vec{a}(t) = \frac{\vec{F}(t)}{m} \tag{3.12}$$

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \vec{a}(t) \times \Delta t \tag{3.13}$$

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \vec{v}(t + \Delta t) \times \Delta t \tag{3.14}$$

where m is mass, $\vec{a}$ is acceleration, $\vec{v}$ is velocity, $\vec{r}$ is position.

3.2.2 MD System Architecture

Here, we cover the four major components inside an MD simulation system, along with some high-level design decisions. We begin with a classic FPGA-based MD force evaluation pipeline and then add several function units that, in previous implementations, were executed on the host processor or embedded cores.

Overall Architecture

Since configuration time is long comparing with iteration time, the design is fixed within a single simulation. A design goal is to give the force computations resources such that their compute times are equalized; resource allocation to summation and

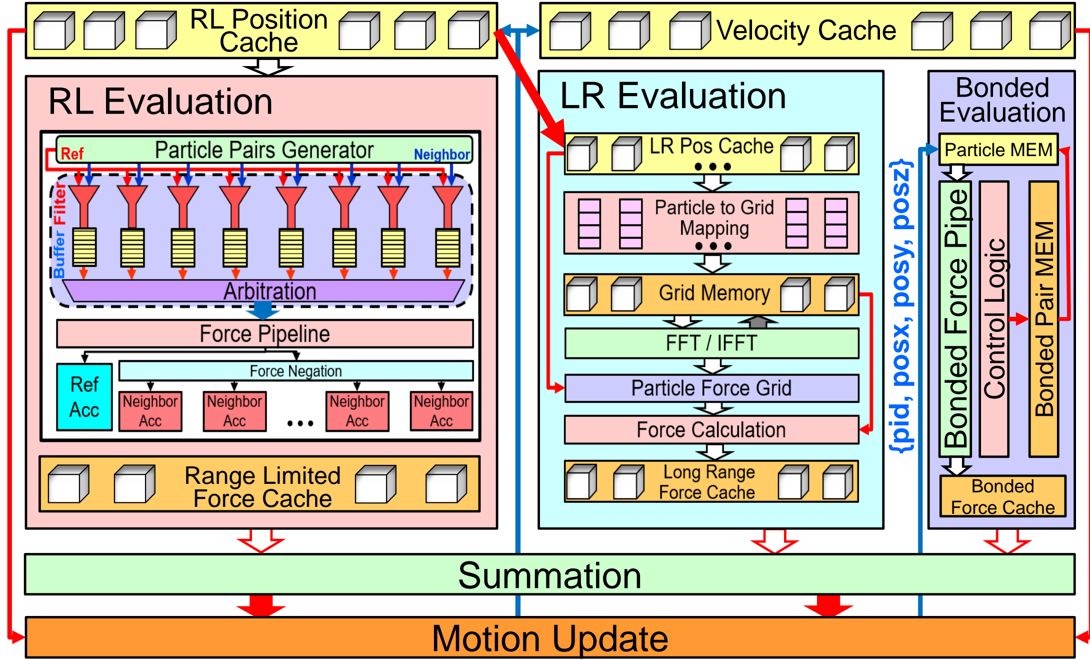

Figure 3·8: MD End-to-End System Overview. Details of each section is covered in following figures.

motion update is analogous. All components (LR, RL, etc.) have parameterized designs with performance proportional to the parallelism applied (and thus chip resources used). This applies also to fractional parallelism: some components can be *folded*, e.g., to obtain half performance with half resources.

Figure 3·8 depicts the proposed FPGA-MD system. The **RL** units evaluate the pair-wise interactions. Since this is the most computationally intensive part, the base module is replicated multiple times. The **LR** unit includes: (i) mapping particles to a charge grid, (ii) conversion of charge grid to potential grid via 3D FFT (and inverse-FFT), and (iii) evaluating forces on individual particles based on the potential grid. Since our timestep is small ($2fs$), LR is only updated every few iterations. The **Bonded Evaluation** unit has pipelines for the three parts (see Equation 3.1). At the end of each timestep, the **Summation** unit sums the three partial forces and sends the result to the **Motion Update** unit to update position and handle particle

migration among adjacent cells.

Range-Limited Force Evaluation

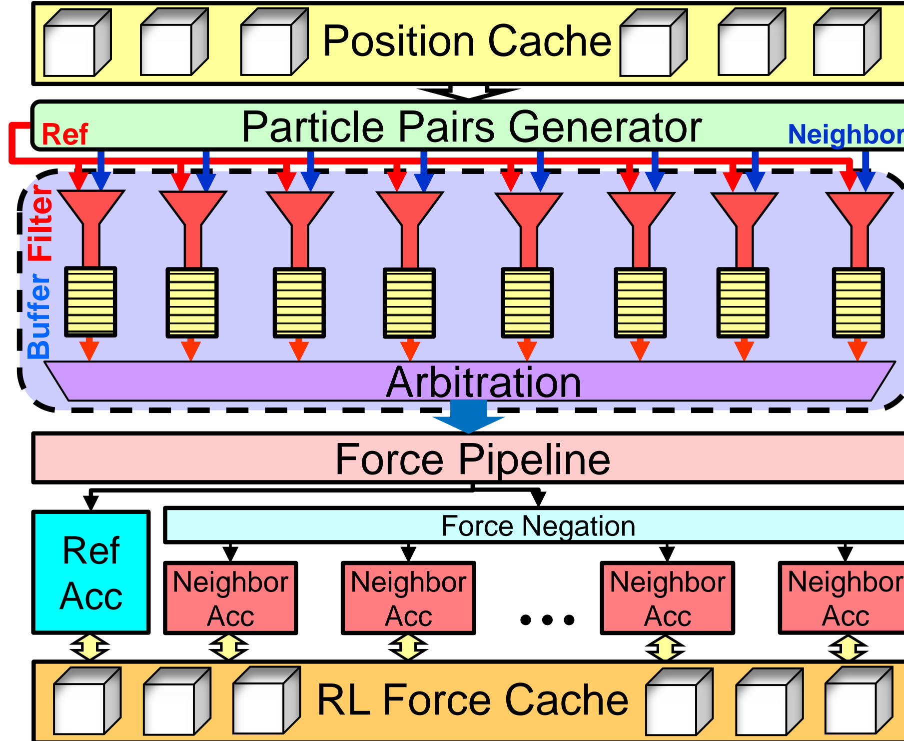

Figure 3·9: RL Evaluation Architecture Overview

The RL force evaluation pipeline (Figure 3·9) is based on a design first proposed in (Chiu and Herbordt, 2009), although nearly all parts have been redesigned from scratch. The particle position cache holds the initial position of each particle. Modern high-end FPGAs like Intel Stratix 10 (Intel, 2018a) provide enough on-chip storage to hold particle data for our range of simulations. Next is a set of filters that performs a distance evaluation of possible particle pairs (in certain neighboring cells) and only pass the pairs within the cutoff radius. Remaining data then enter the most computationally intensive part in the process: force evaluation. Since each particle is contributing to multiple pair-wise interactions, we design an efficient accumulation mechanism on the output of the force evaluation pipeline, to sum up, the partial

forces on each particle.

Particle-Pair Filtering. Mapping among cells, BRAMs, and filters is complex and is described below. Once a particle pair is generated and sent to a filter, its distance is compared with r_c (actually r^2 with r_c^2 to avoid the square root). Possible neighbor particles can reside in 27 cells in 3-dimensions (13+1 if considering N3L, as shown in Figure 3·10). Since the average pass rate is only 15.5%, providing a force pipeline with at least one valid output per cycle requires a bank of at least seven filters plus load balancing (we use eight). If there are multiple valid outputs, round-robin arbitration is used. The not-selected valid outputs are stored in the filter buffer (on the output side of each filter) as shown in Figure 3·9.

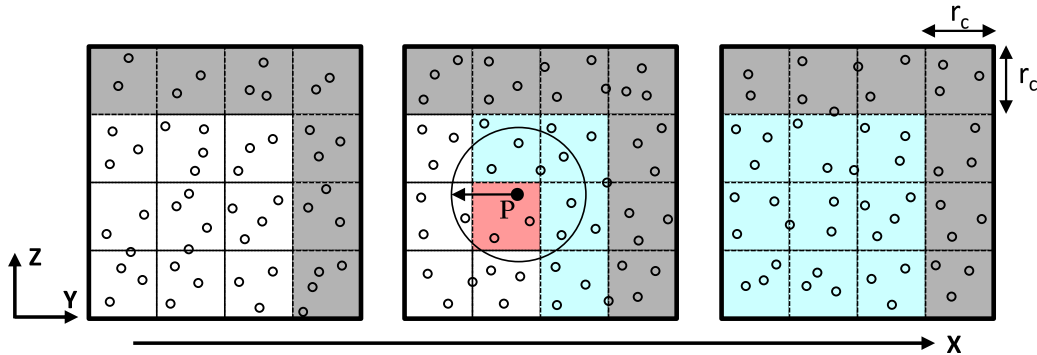

Figure 3·10: Simulation space about particle P. Its *cell neighborhood* is shown in non-gray; cell edge size is the cutoff radius (circle). After application of N3L, we only need to consider half of the neighborcells (blue) plus the homecell (red).

Force Evaluation. Various trade-offs have been explored in other FPGA/MD work (Gu et al., 2006a; Gu et al., 2008). These are two of the most important.

1. **Precision and Datatype:** CPU and GPU systems often use a combination of single-precision, double-precision, integer, fixed-point, and floating-point. ASIC-based systems have complete flexibility and use non-standard types and precisions. FPGAs have multiple implementation possibilities. If logic cells

alone are used, then ASIC-type designs would be preferred for fixed (Fukushige et al., 1996; Komeiji et al., 1997; Chiu and Herbordt, 2010a) or floating-point (Scrofano et al., 2006; Chiu et al., 2011). Modern FPGAs, however, also have many thousands of embedded ASIC blocks, viz. DSP and/or floating-point units. So while the arithmetic design space is still substantial, preferred designs are likely to be quantized by these fixed-sized *hard* blocks. We find that, in contrast with earlier FPGA-MD studies, there is less advantage to use integer and fixed-point; rather we primarily use the *Native Floating-Point IP core.* For certain computations where accuracy is critical, we also employ fixed-point arithmetic; this is at the cost of high resource overhead.

2. **Direct Computation vs. Interpolation with Table-lookup:** The RL force calculation requires computing r^{-3}, r^{-8} and r^{-14} terms. Since r^2 is already provided from the filter unit, a total of 8 DSP units (pipelined) are needed to get these 3 values (based on the force pipeline proposed in (Chiu and Herbordt, 2010a)). Plus, we need 3 extra DSP units to multiply the 3 indexes, QQ_{ab}, A_{ab} and B_{ab}, with r^{-3}, r^{-14} and r^{-8} respectively. To reduce DSP usage, we use interpolation with table-lookup. As is common with this method, we divide the curve into several sections along the X-axis, such that the length of each section is twice that of the previous. Each section has the same number of intervals with equal size. We implement 3 sets of tables for r^{-3}, r^{-8} and r^{-14} curve. We use r^2, instead of r, as the index to further reduce resource consumption that would be needed when evaluating square root and division.

RL Workload Distribution. FPGAs provide abundant design flexibility that enables various workload to bare metal mapping schemes. In this subsection, we introduce two levels of mapping: particles onto BRAMs, and workload onto pipelines.

Figure 3·11 lists two of many possible mapping schemes, which we refer to as **Mem**

1 and **Mem 2**.

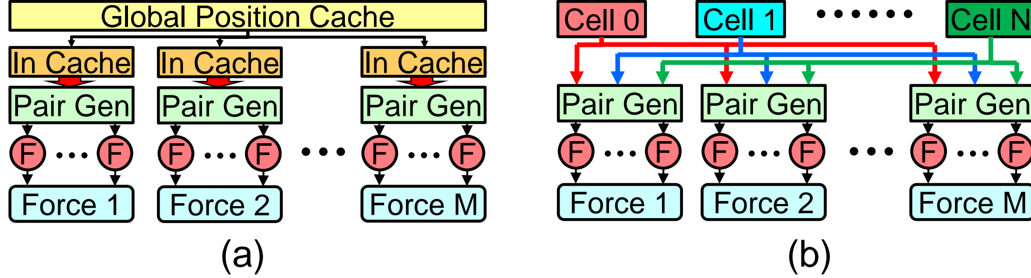

Figure 3·11: Cell to RAM Mapping Schemes: (a) Mem 1: all cells mapped onto a single memory module; (b) Mem 2: each cell occupies an individual memory module.

1. ***Mem 1:*** A single global memory module holds position data for all particles (Figure 3·11(a)). This design simplifies the wiring between position memory and the hundreds of pipelines. To overcome the bandwidth bottleneck, we insert an input cache at the start of each pipeline to hold the pre-fetched position data.

2. ***Mem 2:*** The bandwidth problem can also be overcome by having each cell map onto an individual memory unit (Figure 3·11(b)). But when there are hundreds of pipelines and cells, the all-to-all connect incurs large resource consumption and timing challenges.

The simulation space is partitioned into cells. We successively treat each particle in the homecell as a *reference* particle and evaluate the distance with *neighbor* particles from its homecell and 13 neighborcells (N3L). The system then moves to the next cell and so on until the simulation space has been traversed. There are a vast number of potential mapping schemes; due to limited space, we present just three of the most promising.

1. ***Distribution 1:*** **All pipelines work on the same reference particle (Figure 3·12(a)).** A global controller fetches a particle from the current homecell

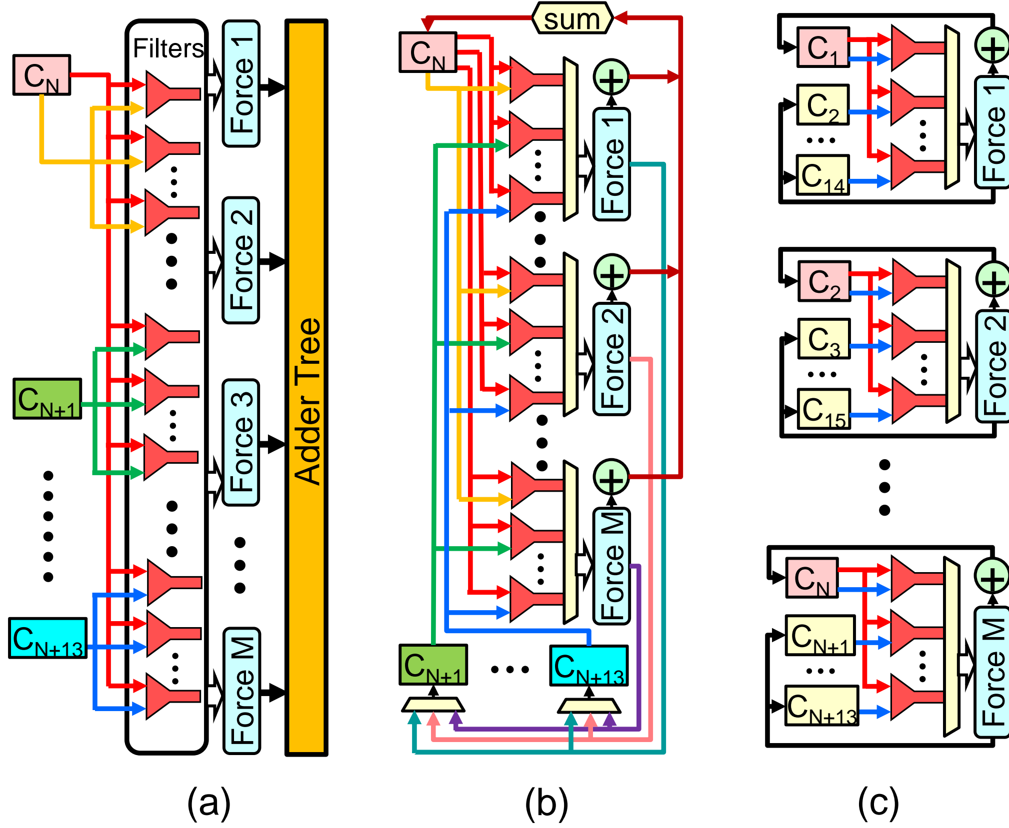

Figure 3·12: Workload mapping onto force pipelines: (a) all pipelines work on the same reference particle; (b) all pipelines work on the same homecell, but with different reference particles; (c) each pipeline works on a different homecell.

and broadcasts it to all the filters in the system, around 1000. Potential neighbor particles from home and neighbor cells are evenly distributed among all the filters. The evaluated partial force output from each pipeline is collected by an adder tree for summation and written back. At the same time, the partial forces are also sent back to the neighborcells and accumulated inside each cell. This implementation achieves the workload balance on the particle-pair level. However, it requires extremely high read bandwidth from the position cache to satisfy the need for input data for each filter, and requires high write bandwidth when accumulating partial forces to neighbor particles, since the Read & Write

only targets 14 cells at a time.

2. ***Distribution 2:*** **All pipelines work on the same homecell, but on different reference particles (Figure 3·12(b)).** To start, the particle pair generator reads out a reference particle from the homecell for filters belonging to each force pipeline. During the evaluation, the same neighbor particles are broadcast to all filters (belonging to different force pipelines) at the same time, since the neighbor particle set for every reference particle is the same as long as they belong to the same homecell. Compared with the first implementation, this one alleviates the pressure on the read port of the position cache. The tradeoff is that partial forces targeting the same neighbor particle may arrive at the neighborcell at the same time; thus a special unit is needed to handle the read-after-write data dependency. Since each force pipeline is working on different reference particles, an accumulator is needed for each force pipeline.

3. ***Distribution 3:*** **Each pipeline works on its own homecell (Figure 3·12(c)).** Under this mapping scheme, each filter only needs to interact with a subset of spatially adjacent homecells, along with a set of neighborcells. Compared with the previous two schemes, there is only interaction among a small set of cells. This method not only fully utilizes the parallelism in force evaluation, but also reduces the number of wires between particle caches and force evaluation units. The downside, however, is load balancing. Suppose we have 100 pipelines, but 150 cells. After each pipeline evaluates a cell, half of the pipelines will remain idle while the others evaluate a second homecell. To avoid this waste of resources, an application-aware mapping scheme is required.

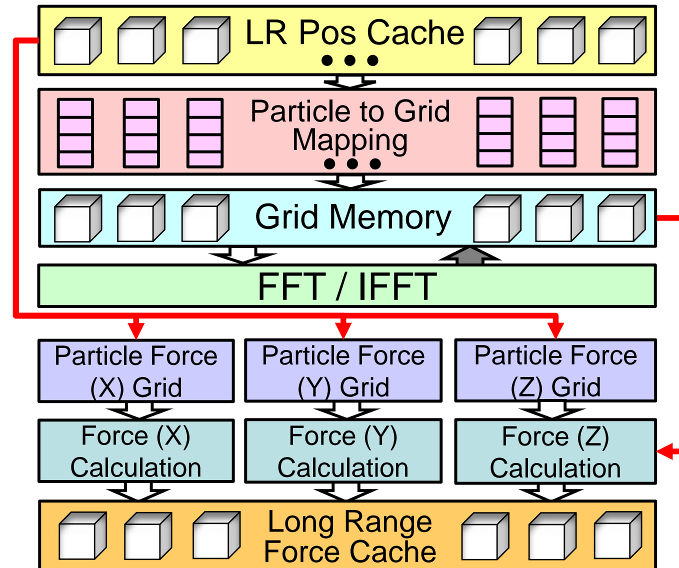

Figure 3·13: LR Evaluation Architecture Overview

Long-Range Force Evaluation

Parts of the LR computation have been explored previously (see (Sanaullah et al., 2016a) on mapping and (D'Alberto et al., 2007; Dick, 1998; Humphries et al., 2014) on the 3D FFT), but this is the first time they have been integrated. LR (Figure 3·13) begins with a cache of position data, which maintains particle information when mapping to the particle grid and the force calculation. The position cache is necessary since positions may change during LR. Particle charges are evaluated and assigned to 64 neighboring cell locations using a third-order basis function, with results stored in *Grid Memory*. After all particle data are consumed, the FFT runs on the resulting grid (through each axis X, Y, and Z). The resulting data, after multiplying with the Green's function, is replaced in the memory grid only a few cycles after evaluation. This is possible because of the pipeline implementation of the FFT. The inverse FFT is then performed on each dimension. Finally, forces are calculated for each particle using the final FFT grid results and the starting particle position information saved previously in the position cache. These are then saved into a force cache which is used

during the motion update phase to apply long-range forces to the particle positions.

Particle-to-Grid Mapping. The third order basis functions Equation (3.15) are used to spread particle charges to the four closest grid points, based on particle position data, and can be independently evaluated for each dimension. After a particle is evaluated in each dimension, values are assigned to 64 neighboring cells and each result is accumulated into grid memory locations. Figure 3·14 shows the process of a single particle's influence on 64 neighborcells and their mapping to the grid memory structure. Parallel particle-to-grid mapping occurs with the use of accumulators before entering grid memory due to restrictions in using BRAMs.

$$\begin{cases} \phi_0(oi) = -1/2oi^3 + oi^2 - 1/2oi \\ \phi_1(oi) = 3/2oi^3 + 5/2oi^2 + 1 \\ \phi_2(oi) = -3/2oi^3 + 2oi^2 + 1/2oi \\ \phi_3(oi) = 1/2oi^3 - oi^2. \end{cases} \tag{3.15}$$

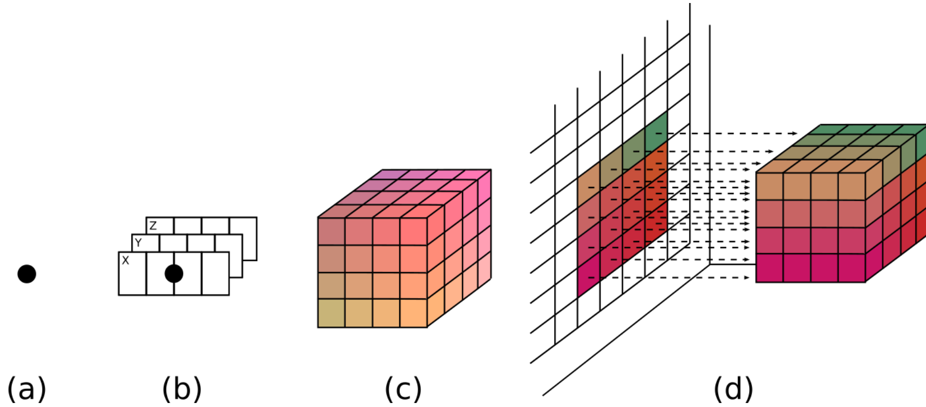

Figure 3·14: Particle to grid flow: (a) Initial particle position data; (b) Particle to 1D interpolation for each dimension using basis functions; (c) Mapping 1D interpolation results to a 4x4x4 3D grid; (d) Final 64 grid points to 16 independent memory banks

FFT. The FFT subsystem performs calculations in parallel using vendor-supplied FFT core configured by Intel Quartus Prime Design Suite FFT IP controller. It has

the capability of dictating the number of streaming values, by which we can change the core to suit the size of our design space (16, 32, etc.) (Intel, 2017). To ensure high throughput memory access, we assign the FFT units to specific banks of the grid memory. As a result, grid data can be continuously streamed through all FFT cores in parallel. While output is being generated for a given vector, a new input is sent for the next set of calculations. Each dimension is performed sequentially until all three dimensions are completed on the memory grid. Once all three dimensions are evaluated and converted into the Fourier-domain, the grid is multiplied with Green's function, before proceeding to the inverse FFT stage going through each dimension again and converting back. Final values at each grid point are used to compute the LR force for each particle based on its position.

Bonded Force Evaluation

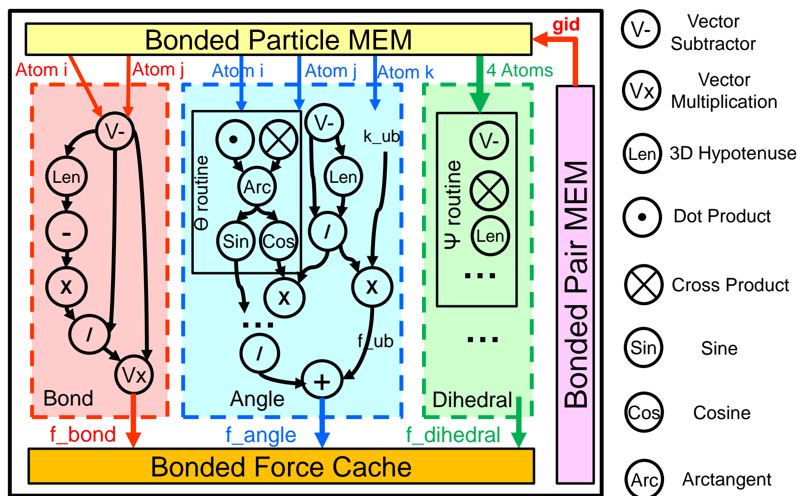

Figure 3·15: Bonded Force Evaluation Architecture

While the bonded force has been explored previously (Xiong and Herbordt, 2017), this design and implementation are entirely new.

As shown in Figure 3·15, we evaluate three types of bonded interactions: bond, angle, and dihedral, which have, respectively contributions from 2, 3, and 4 atoms.

For a given dataset, the covalent bonds remain fixed as long as no chemical reaction is involved. In general, the bonded computation requires only a few percents of the FLOPs, so attenuation rather than parallelism is advantageous: we, therefore, process bonds sequentially.

For LR and RL we organize the particle data based on cells; this proves costly for bonded force evaluation. Rather than particles interacting with others based on their spatial locality, bonded interactions have a fixed set of contributing particles. As the simulation progresses, particles can move across different cells and require extra logic to keep track of their latest memory address. Given the fact that we process bonded sequentially, and this requires little memory bandwidth, we propose a different memory architecture: a single global memory module (*Bonded Particle MEM* in Figure 3·15) that maintains information on each particle position based on a fixed particle global id (gid). The gid is assigned prior to the simulation and remains fixed.

A read-only memory, *Bonded Pair MEM*, holds pairs of gids that form chemical bonds in the dataset. During force evaluation, the controller first fetches a pair of gids along with other parameters from Pair MEM, then proceeds to fetch the related particle position from Particle MEM and sends this for force evaluation. The evaluated bonded force is accumulated in the *Bonded Force Cache* addressed by *gid*. During the motion update, the accumulated bonded force summed with partial results from RL and LR. Finally, Particle MEM receives the updated position, along with the particle gid, to maintain an up-to-date value.

Force Summation and Motion Integration

The three force components must be combined before the motion update. Even with load balancing, RL always finishes last; this is guaranteed, in part, by the small variance in the other computations. Therefore we can assume that the LR

and bonded force caches always have data ready. Thus, as soon as RL of a certain particle is ready, we can perform the summation and motion update. As described in Section 3.2.2, for any given particle, it needs to be evaluated with respect to each neighbor particle from 27 cells. Since we make use of N3L to avoid revisiting particle pairs more than once, we need to keep track of how many times each cell has been visited (as homecell and neighborcells). To handle this we propose a *Score Boarding* mechanism. Once computations on all particles in a cell have finished, the Score Board module will access LR, RL and Bounded forces from the corresponding caches for force summation. By doing so, the positions of particles from the same cell can be updated immediately when a cell is fully evaluated; the motion update is executed in parallel with force evaluation with limited resource overhead; a large fraction of motion update latency can, therefore, be hidden.

After summation for a particle is finished, the aggregated force is sent to the motion update unit, along with particle's position and velocity. Since we organize particle data based on the cells they belong to (except for the bonded unit), particles can move from one cell to another. This creates a challenge on particle memory management: we need to maintain a record of which memory is ready for receiving new particles (due to particles left in the current cell, or the pre-allocated vacant memory space in each cell). It may take multiple cycles to find an available memory slot when the cell memory is almost full, or to find a valid particle in the cell when the cell memory is almost empty. Our solution is to double buffer the particle position and velocity caches; details are given in the next section.

3.2.3 MD System Implementation

In this section, we highlight a selection of implementation details.

Datatype and Particle Cache

The system maintains three sets of information for each particle: position, velocity, and force. The first two need to be maintained throughout the entire simulation, while the force data is flushed after motion update.

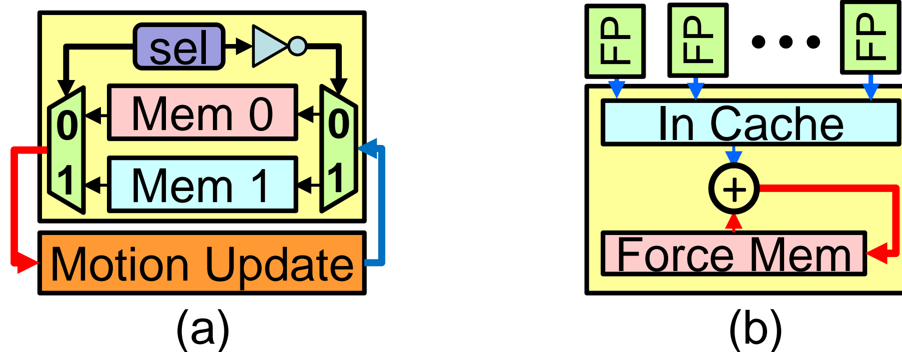

Figure 3·16: (a) Double buffer mechanism inside position and velocity cache; (b) Force cache with accumulator.

RL Position Cache organizes data into cells. Double buffering is implemented (Figure 3·16(a)) with the particle *gid*s being kept along with position data, which is used during summation and motion update process.

RL Force Cache is read and written during force evaluation. Since the system has hundreds of pipelines, that many partial forces must be accumulated each cycle. To manage the potential data hazards, we implement an accumulator inside each force cache module (see Figure 3·16(b)). After the aggregated forces are read out during motion update, they are cleared for the next iteration.

LR Particle Cache. The LR force evaluation is generally performed every two to four iterations while motion update happens every iteration. Since LR evaluation needs the positions to remain fixed, we allocate a separate LR particle cache (Figure 3·8&3·13). Every time LR starts a new evaluation (every second iteration in our experiments), it first performs a memory copy from RL Position Cache. To shorten the memory copy latency, we implement LR cache using *Mem 2*, which provides high

write bandwidth.

RL Evaluation Pipeline

We use the Native Floating-Point IP Core controller inside Intel Quartus Prime Pro 18.1 Design Suite to configure the DSP units on FPGA to realize the IEEE floating-point operations in our design.

As introduced before, we have 8 filters per RL pipeline to statically guarantee a valid output can be provided from the filters each cycle. Round-robin is used to select among filters with a valid output. To reduce the latency in the filter bank, while saving buffer space in the meantime, we have developed an arbitration algorithm (Algorithm 1) that delivers one result per cycle.

Algorithm 1 Filter Arbitration Algorithm

Step 1: Shift the current arbitration result left 1 bit, then subtract 1
Step 2: Perform NOT on Step 1
Step 3: Get valid mask based on data availability in each filter buffer
Step 4: Perform AND on Step 2 and Step 3
Step 5: Perform 2s compliment on Step 4
Step 6: Perform AND on Step 4 and 5, this is **new arbitration result**
Step 7: If current valid mask only has MSB as 1, then omit Step 1
Step 8: If current arbitration result is 0, skip Steps 1-4

The pipeline evaluates forces via interpolation with table-lookup (shown in Figure 3·17). Assuming the interpolation is second order, it has the format:

$$r^k = (C_2(x - a) + C_1)(x - a) + C_0 \tag{3.16}$$

where $x = r^2$, a is the x value at the beginning of the target interval, and $x - a$ is the offset into the interval. Based on different datasets, the interpolation coefficients are pre-calculated, packed into the mif file, and loaded onto the FPGA along with position and velocity data. After the coefficients are read from memory, the pipeline

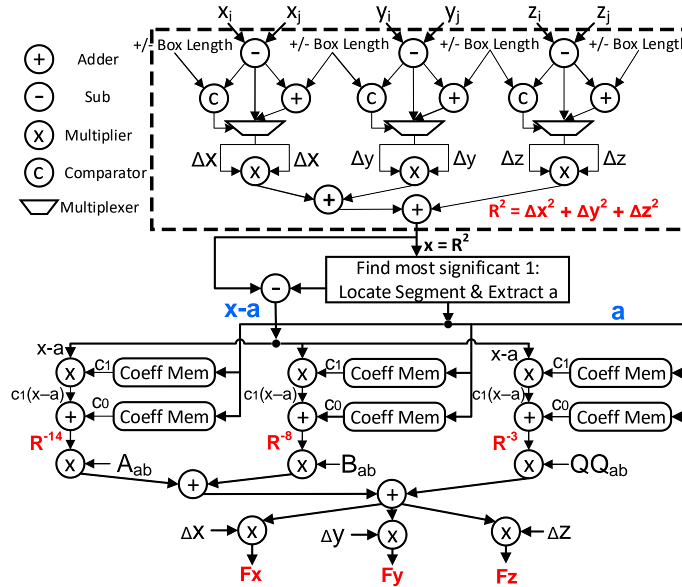

Figure 3·17: Force evaluation with first order interpolation.

performs the evaluation following Equation (3.16). Figure 3·17 shows this for the first order; the actual system supports up to the third order.

LR Evaluation

Due to the large number of particles, the particle to grid mapping must be optimized to avoid adding additional stall cycles when each particle enters the system. This means replication is a must to avoid long delays. The first step is to evaluate each individual basis function per dimension to obtain a single particle contribution to an individual cell. As Figure 3·18 shows, one function takes 5 steps to evaluate a single equation. This unit can be replicated to evaluate all 4 functions simultaneously and each dimension is done in parallel requiring a total of 12 replications of each unit. After all functions are evaluated, values are combined to form 64 unique values representing the $4 \times 4 \times 4$ neighbor grid of cells around the particle. These 64 cells are then accumulated with the previous information, found in their respective cells.

Using the interleaved structure of the grid memory, the FFT implementation

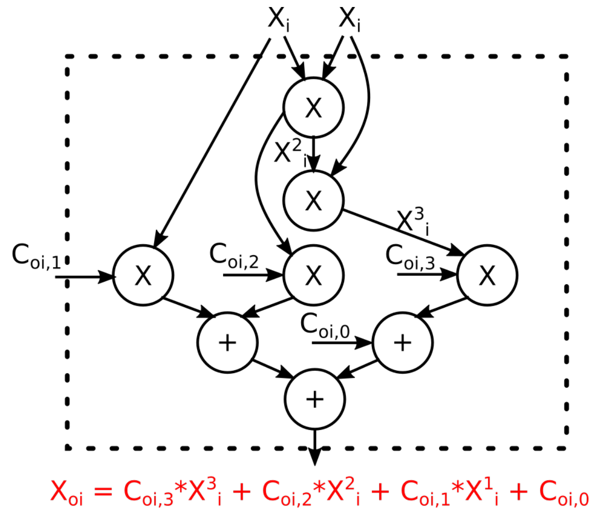

Figure 3·18: One instance of the particle to grid conversion equation: The unit is replicated 12 times. Four instances represent the four basis equations for each dimension X, Y, and Z.

allows for the use of multiple FFT units to evaluate each dimension in parallel. Since this part of LR is not the bottleneck, a modest number of FFT blocks (16) is currently used.

By using a parameterized design, our sample implementation maintains a 2:1 timing ratio between LR and RL. Details are complex, but entail using methods such as folding and reusing logic.

Bonded Force Pipeline

It is possible to stay within the time budget even if only one of the three evaluation pipelines (Figure 3·15) is active in a given cycle. Also, many functions overlap among the three interactions. Therefore, to maximize the DSP units' utilization ratio, we merge the three pipelines into a single one with control registers and muxes at different stages of the pipeline (Figure 3·19).

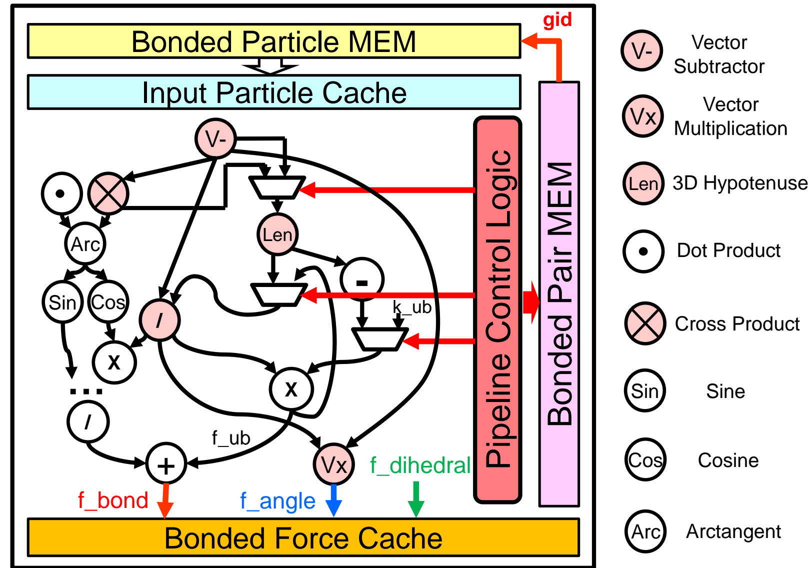

Figure 3·19: Motion Update Pipeline

Motion Update and Particle Migration

Figure 3·20 shows the workflow inside the motion update module. When the updated positions are calculated, the module computes the target cell of the updated particle. Since we are using a short timestep, and we perform motion integration each iteration, particles rarely move across more than one cell. Each cell has a pre-stored lower and upper boundary. If the updated position falls in the current cell range, the output cell id remain the same as the input. Otherwise, the output cell id will add or subtract one depending on the comparison with the boundary value.

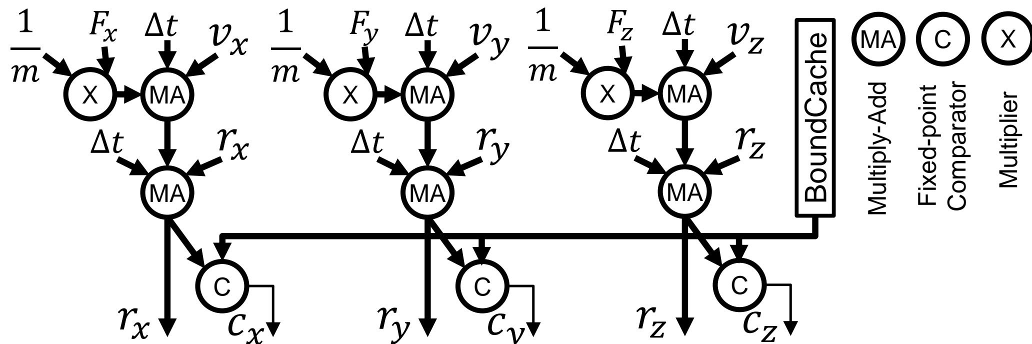

Figure 3·20: Motion Update Pipeline

Table 3.2: MD Design Variations

Design	Memory Mapping Scheme	Workload Distribution Scheme
1	Mem 1	Distribution 1
2	Mem 2	Distribution 1
3	Mem 1	Distribution 2
4	Mem 2	Distribution 2
5	Mem 1	Distribution 3
6	Mem 2	Distribution 4

3.2.4 MD Performance Evaluation

We have implemented, tested, and verified our designs on a Reflex XpressGX S10-FH200G Board which hosts an Intel Stratix 10 1SG280LU2F50E2VG chip (Reflex, 2018). This chip is high-end with: 933,120 ALMs, 11,721 RAM blocks, and 5,760 DSP units, which makes it a good target for implementing FPGA/MD. To get the comparable MD simulation performance on CPU and GPU, we installed Amber 18 (Salomon-Ferrer et al., 2013) on a single node with an Intel Platinum 8160 2.1GHz CPU and various Nvidia GPUs. The operating system is CentOS 7.4.

The dataset we use is Dihydrofolate Reductase (DFHR), a 159-residue protein in water, with 23,558 atoms (Case et al., 2018). The dataset is constrained to a bounding box of 62.23×62.23×62.23 Å, with a cutoff radius of 9 Å. The simulation timestep if 2 fs with Particle Mesh Ewald (PME) every two iterations.

Resource Utilization

The three major components, RL, LR, and Bonded, are designed for a balanced load. To recap, we have two particle-to-memory mapping schemes: mapping all the particle in a single large memory unit and mapping particles onto small block RAMs based on the cell it belongs to. We also have three workload to pipeline mapping schemes: all pipelines work on same reference particle, all pipelines work on the same home cell with different reference particles, and each pipeline works on a different home cell. This yields six different designs as shown in Table 3.2.

Table 3.3: Full system resource usage. Columns 2-4 are post Place & Route. Columns 5-6 give the number of replications of RL pipeline and LR Grid Mapping units in each design. Column 7 lists running frequency of each design. The last two give the stand-alone performance of RL and LR units.

	ALM ($\times 10^3$)	BRAM ($\times 10^3$)	DSP ($\times 10^3$)	RL Num	LR Mum	Freq (MHz)	RL Time(μs)	LR Time(μs)
1	658K(71%)	9.4(80%)	4.4(77%)	52	1	350	64,377	818
2	747K(80%)	9.1(77%)	4.3(75%)	35	2	340	349	513
3	658K(71%)	9.4(81%)	4.0(70%)	52	1	340	969	818
4	747K(80%)	9.1(77%)	3.9(69%)	35	2	340	293	513
5	647K(69%)	9.4(80%)	4.2(73%)	51	2	350	271	513
6	586K(63%)	9.4(80%)	4.0(70%)	41	2	350	260	513

Table 3.3 lists the resource utilization and the number of function units that can fit onto a single FPGA-chip under different RL mapping schemes. We also list the stand-alone performance number for both RL and LR parts. By adjusting the number of *LR Particle to Grid Mapping* modules (column 6), we aim to make the LR evaluation time about twice as much as RL (column 8 & 9).

We note first that Designs 2 & 4 can only fit 35 pipelines. Those two designs have hundreds of memory modules, while the workload mapping requires each pipeline to receive data from all cells. Because of this, a very large on-chip switch (mux-tree based) is required, which consumes a large number of ALMs (196,805). Compared with *Mem 2*, designs using *Mem 1* all have more pipelines, due to the convenience of having a single source of input data. Given the resource usage comparison, it seems that having a global memory provides the benefits of having more pipelines mapped on to a single chip. However, the stand-along RL performance shows otherwise. We describe this next.

MD System Performance

Table 3.4 lists performance numbers for the DHFR dataset on various platforms, including multi-core CPU, GPU, and our six HDL implementations on different FPGAs.

The CPU and Titan XP GPU numbers come from collaborators in an industrial drug design environment. The RTX 2080 and Titan RTX GPU performance numbers are publicly available from Amber (Amber, 2018). Compared with the best-case single CPU performance, the best-case FPGA design has one order of magnitude better performance. The FPGA design has 10% more throughput than that of the GPU performance. Much more evaluation needs to be done, but we believe these results to be promising.

As shown in Table 3.3, RL is the limiting factor on the overall performance. The poor performance of Design 1 is due to the memory bandwidth limitation: for most cycles, pipelines are waiting for data. In Design 2, the distributed memory provides much higher read bandwidth. Design 3 faces a different problem: the number of particles per cell (70) is not a multiple of the number of pipelines (52), which means a set of pipelines (18) is idle after evaluating a single reference particle. It also suffers from memory bandwidth limitations. Design 4 has a happy coincidence that its pipeline count (35) can be divided evenly into 70 and most pipelines will have close to 100% usage (this is subject to the dataset). Designs 5 & 6 are supposed to have similar performance. However, in Design 5, there is overhead on reading the first sets of input data from a single memory unit, while the subsequent read latency can be fully hidden.

Dataset Impact on Mapping Selection

Our system takes advantage of FPGAs' reconfigurability to fully customize the number of pipelines and the mapping scheme of workload and particle storage. Since RL evaluation takes both most of the resources and evaluation time, we focus here on examining the RL performance. Using the provided scripts, we can quickly estimate the number of pipelines and resource usage based on the size of the input dataset and number of cells, along with an estimation of the simulation performance from

Table 3.4: Performance comparison: the middle column shows time to perform one full iteration (23k dataset); the right column shows throughput with a $2fs$ timestep.

Platform	Iteration Time (μs)	Simulation Rate (ns/day)
CPU 1-core	85,544	2.02
CPU 2-core	38,831	4.45
CPU 4-core	21,228	8.14
CPU 8-core	11,942	14.47
CPU 16-core	6,926	24.95
GTX 1080 GPU	720	240.13
Titan XP GPU	542	318.97
RTX 2080 GPU	389	444.05 (Amber, 2018)
Titan RTX GPU	304	567.53 (Amber, 2018)
Design 1: Mem 1 + Distribution 1	64,411	2.68
Design 2: Mem 2 + Distribution 1	370	467.40
Design 3: Mem 1 + Distribution 2	1003	172.36
Design 4: Mem 2 + Distribution 2	313	551.55
Design 5: Mem 1 + Distribution 3	291	593.55
Design 6: Mem 2 + Distribution 3	274	630.25

the six different mapping schemes. In order to further demonstrate the selection of mapping schemes, we use a variety of datasets (5K to 50K) and cutoff radii (leading to different cell sizes). Characteristics are shown in Table 3.5.

Table 3.5: Various testing datasets evaluating the impacts on workload mapping selection

	Particle #	Cell #	Particle #/Cell
Dataset 1	5,000	63	80
Dataset 2	5,000	12	417
Dataset 3	20,000	252	80
Dataset 4	20,000	50	400
Dataset 5	50,000	625	80
Dataset 6	50,000	125	400

The number of pipelines and performance are shown in Figure 3·21. We note first (from Figure 3·21(a)) that the dataset size has little impact on the number of pipelines we can map on a single Stratix 10 FPGA until the dataset grows large enough to cause a resource conflict (in BRAMs). However, this is not the case on simulation performance as shown in Figure 3·21(b). All the performance number is normalized to the Design 1 performance for each dataset. We have the following

observations: **(i)** Design 1 with single particle memory and workload distribution 1 always has the worst performance due to memory bottleneck; **(ii)** When the dataset is sparse (see Dataset 1, 3, 5), Design 6 tends to return the best performance, and the relative performance among the six designs is similar; **(iii)** When the dataset is dense (see Dataset 2, 4, 6), workload distribution 3 provides fewer benefits comparing with workload distribution 2; this is especially clear when the dataset is small and dense.

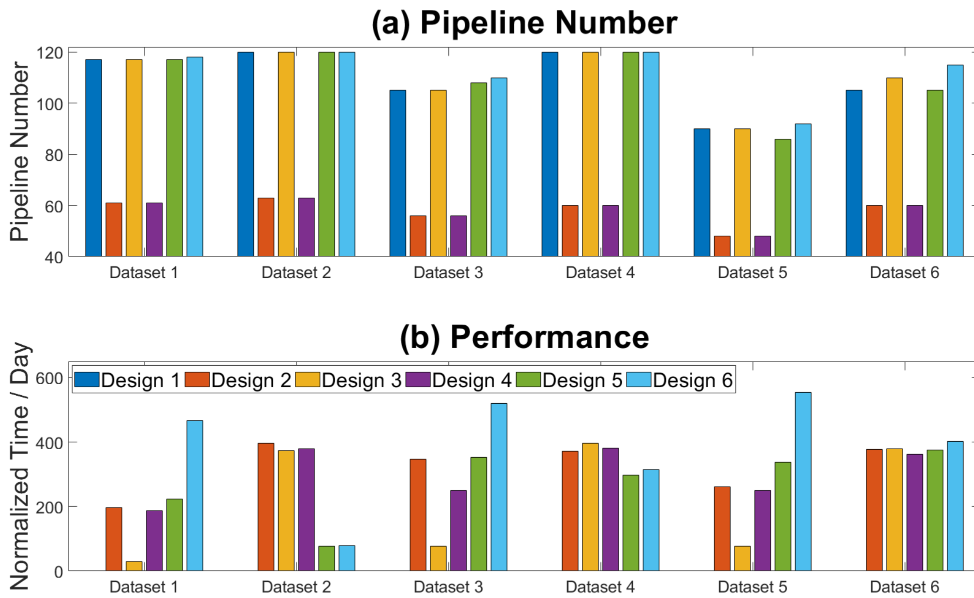

Figure 3·21: Performance with Different Datasets: (a) Number of RL pipelines that can map onto a single FPGA; (b) RL simulation performance, normalized to Design 1 for each dataset.

Verification and Validation

As is usual with complex FPGA designs, we have multiple levels of verification, starting with MatLab models of the computations, HDL simulations of components, HDL simulations of the full system, and the actual implementation. These are also validated with respect to Amber 18.

To validate using energy waveforms, we run two sets of simulations to collect system energy values using different evaluation methods: the FPGA using 1st-order

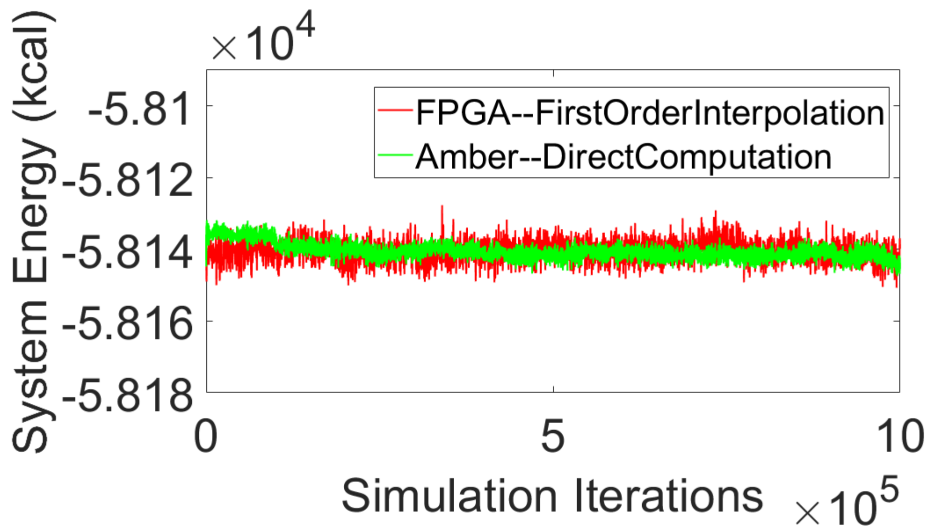

Figure 3·22: Energy Waveform

interpolation and Amber running on a CPU (see Figure 3·22). We note that our simulation system maintains an equilibrium state and that the energy level is similar to Amber's. The variance is likely due to the fact that Amber uses a sophisticated motion integration method (Case et al., 2018) and different smoothing techniques (Case et al., 2005), which are not yet implemented in our system.

3.2.5 MD Strong Scaling onto Multi-FPGA Platforms

Our proposed MD simulation design achieves good performance with a pre-requisites: keeping all the data on-chip. The Stratix 10 FPGA we use provides ∼20 MBytes of on-chip RAM, which is only enough for small datasets with less than 40K particles. Another limitation we have on a single FPGA is the number of pipelines we can fit. Since MD featuring good parallelism at multiple levels, ideally, we can achieve better performance by having more evaluation units in our system.

Another motivation for using multiple FPGAs is FPGA's support for communication. FPGA-clusters with FPGAs directly linked through their MGTs have a proven advantage over other commodity architectures in facilitating communication that is

both high bandwidth and low latency; but also in the collocation of compute and communication on the same device (Sass, R. et al., 2007; Putnam, A. et al., 2014; George et al., 2016).

Given the above reasons, our immediate next step is extending the current single-chip MD simulation system onto an FPGA-cluster.

RL Evaluation on Multi-FPGAs

Since we organize particle data based on their position and group particles within a certain range as a cell, we can specially divide a single dataset and mapping a various number of cells onto each FPGAs. Based on the number of FPGAs and cells, each FPGA can get assigned with a partial cell, a single cell, or multiple cells as shown in Figure 3·23. Each FPGA only need to hold the particle data (position, force, and velocity) within the assigned cells, thus alleviate the resource pressure on each FPGA.

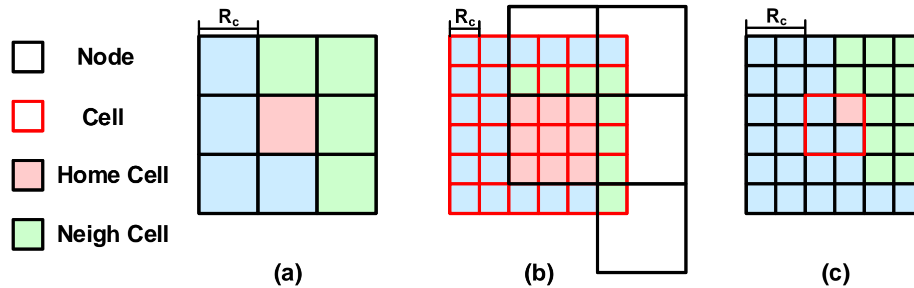

Figure 3·23: 2D analog of cell to node mapping scheme: (a) single cell mapping to single node; (b) multiple cells mapping to single node; (c) single cell mapping to multiple nodes

Each FPGA is responsible for the RL force evaluation, accumulation and motion integration on particles that are assigned to it. Since each FPGA only has partial data from the dataset, multiple phases of communications with other FPGAs are required. At the beginning of force evaluation, each FPGA need particle position data from FPGAs holding the neighbor cells; after force evaluation, each FPGA will send back partial forces of neighbor particles that belong to cells on other FPGAs; during motion

integration process, particles may move from a cell to another, which may be located on different FPGAs. The communication load for sending the position data is also different: (i) When each cell is mapped onto a single node (see Figure 3·23(a)), each FPGA need to broadcast to 13 nearest neighboring FPGAs, the number of particles need to broadcast is the cell particle number; (ii) When there are multiple cells mapping onto the same node (see Figure 3·23(b)), each node still needs to broadcast to 13 nearest neighboring nodes, but the amount of particles to send varies a lot based on the relative location of the neighbor node; (iii) When the number of cells is smaller than the number of nodes, which is the case for strong scaling, each node needs to broadcast to 62 neighboring nodes. In this case, the communication load is the heaviest one.

LR 3D FFT on Multi-FPGAs

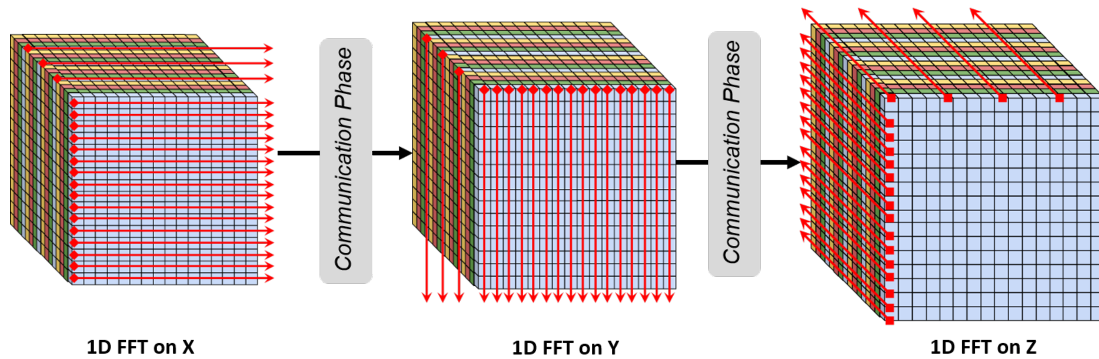

Figure 3·24: 3D FFT on Mult-FPGAs

The main workload of LR evaluation is the 3D FFT. 3D FFT is calculated by decomposing it into 1D FFT and computed successively in each dimension. Since all 1D FFT in a dimension is independent, we divide 3D FFT into three phases with each phase handles a dimension, as shown in Figure 3·24. Between each phases, a transpose communication phase is required to get the data ready for 1D FFT on the next dimension. The strong scaling of 3D FFT is previously been demonstrated

in (Lawande et al., 2016; Sheng et al., 2017).

Need of Communication Support

Due to the physical limitations, a direct all-to-all connection is impossible when numbers of FPGA grows large. Also, communication patterns and data sizes can vary a lot. In order to address those, while making use of the high-bandwidth, low-latency MGTs on each FPGA, a good routing solution is in high demand.

3.2.6 MD Summary

We present an end-to-end MD system on a single FPGA featuring online particle-pair generation, force evaluation on RL, LR, and bonded interactions, motion update, and particle data migration. We provide an analysis of the most likely mappings among particles/cells, BRAMs, and on-chip compute units. We introduce various microarchitecture contributions on routing the accumulation of hundreds of particles simultaneously and integrating motion update. A set of software scripts is created to estimate the performance of various design choices based on different input datasets. We evaluate the single-chip design on a commercially available Intel Stratix 10 FPGA and achieve a simulation throughput of 630 ns/day on a 23.5K DFHR dataset, which is comparable to the analogous state-of-the-art GPU implementations.

3.3 Summary

In this chapter, we use two real-life applications to showcase FPGA's communication and computation capabilities. Both applications show good parallelism. On top of that, MD simulation also requires large on-chip storage space when evaluating large datasets. This makes MD a promising application for acceleration on multi-FPGA systems. What should a real-life multi-FPGA system should look like? And how should we efficiently coordinate the communication and computation capabilities in

such systems? These questions are addressed in Chapter 4 and Chapter 5, respectively.

Chapter 4

FPGA-Centric Cluster Platform

In the previous chapter, we present two model applications that are amenable to multi-FPGA solutions. There are two sets of conclusions. First, those real-life applications demonstrate FCC communication capabilities. Second, those applications also demonstrate the need for a more general multi-FPGA platform that can serve various applications' needs. In this chapter, we present our basic design of the multi-FPGA system. To cover that, we first introduce a few design choices and background on multi-FPGA clusters. This is followed by the architecture of the nominal target FCC, the Novo-G#. After that, we perform experiments on the existing platform to gather baseline communication performance. Next, in order to measure the measured link variance's impact on application performance, we perform a case study using the 3D FFT. Lastly, we compare our system with some existing HPC routing solutions, as well as Network-on-Chip (NoC) designs, and discuss the resulting differences in design philosophy between those systems and FCCs.

4.1 Design Choices

In this section, we compare the FPGA clusters introduced in Section 2.2. Through which, we present some high-level design decisions we adopt and explain the reasons behind it.

4.1.1 Direct and Integrated Network

In most HPC clusters, they adopt the indirection network implementation for the sake of saving efforts on router designs by offloading the communication task onto commodity routers. However, the drawback is the relative longer network latency introduced by various network stacks, especially with near neighbor communications. Apart from Zedwulf (Moorthy and Kapre, 2015) and Catapult I (Putnam, A. et al., 2014), all the multi-FPGA clusters in Section 2.2 have a direct network. While in our case, since we want to take advantage of the low-latency feature provided by MGTs, we choose to organize our FCC in a direct network fashion. To close couple the communication and computation on FPGAs, we also add the routing unit inside FPGA, which falls into the integrated network category.

4.1.2 Network Topology

As we introduced in Chapter 2, for a direct network, the topology generally falls into the category of k-ary n-cube. Modern high-end FPGAs provides multiple high-speed I/O ports, thus our selection of n is more flexible. According to the study in (Bunker and Swanson, 2013), a high-radix router is more desirable due to the shorter network diameter. Thus we adopt **3D-torus** in our FPGA-cluster to not only make use of the 6 transceiver ports on our board, but also matching the logical topology of HPC applications like Molecular Dynamics simulation (Chiu, 2011).

4.1.3 Standard HPC Cluster vs. FPGA Cloud vs. FPGA-Centric Cluster

In conventional FPGA-clusters configured under indirect networks, FPGAs are communicating with each other via the commodity networks. During which, the FPGA initialized data need to reach CPU via PCIe first, then send out on conventional networks, all of which contribute to a millisecond-level latency. Catapult II (A. M. Caulfield et al., 2016) addresses this issue by placing FPGAs CPU and Network Inter-

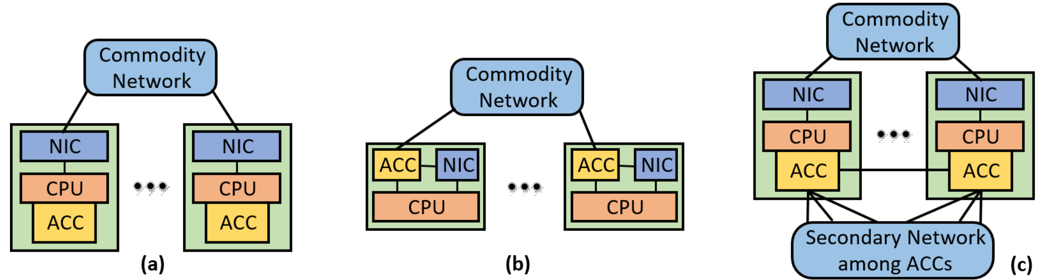

Figure 4·1: FPGA-based HPC system models: (a) stardard HPC cluster; (b) FPGA cloud; (c) FPGA-Centric Cluster

face Card (NIC), which removes the PCIe latency when two FPGAs talking to each other. In Novo-G#, we push this even further by having FPGAs directly connected together via Mult-Gigabit Transceivers (MGT) forming a secondary network, which we referred to as FPGA-Centric Cluster (FCC). To maintain this secondary network, a high-performance router is needed when source and destination are far away and need to traverse multiple hops. Details of which are introduced in Chapter 5.

4.2 Novo-G# Architecture

Three fundamental issues limiting performance are computational efficiency, power density, and communication latency. All of these issues are being addressed through increased heterogeneity, but the last in particular by integrating communication into the accelerator. Novo-G# is designed to be flexible at all levels of the communication and application stacks. With Novo-G# we aim to explore existing communication IP, network architectures, and application-aware communication protocols, therefore we have designed the system to be as reconfigurable as possible.

Novo-G (George, 2010; George et al., 2011) began in 2009 as an effort to create a research cluster using high-density FPGA boards to accelerate scientific applications. The original machine began with a head node and 24 Linux servers, each featuring a quad-FPGA board from Gidel (Gidel, 2010), for a total of 96 Altera Stratix III E260

FPGAs. Over subsequent years, the machine has been upgraded annually and now stands at 192 Stratix III E260 FPGAs in 24 servers, 192 Stratix IV E530s in 12 servers, 64 Stratix V GSMD8s in 16 servers, and the second set of 64 Stratix V GSMD8s in 16 servers under construction. Each server features dual Intel Xeon multicore processors. Server connectivity is provided by gigabit Ethernet and DDR/QDR InfiniBand within the system, and a 10 Gb/s connection to the Florida LambdaRail.

The Novo-G# system is part of our effort to create an FPGA cluster that can handle communication-intensive applications. In keeping with that theme, the system features ProceV boards, which are PCIe-based accelerator boards from Gidel populated with Stratix V GSMD8 FPGAs from Altera. The GS-series devices are optimized for high performance, high bandwidth applications with support for up to 36 on-chip transceivers that can operate up to 12.5 Gbaud. Each FPGA is connected to two 8GB DDR3 SODIMM and two 36-Mbit SRAM memory banks and communicates with the host CPU via PCIe v3. The FPGAs are housed in a 4U chassis with two Xeon E5-2620V2 (Ivy Bridge) processors per server. The servers themselves are interconnected via Gigabit Ethernet and QDR InfiniBand.

Our FPGA platform vendor Gidel has provided invaluable assistance by designing a custom daughterboard that allows external access to 24 high-speed transceivers. The transceivers are grouped into six bidirectional links, each link consisting of four parallel channels, enabling the construction of a 3D torus of arbitrary size. Physical connectivity between the boards is provided by a Commercial Off-The-Shelf (COTS) CXP-3QSFP+ split cable that enables each FPGA to be connected in six different directions. Initial deployment of the Novo-G# system was completed in the second half of 2014 with 32 Stratix V boards housed in eight chassis and supporting up to a $2 \times 2 \times 2$ torus, and an upgrade to 64 nodes ($4 \times 4 \times 4$ torus) was completed in August 2015.

A part of the hardware resources on each FPGA is used to implement a network stack that services the 3D torus network. The network stack is responsible for accepting data from the application logic, packetizing the data, routing packets across the 3D torus network by the shortest route, and delivering the data to the application logic at its destination. These functions represent a subset of the services provided by the lowest three layers of the OSI reference model (i.e. physical, data link, and network layers).

Figure 4·2 depicts Novo-G#'s 3D torus network stack, and the network services associated with each component. The IP cores used for the transceiver interfaces are primarily supplied by Altera and interface directly with hardware resources attached to each transceiver channel. The remaining blocks are implemented as RTL code and therefore do comprise a resource overhead for each FPGA. Data generated by the application logic is packetized by the internal receiver block and stored in a FIFO. Similarly, the external receiver blocks accept packets or streaming data from the transceiver IP. One or more routers are used to route packets from the receiver to the transmitter blocks, which transmit the data further along with the network or to the application at the destination node.

4.3 Novo-G# Link Performance

The inter-FPGA communication is controlled by dedicated ASIC components (SerDes) resides on each FPGA chips. The SerDes module is configured by the vendor-provided IP cores, with support for various link layer protocols like Ethernet, Interlaken, SerialLite, XAUI, etc..

4.3.1 Configurations of MGT

Figure 4·3 presents the datapath inside Stratix V FPGA's transceiver blocks. It has two major blocks: Physical Medium Attachment (PMA) and Physical Coding Sub-

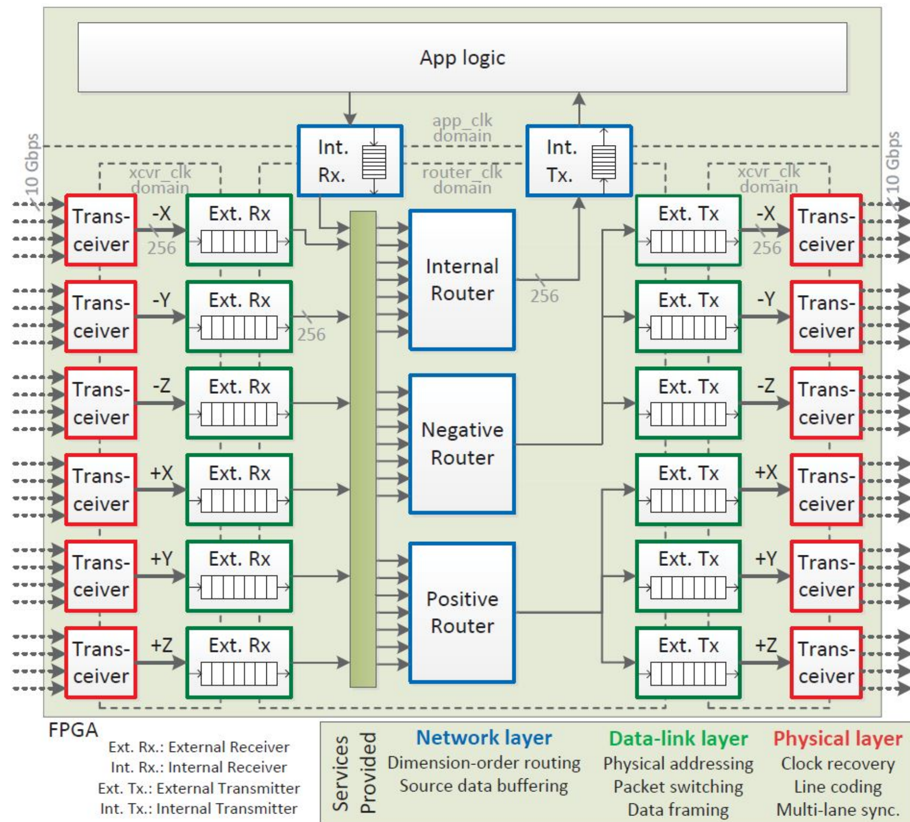

Figure 4·2: Novo-G# node architecture detailing the 3D torus network stack. Services provided to the user through RTL or third-party IP are also shown.

layer (PCS). The PMA block directly connects to the physical channel while provides functions like data conversion between serial and parallel and generate clock signals for/based on TX/RX data stream. The PCS block is in charge of digital processing functions between PMA and FPGA user logic, including but not limited to: error checking, encoding/decoding, link-layer protocols, etc.. Most of the configurations can be finished inside the Transceiver IP core user interface. But when the data rate is high, other vendor-provided tools like *Altera Transceiver Toolkit* is needed in order to find the best analogy parameters on the physical link.

In theory, the per channel (direct connect to transceiver I/O pin on FPGA) datarate on Stratix V FPGA is 14.1 Gbps (Altera, 2014c) working in a full-duplex

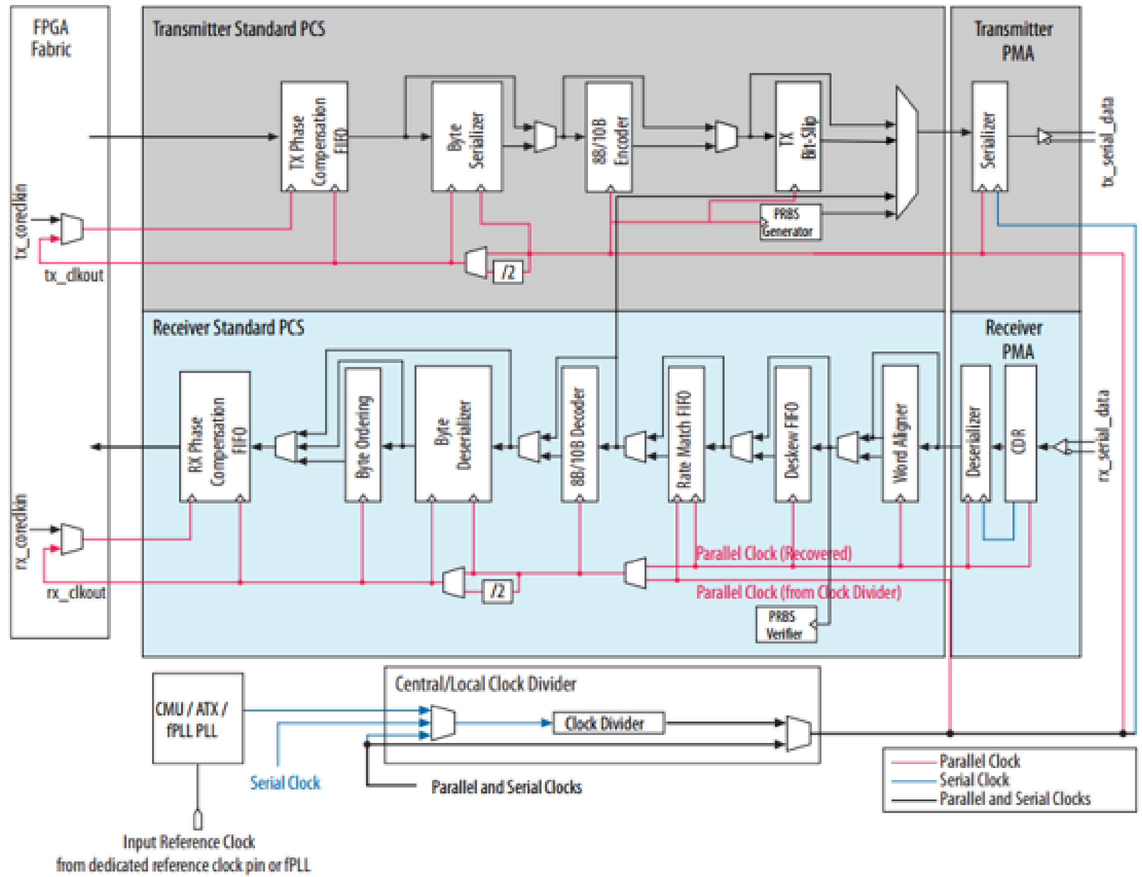

Figure 4·3: Stratix V Transceiver Architecture (Altera, 2014c)

mode. Since we have 4 channels per link, each link could reach an aggregate bandwidth of 56 Gbps. Each port is used to connect to one of the six immediate neighboring nodes in a 3D-torus. Due to the power supply limitation on the board, we detect a high link error rate when having six link working at maximum bandwidth. With the worst case being link connection failure during runtime. In order to minimize link error, we thus cap the link data rate at 20 Gbps in our later experiments.

4.3.2 MGT Link Protocol

We choose Interlaken PHY and SerialLiteIII (Altera, 2015b) as our MGT link layer protocol. Both protocols are based on the Interlaken protocol, featuring low-latency

and high-throughput, which is a perfect fit for our needs.

The Interlaken IP core wrapping multiple function units inside: TX/RX FIFO, MetaFrame generator, 64B/67B encoder, and CRC32 generator. Among those, 64B/67B encoding is based on the IEEE 802.3 64B/66B encoding, which transforms the 64-bit user data into 67-bit that provide enough state changes to allow reasonable clock recovery and alignment of the data stream. The CRC32 is provided as a diagnostic tool on each lane to trace errors. The data in Interlaken PHY are transferred in a frame-by-frame manner, which is called MetaFrame. One MetaFrame is formed by MetaFrame generator including one synchronization word, one scrambler word, one diagnostic word, one or more skip words, and the data payload. One obvious drawback for Interlaken PHY IP is all four lanes inside a single link is working independently (individual FIFOs and recovered clocks), which lead to the desync problem on the receiver side among the 4 64-bit data transferred on a single link.

The second IP we used is SerialLite III. It is based on Interlaken PHY, but on top of which, it provides synchronization among multiple lanes. But this is not coming for free: latency is introduced during the syncing process, and of course, more buffer resource usage.

The comparison between the two protocols is shown in Table 4.1. The latency comes from our on-board measurement.

Table 4.1: Comparison between Interlaken IP and SerialLite III IP

Pattern Name	InterLaken	SerialLite III
Datarate	10 Gbps	10 Gbps
Latency	177 ns	307 ns
ALM Usage	240	1080
BRAM Usage	0	0

4.3.3 Measurement of Link Variance

The inter-FPGA link variance is introduced at multiple places: length of cable, reference clock used by the transceiver IP, the delay inside send/receiver buffer, and clock jitter introduced by on-board oscillators.

In this section, we measure the link latency on four different levels:

- in one lane (channel) over time,
- among different lanes of the same link,
- among different links of the same FPGA chip, and
- among different transceivers of the different FPGAs.

We use the classic ping-pong test and validated by examining the waveforms directly. The FPGA on the right has a traffic generator which sends data to the MGT interface in a parallel format. At the same time, the monitor creates and records a random value for the outbound data stream of the transmitter. As data is transferred to the tx parallel port module, a counter in the monitor is initiated. The data is then sent through the serial link to the FPGA on the left side, after which the data is routed from that FPGAs RX port to its TX port and transmitted back again. The FPGA on the right side eventually receives the return data and a comparator checks it with the recorded sent data. Once that happens, the counter in the monitor block stops. This mechanism is very accurate, working on the frequency from one PLL.

We deployed two FPGAs to determine the variation among the boards. For each board, two MGTs are configured with Interlaken PHY IP to explore the variation among the MGTs on the same board. Each MGT is configured with 4 lanes to discover the variations among the lanes of the same link table.

To explore level-one latency variation (single lane) we sample its latency 75 times and plot the results (see Figure 4·4). We find that distribution appears to be random

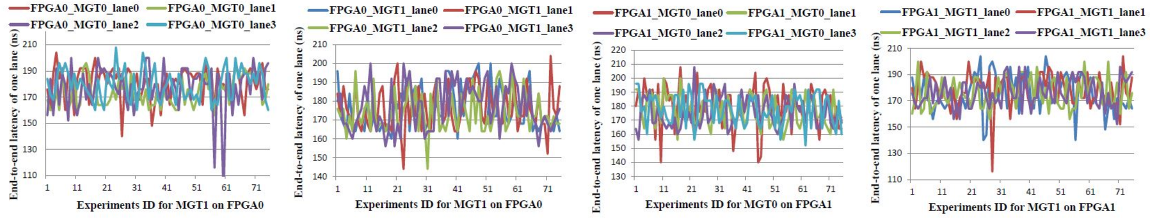

Figure 4·4: Variations of end-to-end latency over time for four lanes caged inside a single QSFP connector

with distribution TBD. The mean and variance are shown in Table II. The level-two variations are found by comparing the different lanes in the same transceiver. The level-three variations exist in the differences between the two MGTs of the same FPGA. The level-four variations can be identified by comparing the differences between FPGA0 and FPGA1. When we look at the variation over time for a single lane, the maximum latency is about 40% larger than the minimum (on average). In some extreme cases (Lane3 of the MGT1 on FPGA0), the maximum latency is close to double of the minimum latency. When we compare the difference among multiple lanes and boards, however, the variation is relatively trivial compared to the fluctuation in the timing dimension.

Figure 4·5 gives us a different perspective with the (a) panel giving latency distributions for different MGTs and the (b) panel different FPGAs. While not a perfect fit, we use a Gaussian with a mean of 176.8ns and Standard Deviation of 12.4 ns to describe this distribution.

4.4 Cast Study on 3D FFT

By conducting several simple experiments, we obtained an estimate of link latency, and variation of link latency. In this section, we apply this information to the simulation of the 3D FFT.

As described in our previous work (Sheng et al., 2014), our FPGA-based cluster

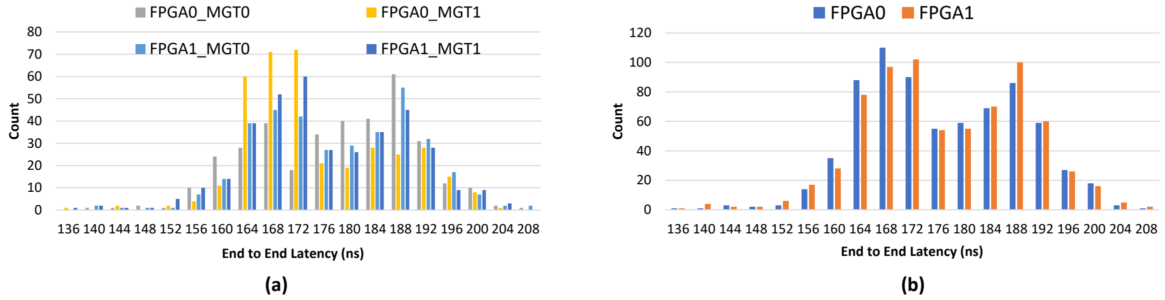

Figure 4·5: (a) Latency distribution comparison among four Multi-Gigabit Transceivers (MGTs) (b) Latency distribution comparison between two Altera Stratix V boards

topology is a 3D-torus. The routing scheme is table-based routing (Dally et al., 1998; Kinsy et al., 2013a). Table-based routing means that there are one or more routing tables in each node, and there is a corresponding entry in each routing table for each incoming packet. The packet header carries index information to find the correct entry. After the correct entry is found, the information in the entry is used to determine the next node for the packet. For the network switch architecture, we use a simple ring-based design (also as described in (Sheng et al., 2014)). The ring is composed of 7 routers, six of which deal with the traffic on the torus links and one which injects or ejects packets to or from the network. If there is no congestion, each packet spends just one cycle on each (internal) router. In general, each packet spends 2-4 cycles on each node.

The 3D FFT is a well-studied application. Our previous work (Sheng et al., 2014) proposes a generalized mapping for 3D FFTs of arbitrary size 3D-tori with arbitrary numbers of nodes. The mapping defines the source and destination of each packet. By knowing these a priori we can use table-based routing to orchestrate the packet routes so as to minimize congestion. We note that the current communication architecture allows sufficiently deterministic packet arrivals to support this form of application-aware routing.

Previously in (Sheng et al., 2014), our simulations assumed fixed link latency.

In this paper, we replace fixed link latency with the varied latencies that obey the Gaussian distribution. Table 4.2 demonstrates the new FFT simulation results and the comparison with the results that use fixed latency. Cluster size is 4^3, FPGA operating frequency is 150 MHz, and the data type is 32-bit floating-point. The right three columns have latencies in microseconds. Fixed Latency denotes average latency with no variance, Variable Latency denotes that variance has been modeled, and Reference denotes previous results where we assumed (very conservatively) 100 MHz FPGA frequency and 500 ns fixed link latency (Sheng et al., 2014).

Table 4.2: 3D FFT Latency Case Study

FFT Size	Fixed Latency	Variable Latency	Reference
16^3	1.63	1.64	3.98
32^3	2.24	2.48	5.46
64^3	6.72	6.92	16.76
128^3	35.13	41.14	101.11

We note that despite the apparent link variance, the two new results are not that different. This is mainly because the variation of the latency on the physical links is small when compared with the latency caused by congestion. But still, the variation of physical links degrade the performance for all cases because they increase the cost for synchronization.

4.5 FPGA-Centric Cluster (FCC) Router Design Constraints

There have been a vast amount of studies on communication infrastructure design, both on NoC (ASIC- and FPGA-based) and cluster. We present a selection of those in Chapter 2. However, due to the underlying hardware configuration and performance requirements, there are great differences in design constraints that lead to different design decisions being preferred for FPGA-Centric Clusters. We list those constraints below and discuss the impact on FCC router designs methodologies by comparing with both NoC and cluster solutions.

4.5.1 Network is direct

Since a central reason for using FCC is collocation of application and communication logic, this choice is obvious.

Single router per chip

Under a direct network, when a packet's source and destination node is not directly connected, it needs to traverse multiple intermediate hops. On each intermediate hop, it first checks if the packet is targeting the current node. If not, then it needs to determine what is the next immediate node to send the packet. When more than one packet is requesting the same output, arbitration is also needed. Plus, a buffer is needed to hold those packet that is not winning the arbitration. Thus, inside a direct network, a router is requested on each node.

Unlike NoC, our FCC implementation only need a single router. However, depending on the applications, there can be NoC implemented on top of the cluster router. As far as this thesis concerns, all the routers talked in this chapter is focusing on the cluster router. Having a single router per chips means we have more resource budget to allocate to the router implementation.

No Global Clocking

Inside commercial switches and NoCs, most function units involved in the routing decision making is working under a single global clock. However, on the cluster level, especially when the network diameter is large, it is impossible to have a global clock. That means data transfer between routers is performed asynchronously. Luckily, our MGT IP provides physical level clock signal recovery from the received patterns. Still, we need a complex scheme to synchronize the send and receive status to coordinate the data traffic.

3D-Torus Topology

As we discussed earlier in this chapter, we adopt the 3D-torus topology on Novo-G#. This topology requires each router inside a node to send and received data to and from 6 ports in parallel. Since the switch complexity grows with the square of the number of ports, our cluster router is larger than 2D NoC routers.

4.5.2 Long Inter-node Latency

From the experiments conducted in the previous section, we know that the link latency is in the neighborhood of 170 ns. Given the fact that applications running on Stratix V FPGAs generally runs at a frequency of $200 \sim 300$ MHz, the average link delay is around 30∼50 cycles, which is quite different from NoC designs (link delay usually within 1∼2 cycles). The long link impacts our router design on multiple levels.

Router pipeline stage

For NoC router, due to the short link delay, Papamichael and Hoe suggest NoC router should have the minimal number of pipeline stages to boost routing performance (Papamichael and Hoe, 2012). While for clusters with indirect networks (using commodity switch), they tend to have longer link delay as FCC does. Thanks for the large packet size and large buffering space on top of the us level latency, the overhead inside the switch itself has limited impact on under those use cases. Similarly, in our case, we have implemented our routers with multiple stages.

Flow control complexities

The short link delay enables deterministic flow control in NoC router: the upstream router knows exactly how much data it can send down (Huan and DeHon, 2012). Commodity switches address this by having a huge buffer to overcome the delay of flow control signals. For Ethernet switches, when the buffer is almost full, it sends out

a "pause" frame to either slow or pause the sender from sending more data. While in our case, we do not have the budget to implement an "infinite" large buffer. Thus a better flow control mechanism is in need.

Large Buffers

Since the sender does not have the immediate status on the receiving side, to avoid long waiting time between each sends operations, FCC router needs to have larger buffers to let the sender keeping sending data before receiving the "stop send" signal initiated by the receiver.

4.5.3 Large Phit and Flit Size

In Wormhole routing (Dally and Towles, 2004), *Phit* is the unit that is transmitted on the physical link; *Flit* is the basic unit for flow control. When implementing on NoC environments, the phit size equals to the number of wires between to routers. The flit width is usually the same as phit size, thus to provide enough data on the inter-router link. To reduce the number of wires between two routers, as well as reduce buffer resource usage for flits, the phit size is usually small (less than 32-bit). While for both clusters and FCCs, data on the physical link is transmitted serially for better quality. Both 10G Ethernet and Infiniband use 8B/10B encoding (turns 8-bit data into 10-bit to ensure enough data transition in each transmit). Thus the unit size for each transmission is 8-bit. Cables used for 10G ethernet composed of 4 differential pairs with each one running at 10 Gbps. Ethernet sends data one byte at a time on each pair. Infiniband cable also featuring differential pairs, based on speed grade, they can have 1, 4, 12 such pairs on each link. The FPGAs we used on Novo-G# featuring QSFP connectors, which housing 8 differential pairs with 4 for transmission and other 4 for receiving.

Flit width smaller than phit

However, our MGT IP controller returns a user interface of 256-bit data. Thus in FCC, the phit size fixed as 256-bit. If we use the same size for flow control, it requires large buffer space to hold those flits, plus increasing the wiring complexity of switch. Thus we choose a smaller flit size. However, to provide enough data for transmission, our router needs to run at a frequency that is multiple on top of the MGT returned clock signal.

4.6 Summary

In this chapter, we present the high-level design decisions made for FCCs as well as the resulting underlying hardware architecture. We measure the communication between communication performance between pairs of FPGA chips at multiple levels. We conduct a case study on the 3D FFT and demonstrate complex multi-hop communication support on FCCs. Further, we confirm that the low link variance on FCC has little impact on the application, which in turn, makes static-scheduled routing feasible. We compare with routing solutions for HPC systems and NoCs. Using differences in the underlying working environment enables the enumeration of router design constraints on FCCs, which influence the router design. Details are presented in Chapter 5.

Chapter 5

FPGA-Centric Cluster Router, Part 1: Dynamic Router Design for Unicast Traffic

The goal of this chapter is to find the best router designs for FCCs for unicast operations. To achieve that, we start with conventional router designs: based on the difference of hardware constraints and characteristics when compared with other platforms, we identify the major functional units needed in our FCC router, while making modifications based on the FCC-specific characteristics. Three different dynamic router designs are covered in this chapter: two wormhole style routers and one virtual cut-thorough style router. We end this chapter with performance comparisons among the three designs under various workloads.

5.1 Conventional Design: VC-based Wormhole Router

The VC-based wormhole router was first proposed by William Dally in (Dally, 1990; Dally, 1992; Dally and Aoki, 1993). Featuring low resource utilization and high throughput (Dally and Seitz, 1986), the wormhole router quickly became the *de facto* router architecture in multicomputer systems (Lin and Ni, 1991; Ni and McKinley, 1993; Draper and Ghosh, 1994) and NoCs (Mello et al., 2005; Shi and Burns, 2008; Papamichael and Hoe, 2012). In this subsection, we briefly walk-through the major components and flow control mechanism inside the conventional wormhole router.

Wormhole router break packet into small units called *flit*, which is the smallest unit on which flow control is performed: each component inside a wormhole router process or buffer a single flit at in each cycle. The information transferred over physical links is call *phit*, which is usually the same size or smaller than a flit (Dally, 1992). Wormhole switching breaks a large packet into multiple flits. Based on the flit ID inside a packet, they are referred to as the head, body, and single flit. The packet information, including size, source, destination, priority, etc., is only carried inside the head flit, while the rest flits from the packet are following the head flit and relying on the head flit to set up routing path for them. Since only the head flit carries routing information, once router resource is assigned for a particular packet, it cannot be reused by other packets unless the current packet finished traversing the router. Wormhole router tends to have smaller buffer space, thus flits from the same packet may be scattered on multiple hops and flits are forwarded one after another on each hop, as indicated by the name "wormhole". If the head flit is blocked on a particular node, then the rest flits from the same packet is blocked in place, which stops other packets from using the routing resources (Dally, 1992). To address this blocking issue, Virtual Channel (VC) is introduced as a buffer that can hold one or more flits of a packet along with associated state information (Dally and Seitz, 1987). Multiple VCs can share the bandwidth of a single physical channel.

Figure 5·1 depicts the block diagram of the conventional VC-based wormhole router. Each flit arrives the router at the *Routing Compute (RC)* unit, which determines the output port through which the flit leaves. Carrying the RC result, the flits enters the input unit, which consists of a set of *VCs*. For a newly arrived head flit, the *VC allocator* select among the free VCs and assign one with match class for the packet. The head flit and a few following flits from the same packet are buffered inside VC until the required output port is available for that flit. One thing worth mention

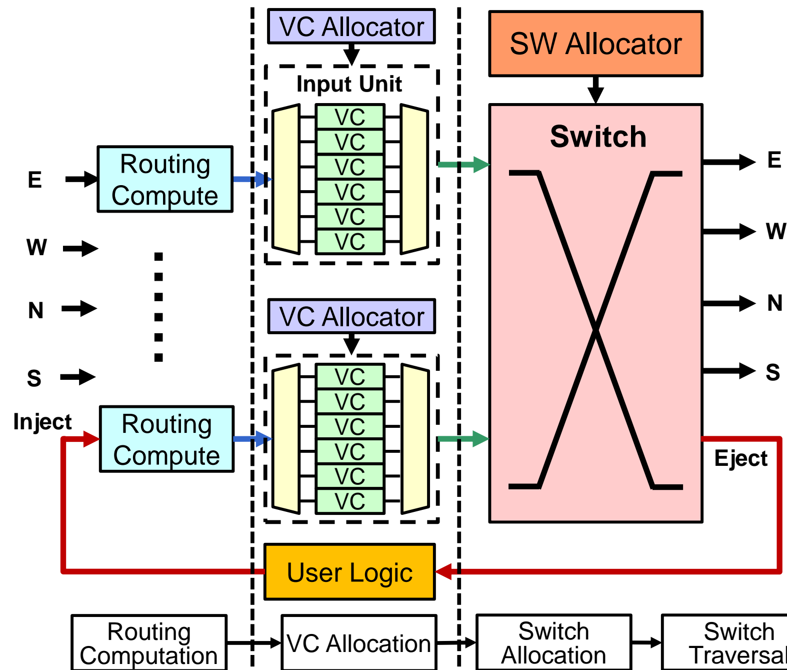

Figure 5·1: Classic VC-based Wormhole Router Architecture

is that VCs are not necessarily large enough to hold the entire packet, as a matter of fact, VCs inside wormhole router usually has a buffer depth of one or two flits. Each input unit is also featuring a finite state machine which manages each VC's status (indicating if particular VC is idle, or notifying switch which output port the flit is requesting) and forwarding the flits inside VC to switch based on the feedback signal from *Switch Allocator*. On the switch side, whenever an output port is available, it arbitrates among VCs that requesting that output. Only a single VC can win the arbitration at a time. Once granted access, VC starts forwarding the buffered flits to the switch output and keeping that output occupied until the entire packet is sent out.

As we mentioned before, if one packet needs to hold all the router resources until all the flits from it are passed, it could cause blocking issues. The use of VCs comes in handy in solving this issue on two levels. First of all, having multiple VCs enables bypassing the blocked packet and temporarily assign the routing resource to other

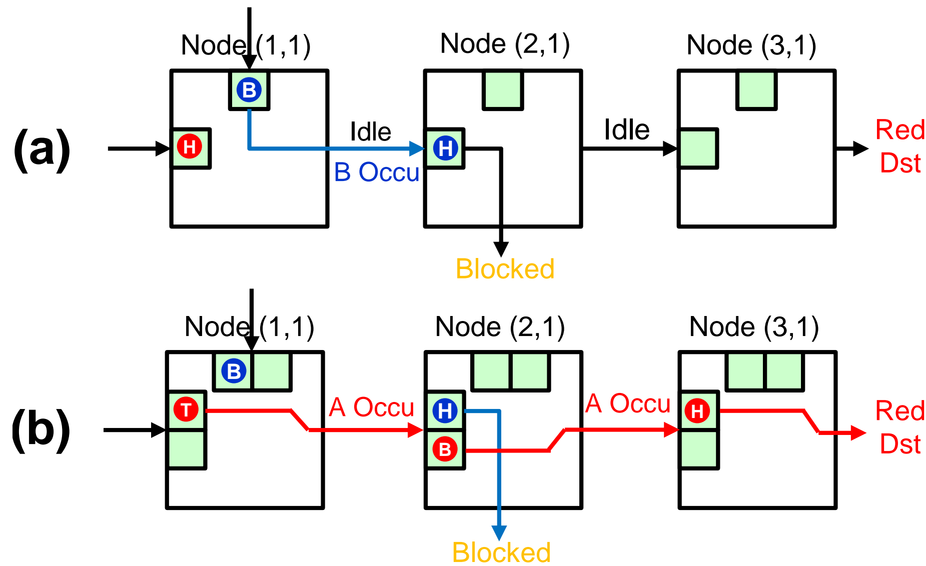

Figure 5·2: An exmaple showing how VC-based flow control solves the blocking issue: (a) Non-VC wormhole switching; (b) VC-based wormhole switching.

non-blocked packets. One example is illustrated in Figure 5·2. For the non-VC implementation, the head flit from the blue packet is blocked on Node (2,1), with the immediate body flit from that packet blocked on Node (1,1) and occupying the east link. In the meantime, the red packet is waiting for the east link to be freed and move to Node (3,1) via Node (2,1), which is not being utilized. Any attempts to move red packet is not viable in this case since the input slot on Node (2,1) is already taken by blue ones. However, this is a problem in the VC-based router. By multiplexing multiple VCs on a single physical link, it provides an extra slot for the red packet, while holding blue packet's head flit in flit. In this way, the red packet is no longer blocked. Theoretically, this solution is perfect. While in practice, since only the head flit carries the routing information, when the blue packet gives away the link control to the red packet, it also lost the connection to the exact VC that the head flit is utilizing. Later on, when the red packet is giving link control back to the VC that is holding the blue packet, it also needs to recover the downstream VC id to

which the blue flit should be forwarded. There are two potential solutions. **(i)** Perform downstream VC allocation on the upstream node, in that case, the VC that is holding blue packet knows exactly which downstream VC it should reach. Accordingly, an extra segment inside each flit is required to let the receiving node know which VC this packet should enter. **(ii)** The downstream node performs VC allocation in place, while maintaining a lookup table recording which packet should enter which VC. The table is indexed by packet id, which is also need be carried inside each flit. Both solutions have their pros and cons, plus it requires extra bits in each flit. The details of which is further discussed in Section 5.4. Another benefit brought by VC is that it can solve the deadlock issue by breaking channel dependency loops. We will further discuss this in Section 5.2.4.

Keep in mind that all three proposed designs in some level share a similar flow control manner with the classic VC-based wormhole router. However, of course, modifications are needed to adjust to the FCC working environment.

5.2 Shared Design Components

We proposed three different sets of designs in this chapter with different levels of resource usage and flow control mechanisms. However, still, they share a few characteristics in common, which is introduced in this section. We first introduce the packet formatting we defined in our router. Following that, we list a few routing algorithms and switch arbitration, that we believe are suitable for FCC communication workload. Later on, we list a selection of measures we take to avoid deadlock and livelocks.

5.2.1 Packet Formatting

We aim to support an arbitrary size of packets. To achieve that, we support five different types of flits: head, body, tail, single, and credit. Usually, a packet starts with a head flit, which is followed by several body flits and ended by a tail flit. If the

packet only has one flit, its flit type is single flit. The credit flit is used to transfer credits information from downstream nodes to upstream nodes.

All kinds of flits have a flit-type field and a payload field. Besides the payload field in credit flit contains the credit information of a downstream node, the payload in other flit types all contain the payload data. A single or head flit also includes the destination field, VC class field, priority field, as well as misc field required by different routing algorithms and flow control mechanisms (details of which is further introduced later in this section). A head flit also required to carry a packet information field indicating packet size. The priority field is used when multiple packets are competing for the same output port. Its content might be packet age, distance from the destination, or both.

The phit size on FCC is determined by the FPGA fabric and MGT controller IP. Each FPGA board provides six QSFP connectors housing four duplex transceiver channels each. The MGT IP provides a 64-bit user interface for each channel (Altera, 2014c). Thus, on each port, the concatenated read & write datawidth is 256-bit, which is the phit size. In traditional NoC design, the flit size tends to be equal or larger than phit size. However, this costly in our case since the wire and buffer size scales linearly with flit size. Our solution is to use a smaller flit size while increasing the router operating frequency. On one hand, since we only need a single router per chip, we have more resource budgets comparing with NoC router. Plus larger flit size contributes to better payload-to-control ratio. But on the other hand, we want our router running at a higher frequency. The routers throughput should in line with the transceiver bandwidth. Configured at 20 Gbps, the MGT IP returns a user clock running at around 150 MHz. To meet the throughput requirements, our router should run at the same frequency with 256-bit flit, or double that with 128-bit flit. Due to FPGA's place & route limitation, a running frequency of more than double the MGT

returned frequency is hard to reach.

Figure 5·3 gives an example of the packet format, including the bit order for each segment for an $8 \times 8 \times 8$ network with a 128-bit flit.

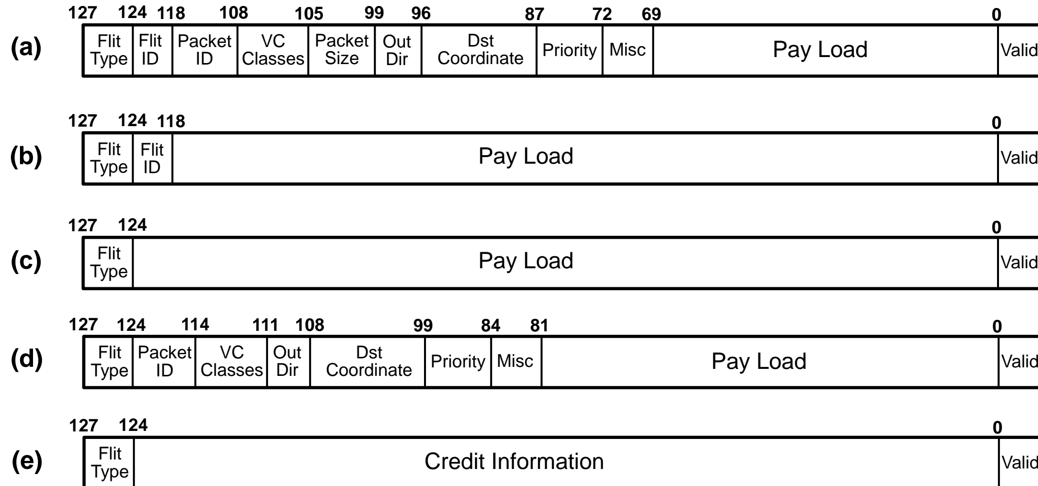

Figure 5·3: Packet Formatting Example: The exact bit field may vary based on application requirements. For the shown example, the constraints are: Network Size $8 \times 8 \times 8$; Max Supported Packet Size: 64; Max Packet ID: 1024; Priority Value range: $1 \sim 2^{15}$; Misc Routing Info (for O1TURN and RLB): 3-bit; (a) Head Flit; (b) Body Flit; (c) Tail Flit; (d) Single Flit; (e) Credit Flit.

5.2.2 Routing Algorithms

Routing algorithms play a key role in determining an optimal path among hundreds of possible routes for any given source and destination node. In Section 2.2.1, we briefly introduced a variety of routing algorithms. Here, we have a closer look at five different routing algorithms and introduce the adaptation we made based on FCC working environments. Among the five algorithms, four of which are oblivious ones, with the last one being adaptive. All the oblivious algorithms can be implemented either dynamically using on-chip computation units, or statically using routing tables. However, the last one can only be implemented dynamically. To keep the design consistency among all the routing algorithms, and reduce the packet overhead, we

choose the distributed routing computation method and implement all the algorithms in a dynamical evaluation manner using on-chip computation units.

Keep in mind that all the implemented routing algorithms here are supported in all three proposed architectures, the selection of which, is served as a dimension of our design space.

Dimensional Order Routing (DOR): Oblivious, Minimal

In 3D-torus, the topology adopted in Novo-G# (George et al., 2016), there are six possible dimensional orders: XYZ, XZY, YXZ, YZX, ZXY, and ZYX. Since the topology is decentralized and symmetrical in each dimension, there is no performance difference among those six. So we select XYZ as our order. The implementation cost is meager: three comparators are used for selecting among dimension. Since our inter-FPGA link is fully-duplex, on each dimension, there are two ways to move along. Thus we add another comparator to select the direction with minimal distance.

DOR is inherently deadlock-free. Thus no extra work is needed for deadlock avoidance.

Orthogonal One-turn Routing (O1TURN): Oblivious, Minimal

When all the packets following the XYZ order, one obvious drawback is the workload imbalance among X links and Z links at the beginning of batch workload since all the packets are flowing on the X links. O1TURN address this issue by randomly selecting among the 6 different orders, thus diverse the routing paths. Once the dimensional order is determined, the routing behavior is similar to that of DOR.

The hardware overhead is minimal: a random number generator is needed during the packet generation stage to select among the six orders; a 3-bit segment is allocated inside head flit carrying the order information.

The deadlock avoidance mechanism is introduced in detail in Section 5.2.4, which

including forbidden turns. If any routes among those six violate that, it is removed from the selection list.

Randomized Minimal Routing (RMR): Oblivious, Minimal

This algorithm is derived from ROMM, but with multiple modifications. First of all, since the original ROMM is optimized for 2D-mesh or torus, its selection of intermediate nodes are limited in the minimal sub-mesh between source and destination. When adopting this in 3D-torus, we extend the selection range to the minimal subcube constrained by source and destination. Next, we removed the constraints on the number of phases. We make this change for a couple of reasons: **(i)** it is hard to force the same number of phases for all the packets due to the Manhattan distance of from source to destination can vary a lot among different packets for nonuniform workload; **(ii)** generating the intermediate node requires each node have a random number generator to select intermediate nodes, but this module is not used if not on the intermediate node, which leads to idle cycles on those modules; **(iii)** each packet needs to carry the information for both the real destination and the intermediate destination.

Given the above reasons, we apply the following changes: when arriving on an intermediate node, the packet could have at most 3 output directions if following the minimum path; the routing decision is made among all the possible directions randomly; a dedicated Pseudo-random Binary Sequence (PRBS) generator with limited resource usage is added to each node, details of which are introduced later. The third modification we made, related to VC selection. Conventional ROMM use s VCs for s-stage routing. Whenever a packet passes an the nth intermediate node, it can only enter the nth VC before reaching the $(n + 1)$th intermediate node. Since we perform random selection on each node, we abandon this setting to reduce the number of VCs.

Similar to O1TURN, the random selection for output should comply with the

forbidden turns.

Randomized Load Balance Routing (RLB): Oblivious, Non-Minimal

RLB follows DOR routing philosophy with the difference being packets not necessarily have to move along the shortest path on each dimension. On a bidirectional torus, packets can take either a positive or negative direction and are guaranteed to reach the destination on that dimension. Most of the time, only one of the routes are minimal and is always taken by DOR. RLB doing this differently in the sense that there is a probability that packets may take the longer path, thus to alleviate the routing pressure on a single direction when traffic load is unbalanced. Assume the number of nodes on one dimension of FCC is N, and the shortest distance between the source and destination is P on that dimension. When $P < N$, the probability that the packet is routed along the shorter path is $\frac{N-P}{N}$, while the probability of taking the longer path is $\frac{P}{N}$. In this way, packets still have a high chance of taking the shorter path, while a chance of taking the shorter path is still provided.

In our HDL design, a PRBS generator is in place to generate a random integer between 0 and $N-1$. If the generated value is larger than P, the packet takes the short path and vice versa. Similar to O1TURN, the path to take on each dimension should be determined at the packet generation stage. The 3-bit section allocated for O1TURN inside head flit can be reused for indicating the direction to take on each dimension.

RLB is deadlock-free since it follows the dimensional order. Even though it is a non-minimal routing method, RLB is also livelock free. Once a path has been selected for a packet, it monotonically advances along the route, thus shorten the distance between the packet and destination at each hop. Since there is no incremental misrouting, all packets reach their destinations after following a predetermined, bounded number of hops.

Credit Count Adaptive Routing Algorithm (CCAR): Adaptive, Minimal

Adaptive routing algorithms can make dynamic routing decisions based on a variety of networks information: based on remaining hop count in each dimension (Badr and Podar, 1989); based on whether the current moving path is congested (Glass and Ni, 1994); based on downstream free VC numbers (Dally and Aoki, 1993); or based on output buffer available slots (Singh et al., 2003). The above mentioned adaptive solutions all have their drawbacks: either lacking adaptation to real-time traffic load or requesting information that cannot be provided in-time when working on FCCs. Kim, et. al. proposed the CCAR algorithm in (Kim et al., 2005) by counting the downstream returned credits. In our proposed FCC router designs, we adopt the credit-based flow control mechanisms, but with modifications to accustom to the FCC working environment (details of which is introduced later). Due to the long link latency, the upstream node cannot get the real-time status from the downstream, thus not able to have immediate reactions on downstream status changes. Depending on different router architectures that we are going to cover later in this chapter, the "credit" here can represent different types of information from downstream node: input buffer's available flit space; each VC buffer's available slots; or downstream input buffer's packet slots.

CCAR requires two extra units. Firstly, since we only receive the credit information every few cycles, a downstream credit manage unit is needed to manage the credit value: if a packet is assigned to a particular output direction (or particular downstream VC), related credit value should be reduced. Another unit is the selection unit, which in charge of finding the direction that has the maximum "credit" value and performing arbitration when multiple directions have the same credit.

Just like RMR, when choosing potential output directions, it needs to follow the forbidden turn rules.

5.2.3 Switch Arbitration Policies

When the head (or single) flits from two or more packets arrive on the switch input and contesting for the same outgoing port, an arbitration mechanism is in need to resolve the contention. The winning packet gets a hold on the switch output and start forwarding it to downstream node, while the failed packets are being blocked in place until the current packet give up the occupation, either because the entire packet finishes traversing the switch, or the current packet is blocked elsewhere and let other packets use the switch before it can proceed again.

Inside all three proposed FCC routers, we provide support for three different switch arbitration policies. The arbitration policies are implemented in a way that minimizes the impact on overall router architecture. Thus, the selection of those policies also contributes to the design space expansion. In this subsection, we describe the arbitration working process as well as some implementation details.

Farthest First (FF)

Under this policy, when multiple packets contending for the same output, whichever packet has a largest Manhattan distance between source and destination wins the arbitration. Each packet can have two types of Manhattan distance, whether it is between the packet source and destination, or it is between the packet's current node and destination. We adopt the later one in (Sheng et al., 2018b). There are two ways to keep a record of the Manhattan distance. The first solution is to make use of the destination information carried inside the head flit and evaluate the distance in place. This solution is flexible but requires extra hardware. Another solution is allocating a dedication segment inside the head/single flit, which initialized as the distance between the source and destination node, which is referred to as priority value. When head flit left an intermediate node, the value is subtracted by one. The width of the

segment is determined by network size. This solution trade packet overhead for less resource consumption. When the network size is not so large, while FPGA on-chip resources are limited, adoption of the latter one is more reasonable. When multiple packets have the same priority value, which often happens when multiple packets share the same final destination, a round-robin mechanism is applied.

Oldest First (OF)

Oldest first means the packet with the oldest age has the highest priority. The packet age we use here is the inject timestamp, which is carried inside the head/single flit's priority segment. The implementation for this one is simpler. We compare the packet's priority value, whichever has the smallest value, indicates having the oldest age, wins the arbitration. Since FCC is designed to run for a long time before the reset, the injection timestamp needs to be infinitely wide, which incur large overhead in each packet. To address that, we assign a fixed bit width, P, for the priority value. When comparing two packets priority value, if one is significantly less than the other, then an extra 1-bit is affixed to the smaller one form a $P + 1$ bit value, while the larger one also extending to $P + 1$ bit but with Most Significant Bit (MSB) set as 0. Our experiment shows that $P = 15$ is enough for most cases. When multiple packets have the same age, round-robin mechanism is used as well.

FF Then OF (Mix)

This policy is a combination of FF and OF. The packets are arbitrated based on distance first. However, if one packet has an age value (current timestamp - priority age value) that is higher than a preset threshold, it wins arbitration immediately no matter the distance. If more than one packets have an age value large than the threshold, then the oldest packet wins. The overhead for this implementation is apparent: both the injection timestamp and Manhattan distance value need to be

carried inside the head/single flit. What is more important is that threshold value should be carefully set with regarding different patterns: if too large, the policy turns into FF; too small, turn into OF. Based on our experiment, a threshold value that is relatively high than the average packet end-to-end latency can potentially provide good performance.

5.2.4 Deadlock & Livelock Avoidance

Deadlock Avoidance

Deadlock happens when a sequence of packets are blocked forever in the network (Ni and McKinley, 1993). Deadlock usually starts with packets holding some resources on one node and requesting for more on other nodes. When multiple packets are having similar behavior while forming a dependency loop, deadlock is highly likely to occur. The deadlock has a huge impact on network in the sense that not only those packets can never reach its destination, but also all the occupying resources by those packets are virtually removed from the network, thus lead to further blocking one various other packets.

To avoid deadlocks, the most critical measure to take is to avoid the forming of dependency loops. In a 3D-torus topology, due to the existence of the wrap-around link, such cases can happen on two levels: on a single dimension, and across multiple dimensions. We take measures to address those on both levels.

On any given dimension, we make use of VCs and adopt the dateline technique as proposed in (Dally and Seitz, 1986). To break the resource dependency loop, we divide all the packets in the network and VCs into two groups: class 0 and class 1, as shown in Figure 5·4. Each packet is assigned a class type (either 0 or 1) when it is injected into the network. A packet with a specific class type can only request and use VCs with the same class type. To altogether avoid the dependency loops, packets within that loop must not be competing for the same set of resources. As a

result, a virtual **Dateline** is added on each torus loop, which are the links between Node $(N-1)$ and Node 0 (N equals to the number of nodes per dimension). After a packet traversed the dateline, no matter from Node 0 to Node $(N-1)$ or the other way around, the packet class flipped. Since our routing algorithms guaranteed that no packet could traverse the dateline more than once within each dimension (packets only move toward a predefined direction on each dimension), thus flits cycling on any given torus loops are necessarily contesting two different sets of VCs. Since we are targeting a bidirectional torus network, the initial packet class is different based on injection directions. If injected on a positive direction (X+, Y+, or Z+), the packet is set as 0, while on negative directions (X-, Y-, or Z-), the packet class is set as 1. Similarly, when packets' incoming and outgoing direction are on different dimensions, the packet class flips in the same manner: if turn to the positive direction, set as 0, and vice versa.

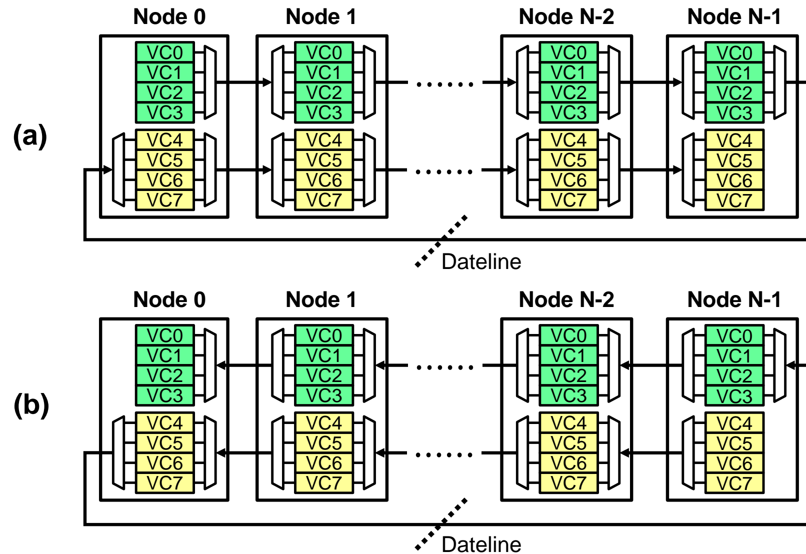

Figure 5·4: Dividing VCs into two classes and applying a dateline: (a) On positive link class type starts with 0; (b) On negative link class type starts with 1.

One may notice that in Figure 5·3, we allocate 3 bits for VC classes, while we only

have two classes to begin. For algorithms that not following a specific dimensional order, like RMR or CCAR, the packet may actively switch outgoing directions. If we use a single bit for the VC class, we may encounter unexpected class flipping. For example, when one packet just crossed the dateline on along the X+ direction and have a new class type as 1, but the outgoing direction is on Y+, which leads to class type flip back to 0 on the next node; later on when packet turned back to the X+ direction, the class type is now back 0, even though the packet has already crossed the dateline on X+ direction. Therefore, to overcome such cases and to accommodate for various routing algorithms, we allocate a single bit for each dimension and manage those bits individually depending on from which direction they arrive.

Another obvious problem of dividing VCs into two groups is the resource overhead. If we aim to achieve a similar level of performance, ideally we need to double the buffer space within each VC. However, in reality, when using minimal routing algorithms, only a small number of packets are crossing the dateline, and for those packets that crossed the dateline, they are close to reaching the dimensional destination or final destination after that. As a result, resource utilization is imbalanced among the two groups of VCs. In the care of this, we overlap the two VC classes as suggested in (Dally and Towles, 2004). If all the VCs are shared, then packets are still contending for the same set of resources. Thus, as illustrated in Figure 5·5(b), seven out of nine VCs on each input port is shared by both classes, while the first and last VC are exclusively used by class 0 and 1, respectively. The fundamental idea behind this solution is providing an escape route for every packet inside a cyclic channel dependency graph. Even though most VCs are shared, as long as the two exclusive VCs are there, packets could make use of that to break the resource lock when dependency loops forms and proceed to its final destination.

By classifying VCs into two classes, we address the issue of deadlock on each torus

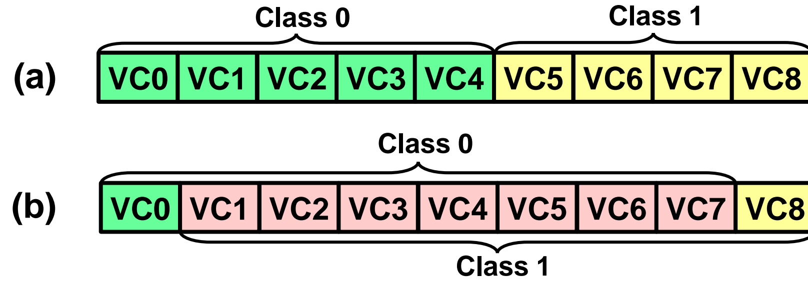

Figure 5·5: VC devided into two classes: (a) VC exclusively used for a certain class; (b) Overlapping VCs from both classes to rebalance buffer usage

loop. If all the packets follow the same dimensional order, then the method mentioned above is enough. However, algorithms like RMR, CCAR, etc. are actively switching routing dimensions. Since we are targeting a 3D-torus topology, dependency loops can quickly form across multiple dimensions. A simple solution is adapting the VC class idea, and further divide the VCs into more classes. With the extreme case being: each time a packet traverses an intermediate hop, the class number increase by 1, and can only enter VCs with the same class value. The implementation cost for this method is too high, especially when the network size is large. A better solution is to apply **Forbidden Turns** on the routing algorithm level: when packets entered a particular direction, it is not allowed to turn back to a subset of directions. Too many or too complicated forbidden turn rules reduce the routing flexibility and increase the computation cost. We find that the following six forbidden turns is sufficient for breaking all the dependency loops inside 3D-torus: Y- to X+, Y- to X-, Z- to X+, Z- to X-, Z- to Y+, Z- to Y- (shown in Figure 5·6). The first two turns are essential to stop packets that entering the Y- direction from turning back to the X direction, which avoids the deadlock in the XY plane. The later four make sure packets entering Z- direction cannot turn to any other directions, thus avoid the deadlock on XZ and YZ planes. This scheme defines that for any routing algorithms, Z- is always the last direction to go for all packets. The Z- last breaks all the loops covering multiple 2-D

planes because all the loops are covering multiple 2-D planes must have an edge on Z- direction.

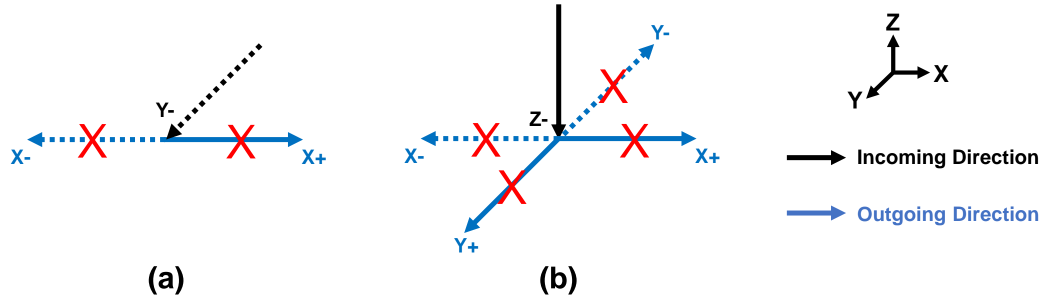

Figure 5·6: Forbidden Turns: (a) forbid Y- to X turn; (b) forbid Z- to X and Y turn.

Livelock Avoidance

Livelock occurs when a packet is not able to make progress in the network and not able to reach its destination. However, unlike deadlock, a livelocked packet continues to move through the network (Dally and Towles, 2004). Our design supports five different routing algorithms, four of which are minimal algorithms, which is inherently livelock free. For the non-minimal one, RLB, the packet routes are determined during packet generation phase. Once a packet is injected into the network, is follow a deterministic route and make progress towards the final destination on each intermediate hop. Therefore, RLB is also livelock free.

5.3 Proposed Router Architecture 1: VC-based Wormhole-Style Router

We briefly covered classic VC-based wormhole router in Section 5.1. However, as described in Section 4.5, properties of FCC and their use provide various constraints on the design space comparing with the traditional wormhole router use case. Modifications and optimizations have to be made on multiple components of traditional

wormhole router to accommodating the FCC working environment. In this section, we go over the modifications made on each function unit following the data flow order.

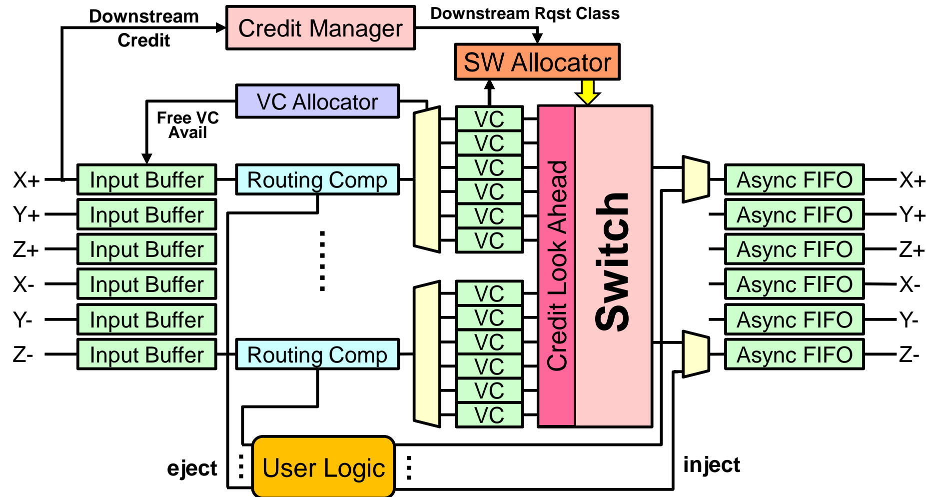

Figure 5·7: Proposed FCC VC-based Wormhole Router Architecture

5.3.1 General Work Flow

Figure 5·7 depicts our proposed router's general work flow: the packets enter *Input Buffer* upon arrival; whenever there is a *VC* available, *VC allocator* enables buffer read and pop a head flit out from a new packet; the read out head flit goes through the *Routing Computation (RC)* process to find the output direction; based on RC result, VC allocator assigns a free VC for the post RC head flit; if the packet has more than one flit, all the following flits must follow the head flit and enters the assigned VC one after another; after that, the *Switch* selects among valid outputs from VC and forward those selected packets to the designated output; if more than one packet is contesting for the same output port, an arbitration process is performed inside the switch; lastly, those packets that traversed switch are sent to a shallow output FIFO (only there to handle the potential meta-stability issue when crossing from the router clocking region to the MGT user clock region) and send out via the MGT controller immediately.

From the description above, we notice the datapath forks at two places: from RC to VC and from switch input to output. Selections are made, and datapaths are established after the head flit goes by. Since only the head flit carries routing information, the body flits must follow its previous flit and go through the established routes. If interrupted by other packets, the router lacks information to reestablish the previous route and lead to flit losses. Datapaths are freed and made available to other packets until the last flit from the previous packet is left.

Congestion is easy to take place under the "hold and wait" mechanism, especially when the packet size is large. To alleviate the potential congestion, we take different measures inside each router components, details of which are introduced in the subsections to follow.

5.3.2 Addition of Single Large Input Buffer

We perform a series of experiments on Novo-G#'s inter-FPGA link as described in Section 4.3.3. The loop-back test returns an average link delay of 176.8 ns. In conventional wormhole routers, since the link delay is usually within $1 \sim 2$ cycles, the upstream knows precisely how many available buffer (VC) space before sending the flits down, thus avoiding overflow. When set the running frequency as 300 MHz, the link delay is approximate 53 cycles, which means the turnaround delay between the two directly connected node is more than 100 cycles. Thus it is impractical for the upstream to control the number of flits to send deterministically. To accommodate for this, we implement a large input buffer on each one of the six input directions. The depth of the buffer is large enough to cover the flow control signal's turnaround time, plus a little overhead accommodating the flow control signal intervals. All the arrival flits are held inside the input buffer before entering the routing pipeline. The buffer status is reporting back to the upstream node periodically. Based on the downstream available buffer space, the upstream node controls the sending of flits.

Overcome Head-of-Line Blocking

In classic NoC design, buffers tend to implemented as First-In-First-Out (FIFO), through which the output follows the exact sequence as the input order. FIFO excels at simple control logic and maintaining the flit order. However, when we have more than one packets buffered inside a FIFO, it can lead to the Head-of-Line (HOL) blocking. For example, at a given cycle, the class 0 VCs are all occupied, while a remaining class 1 VC is still empty. However, the packets requesting class 1 VC are still held in the middle of the input buffer, blocked by the class 0 packet on top of FIFO. Since our router is relying on making use of the free class 1 VC to break the dependency loop, the HOL blocking inside input buffer is jeopardizing our deadlock avoidance mechanism. To address that issue, we come up with a smart buffer architecture (shown in Figure 5·8) aiming to address four design challenges listed below:

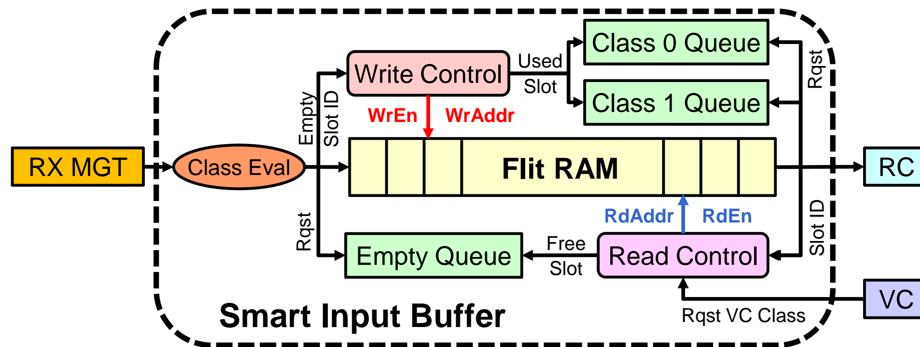

Figure 5·8: Smart Input Buffer

1. The input buffer need out-of-order read capability, which cannot be full-filled by FIFO. Thus, we use Random Access Memory (RAM) to hold received flits.

2. For ease of access flits from the same packets should be stored in adjacent memory addresses. We divide the RAM into chunks called “slots”. Each slot is enough to hold the largest possible packets.

3. The router needs to know which slots have valid packet without scanning the entire buffer (the timing cost of which is linear to the number of slots, which has a large impact on the routing performance). In the care of this, we implement two sets of "token" FIFOs, one for each class. Each token is pointing to the slot id and starting address of that slot within the RAM. A token is generated for each valid packet stored in the buffer and send to the related "Queue".

4. The buffer needs to know which slot is ready to take new data immediately since the received flit only exists on the read port for a single cycle. Our solution is to add another FIFO serving as 'Empty Queue". All the empty slots have a related empty token and enqueued into the empty queue during router reset stage. When new packets arrive, an empty token pops out and provide a write address to write controller.

The working flow for the smart buffer is as follows: When a head flit or single flit is received from receiving MGT's output, the smart buffer first evaluates its class based on the incoming direction. A token is then read out from empty queue and write address is send to write controller. In the meantime, a new token with the same slot id is generated and send to either class 0 or 1 queue based on the class evaluation result. For packets have more than one flit, the write address increments automatically until the tail flit is written into RAM. When there is VC available, VC allocator issues a read request to the buffer stating which class of packet it needs. The read controller requests a token from the related queue and extracts read address. If VC can take both classes, a round-robin mechanism is taken place as long as both queues are not empty. Similar to write address, the read address also increments automatically when packet size is significant than one. When a tail or single flit is sent out, an empty token is generated with current read slot id and enqueue into the empty queue.

5.3.3 Optimization on Routing Computation (RC)

Our proposed router design supports a variety of routing algorithms, listed in Section 5.2.2. RC returns the immediate outgoing direction in a single cycle. The implementation of those algorithms returns similar resource usage, but with different requirements at certain evaluation steps, which is listed below:

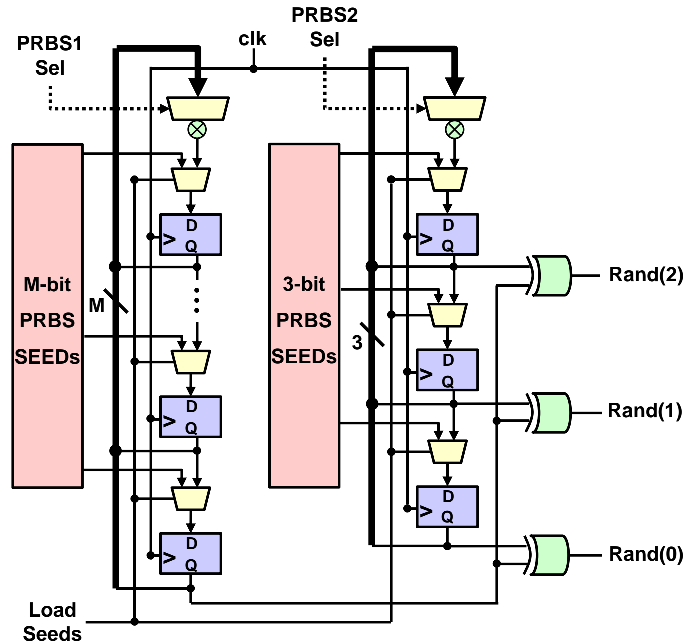

Figure 5·9: 3-bit pseudo-random binary sequence generator. The 3-bit PRBS's output is XOR'd with the M-bit PRBS's result to create a 3-bit output. Even though a single PRBS sequence can achieve similar results, when there are only 3 bits, the generated number is easy to get correlated with the number of cycles.

1. DOR is the most straight forward: given the current and destination coordinates, it provides immediate outgoing direction immediately.

2. O1TURN requires extra information: the selection among 6 possible dimensional orders. Since that information only needs to generate once in a packet's lifespan, we offload that function to user logic during packet generation.

3. Similarly, RLB need to select the initial output direction at each dimension, which is offloaded as well.

4. RMR requires a random number generator to select among all the possible directions that are complying with our deadlock avoidance mechanism. We implement a Pseudo-random Binary Sequence (PRBS) generator proposed in (Sheng et al., 2015a) with minimal resource usage (see Figure 5·9).

5. CCAR requires link usage status, which is the credit value from each one of the six immediate neighboring nodes (details of which is introduced in Section 5.3.8). In each cycle, RC needs to find the output direction with maximum credits as long as in line with the forbidden turn rules.

The routing computation only performed on head flit and single flit, while other types of flits (body and tail) bypass this unit and directly enter the designated VC set up by the head flit. The RC evaluated output direction, and class value replace the related segment inside head/single flit. As we designed in Section 5.2.4, head flit carries individual bits indicating class on each dimension and manage them separately. Thus RC unit on different input ports only updates a specific class bit: RC on X+ and X- only update the 1st class bit, with Y+ and Y- updating the 2nd bit and the remaining two updating the 3rd one.

5.3.4 Optimization on VCs

As discussed in Section 5.2.4, we divide the available VCs into two classes. Even though we can overlap those two classes, there should be at least a single VC that is exclusively used for each class. To achieve that, the minimal number of VCs on each input port should be 2. On one hand, the more VCs we multiplex on each input, the more routing flexibility there is; On the other hand, resource usage and arbitration

complexity scale faster than the VC numbers. The number of VCs serves as one design parameter that is further explored in the evaluation section.

Perform VC Allocation in Place

Given the most up-to-date downstream hop information, traditional NoC wormhole router performs VC allocation on the next hop before sending the packet down. The benefit for this implementation apparent: since the sender knows the exact downstream VC id, it can give away the link to other packets when congestion occurs. While upstream hop can still resume sending when the requested downstream VC is available again. If we stick with that workflow in FCC environment, the long link latency (50 cycles) means we need to wait for double that period before we can send the next flit to make sure we have the latest downstream VC usage information. Alternatively, a more reasonable solution is that, we directly send the packets to the downstream input buffer. When packets read out from the input buffer, the downstream router performs VC allocation for the current node. As a result, the upstream must send down the entire packet without interruption to maintain the correct flit sequence. The input buffer size should also enlarge to mitigate the congestion issue.

VC Buffer Entire Packet

Based on evaluated VC class information from the RC stage, the VC allocator finds the unoccupied VC with a matching class for each newly arrived packet. In NoC design, the VC buffer tends to be small (usually enough to hold a few numbers of flits) to preserve resource consumption. However, this turns out to be inefficient in FCC router design. As introduced before, flit paths within the router are established and held until the entire packet is transmitted. When the packet size is large, a small VC buffer can only hold a few flits from a packet, while the rest flits are still holding inside input buffer. Before the current VC is selected, it can take no more flits from

the input buffer. As a result, the flits on top of the input buffer is blocking all other packets since the input buffer only has a single output port. When that happens, mostly only a single VC is being used at all time. To address this blocking issue, we allocate enough VC buffer space that it is just large enough to hold the largest possible packet. In this way, even if a VC is not selected or blocked in the way, it still has enough space to hold the packet while letting other packets utilize other available VCs. Our router is designed to accommodate different applications. Ideally, the application could have arbitrary large packets, which requires infinite large buffer size. Thus, our router puts a cap on packet size. If a packet is larger than the limit, we have to break that packet into multiple smaller packets.

5.3.5 Optimization on User Logic with Direct Injection and Ejection Capability

Comparing Figure 5·1 and Figure 5·7, a noticeable difference is that classic wormhole router add extra input & output ports for local traffics apart from bypassing traffics. While in our design the injection and ejection packets avoid the switch: packets are directly ejected after RC; injection packets take the output link if there is no bypass traffic occupying it. As a result, we have six injection and ejection datapaths inside our router. This design has a few pros and cons. On the bright side, the highest possible offered injection throughput can achieve the full outgoing bandwidth of one node. Therefore, the nearest neighbor pattern can achieve "ideal" performance (ideal means 100% utilization of the link bandwidth without congestion). This design is especially beneficial for our model application: Molecular Dynamics (MD) simulation, where nearest neighbor communication is required on each node at the beginning of each simulation timestep. Another benefit is the reduced complexity in the switch because of the fewer in & out links.

Of course, this design has some overheads. Since we offered six injection direc-

tions, user logic now also responsible for selecting injection directions after the packets are assembled. The selection of injection direction must obey the deadlock avoidance rules stated in Section 5.2.4. Another task for user logic is to setup routing parameters for algorithms like O1TURN and RLB. O1TURN requires selection among six possible dimensional orders. RLB requires routing direction on each dimension. These parameters should also in line with deadlock avoidance rules and are needed before the injection direction is determined.

5.3.6 Optimization on Switch

Individual Input for Each VC

There are a variety of ways of designing connections between VCs and switch. Figure 5·10(a) gives a naive implementation, which directly connects all the VCs from all input ports to switch, with six outputs for each VC. On the switch output, a packet is selected among all the packets contending the same output direction. This implementation requires no arbitration inside the switch, but the resource cost is extremely high since the switch complexity increasing quadratically with the number of input & output paths. Figure 5·10(b) depicts an efficient solution to preserve resource usage for designs under tight budget: all the VCs multiplexed on the same physical channel are multiplexed again before they are connected to switch, while grant access to no more than one input for each output. Most NoC switches follow this method to optimize their resource usage. However, compactness is not achieved freely. First of all, this design requires two levels of arbitration: input VC arbitrate for access to switch and switch input arbitrate for output. Secondly, the VC that is granted for switch traversal may block other VCs that are multiplexed on the same physical channel. One example is shown in Figure 5·10(b). After routing computation, the output port of packet colored in blue is Y-, which is idle. However, it cannot advance because the VC of the red packet is the one that is granted for switch traversal, al-

though it is blocked at output port Z+. While in Figure 5·10(a), this issue is not a problem because all the VCs are independently routed. Since FCC only requires a single router per FPGA, more resources can be allocated for the switch. Thus, an intermediate solution is shown in Figure 5·10(c) is adopted: switch provides separate inputs for each VC, while multiplex outputs based on outgoing direction.

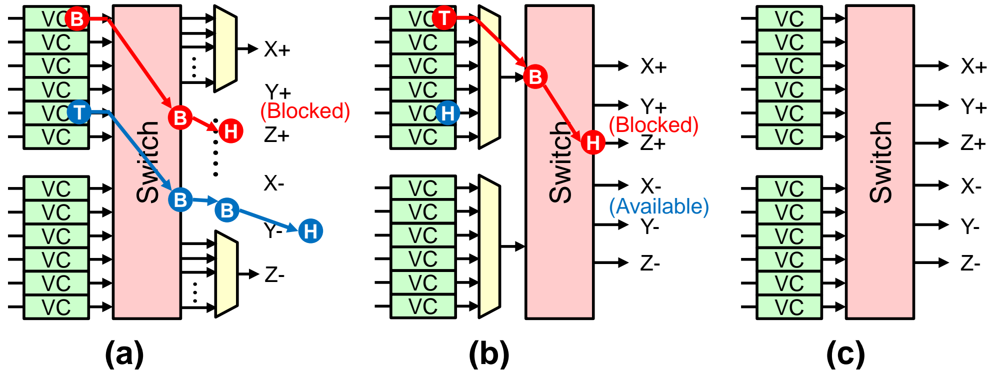

Figure 5·10: Implementation of connection between VCs and switch (assume M ports and N VCs per port): (a) fully connection: VCs direct connect to switch with individual switching for each VC (MN to M^2N); (b) fully mux: Mux VCs attached to the same physical link and grant no more than single input for each output port (M to M); (c) intermediate: VCs direction connect to switch while grant access to no more than a single input for each output (MN to M).

Reduction-Tree Based Switch Micro-Architecture

Now that we decide to provide individual input for each VC, and arbitration among packet contesting the same output direction, how shall we design an efficient switch that meets our design requirements? Assume we have M input ports ($M = 6$ in our case), and N VCs multiplexed on each input. Then we have a $M \times N$ to N switch. Conventionally, in NoC implementation, the switch is usually designed as a crossbar (Papamichael and Hoe, 2012; Huan and DeHon, 2012), or even a simple mux (Kapre and Gray, 2015) as those designs are targeting 2D topology with a smaller

number of input & output ports. If we adopt such kind of structure inside the FCC router, it causes two problems: complex arbitration logic and poor scalability. Our proposed arbitration policies require finding the packet with maximum priority value as described in Section 5.2.3. Since the switch has no less than 12 ($M = 6$, the minimal value of N is 2 as analyzed in VC implementation section) input ports, it is impossible to meet the timing requirements if we want to finish the arbitration process in a single cycle. To address this issue, we propose a reduction-tree-based switch. For each output port, a $M \times N$-to-1 switch is formed. Each reduction tree is built out of multiple levels of 2-to-1 or 3-to-1 muxes. Figure 5·11 shows an example of 6×3 switch. Based on the RC returned result, each VC with a valid head or single flit forwards that flit to the related tree input port that belongs to the request output port. Each reduction tree in this example has two levels of reductors: the first level has dual 3-to-1 reductors followed by a single 2-to-1 reductor. Inside each N-to-1 reductor, a shallow FIFO is in place to temporarily hold the flit for arbitration. When N inputs are competing for the output, only a single flit can grant access while the other flits stay in place of the FIFO. Once a head flit is granted, the current reductor is locked to that packet until the last flit passes. Each flit spends one clock cycle on each level reductor. Once freed, the reductor checks all the buffered head flits and make the selection once again.

We choose 3-to-1 and 2-to-1 reductors to form the reduction is to satisfy the post place & route Fmax requirements. In Table 5.1, we list the configuration of the reduction tree with regarding to different N value. The first column sets the number of VCs multiplexed on each input port; the second column denotes the switch size; the third column gives the types of reductors on each level (for instance, 3×2 indicates from input to output, the first level is built with 3-to-1 reductors, the second level is built out of 2-to-1 reductors); the number of reductors on each level is presented in

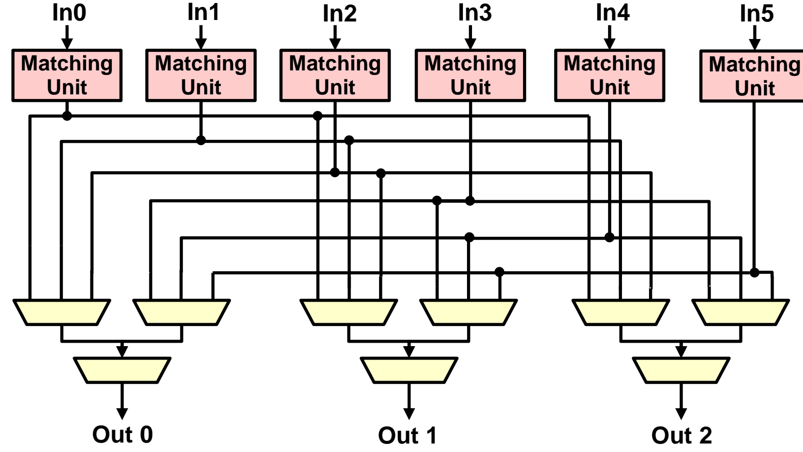

Figure 5·11: Reduction-Tree-Based Switch

the fourth column. In Table 5.2, we list the post place & route resource usage on both Stratix V and Stratix 10 FPGAs. We stop at 10 VCs per physical channel because we believe ∼20% of the complete chip resource is the upper limit for a router. Later our experiments show that performance gain stops when VC reaches a certain number. Since each flit spends a single cycle at each level of reduction tree, we can tell a flit takes about 2-5 cycles to traverse the switch, depending on the number of VCs per input port.

Table 5.1: Reduction tree configuration and resource usage with regarding to different number of VCs multiplexed on each input port.

VC# on Each Input	Switch Size	Tree Config	Reductors #
2	12×6	$3 \times 2 \times 2$	4:2:1
3	18×6	$3 \times 3 \times 2$	6:2:1
4	24×6	$3 \times 2 \times 2 \times 2$	8:4:2:1
5	30×6	$3 \times 3 \times 2 \times 2$	12:4:2:1
6	36×6	$3 \times 3 \times 2 \times 2$	12:4:2:1
7	42×6	$3 \times 3 \times 2 \times 2 \times 2$	16:8:4:2:1
8	48×6	$3 \times 3 \times 2 \times 2 \times 2$	16:8:4:2:1
9	54×6	$3 \times 3 \times 3 \times 2$	18:6:2:1

Class Look-Ahead (CLA) Unit

To avoid deadlock, we divide VCs into two classes with exclusive VCs to each class. It is critical always to make use of the free VCs, especially when congestion occurs,

Table 5.2: Switch resource usage on different FPGA chips.

VC# on Each Input	Stratix V		Stratix 10	
	ALM	Percentage	ALM	Percentage
2	11,220	4.3%	4,752	0.5%
3	17,340	6.6%	7,944	0.9%
4	23,460	8.9%	9,816	1.1%
5	35,700	13.6%	16,200	1.7%
6	35,700	13.6%	16,200	1.7%
7	47,940	18.3%	19,944	2.1%
8	47,940	18.3%	19,944	2.1%
9	54,060	20.6%	25,116	2.7%

thus break the dependency loops. As analyzed in Section 5.2.4, packets are less likely to cross the dateline. Therefore, most packets on a single torus loop are sharing the same class type. If upstream keeping sending packets down without knowing what is needed, the downstream input buffer can easily get flooded by packets with the same class, thus fail to use the free VC(s) dedicated for the opposite class. The solution to this is closely coupled with the credit-based flow control described in Section 5.3.8. Inside each credit flit, it carries the needed class type on downstream. When switch performs arbitration, it only selects among packets with needed class type. The pitfall here is the class type is not from the RC evaluated class on the current node, but from what is going to be when received on the downstream node. Thus an extra *Class Look-Ahead (CLA)* unit is needed to evaluate the class type on the next node. Given the output direction and current node's coordinate, the class on the next hop is determined. In the current implementation, we put the CLA unit on the input side of the switch. Every time a new head flit appears on the switch input side, the CLA unit is activated, and the evaluated result is saved in a register.

5.3.7 Addition of Asynchronous Output FIFO

We add a shallow asynchronous on the router output side for two reasons: Firstly, our router working in a different clock region than the MGT controller. The asynchronous FIFO bridges the clock domain difference and avoids the potential metastability is-

sue. Secondly, there is a mismatch between our 128-bit flit and 256-bit phit. We configure the FIFO with a 128-bit input width and a 256-bit output. In this way, the asynchronous FIFO helps to combine two flits into a single phit on the way.

5.3.8 Adaptation: Credit Flit Based Flow Control

When the receiving buffer on the downstream side is full, it needs to send a stop or pause signal back to the upstream node. Due to the limitation of vendor-provided MGT IP cores, which provides no functionality on passing backpressure, we need to implement our mechanism to generate the backpressure signal and notify the upstream node.

Credit-based flow control is a popular solution existed in NoC designs. Conventional credit-based flow control working as follows: every time downstream node consumes a flit from buffer (which means one more space available to accept a flit from upstream node), it generates a "credit" and send it back to the sender side; on the upstream node, it is not sending a flit down until a credit is received. This mechanism guarantees the buffer space for all the received flits at the cost of frequent credit traffic.

Now the question is: how shall the downstream node send the credit back? Dally and Towles provide a few solutions in (Dally and Towles, 2004). **Method 1:** a credit-only channel next to the data link, which is simple and has minimal impact on general traffic. However, it is very costly to add additional links (essentially doubles the link count) on cluster implementation. Plus, high-end FPGAs like Stratix Series from Intel (Altera, 2014c; Intel, 2018a) provides a limited number of off-chip I/O ports. As a result, it is impractical for a dedicated credit link in FCC environment. **Method 2:** credit "piggyback" on each data flit, with which each data flit allocates a specific number of bit for credit information. This method requires no modification on FCC hardware configuration but has high overhead within each data flit. **Method 3:**

multiplex data flits and credit information at the phit level. Each flit has a flit type section which indicates if the current flit is data or credit flit. The receiver node parses the flit and use the payload in each flit accordingly. When used in NoC or switching chips, the inter-hop latency is short, usually within a few cycles. To let the sender have the most up-to-date information on the receiver side, it requires credit flits to generate frequently. The overhead is significant. Under the worst-case scenario, half the link bandwidth is occupied by the credit flits.

Send Credit Flit with Intervals

Now look back at our FCC working environment. The long link latency makes it impossible for upstream node know the real-time downstream status (even with a dedicated credit port, the sender can only get the receiver information from 50 cycles ago). Moreover, we do not want to let credit flits occupy too much bandwidth. Therefore, in our design, we send one credit flit from downstream to upstream node every few cycles. The traditional wormhole router assigns a credit flit every time downstream router consumes a flit from the buffer. While in our case, we directly collect the unused buffer slot as the credit numbers.

The credit-based flow control in our design works as follows: when the countdown timer times out (the countdown value is referred to as *Credit Back Period (CBP)*), we pause the general traffic and collect the downstream buffer information. A credit flit is then assembled and sent on the upstream link. While on the upstream side, if it receives a credit value that is less than the sum of link latency, CBP, and a small threshold, it stops sending. In this way, all the on-the-fly flits are guaranteed to have space inside the downstream buffer. As for the value of CBP, that is a parameter we evaluate, which also serves as another dimension of our design space exploration.

Attach Needed Class in Credit Flit

As introduced in Section 5.2.4, we divide VCs into two different classes. To guarantee the VCs are working as intended in breaking the dependency loops, especially when the network has heavy traffic load, special care is already taken inside the input buffer by support out of order fetching. However, under the extreme circumstance, the input buffer only has packets with the same class given the fact that the upstream node does not know what type of packets downstream need. To address this, we insert an additional 1-bit flag for each class. The flag is asserted as long as one or more VCs belong to that class are available. On the upstream side, the switch arbitrates among either class packets if, and only if, the available flag for that class is set. Even though the upstream node has that information tens of cycles later, this can effectively avoid the cases that input buffer is flooded by packets from a single class which is not needed, while packets with needed class cannot send down since input buffer has no available space.

5.3.9 Summary on Current Design

We propose our first design in this section. Keeping the difference in hardware configuration in mind, we make modifications inside each class VC-based wormhole router components. The major changes we introduce is the addition of a single large input buffer and the resulting credit flit based flow control. This design provides a baseline solution for packet switching on FCCs. However, due to the limitation of link delay, and the existence of input buffer, the current design put a cap on maximum packet size, and lacks flexibility in dealing with congestion. To address that, we cover various modified designs on top of the baseline design in the next few sections to follow.

5.4 Proposed Router Architecture 2: Wormhole Router with Advance Flow Control

Following the classic VC-based wormhole router architecture, we modified each module to preserve the traditional wormhole routing workflow, the result of which is our first proposed architecture. Due to the long link latency, certain design philosophies work well for NoC wormhole router no longer fits the FCC environment. Thus in this section, we first analyze the drawbacks existed in the first design, then make further optimizations to address those.

5.4.1 Design 1 Drawbacks

High Resource Consumption on Storage

Our first design is featuring an input buffer at each input port to overcome the flow control delay due to long link latency. In classic VC-based wormhole router, the prerequisites for VC being able to alleviate the blocking issue shown in Figure 5·2 is that flits transmission can be intervened by flits from other packets while still being able to recover the previous route after blocking is removed. While in our case, the upstream nodes only have limited downstream information (buffer available space and needed class type), thus making it hard to make VC arbitration ahead of time. Since only the head flit has the routing information, in our first design, we force the consistency of the flits that belonging to the same packet to be maintained both on the physical link and inside the input buffer. To alleviate the potential blocking issue, we enlarge our input buffer and VC buffer that it can at least hold an entire packet. Thus remove the link occupation on its path even a packet is blocked at an intermediate node. The cost of which is the high resource consumption on on-chip storage.

Repeated Work on Class Evaluation

To forward packets with the needed class type, we add new segment inside credit flit indicating the needed class types. Since packets may change the class value after traverse the immediate next link, we also add an extra unit, CLA, inside the switch to deterministically calculate the class on the next node. The switch only forwards a packet if the packet's look ahead class type matching the downstream node requested type. While on the downstream node, the same class evaluation process is performed again inside the RC unit. Intuitively, we should only perform this evaluation once inside the RC unit. That is, inside the RC unit, we can directly calculate the class for the next node after the output direction is calculated, and override the class segment inside head flit with the class value on *next node*. However, this is not feasible since the post-RC head flit is used for VC allocation, which needs the class value on the *current node* to select from idle VCs with a matching class. One potential solution to address this problem is to add a new segment inside head flit, recording both the class value for the current node and the next node. However, this adds extra overhead to head and single flit.

5.4.2 True Wormhole Architecture

Our second design shares a similar architecture as our first one. However, in care of the drawbacks as mentioned above, we make changes on two levels: function unit rearrange and flow control mechanism optimization. The overall architecture for the second design is shown in Figure 5·12.

Removal of Input Buffer

Since the very existing of input buffer forces flits within the packet transmitted without interruption, we decide to eradicate the input buffer. To compensate for the long link latency and even longer flow control signals, we extend the VC buffer size inside

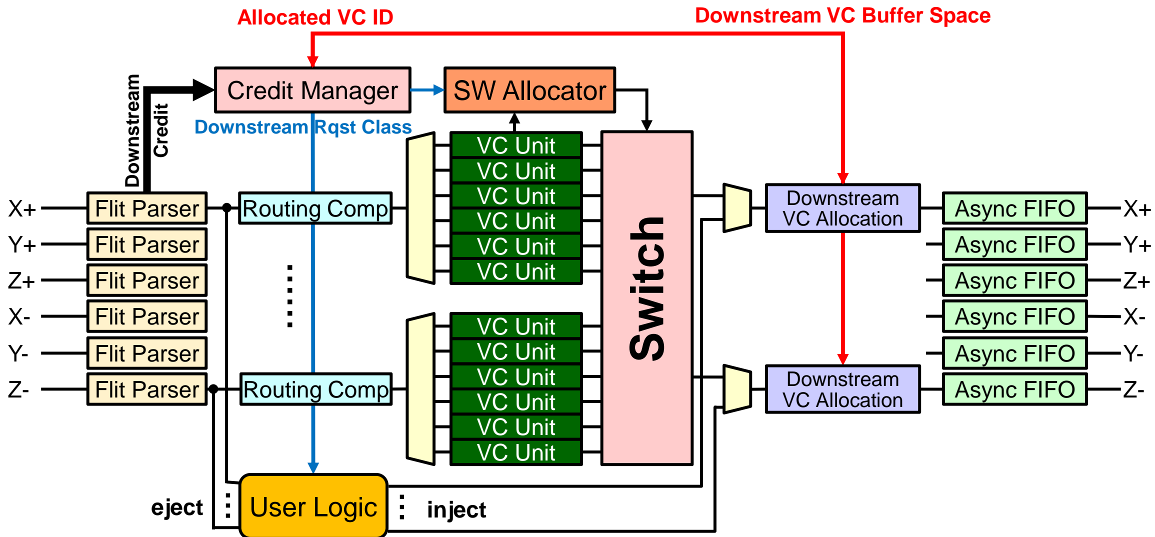

Figure 5·12: Proposed Wormhole Router with Advance Flow Control

each VC, acting like multiple individual input buffers.

Given the fact that our design supports as much as 9 VCs multiplexing on each physical link, this change seems expensive. However, the way Intel FPGAs configure and make use of its on-chip SRAM units provides an excellent opportunity for this change. Intel Stratix series FPGAs have 3 types of on-chip memory blocks: eSRAM, M20K, and MLAB (Intel, 2018b). eSRAMs have a large capacity and low latency; but also fixed data width, limited implementation flexibility, and only exists in high-end models. MLABs are built on top of the FPGA's logic element, the ALM, which is the resource bottleneck of our implementation and is only suitable for shallow memory arrays. M20Ks are small SRAM blocks (BRAM) distributed over the entire chip, feature both low latency and a large number of independent ports. The M20Ks are thus the ideal memory for our implementation of flit buffers. As the name indicates, each M20K SRAM block provides a theoretical capacity of 20K bits. However, it is hard to achieve 100% utilization on them due to the way it is organized inside. The users can configure the read & write datawidth as 10, 20, and 40 bits, and the resulting memory depth is 2048, 1024, and 512, respectively. In our case, since the flit size is

128, the minimal number of M20Ks we need to implement the flit buffer is 4 (when datawidth is set as 40), if not considering the buffer depth requirements. Under that configuration, 4 M20Ks are sharing the same address, while each manages different bit ranges (0-39, 40-79, 80-119, 120-128) of a single input and output. Since a single M20K block can only be used by a single memory module, thus a buffer depth of 2 and 512 consume the same number of on-chip resources. In this way, we can expand the VC buffer depth to 512 for free. Similarly, if more space is needed (within 1024), we can configure each M20K block as 20×1024 and use 7 BRAMs instead of 8. Based on our simulation, buffer size exceeding 1024 provides minimal performance gain.

VC Unit

Just like our first design, we still classify the new VC units into two groups with overlapping, as shown in Figure 5·5.

The significant components inside the VC unit are the flit buffer. Since we extend the buffer size, now each one can hold more than one packet. We can organize the VCs in two fashions. One way is to assign an output direction to each VC. Only packets with the matching output direction can enter that VC. In this way, we can implement the VC buffer as a FIFO, since no HOL blocking is taking place. The other way, we perform VC allocation on the upstream node, which means the upstream node need to predict for the output direction ahead of time, too. For routing algorithms following dimensional order, we can have 100% accurate prediction since the location of the intermediate node solely determines it. However, for an adaptive routing algorithm like CCAR, we do not have enough information to make that routing decision. Besides, for non-uniform patterns, this leads to buffer usage imbalance. A better solution is to treat each VC equally and always assign a new packet to the VC with the matching class type that has the maximum credits. To solve HOL blocking in this case, we adopt a similar RAM-based buffer architecture as described

in Section 5.3.2. With the difference being: the class we used here is the class value on *downstream node* provided by the current RC unit.

Credit Flit Carry Each VC's Buffer Status

In our first design, we use the number of available space inside the downstream input buffer as the credit value and attach it as payload inside credit flit. With the removal of the input buffer, the backpressure should be generated based on each VC buffer's available space. Similarly, we divide the VCs into two classes, with the first and last VC exclusively used for each class while the rest are sharing among both classes. Since different packets may enter different VCs, it is hard to use any specific VC's buffer usage as an indication of backpressure. The opportunity is that credit flit can carry 125-bit of payload each time (as shown in Figure 5·3(e)), which is wide enough to carry all buffer's available space.

Now that we have each VC's information, and each VC provides enough buffer space to cover the link delay, we can move the VC allocation ahead of time and perform it on the upstream side.

Advance Flow Control

Since we aim to perform VC allocation on the upstream side, the workflow should be altered. Following the flit moving order, we cover each module in this subsection.

When flit arrives on each input port, the *Flit Parser* first check if the flit is a credit flit. If so, the payload, which is the free buffer space inside each downstream VC buffer, is extracted and send to the *Credit Manager*. Inside which, the credit value is managed individually: the newly received credit value always replace the old value; and whenever a flit is sent down, based on the targeting VC id on downstream node, the matching credit value is decreased by one (since the router is not able to predict downstream VC buffer consumption, the upstream always assume downstream

VC has no consumption to prevent buffer overflow from happening). If receiving a head/single flit, then it is sent to the RC unit. The new RC unit still supports all five routing algorithms as described before. The class type is also evaluated inside RC; however, this time, the evaluation is based on the immediate output direction on the current node, which is primarily evaluating for the next node. Suppose the upstream has already performed VC allocation for the current node, a new segment is preserved in each flit (including body and tail flit) specifying the VC ID the packet should enter. In this way, the post RC flits can directly enter the designated VC buffer and wait for switch allocation. Credit manager provides available class type to switch allocator based on the locally managed downstream VC buffer space. After flits traverse the switch, they enter the last stage: downstream VC allocation. Our allocator aims to balance the buffer usage inside each VC. Thus when detecting a head or single flit on the switch output, or the injection port, it looks for the downstream VC with maximum credit value among the ones with matching class type. If credit values from all the VCs are below the threshold, then backpressure is generated to pause switch traversal and local injection, until the newly received credit flit reporting a larger than threshold credit value.

We perform downstream VC allocation as the last stage for the following reason: depending on VC buffer size and arbitration policy, the delayed time for each flit may vary. If we conduct switch allocation before entering VC buffer or before switch allocation, that arbitrary delay may lead to VC buffer usage imbalance. To make matters worse, our credit manager only deducts credit value when a flit is sent on the link. If flits targeting a particular VC are stuck for a long time, the allocator may keep assigning more packets to that one, with the worst case being buffer overflow.

5.4.3 Summary on Current Design

By enabling VC allocation on the upstream side, our router now can recover the connection between the upstream node and downstream VCs. When congestion occurs (the designated VC buffer is full), the new router release link occupation to other packets that are targeting other VCs with available space. Since each flit now carries the targeting VC id on the downstream, we can resume the transmission later when that VC has more free storage space.

More importantly, being able to give away routing resource occupation and resume later means this router no longer requires VC buffer to be large enough to hold the entire packet, thus enables support for huge packet size. Just like conventional wormhole router, where flits from the same packet are scattered on multiple the intermediate nodes, our new router can work in the same manner, but with longer “worms”.

5.5 Proposed Router Architecture 3: Virtual Cut-Through Style Router

We design our first two routers based on classic wormhole switching method. Wormhole switching consumes less resource while provides high routing efficiency when compared with other techniques. The cost of which lies in the complex flow control mechanism. When applied in NoC designs, the overhead can be overcome by adding extra wires between adjacent routers. However, this is not feasible in FCC configuration. We take multiple measures to achieve the complex flow control at the cost of on-chip storage resource consumption. In Section 2.1.3, we list a variety of other switching techniques. In this section, we look at one of the others, Virtual Cut-Through (VCT), and discuss the changes we need to make to meet FCC design constraints.

5.5.1 Virtual Cut-Through Background

VCT is another widely used switching technique, which only store packets at an intermediate node only if the next required channel is busy. As long as the downstream node has enough buffer space, the upstream can directly forward the first flit that arrived without waiting for the entire packet to arrive. Comparing with wormhole switching, VCT consumes buffer space since it needs each intermediate hop to provide enough buffer space for an arbitrarily sized packet. However, since we extend the buffer size inside our wormhole router, VCT is not necessarily taking much more. One key advantage brought by VCT is the more straightforward flow control mechanism: theoretically the upstream only need to know if the downstream buffer has enough space, then flits inside a packet can start forwarding. VCT has another advantage of immunization to link congestion. Whenever a packet is blocked on an intermediate node, it is buffered entirely in place, which free all the physical links on the path.

Given the above-stated features, we adopt the VCT switching mechanism and design FCC version of VCT router. Details of which are given in the subsections to follow.

5.5.2 Overall Architecture

The overall architecture of Design 3 is presented in Figure 5·13. The obvious change is the removal of all the VCs and replacing it with a new input buffer unit when compared with the previous two architectures. As a result, we no longer need detailed downstream information since no VC allocation is needed. Also, BRAM consumption is reduced. On the other hand, dependency loops are still forming in a 3D-torus network. We used to rely on VC to break the dependency loop. Since VC no longer exists in our design, we make further changes inside the input buffer unit; details are introduced blow.

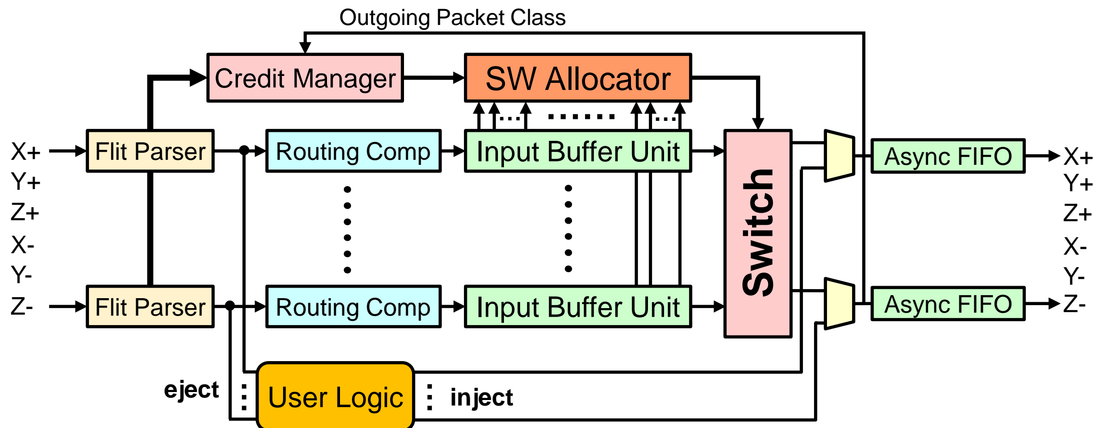

Figure 5·13: Proposed Virtual Cut-Through Router Architecture

5.5.3 Input Buffer Unit

Apart from holding the blocked packets, the input buffer unit also plays a key role in break dependency loops.

Similar to the input buffer implemented in our first design, we implement packet buffer as a RAM and divide the ram address into slots that are large enough to hold a single packet. Two two sets of FIFO queues, *Available Queues* and *Empty Queue*, are maintained for quick access to slot addresses (see Figure 5·14). The basic unit inside those queues is called *Token*. Each token carries a slot id, as well as the starting address and size of that slot. During the network initialization stage, all the tokens are enqueued inside the empty queue.

We keep the idea of dividing the buffer resources into two groups (class 0 and 1) and assigning each packet with a class type just as we do for the wormhole router. When a head flit is detected on the MGT output, we perform the class evaluation first. A token is then popped out from the empty queue, and the head and following flits from that packet are written inside the designated slot. A new token is generated based on the received packet's class type and output direction and written into the matching available queue.

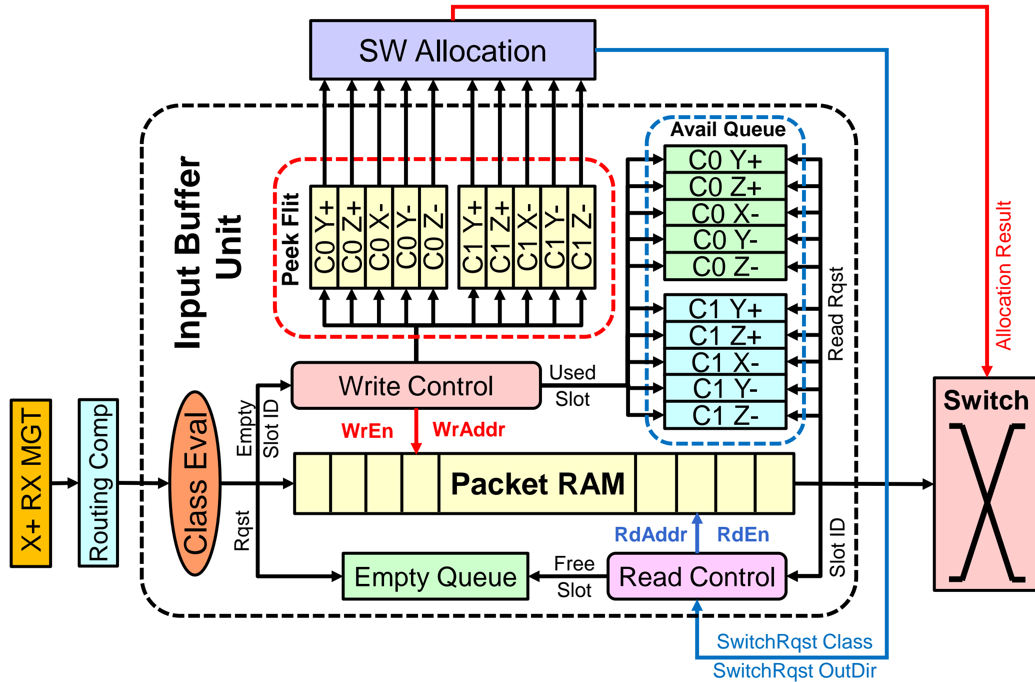

Figure 5·14: Design 3 Input Buffer Unit Architecture

Since our proposed router only has six such input buffer units, the HOL blocking is more likely to happen, especially when multiple input buffers' outputs are requesting the same output port. In the care of that, we increased the number of avail queue to 10 (there is no need to assign the queues for the incoming direction since we forbid bouncing back in our routing algorithm), with each one dedicated to a specific output direction and class type. We implement 10 register-based *Peek Flit*, with each holding a copy of the head flit from the packet on top of each available queues. The peek flit is directly connected to the *Switch Allocation* unit, based on which the switch arbitration is performed.

After the switch allocator notifies each input unit with the needed class and output direction, the matching available queue pops out a token and start reading out flits from the slot recording in that token. When the packet is sent out entirely, an empty token is generated with slot id and entered the empty queue.

To break dependency loops, we need to have specific resources that are exclusively

used by a particular class. If we assign a specific slot for each one of the two classes, it further complicates our buffer management. Instead, we keep monitoring the available queue usage for each class. If one class occupying several RAM slots that is larger than a preset threshold, it stops accepting any more packets with that class value. In this way, we successfully avoid the case when the entire buffer is occupied by a single class packet, while also providing flexibility to buffer read & write control. Our packet RAM should be at least large enough to hold two packets to guarantee slots for each class type.

5.5.4 Switch Allocation

As we have a single input buffer unit with a single output port on each physical link, this design only needs a simple 6×6 switch. In this case, we no longer need the reduction-tree style switch. Instead, a simple 5-to-1 mux (ruling out the incoming direction) is implemented for each output direction.

As we described before, the input unit provides a set of peeking flits to the switch allocation unit to let it selected among packets inside the buffer based on class value and output direction. For the worst case, the allocator need to compare the priority value among 10 possible packets (each input unit provides two packet header with different class type, multiply by 5 input directions). What is more, since each input unit can only provide a single flit per cycle, the allocator cannot grant each input unit to more than one output directions. Considering our forbidden turn rules makes Y- and Z- the last two directions to go, we give higher priority to the positive links and perform arbitration in dimensional order (X+, Y+, Z+, X-, Y-, Z-). If an input unit is already granted to an output, packet header from that direction is omitted for later output directions.

5.5.5 Flow Control

We stick with credit-based flow control as we have in the previous two designs. This time, the payload inside each credit flit is the available slot number for each class type inside packet RAM: by subtracting the sum of tokens inside the available queue for each class type from the total allowed token number for that class. On the upstream side, when receiving credit flit, the *Credit Manager* extracts the allowed slot number for each class and hold it in the designated register. Whenever a new packet is sending down, the packet class is returned to the credit manager and subtract one from the registered credit value. If the credit value on a downstream node is below a threshold, then the switch stops performing arbitration for that output, and injection is also paused.

5.5.6 Summary

This design has a few advantages comparing with previous designs. First, it reduced the resource consumption on BRAMs by reducing the number of the storage unit to one per port. Second, the flow control mechanism is simpler since it requires no VC allocation. As long as downstream have enough buffer space, a new packet can start forwarding. On the other hand, one major drawback is the limitation on packet sizes since the packet RAM need to allocate space for the maximum possible sizes. However, this can be alleviated by application-aware design practice: packet RAM size can be adjusted based on different applications, and user logic can always break large packets into smaller ones. Another drawback is the throughput bottleneck due to the limited number of buffers, the performance of which is further discussed in the evaluation section.

5.6 Router Performance Evaluation

In this section, we compare the routing performance among the three proposed router designs. We start by defining the performance metric. Then the resource utilization is reported. Later on, we present the per router design performance with regarding different workloads.

5.6.1 Routing Performance Metrics

We envision two types of communication loads: one with constant traffic in the network with each node process the incoming data at line rate; the other with periodical traffics where communication and data processing in each node are interleaved. We believe the performance metric should vary for those two types of workload, which we listed below.

Packet Delay Ratio and Network Saturation

For non-uniformed patterns, the distance between the source and destination node may vary a lot for different packets. Thus, it is impractical to set a fixed expectation on packet delivery time. Instead, in our simulator, we measure the *Delay Ratio* of each packet. As suggested by its name, it represents the ratio between the actual packet transmission time in real-life network and the ideal packet latency in the same network when that packet is the solely served one. Ideally, a delay ratio of 1 means 100% network efficiency. While in reality, due to congestion on the intermediate node and buffering delay, that ratio is usually larger than 1.

At a particular cycle, when the delay ratios for a certain amount of packets are above a preset threshold, we can say the network is entering *Saturation Status*. When that happens, the network performance degrades rather quickly. However, that is also the cases we care most: when network is under heavy workload, how are those different

designs can make a difference in delivering packets.

Continuous Mode

Under continuous mode, every node is injecting flits into the network at a given rate. Our injection mechanism differs from the NoC router in two ways. First of all, we enable direct injection from user logic to all six ports. Thus our router can inject at most six flits into the network per cycle. When measuring injection ratios (the average number of flits injected into network from each node), the range is between 0 and 6, while for NoC that range is 0∼1. Secondly, NoC usually inserts gaps between two flits' injection to alleviate network congestion. This method is feasible in NoC for two reasons: NoC only featuring small buffer size and the sender knows receivers buffer status. Things are quite different in FCC router. For our first design, we need to maintain the flits order inside the input buffer. On top of that, our buffer depth is much larger than NoC. Given those reasons, once the injection of a packet is started, the router keeps injecting a flit on the designated port each cycle until the entire packet is injected. To reduce congestion, we reduce the injection speed by inserting gaps between two adjacent packets' injection.

When working under the continuous mode, the critical parameter we observe is the *Network Throughput*. Since most of our routing algorithms are minimal routing, which means each time a flit moving on the link is making progress towards the destination. Thus we can collect the average throughput on each link during a certain period. The higher the average throughput is, the better network working status is in carrying packets from source to destination. To rule out the low throughput portion during the network "warm-up" process, we only measure the throughput value when the network enters the saturation status, thus demonstrating the capability of each routing algorithm and arbitration policy combination's capability in solving the link congestion when network load is heavy.

Batch Mode

Under the batch mode, each node initiates a set of packets that are targeting different destinations based on different patterns. Usually, there is only a single packet is needed between each source and destination pair. Many communication patterns derived from the Molecular Dynamics simulation falls into this category. Details of which are further discussed in the next section. When working in batch mode, the key performance parameter we care about is the "tail latency", referred to as *Batch Latency*, which is the time between the first node starting injection and the last flit left the network. For any given pattern with fixed packet size, the shorter the batch latency is, the better performance it provides.

5.6.2 Communication Patterns

Given the fact that we have a large design space to explore, in this section, we use different communication patterns, including both synthetic ones and real-life ones extracting from HPC applications, to test on our proposed routers with regarding different design parameters.

In Table 5.3, we list 7 communication patterns we used in evaluating our router performance. Among those, BC, Transpose, and Tornado are synthetic patterns that are widely used to evaluate NoC router performance. Another common characteristic shared by those is that each node only has a single destination. Thus we refer to those patterns as *simple patterns*. The rest patterns are all derived from real-life HPC applications. CUBE-NN and ALL derived from Molecular Dynamics (MD) simulation. CUBE-NN communication is needed twice for each Range-Limited force evaluation iteration: at the beginning of the evaluation, each node needs to broadcast the particle data to its neighbor nodes; after the force evaluation finishes on each node, it needs to send back the partial forces back to its 26 neighbors. ALL is needed

when performing Long Range potential evaluation (the core of which is 3D FFT) on multiple FPGA. NN and 3H-NN often used in stencil computation. Among those, the nearest neighbor pattern always has 100% link utilization because we enable direct injection onto 6 ports.

Table 5.3: Workloads to evaluate router performance. For tornado pattern, the YSIZE is the number of nodes on the Y dimension.

Workload Name	**Abbr.**	**Pattern**
Nearest Neighbor	NN	(x,y,z) → (x+1,y,z),(x-1,y,z),(x,y+1,z),(x,y-1,z),(x,y,z+1),(x,y,z-1)
3-Hop Diagonal Nearest Neighbor	3H-NN	(x,y,z) → (x+1,y+1,z+1),(x+1,y+1,z-1),(x+1,y-1,z+1),(x+1,y-1,z-1),(x-1,y+1,z+1),(x-1,y+1,z-1),(x-1,y-1,z+1),(x-1,y-1,z-1)
Cube Nearest Neighbor	CUBE-NN	(x,y,z) → 26 nodes with the range of [x-1,x+1][y-1,y+1][z-1][z+1]
Bit Complement	BC	(x,y,z) → (bitcomplement(x,y,z))
Transpose	TRAN	(x,y,z) → (z,x,y)
Tornado	TOR	(x,y,z) → (x,y+YSIZE/2-1,z)
All-to-all	ALL	(x,y,z) → all other nodes in the network

5.6.3 FPGA Resource Usage

We have implemented, tested, and verified our designs on a four FPGA system. Each FPGA card hosts an Altera Stratix V 5GSMD8 chip and supports MGT six links. Stratix V FPGA is the one we used inside Novo-G#.

In the future, we plan to make use of our current router design to serve the MD simulation accelerator on multi-FPGA systems. For which, we use a high-end Stratix 10 1SG280LU2F50E2VG FPGA with even more resource. Thus, we also evaluate the resource consumption of such FPGAs.

Among the 3 proposed router designs, resource usage of Design 1 and 2 is dominated by the number of VCs multiplexed on each physical link. We list their resource

usage with regarding to different number of VCs in Table 5.4 and Table 5.5, respectively. For both of the two designs, we notice that the resource usage almost growing linearly with the number of VCs, with Design 2 consumes about 10% more ALMs than Design 1 when having the same VC number. Thus, from a resource usage point of view, we prefer less number of VCs. On the other hand, the number of VCs have a direct impact on routing performance. Thus finding the right number of VC is the immediate next task we need to do.

Table 5.4: Design 1: Baseline Router Resource Usage on Stratix V & Stratix 10 FPGAs

VC#	Stratix V				Stratix 10			
	ALM	Percent	BRAM	Percent	ALM	Percent	BRAM	Percent
2	12,748	4.9%	138	5.4%	6,472	0.7%	138	1.2%
3	18,976	7.2%	162	6.3%	9,766	1.0%	162	1.4%
4	25,204	9.6%	186	7.3%	11,740	1.3%	186	1.6%
5	37,552	14.3%	210	8.2%	18,226	2.0%	210	1.8%
6	37,660	14.4%	234	9.1%	18,328	2.0%	234	2.0%
7	50,008	19.1%	258	10.1%	22,174	2.4%	258	2.2%
8	50,116	19.1%	282	11.0%	22,276	2.4%	282	2.4%
9	56,344	21.5%	306	11.9%	27,550	3.0%	306	2.6%

Table 5.5: Design 2: Advance Flow Control Router Resource Usage on Stratix V & Stratix 10 FPGAs

VC#	Stratix V				Stratix 10			
	ALM	Percent	BRAM	Percent	ALM	Percent	BRAM	Percent
2	13,318	5.1%	120	4.7%	7,918	0.8%	120	1.0%
3	20,206	7.7%	168	6.5%	12,226	1.3%	168	1.4%
4	27,094	10.3%	216	8.4%	15,214	1.6%	216	1.8%
5	40,102	15.3%	264	10.3%	22,714	2.4%	264	2.3%
6	40,870	15.6%	312	12.2%	23,830	2.6%	312	2.7%
7	53,878	20.5%	360	14.0%	28,690	3.1%	360	3.1%
8	54,646	20.8%	408	15.9%	29,806	3.2%	408	3.5%
9	61,534	23.5%	456	17.8%	36,094	3.9%	456	3.9%

Design 3 has no VCs. Thus the resource usage is fixed (shown in Table 5.6). Comparing with the other 2 designs, it consumes more BRAM, especially when the VC number is not large in the first two designs. ALM usage has a similar trend. However, when VC number grows larger than 5, Design 3 is the winner resource

usage wise.

Table 5.6: Design 3: VCT Router Resource Usage on Stratix V & Stratix 10 FPGAs

Stratix V				Stratix 10			
ALM	Percent	BRAM	Percent	ALM	Percent	BRAM	Percent
21,492	8.2%	336	13.1%	14,508	1.6%	336	2.9%

5.6.4 Experiment Setup

To cover as many combinations of design parameters, we use our proposed cycle-accurate simulation to evaluate the network performance (details is introduced in Chapter 7). The simulator is fully validated with respect to the four-node test fixture. To reflect our proposed routers' performance in the real-life use case, we set the simulation network size at $8 \times 8 \times 8$.

The targeting link bandwidth is set as 20 Gbps, with a 256-bit phit size, the MGT's recovered user input frequency is around 80 MHz. We set our flit size as 128-bit. Thus our router targeting frequency is 160 MHz, which meet our post place&router timing requirements on Stratix V FPGA. Based on our measured link performance in Chapter 4, we set our link delay as 175 ns, which equals to 28 cycles. The interval for credit flit generation is set as 9 cycles, with which the flow control signal occupies 10% of link usage.

5.6.5 Routing Performance under Different Workloads

For each router design, with different design parameters, there supposed to be 15 different performance numbers with each one matching each combination of routing algorithm (see Section 5.2.2) and arbitration policy (see Section 5.2.3). From Section 5.6.3, we know that the router resource usage is determined by two design factors: router design, and VC numbers. The methodology we used in finding the best routing performance is as follows: for each pattern, for each one of the three designs, we pick

the best case performance among the 15 results with regarding different VC numbers. Since the router resource usage is proportional to VC numbers, our goal here is to find the optimal VC number that can provide best-case performance.

We organize our experiments on three levels:

- Working mode: we measure batch latency under the batch mode, and throughput under continuous mode with regarding different injection ratios.
- Two groups of patterns: simple patterns with a single destination node, including BC, TRAN, and TOR; complex patterns with multiple destination nodes, including 3H-NN, CUBE-NN and ALL (The NN pattern always has 100% efficiency. Thus we do not include that pattern in the evaluation).
- Different packet size: each pattern has two different packet sizes: for the simple pattern, the packet sizes are 0.5 kb and 8 kb; for the complex pattern, the packet sizes are 0.25 kb and 2 kb.

Batch Mode

Figure 5·15 and 5·16 gives VC number's impact on batch latency performance. For each pattern, under each design combination of router architecture and pattern size, we select the best case performance among the 15 combinations of algorithm and policy. We also use two different sizes for each pattern, which is listed above. Since Design 3 has no VC inside, thus the performance remains constant with regarding VC numbers.

Based on our observation, we found a few interesting points in general. First of all, routing performance is not scale linearly with the VC numbers. For most patterns under test, after reaching a certain VC number, the routing performance stops improving. Also, that optimal VC number is less than 9 for all designs. Secondly, Design 3 seems to have better performance in general. This phenomenon is easy to understand since those designs provide large enough buffer space. Even under heavy

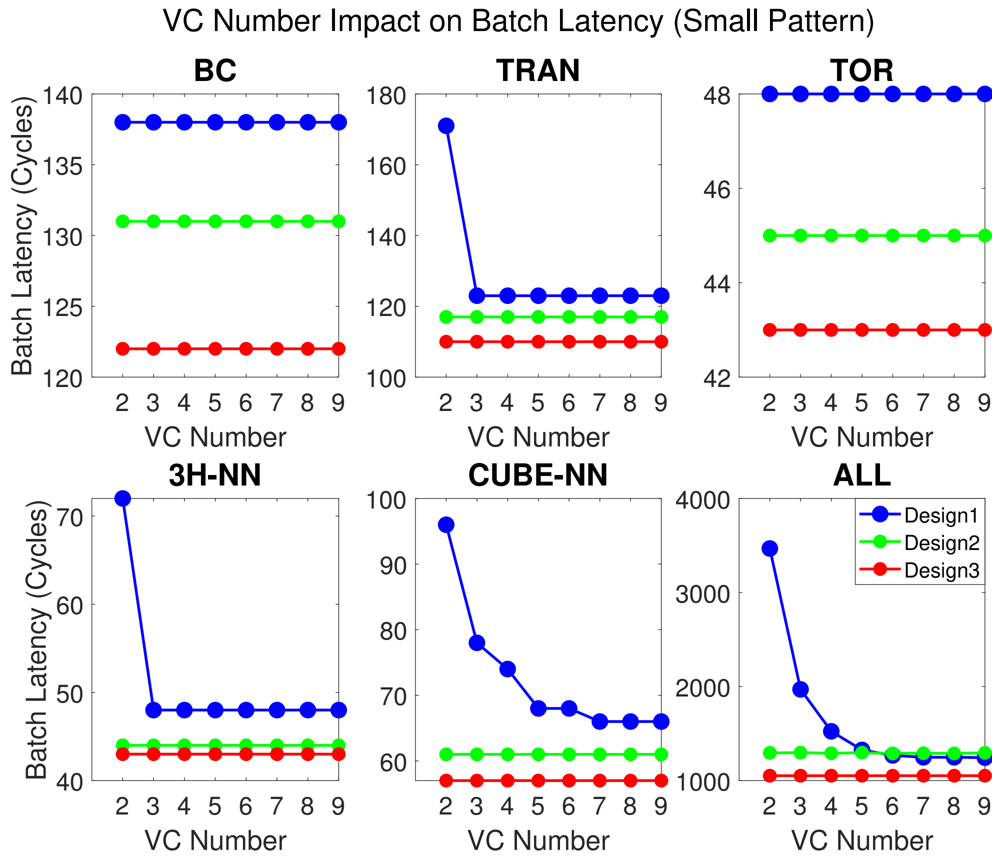

Figure 5·15: VC number impact on batch latency when packet size is small. The lower the value is, the better.

workload, Design 3 provides enough space to buffer the entire packet in place, which relieves the link from being occupied by blocked packets. What is more, the network load (packet size) has an impact on the optimal VC number selection. Usually, the heavier the network load, the more VC is needed to achieve better performance. This is especially obvious with those simple patterns like BC, TRAN, and TOR. Last but not least, Design 1's performance is sensitive to the VC number, with the reason being Design 1 has tiny VC buffer. When packet size is large, one packet tends to occupy a single VC for an extended period. Thus the more VCs multiplexed on each link, more routing flexibility there can be. In comparison, Design 2 is less sensitive

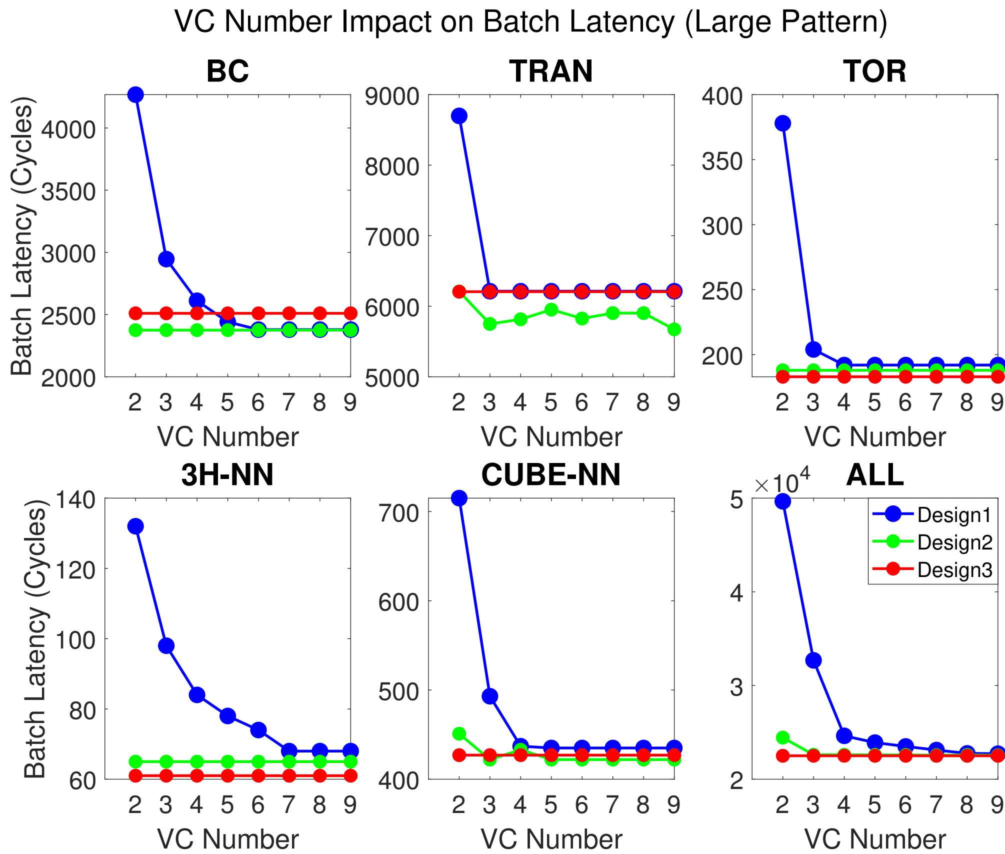

Figure 5·16: VC number impact on batch latency when packet size is large. The lower the value is, the better.

to the VC number due to the large buffer size.

Now let's have a closer look in selecting optimal VC numbers for each pattern. For BC, when the packet size is small, the VC number makes no difference since the workload in the network is not heavy enough to saturate the buffers. However, when the workload increases, we start to see differences in Design 1, with an optimal VC number of 6. TRAN is interesting. When the packet size is small, Design 3 has better performance. However, when packet size grows larger, Design 2 is in the leading position. This change is caused by Design 3's limitation on buffer throughput. When multiple packets targeting different output port enters the same input buffer, even

though there are free ports available, they are still blocked by the currently selected packet, because there is only a single read port on the RAM buffer. The larger the packet is, the longer the blocking time is. The optimal VC number for TRAN should be 2. TOR is the most straightforward pattern in our experiment, which only utilizes the Y link, specifically the Y+ link when using a minimum routing algorithm. Since the pattern is too simple, even with large packet size, the latency is not influenced by much from the VC number. Even categorized among the complex patterns, 3H-NN is relatively simple, which only need to send packets to the 8 nearest corner nodes. When packet size is large, there are lots of packets contending for the same link at the same time. When the VC buffer size is not large enough (Design 1), the performance is benefited by large VC numbers. However, when buffer size is large (Design 2), the link contention is hidden due to the lack of packets to saturate the buffer. Both CUBE-NN and ALL featuring a mix of destinations, ranging from nearest neighbor to nodes from multiple hops away, while ALL having even more destinations and even further away nodes. With that many packets inside (or to be injected into) the network, intuitively, they should require for more VCs to alleviate the link contention. For Design 1, this is true. However, for Design 2, the performance is counter-intuitive. It has little influence from the VC numbers. What's more, even Design 3 have larger buffers, Design 2's performance still provides close (if not better) performance when the packet is large for both patterns. Similar to TRAN, Design 3's performance is limited by the blocking issue inside the input buffer, and this is especially obvious when the packet size is large.

Continuous Mode

We perform a similar set of experiments under the continuous mode, but this time we use large packet size while tweaking the injection rate and measure the best case performance for throughput with regarding different VC numbers. The results are

shown in Figure 5·17 and 5·18.

In general, the throughput tells a different story in locating the optimal VC numbers. On to top of what we find in the batch mode, we few more findings are the followings. First of all, even though Design 3 provides overall better batch latency performance, in continuous mode, Design 2 tends to have match performance. This result confirms our previous analysis that Design 3 suffers from the buffer blocking issue due to the single output port. Secondly, Design 1 is still the one that influenced by VC numbers most. Thirdly, under the fast injection rate, more VCs generally returns better performance. Lastly, even with large packet size, if the injection ratio is low, the throughput performance is less impacted by the VC number.

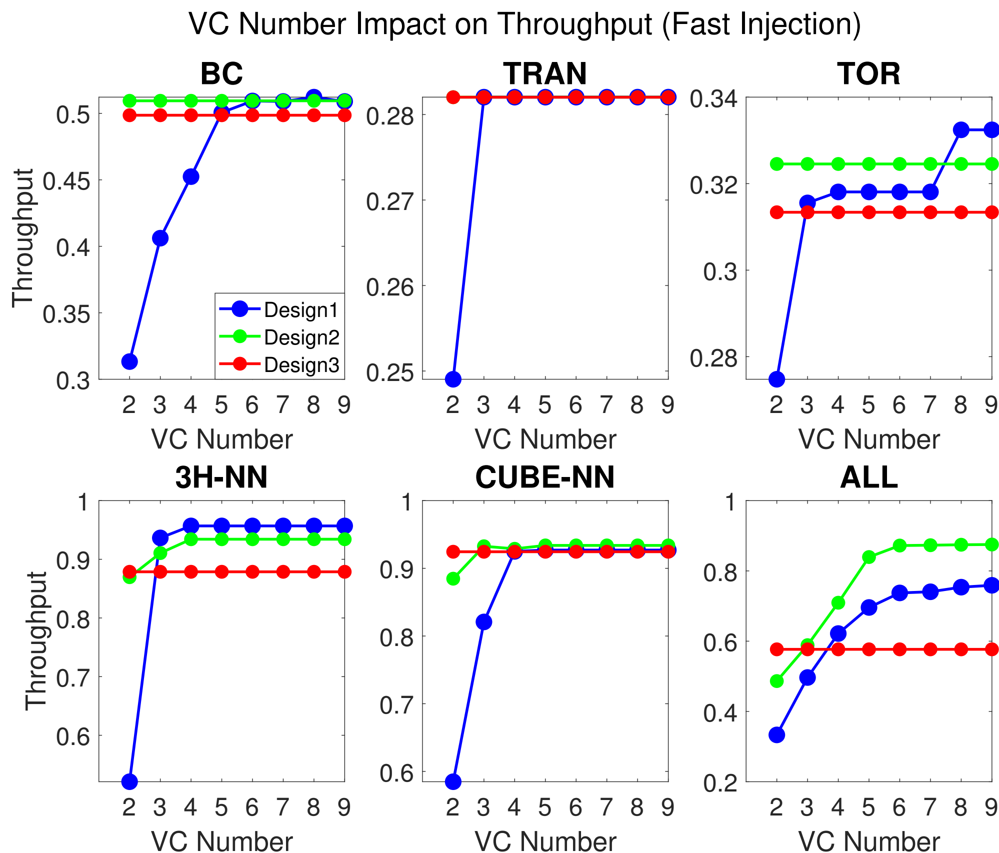

Figure 5·17: VC number impact on throughput when packet injection speed is fast. The high the value is, the better.

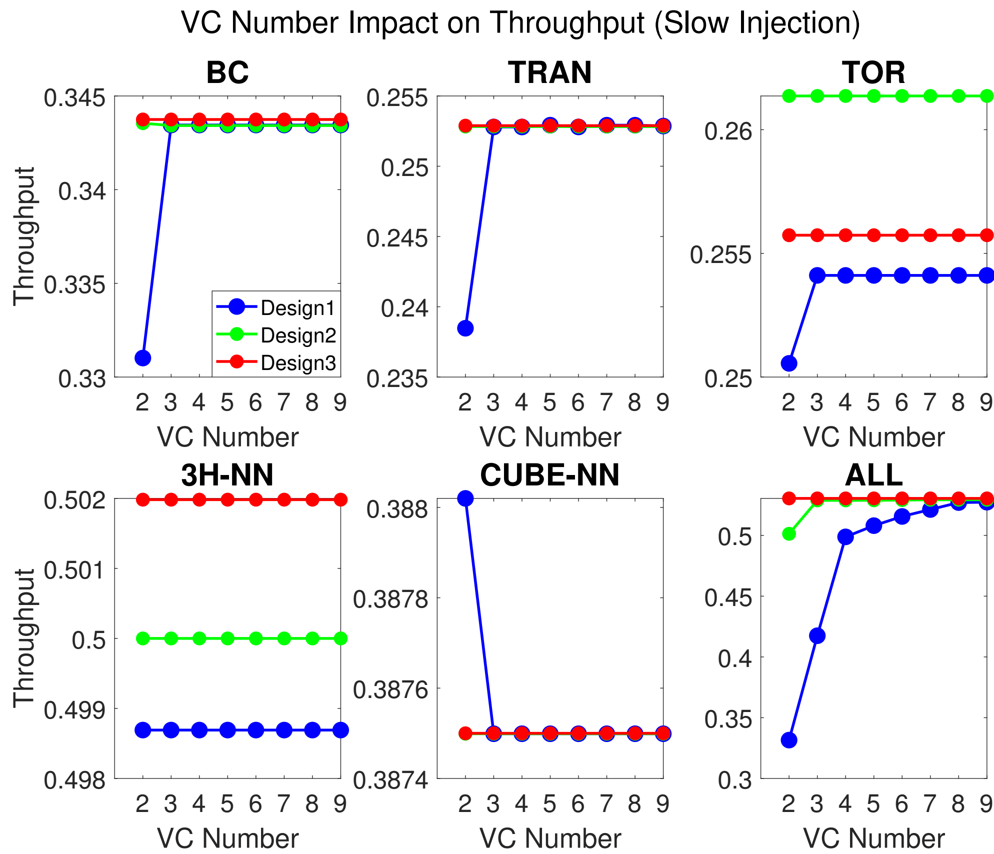

Figure 5·18: VC number impact on throughput when packet injection speed is slow. The high the value is, the better.

The per pattern analysis is presented here. Since at lower injection ratio, the VC number has limited influence on performance, we focus on the high injection ratio case here. For BC and TRAN, the throughput performance of Design 2 and Design 3 routers are very close, with BC needs more VCs (6) to achieve the same level of performance when adopting Design 1. For TOR, Design 2 is not affected by VC number while providing slightly better performance. Design 1 can provide even better performance when the VC number is 8. Looking at 3H-NN and CUBE-NN, we find that the VC number start to affecting Design 2's performance with a VC number of 4 being the best. For ALL, Design 1 and 3 have apparent benefits over Design 2.

Plus, for such intricate patterns, we start to notice the VC numbers impact on Design 2, with 6 VCs per port giving the best performance.

Optimal VC Number

More general conclusions on VC number selections:

- In general, more VCs means better performance. However, performance does not grow linearly with VC numbers. For each pattern and design, there is a cap of VC number. Above which, there is very limited perform gain by increasing VC number.
- There is no single optimal VC number that can fit all the needs. Depending on pattern type, packet size, injection ratio, and working mode, the optimal VC number can range from 2 to 8.
- Design 1 is more sensitive to VC numbers; when VC buffer size grows in Design 2, it mitigates the benefits of having more VCs.
- The pattern size has a direct impact on VC number selections;
- Depending on the working mode, the optimal router architecture can vary, with Design 2 returns better batch latency performance, while Design 2 has better throughput result.

A general rule of thumb in selecting VC number is: when the pattern is small and straightforward, few numbers of VCs (usually less than 4) is sufficient; when the pattern is large and complex, with a fast injection ratio, a large VC number is preferred (in the neighborhood of 7).

5.7 Summary

In this chapter, we first introduce the conventional VC-based wormhole router architecture. Then we present five routing algorithms and three switch arbitration

policies supported in the router design. We examine three router designs specifically optimized for FCCs. These designs feature different resource usage and performance under different work loads. Those router designs, along with the internal design parameters and choice of routing algorithm and switch arbitration policy, combine to create a large design space. We find that for different patterns and workloads, the optimal router design can vary. We explore this further in Chapter 7.

Chapter 6

FPGA-Centric Cluster Router, Part 2: Dynamic Router Design for Collective Traffic

In previous chapters we have constructed a router design space with three different router architectures, various combinations of routing algorithms and arbitration policies, as well as design parameters such as number of VCs and buffer depth. So far, we have optimized the proposed designs for point-to-point workloads. In this chapter, we further extend our design space by introducing application-specific optimizations for our model application, MD simulation. In particular, we extend design and evaluation to collectives including multicast and reduction.

6.1 Background

Collective operations can be implemented either with unicast or multicast (see Figure 6·1). Figures 6·1 (a) and (c) show multicast and reduction using unicasts. All the packets are unicast but share the same source (or destination). Figure 6·1 (b) and (d) show multicast and reduction based on a tree topology; the communication burden is drastically reduced.

Tree-based collective routing algorithms have been much studied recently. They took one of the two approaches in their design: either by having an offline generated multicast/reduction table (Jerger et al., 2008; Abad et al., 2009; Wang et al.,

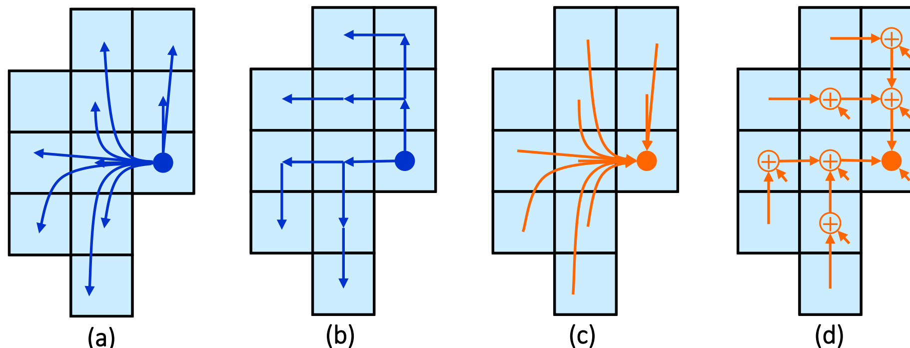

Figure 6·1: Collective operations: (a) Unicast-based multicast; (b) Tree-based multicast; (c) Unicast-based reduction; (d) Tree-based reduction

2009; Sheng et al., 2016a), or by inserting multicast information inside the packet header (Wang et al., 2009). The first design requires generating the table prior and load to on-chip ram before runtime; while the second design added extra overhead inside the packet header, especially when the broadcast path is long. Most of the previous works targeting a 2D mesh topology that are inherently deadlock-free. Besides, those designs working in a multicast/reduction only network, without considering the impact from general non-collective traffics.

In our design, we propose an architecture that having run-time support for both multicast and reduction operations, which completely removes the need for loading an off-line generated table, while minimizing the overhead introduced in packet headers. We design our multicast/reduction unit in a way that it has minimal impact on the normal non-collective traffics with minimal resource overhead. To avoidance introducing deadlock possibilities, our multicast/reduction algorithm follows the forbidden turn rules introduced in Section 5.2.4

6.2 Motivation

Our model application, MD simulation, drives the support for collectives. As we introduced in Chapter 4, the Range-Limited (RL) force evaluation requires two phases

of near-neighbor communications. First of all, prior to force evaluation, each node needs to broadcast the particle position data that is held on that node to its neighboring nodes. Depending on the cell to node mapping, the broadcasting range can vary a lot (details of which is discussed in Section 3.2.5), which requires multiple different sets of tables if implemented in a table lookup manner. The second phase happens after the RL force evaluation on every node. Since each node now has partial force value belongs to particles that are "borrowed" from neighboring cells on different nodes, it needs to send back. Luckily for us, those partial forces accumulate to the calculated value on its home node. We, therefore, can perform the computation in the network, by having partial forces that targeting to the same particle added up on those intermediate nodes, thus reducing the number of packets in the second phase.

For reduction packets that contain the partial force for the same particle, they may arrive at different cycles. Thus, we need to buffer the temporal payload until all the needed packets have arrived. Keeping that in mind, since we already have large buffers in Design 3, the broadcast and multicast modules are implemented on top of that.

6.3 Multicast Support

In our current implementation, we limit the multicast pattern as a cube with the source node sitting in the center, which is a perfect mapping to MD multicast pattern. To support different sizes of multicast, we introduce the multicast radius, R, which denotes the absolute maximum distance between source and destination in each dimension. For example, R=1 results in a $3 \times 3 \times 3$ cube. The radius, as well as the source node location, is carried in the multicast packet's head flit used to locating its virtual location in the multicast tree on each node. To keep the multicast packet's routing path in line with our forbidden turns, we grant priority to X directions by

slicing the cube into $2R+1$ YZ planes, as shown in Figure 6·2.

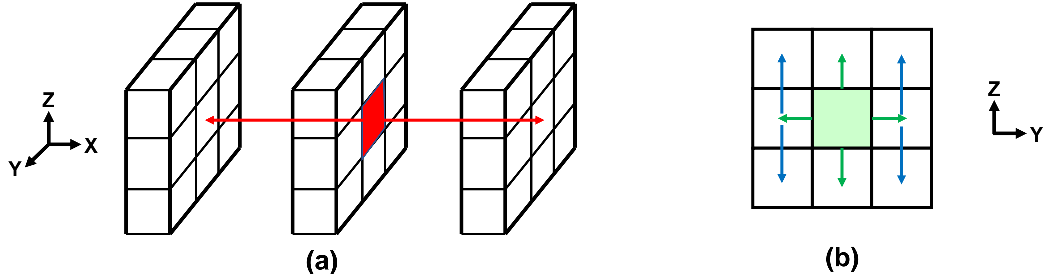

Figure 6·2: Multicast packet path: (a) slicing the cube into multiple YZ planes; (b) inside each YZ plane, grant priority to Y direction, then turn to Z at last

Algorithm 2 Multicast Output Evaluation Algorithm

```
if cur_y = src_y and cur_z = src_z then
    if cur_x = src_x then
        return X+, X-, Y+, Y-, Z+, Z-;
    else if abs(cur_x - src_x) < R and cur_x > src_x then
        return X+, Y+, Y-, Z+, Z-;
    else if abs(cur_x - src_x) < R and cur_x < src_x then
        return X-, Y+, Y-, Z+, Z-;
    else
        return Y+, Y-, Z+, Z-;
else if cur_z = src_z then
    if abs(cur_y - src_y) < R and cur_y > src_y then
        return Y+, Z+, Z-;
    else if abs(cur_y - src_y) < R and cur_y < src_y then
        return Y-, Z+, Z-;
    else
        return Z+, Z-;
else
    if abs(cur_z - src_z) < R and cur_z > src_z then
        return Z+;
    else if abs(cur_z - src_z) < R and cur_z < src_z then
        return Z-;
    else
        return None;
```

When detecting a multicast packet, it first sends on the ejection port. Then it goes through our multicast unit inside which output evaluation is performed based on

Algorithm 2. The evaluated output direction is kept inside the packet header (6 extra bits for each output direction). We put our multicast unit before the input buffer, as shown in Figure 6·3. When the incoming packet has multiple output directions, we only keep a single copy inside the input buffer (thus avoid packet ram writing contention when we need to write multiple words into a RAM in a single cycle). We add an extra avail queue designated for the multicast packets on top of the 10 existing queues shown in Figure 5·14. Each time the packet is finished sending on an output port, the related output request bit in the header is cleared. Instead of popping a token out of the avail queue immediately after sending operation starts on that packet, we keep the token on top of the available queue unless the current pointed packet is sent on all the requesting output ports.

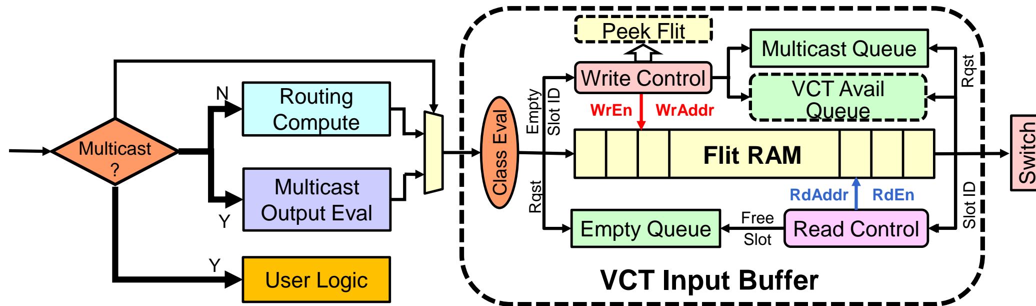

Figure 6·3: Architecture of Multicast Unit

6.4 Reduction Support

Similar to our multicast algorithm, our reduction algorithm also divide the cube into slices. However, this time, we slice along the Z dimension while still grant priority to X directions 6·4. The reduction unit is placed at a similar location as of the multicast unit. Unlike multicast unit, we can no longer share the input buffer since we need frequent access to the buffered data for accumulation.

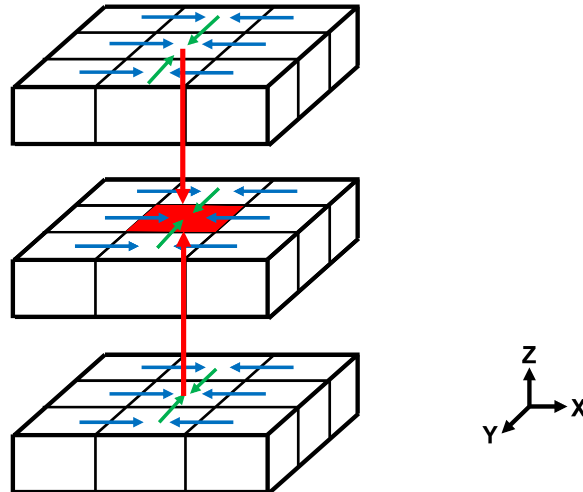

Figure 6·4: Reduction packet path: slicing the cube into multiple XY planes; inside each XY plane, grant priority to X direction, then turn to Y, with Z being the last direction.

Algorithm 3 Reduction Waiting List Generation

```
if cur_x ≠ dst_x then
    if cur_x > dst_x then
        return Wait for X+;
    else if cur_x < dst_x then
        return Wait for X-;
    else
        return Wait for None;
else if cur_y ≠ dst_y then
    if cur_y > dst_y then
        return Wait for X+, X-, Y+;
    else if cur_y < dst_y then
        return Wait for X+, X-, Y-;
    else
        return Wait for X+, X-;
else if cur_z ≠ dst_z then
    if cur_z > dst_z then
        return Wait for X+, X-, Y+, Y-, Z+;
    else if cur_z < dst_z then
        return Wait for X+, X-, Y+, Y-, Z-;
    else
        return Wait for X+, X-, Y+, Y-;
else
    return Wait for X+, X-, Y+, Y-, Z+, Z-;
```

The reduction unit is shown in Figure 6·5. During the initialization stage, a reduction table is generated online based on Algorithm 3. For each potential reduction destination, the algorithm returns the input direction from which it needs to collect first before generating a new packet and forward to the next hop that destination. The destination ID organizes the storage unit inside the reduction unit. We use the same radius, R, as multicast. For a reduction radius of $R = 1$, reduction packet injected from each node could have 27 possible destinations (including itself). Based on destination nodes' relative location to the current node, those destinations are numbered from 0 to 26 following Equations (6.1) to (6.4). An address space is pre-allocated for each destination. When a reduction packet is received, the destination id is evaluated, and the memory address is generated. The payload of which is accumulated to the existing value and written back to the same address.

$$dx = dst_x - cur_x + R \tag{6.1}$$

$$dy = dst_y - cur_y + R \tag{6.2}$$

$$dz = dst_z - cur_z + R \tag{6.3}$$

$$\mathbf{dst_id} = (2R+1)^2 \times dz + (2R+1) \times dy + dx \tag{6.4}$$

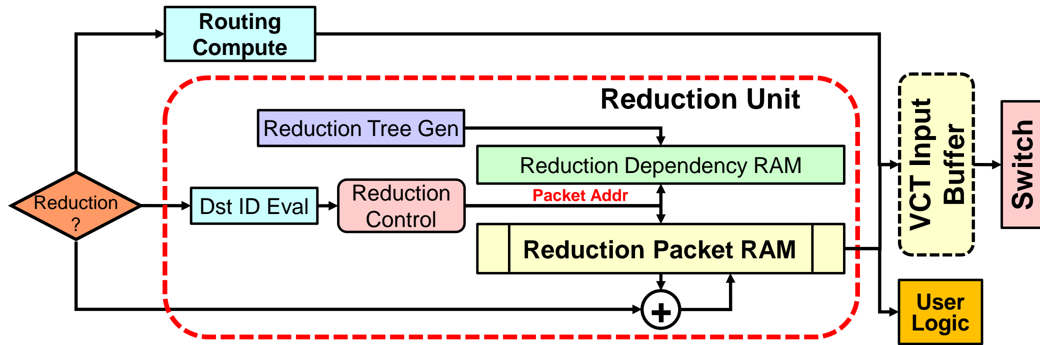

Figure 6·5: Architecture of Reduction Unit

6.5 Performance Evaluation

6.5.1 Resource Usage

We design our application-aware support with on top of the virtual cut-through router. Our multicast and reduction support has minimal impact on the existing designs. Thus the resource overhead is minimal (see Table 6.1).

Table 6.1: Application-Specific Support: Resource Usage on Stratix V & Stratix 10 FPGAs

Function	Stratix V				Stratix 10			
	ALM	%	BRAM	%	ALM	%	BRAM	%
Multicast	22.080	8.4%	348	13.6%	14,958	1.6%	348	3.0%
Reduction	23,250	8.9%	384	15.0%	15,588	1.7%	384	3.3%

6.5.2 Collective Performance

Since our support for collectives aims for reducing the number of packets transmitted inside the network, our performance evaluation on those parts are focusing on the batch mode. Similar to the previous experiments, we envision four cases: small pattern with slow or fast injection, and large pattern with slow or fast injection. The performance is shown in Figure 6·6

The pattern we use is the CUBE-NN for both multicast and reduction experiments. The small pattern size is 0.5 kb, while the large pattern is 16 kb. For the multicast support, we find that it provides better performance over the unicast-based method. The batch latency improvements are more obvious when the workload is heavy, with the best improvements being 75%.

However, on the reduction side, the improvements are not so much. When the workload is light (small pattern with slow injection), the reduction unit provides 10% improvements on batch latency. However, for other cases, the latency performance is worse comparing with unicast-based solutions. The main reason for this unpleasant

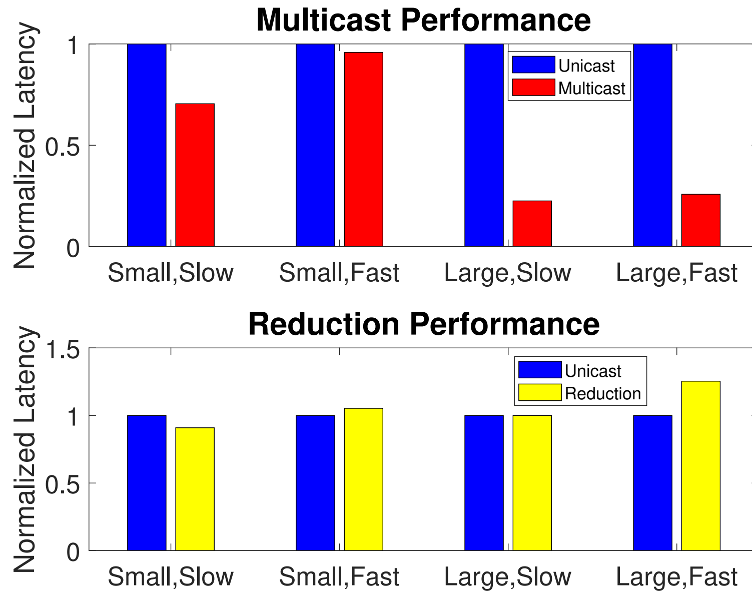

Figure 6·6: Collective Performance

result comes from the fact that the reduction is performed in a store-and-forward manner. For each reduction packets, we need to wait for all the partner packets to arrive and perform summation before generating the new packet. In this way, even packets number in the network are reduced, many cycles spend on waiting for other packets to arrive. As a result, when the packet size is large, the waiting time is longer, hence our experiment results.

6.6 Summary

In this chapter, we introduce a dynamic router architecture optimized for collective workloads based on the proposed virtual cut-through style router (Design 3). We introduce online multicast and reduction algorithms that have minimal impact on general non-collective traffic by obeying the same set of deadlock avoidance rules. Our evaluation result shows that our multicast support provides 5∼78% improvements on batch latency when compared with unicast-based solutions.

Chapter 7

FPGA-Centric Cluster Router, Part 3: Application-Aware Framework

In Chapter 5 we introduced a baseline FCC router design and three variations: Baseline wormhole router (Design 1), Advanced Flow Control (AFC) wormhole router (Design 2), a Virtual Cut-Through (VCT) router (Design 3). We also presented various features that are likely to be beneficial in application-aware support. Inside each design, there are various parameters that can have a direct impact on routing performance. On the algorithm level, there are routing algorithms arbitration policies. On the architecture level, there are VC number and buffer depth. For a given workload, it is unreasonable for users to determine which configuration is optimal for their workloads.

One way to get an estimation on router performance is by performing HDL simulation. However, even for a network sizes as small as $4 \times 4 \times 4$, the HDL simulation could take hours to finish. We thus propose a framework that, given a communication pattern and its evaluation metric, estimates the performance for all possible configurations. Moreover, this framework generates the HDL design for that router configuration. This framework has good extensibility with respect to new algorithms and arbitration policies.

7.1 Application-Aware Framework

7.1.1 Software Based Cycle Accurate Simulator

Due to the existence of congestion and randomize routing decisions, it is impossible for a theoretical model to accurately presents the working status at each cycle inside the router. While the HDL simulation can provide detailed and accurate router status cycle by cycle, the simulation time required for a large scale network is unacceptable, not to say users need to test on various combinations of parameters. To provide fast and accurate network performance with different parameters, we propose a C++ based cycle-accurate simulator.

Every module inside our RTL code is implemented as a class in our simulator. Those classes are organized in the same hierarchical order as our HDL implementation. Each class has a standardized input and output variables, as shown in Figure 7·1. Since our router is working in a pipeline manner, each class receives input data from its upstream module, performing the evaluation, and send to the downstream. Following that manner, we can classify those variables into two sets: the upstream interface and downstream interface, each with three variables. For upstream interface, the first one is *data_in*, which is the pointer referencing the upstream's output. The second one is *data_in_latch*, which dereference that pointer. The third one is a boolean value, *in_avail*, which is generated inside the current module and notifying the upstream module if the current module is ready to take a new set of data in the next cycle. On the downstream interfacing side, first, we have the *data_out* which holds the data to transfer to the downstream side. Then there is the pointer, *out_avail*, which referencing downstream's in_avail value. Similarly, *out_avail_latch* is there to dereference the previous pointer each cycle. If the value inside out_avail_latch is "false", the upstream node holds the value in place until that value turns back to "true".

To mimic the clock cycle, we put the entire simulator inside a *while* loop. Each

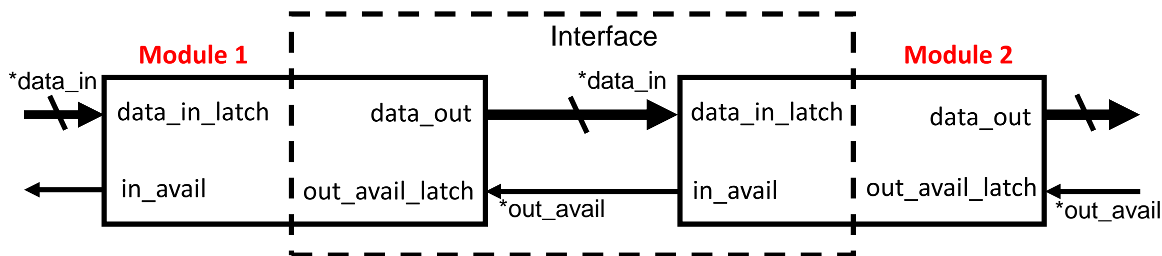

Figure 7·1: Interface Between two Classes inside Cycle-Accurate Simulator

loop iteration is recorded as one cycle. We adopt the "producer-consumer" model introduced in (Sheng, 2017), with each module act as both a producer and consumer. At the beginning of the simulation, there is an *Initialization* phase, which points those pointers to the right variable address inside both upstream and downstream modules, followed by calling initialization function on those submodules. Every cycle is starting with the consume phase, during which, the simulator fetches the value pointed by *data_in* and *out_avail*, and copy the value into those two latches accordingly. Then any modules that are directly under the current one is also performing the consume task as described above. Following the consume phase is the produce. During which, the simulator first calls the produce function of each submodule, then execute the evaluation function on the current module. In the end, the simulator updates the output ports. After the end of produce phase on the top module, the cycle counter increments by one and move on to the next cycle.

With the help of cycle-accurate simulator, we can exhaustively simulate all the possible router configurations, and find out the one with the best performance.

7.1.2 HDL Code Generation

Our HDL coder generator has two parts: the HDL code itself, along with a software script used to generate the HDL code. Based on the simulation results, the users can select among designs with optimal performance numbers. The parameters including

router architecture, VC numbers, buffer depth, and whether they require specific supports like collective.

We have three sets of base HDL code, matching our three router architectures. Parameters like buffer depth and VC numbers can easily be applied by changing the parameters on the top module of the HDL code. While other architectural changes, for example, changing the fan-in of the switch (need to change the reduction-tree configuration), need to use the provided software script. Inside the software script, we pre-stored eight different configurations for the reduction-tree, as listed in Table 5.1. Based on user input, it generates an HDL code formed by six such reduction-trees. Similarly, we have a script for the RC unit, which is able to generate the first different types of RC units for those five routing algorithms we support.

With the help of the software simulator and HDL code generator, our application-aware framework helps end-users find the design suits their application needs and generate the HDL code that is ready to run on FPGAs.

7.2 Evaluation of Application-Aware Selection Benefits

In this section, our goal is to find the "Power of Reconfigurability". We design three different sets of experiments: first of all, we show that the optimal routing algorithm and arbitration policy can vary with regarding to different patterns and workload; secondly, we defined three scenarios with different router resource budget, with which, a "global optimal" router design that can provide overall best performance across all patterns is located; thirdly, we find the per pattern best router design, and compare with the global optimal one, to determine the benefits of our application-aware framework.

7.2.1 Searching for Optimal Combinations of Routing Algorithm and Arbitration Policy

From the results reported in Chapter 5, we find that optimal VC number for different patterns is different, thus, before we start our search for best routing algorithm and arbitration policy, we designate a VC number for each pattern that potentially provides the best results for all types of workload in Table 7.1.

Table 7.1: VC number assigned for different patterns

Pattern Name	VC Number
BC	6
Tran	3
Tor	3
3H-NN	7
CUBE-NN	5
ALL	6

Optimal Design for Batch Latency

Using the selected VC numbers listed in Table 7.1, we list the batch latency performance for all 15 combinations of routing algorithm and arbitration policy inside each one of the three proposed design, with regarding to different patterns and pattern sizes in Figure 7·2 and 7·3.

In general, we notice that the RLB routing algorithm always returns the worst latency performance. Even for TOR, of which RLB is specifically designed for, it still has the worst performance. The long link latency kills the performance of any non-minimal routing algorithms. Also, we notice that the optimal combinations for each pattern tend to be the same across all three router designs. Thirdly, different arbitration policies seem to have less impact on latency performance than routing algorithms.

As shown in Figure 7·2, we notice that when network load is small, batch latency difference among the 12 minimal-routing combinations is not much, especially for

simple patterns like BC, TRAN, and TOR. For BC, DOR and O1TURN all return similar levels of performance, with CCAR also being optimal for Design 2 and 3 as long as the arbitration policy is not OF. For TRAN, the optimal algorithm on all three routers is O1TURN, with best arbitration policy being either FF or Mix. TOR is simple, as long as the routing algorithm is not RLB, the performance is the same. For 3H-NN, all combinations have the same performance. For CUBE-NN, RMR, CCAR, and O1TURN share the same performance on Design 1. But on Design 2 and 3, the optimal algorithm is O1TURN. Finally, for the most complex pattern, ALL, those combinations start to show the differences. With CCAR-FF being the best on Design 1, O1TURN-FF on Design 2, and O1TURN on Design 3.

When pattern is large, the selection is different (see Figure 7·3). For BC, DOR with FF is the best. In case of TRAN, DOR and RMR are better on Design 1, but CCAR with FF is the best on Design 2, while RMR and O1TURN with OF the best on Design 3. TOR is the same with every combination have the same performance. 3H-NN is better with RMR, CCAR, and O1TURN on both Design 1 and 3, with CCAR has slightly worse performance on Design 2. CUBE-NN benefits most from RMR and CCAR with CCAR-OF being best on all router architectures. As of ALL, O1TURN seems the best choice.

Optimal Design for Throughput

Figure 7·4 and 7·5 lists the detailed throughput performance. Similar to batch mode, those combinations only starts to show its difference when the workload is heavy. Thus we are focusing on analyzing the best combination under the fast injection case.

As shown in Figure 7·5, for BC, there is a significant variance in throughput with different combinations, with DOR-OF being the best for all three router architecture. For TRAN and TOR, RLB provides better throughput, which is quite different from what we get under the batch mode. For 3H-NN, RMR, CCAR, and O1TURN all

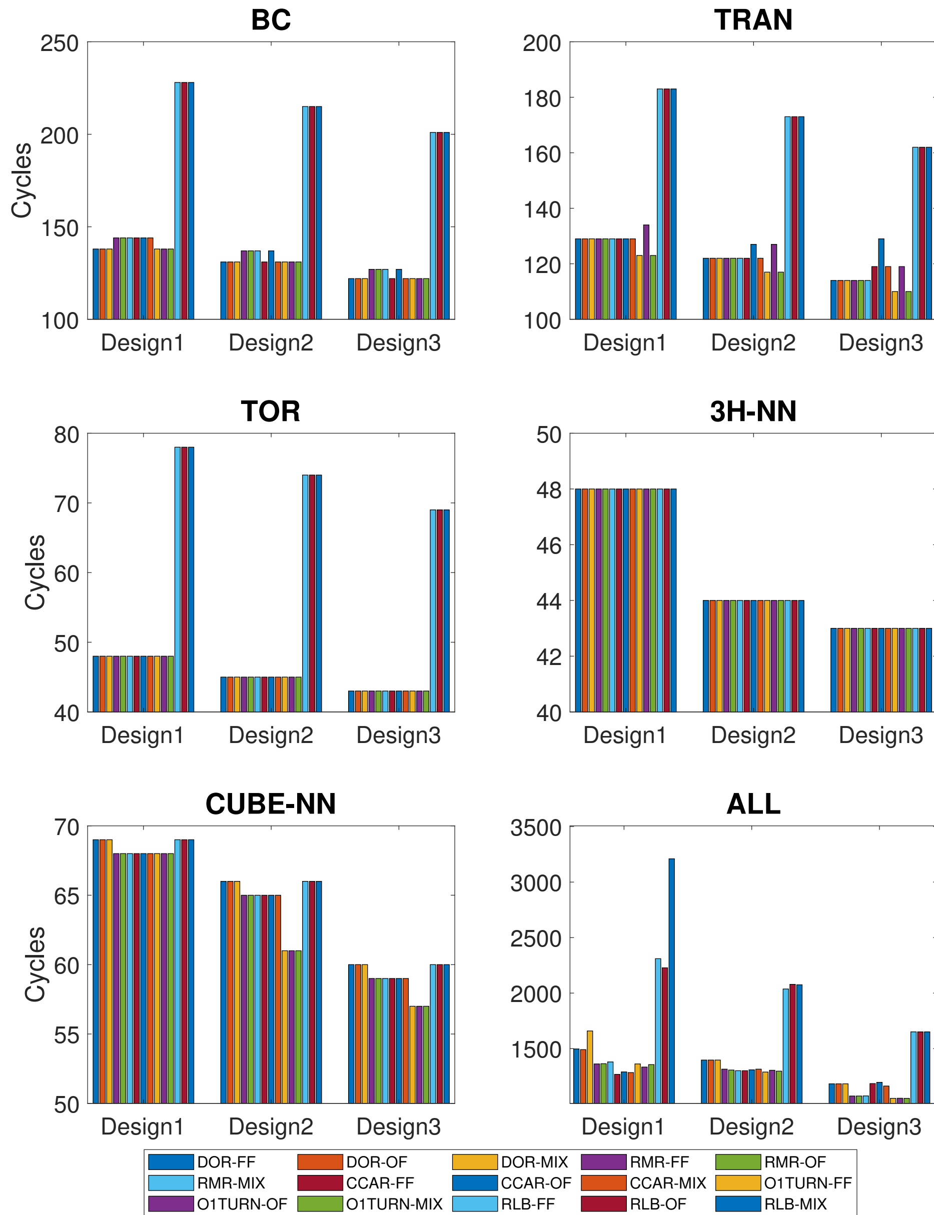

Figure 7·2: Batch lantency performance with regarding to 15 combinations of routing algorithm and arbitration policy (small pattern). The lower the value is, the better.

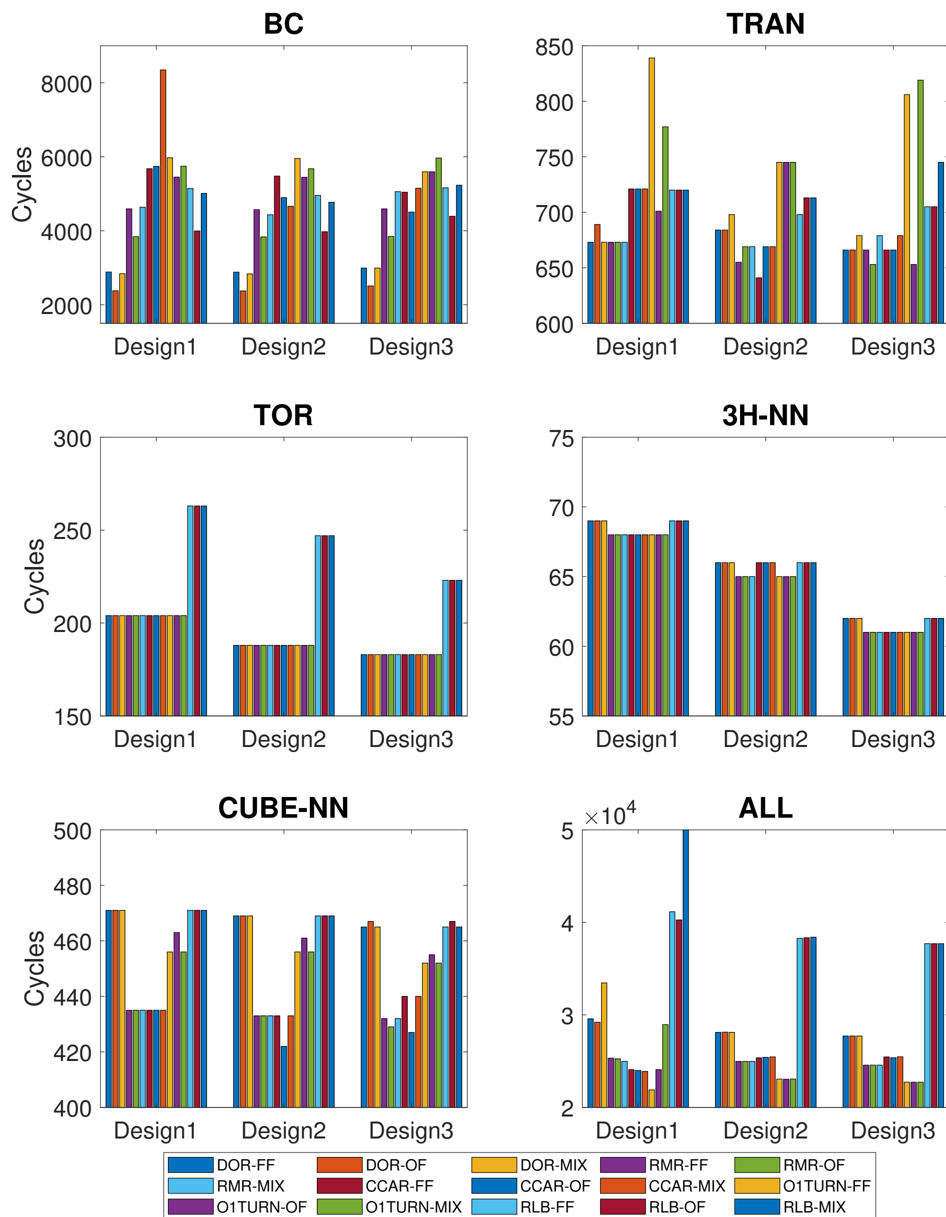

Figure 7·3: Batch latency performance with regarding to 15 combinations of routing algorithm and arbitration policy (large pattern). The lower the value is, the better.

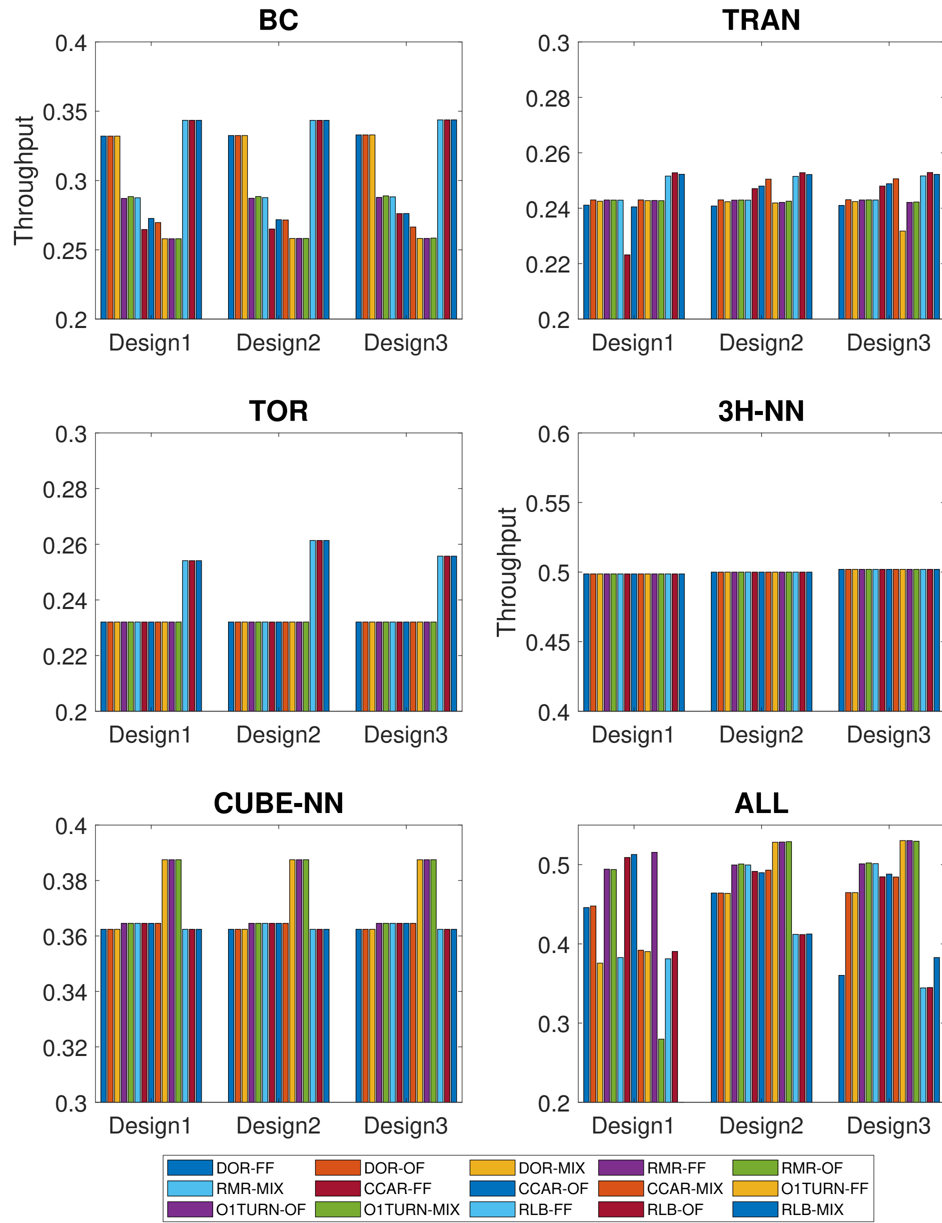

Figure 7·4: Throughput performance with regarding to 15 combinations of routing algorithm and arbitration policy (slow injection). The higher the value is, the better.

work well for Design 1, with only RMR and O1TURN works well for Design 2 and 3. CUBE-NN wise, RMR and CCAR are best for Design 1, CCAR-OF is best for Design 2, and O1TURN is best for Design 3. ALL have the most performance variance with CCAR-OF best for Design 1, O1TURN-OF best for Design 2 and 3.

Summary on Algorithms Selection

Based on the above analysis, we have the following general guidelines in finding the best combination of routing algorithm and switch arbitration policies:

1. Non-minimal routing is not desired.
2. There is not an optimal global algorithm and arbitration policy combination that fit for all; as a matter of fact, the performance for different routers can vary a lot.
3. Router architecture difference does not have much impact on per pattern optimal combination selection.
4. The performance differences among the 15 combinations only starts to show when the workload is heavy.

7.2.2 Locate the "Global Optimal" Design

Our design space is featuring three different router designs and various other design parameters. Without the application-aware framework, usually, there can only be a single router design that is for all the patterns and workload. To find what benefits application-aware selection can bring over an "one-for-all" router design, we conduct the following experiments.

One big question is how to locate the "global optimal" design? To answer that, we divide the question into two smaller ones: **(i)** Which router architecture with what

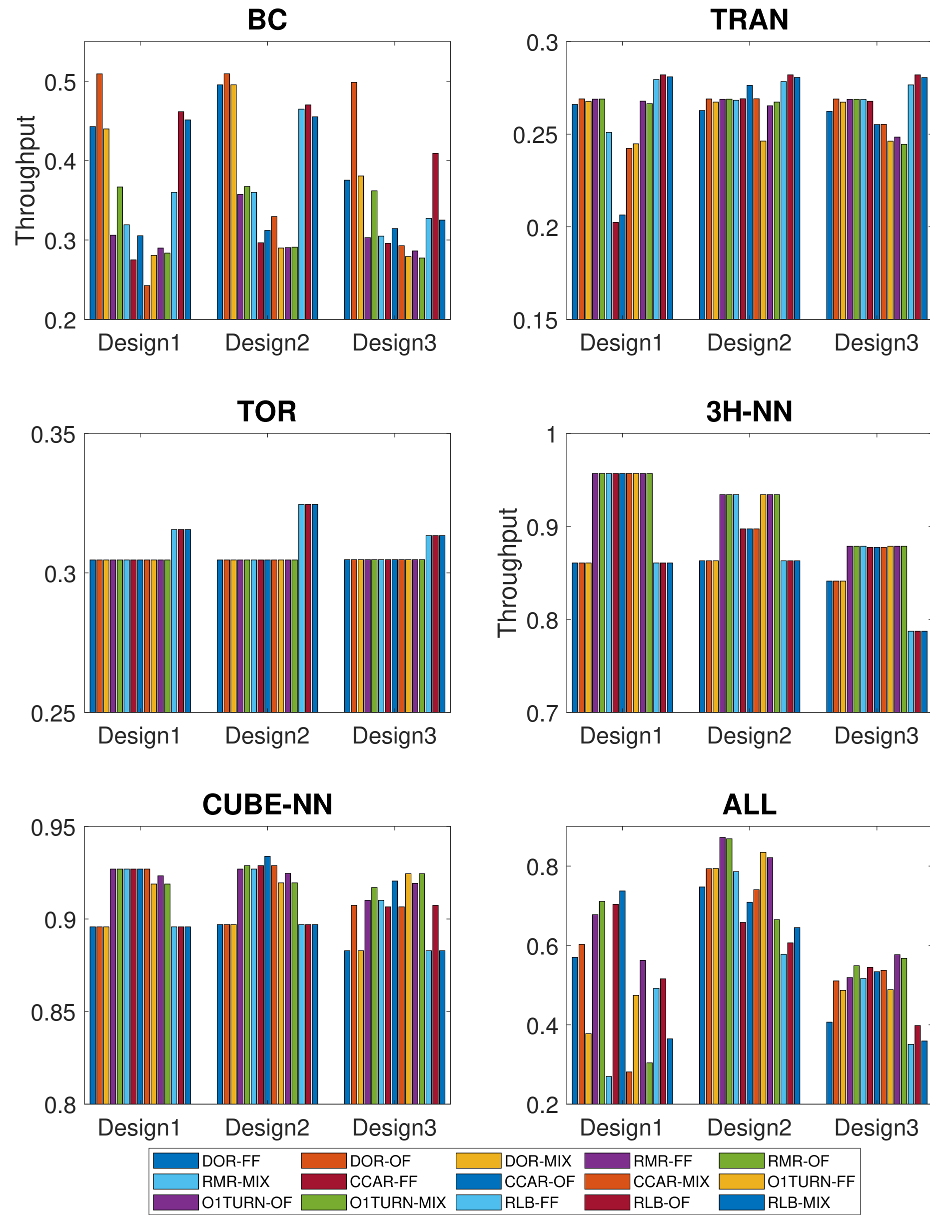

Figure 7·5: Throughput performance with regarding to 15 combinations of routing algorithm and arbitration policy (fast injection). The higher the value is, the better.

VC number should be considered the best design? **(ii)** Which routing algorithm and switch arbitration policy should be the overall best?

We answer the first question from the resource usage perspective since different router designs have different resource usage. We imagine three possible scenarios:

- The applications itself consumes most of the chip area, thus requires the router to consume as less resource as possible. Based on resource usage report in Table 5.4, 5.5, and 5.6, the designs with minimal ALM usage is Design 1 with VC number of 2.
- The applications itself is simple, thus leaves large resource budget for router implementations. In general, more resource usage tends to provide better performance. Thus, we select the router design with most resource usage, Design 2 with VC number of 9.
- The applications have high demand on on-chip storage units, but also want to maintain a certain level of routing performance. Based on the results presented in Figure 5·15, 5·16, 5·17, and 5·18, along with resource usage reported in Table 5.4, 5.5, and 5.6, we choose Design 2.

To answer the second question, for each one of the three listed router designs, we find the combination with on average best performance for each one of the six patterns we have. Due to the performance variance under two different working mode, we thus find the optimal combination under for each working mode individually (batch mode with a large pattern, and continuous mode with fast injection rate). To assign an equal weight for each pattern, we first normalize the performance number inside each pattern to the largest performance value among the 15 results. Then we sum up all the normalized performance from the six patterns for each one of the combinations. Then based on the working mode, we find the combination with the best number (minimum for batch mode, and maximum for continuous mode). The

detailed aggregated performance for each combination is shown in Figure 7·6.

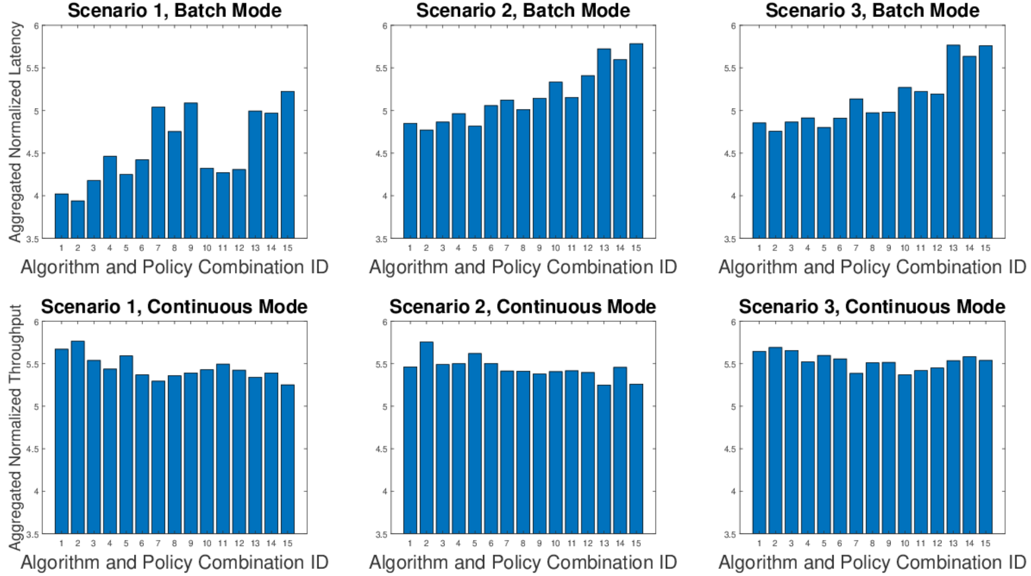

Figure 7·6: The aggregated performance number for all six patterns under different scenario and workload.

Based on the result listed in Figure 7·6, we select the best combination for each scenario and work mode and list in Table 7.2. Surprisingly, *when the VC number is small, **DOR-OF** seems to have the best overall performance for all cases.*

Table 7.2: Optimal Designs Under Different Scenarios

Scenario	Router Design	Work Mode	Optimal Combination
1	Design 1 (VC=2)	Batch	DOR-OF
2	Design 2 (VC=9)	Batch	DOR-OF
3	Design 3	Batch	DOR-OF
1	Design 1 (VC=2)	Continuous	DOR-OF
2	Design 2 (VC=9)	Continuous	DOR-OF
3	Design 3	Continuous	DOR-OF

7.2.3 Application-Aware Speedup over Global Optimal Design

Now that we have located the global optimal design for each scenario, now it is time to find the benefits for having an application-aware framework. We organize the experiments in three parts, each related to one of the three scenarios. Inside each

experiment, using the same workload as we used for locating the "global optimal" design, we explore the best-case performance we can get from all three router designs and with a few different VC numbers. Then we compare the performance between the application-aware-optimal design and global optimal design and find the improvements in performance, and discuss the trade-offs between performance the resource usage.

Under each scenario, we examine various design choices (by changing architecture and VC number) in addition to the assigned router design. Inside each design, we locate the application-aware optimal combination of algorithm and policy with regarding each one of the six patterns. For simplicity, we assign a number of each one of the 15 combinations (listed in Table 7.3). Since we use the same pattern size for different designs, the application-aware optimal design is fixed among all three scenarios and presented in Table 7.4.

Table 7.3: Number of each combination of routing algorithm and switch arbitration policy

#	Name	#	Name	#	Name
1	DOR-FF	2	DOR-OF	3	DOR-Mix
4	RMR-FF	5	RMR-OF	6	RMR-Mix
7	CCAR-FF	8	CCAR-OF	9	CCAR-Mix
10	O1TURN-FF	11	O1TURN-OF	12	O1TURN-Mix
13	RLB-FF	14	RLB-OF	15	RLB-Mix

Now we have the optimal design for each pattern, each design, and each work mode. We then measure the application-aware speedup over the global-optimal designs. For difference scenarios, we choose different VC numbers to match the resource usage from the given requirements. Details of the performance improvements for each application-aware optimal design over the global optimal design is listed in Table 7.5, 7.6, and 7.7.

In scenario 1, we use the design with minimal ALM usage (Design 1 with a VC number of 2). Let's first have a look at the batch mode. Within the same design, we

Table 7.4: Application-aware optimal designs targeting batch and continuous mode under a certain workload

Mode	Design	VC#	App-Aware Optimal Combination					
			BC	TRAN	TOR	3H-NN	CUBE	ALL
Batch	Design 1	2	2	2	1	13	10	8
Batch	Design 1	3	2	1	1	4	4	8
Batch	Design 1	4	2	4	1	1	4	7
Batch	Design 1	5	2	4	1	1	4	8
Batch	Design 1	6	2	4	1	4	4	7
Batch	Design 2	2	2	4	1	4	10	10
Batch	Design 2	3	2	7	1	4	8	4
Batch	Design 2	4	2	4	1	4	4	10
Batch	Design 2	5	2	4	1	4	8	6
Batch	Design 2	6	2	4	1	4	8	10
Batch	Design 3	N/A	2	5	1	4	8	10
Cont.	Design 1	2	2	14	10	1	10	8
Cont.	Design 1	3	2	14	13	10	11	8
Cont.	Design 1	4	2	14	13	4	4	7
Cont.	Design 1	5	2	14	13	4	4	8
Cont.	Design 1	6	2	14	13	4	4	7
Cont.	Design 2	2	2	14	13	4	10	10
Cont.	Design 2	3	2	14	13	4	8	12
Cont.	Design 2	4	2	14	13	7	5	10
Cont.	Design 2	5	2	14	13	4	8	10
Cont.	Design 2	6	2	14	13	4	8	10
Cont.	Design 3	N/A	2	14	13	4	10	10

see that for BC, TRAN, and TOR, the global optimal combination is the application-aware optimal. However, for the other three complex patterns, 3H-NN, CUBE, and ALL, application-aware provides 4.6% to 17.5% improvements. If the users are willing to allocate slight more ALM for the router, which increases the usage from 4.9% to 5.1% (Design 2 with VC = 2), there will be a substantial performance improvements of 79.8%, 36.1%, 101.7%, 112.4%, 68.8%, and 38.7% for the six patterns, respectively. Moreover, a similar trend is also observed for the throughput improvements in continuous mode.

In scenario 2, the routers are not constrained by resource. Based on our previous simulation results, we learned that more resource usage usually generates better performance. Thus, we select the largest router design, Design 2, with a VC number of 9, as the global optimal. Within the same configuration, we use our frame-

Table 7.5: Application Aware Optimal Speedup for **Scenario 1** (Global Optimal Design: Design 1, VC = 2, Combination 2)

Mode	Design	VC#	App-Aware Optimal Improvements					
			BC	TRAN	TOR	3H-NN	CUBE	ALL
Batch	*1*	*2*	*0%*	*0%*	*0%*	*4.6%*	*6.5%*	*17.5%*
Batch	1	3	44.9%	32.4%	85.3%	40.9%	54.4%	35.0%
Batch	1	4	63.5%	38.8%	96.9%	64.3%	74.2%	41.2%
Batch	**2**	**2**	**79.8%**	**36.1%**	**101.7%**	**112.4%**	**68.8%**	**38.7%**
Batch	2	3	79.8%	39.0%	101.7%	112.4%	80.4%	38.1%
Batch	2	4	79.8%	39.0%	101.1%	112.4%	75.8%	38.9%
Batch	3	N/A	70.1%	36.5%	106.6%	126.3%	78.3%	70.2%
Cont.	*1*	*2*	*0%*	*3.9%*	*2.4%*	*0%*	*6.0%*	*13.7%*
Cont.	1	3	29.7%	17.7%	17.5%	79.8%	48.8%	26.5%
Cont.	1	4	44.4%	17.7%	18.5%	83.8%	67.6%	30.4%
Cont.	**2**	**2**	**62.7%**	**17.7%**	**20.9%**	**67.0%**	**60.3%**	**28.6%**
Cont.	2	3	62.7%	17.7%	20.9%	74.9%	69.0%	27.8%
Cont.	2	4	62.7%	17.7%	20.9%	65.9%	68.3%	29.0%
Cont.	3	N/A	59.2%	17.7%	16.7%	68.8%	67.5%	48.5%

work to find the best case performance among the 15 combinations to determine the application-aware best performance for each one of the six patterns. The performance improvement is not so much with ALL being the best at 12.3% and 10.6%, for both batch and continuous mode, respectively. However, when we reduce the VC number to 5, we observe a similar level of performance. However, comparing with the global optimal design, this configuration reduces ALM consumption from 23.5% to 15.3%, and reduce BRAM consumption from 17.8% to 10.3%. Thus, with the help of our application-aware framework, the users could get a similar level of performance by using fewer resources.

In scenario 3, we choose a design (Design 3) with consideration for both resource usage and performance. In batch mode, the application-aware selection on routing algorithm and arbitration policy combinations provides good latency improvements for CUBE and ALL within the same router design. Looking at other designs, apparently, Design 1 does not have matching performance even with less resource consumption for most patterns except for TRAN and CUBE. When we increase the VC numbers on Design 2, the performance improvements also not by much. Looking back at the

Table 7.6: Application Aware Optimal Speedup for **Scenario 2** (Global Optimal Design: Design 2, VC = 9, Combination 2)

Mode	Design	VC#	App-Aware Optimal Improvements					
			BC	TRAN	TOR	3H-NN	CUBE	ALL
Batch	1	4	-9.1%	4.3%	-2.1%	-21.5%	7.4%	-8.4%
Batch	1	5	-2.7%	4.3%	-2.1%	-15.4%	7.9%	5.4%
Batch	1	6	-0.2%	4.3%	-2.1%	-10.9%	7.9%	10.2%
Batch	1	7	-0.2%	4.3%	-2.1%	-3.0%	7.9%	12.0%
Batch	1	8	-0.2%	4.3%	-2.1%	-3.0%	7.9%	12.0%
Batch	1	9	-0.2%	4.3%	-2.1%	-0.3%	7.9%	12.3%
Batch	2	4	0%	4.4%	0%	1.6%	8.4%	7.8%
Batch	**2**	**5**	**0%**	**4.4%**	**0%**	**1.6%**	**11.2%**	**7.9%**
Batch	2	6	0%	4.4%	0%	1.6%	11.2%	8.3%
Batch	2	7	0%	4.4%	0%	1.6%	11.2%	8.4%
Batch	2	8	0%	4.4%	0%	1.6%	11.2%	8.4%
Batch	*2*	*9*	*0%*	*4.4%*	*0%*	*1.6%*	*11.2%*	*12.3%*
Batch	3	N/A	-5.4%	2.5%	2.8%	8.2%	9.9%	32.8%
Cont.	1	4	-11.2%	4.9%	4.5%	10.9%	3.2%	-2.5%
Cont.	1	5	-1.8%	4.9%	4.5%	10.9%	3.4%	8.6%
Cont.	1	6	-0.1%	4.9%	4.5%	10.9%	3.4%	11.9%
Cont.	1	7	-0.1%	4.9%	4.5%	10.9%	3.4%	12.7%
Cont.	1	8	-0.1%	4.9%	9.2%	10.9%	3.4%	12.3%
Cont.	1	9	-0.1%	4.9%	9.2%	10.9%	3.4%	12.3%
Cont.	2	4	0%	4.9%	6.6%	0.1%	3.6%	10.4%
Cont.	**2**	**5**	**0%**	**4.9%**	**6.6%**	**8.3%**	**4.2%**	**10.7%**
Cont.	2	6	0%	4.9%	6.6%	8.3%	4.2%	10.8%
Cont.	2	7	0%	4.9%	6.6%	8.3%	4.2%	10.5%
Cont.	2	8	0%	4.9%	6.6%	8.3%	4.2%	10.9%
Cont.	*2*	*9*	*0%*	*4.9%*	*6.6%*	*8.3%*	*4.2%*	*10.6%*
Cont.	3	N/A	-2.13%	4.9%	2.9%	1.9%	3.1%	27.5%

continuous mode, this time Design 2 with a VC number of 5 provides good improvements on throughput for most patterns except ALL. However, the cost is increasing ALM usage from 8.2% to 15.3%. In this scenario, we can say the global optimal is the application-aware optimal for most patterns. Still, under different patterns, application-aware selection can provide marginal performance improvements.

7.3 Summary of Application-Aware Selection

In this chapter, we compare a variety of application-aware optimal designs with a set of global optimal designs under different scenarios. Compared with the global

Table 7.7: Application Aware Optimal Speedup for **Scenario 3** (Global Optimal Design: Design 3, Combination 2)

Mode	Design	VC#	App-Aware Optimal Improvements					
			BC	TRAN	TOR	3H-NN	CUBE	ALL
Batch	1	3	-14.8%	-1.1%	-10.3%	-36.8%	-5.3%	-40.0%
Batch	1	4	-3.9%	3.8%	-4.7%	-26.2%	6.9%	-22.4%
Batch	1	5	2.9%	3.8%	-4.7%	-20.6%	7.4%	-10.8%
Batch	2	3	5.7%	3.9%	-2.7%	-4.7%	10.7%	-8.8%
Batch	2	4	5.7%	3.9%	-2.7%	-4.7%	7.9%	-8.4%
Batch	2	5	5.7%	3.9%	-2.7%	-4.7%	10.7%	-8.8%
Batch	*3*	*N/A*	*0%*	*2.0%*	*0%*	*1.7%*	*9.4%*	*12.5%*
Cont.	1	3	-18.6%	4.9%	3.6%	11.4%	-9.6%	-28.5%
Cont.	1	4	-9.3%	4.9%	4.4%	13.8%	2.0%	-14.0%
Cont.	1	5	0.5%	4.9%	4.4%	13.8%	2.2%	-4.3%
Cont.	2	3	2.2%	4.9%	6.5%	8.3%	2.8%	-3.0%
Cont.	2	4	2.2%	4.9%	6.5%	2.7%	2.4%	-2.7%
Cont.	2	5	2.2%	4.9%	6.5%	11.1%	3.0%	-3.3%
Cont.	*3*	*N/A*	*0%*	*4.9%*	*2.9%*	*4.5%*	*1.9%*	*12.5%*

optimal design, our application-aware optimal designs achieve improvements on two levels: (i), they achieves on average 73% of performance improvements with slight resource overhead; and (ii) they achieve a similar level of performance, but with 35% less resource usage.

Chapter 8

FPGA-Centric Cluster Router, Part 4: Statically Scheduled Router Design

In the previous chapter, we introduce several router designs targeting dynamic communication patterns. And we find that application-aware optimization can bring good benefits. However, there are cases when communication patterns are static and known *a priori*; in this case judicious routing can reduce congestion, latency, and hardware required. This method is often referred to as "offline" or "static-scheduled" routing. In this chapter, we explore applying the method of offline/static routing to collective operations, in particular, multicast and reduction.

8.1 Table-based Static-Scheduled Router Architecture

In this section, we explore applying the method of offline/static routing to collective operations, in particular, multicast and reduction. New communication infrastructure is proposed and implemented, including switch design and routing algorithm. A substantial improvement in performance is obtained, especially for multicast. We believe that this is one of the few general offline/static routing solutions for real HPC clusters, and FCCs in particular.

In our previous application-specific design, we aim to provide support for multicast and reduction in a dynamic fashion. However, to limit the complexity in generating the dependency trees online, we only provide support for collective operations happens in a cube. However, with our new table-based solution, we can have more flexibility on

multicast and reduction patterns. The general working process is as follows: assuming the collective communication patterns have been extracted from the application (as is done, e.g., in (Grossman et al., 2015)), we can use an offline routing algorithm to build an optimized tree topology for each collective operation. These are fed into scripts to generate the routing, multicast, and reduction tables. Finally, the routing data are downloaded into the appropriate tables within the switches.

8.1.1 Table-based Routing

Table-based routing can be implemented in two ways: source routing and node-table routing (Kinsy et al., 2013a). Since source routing requires packets to carry table indexes, which consumes extra bandwidth, we instead use node-table routing. There the resident routing table preserves a table entry for each incoming packet (see Figure 8·1).

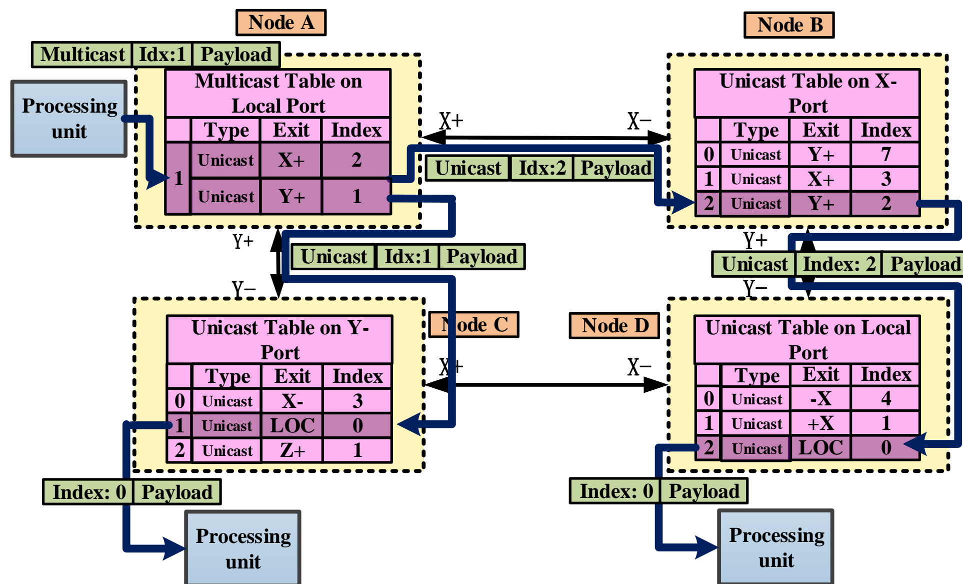

Figure 8·1: Node Table Routing Example

In this example, node A dispatches a multicast packet that carries three fields: packet type, table index, and payload. The router routes the packet to either a

unicast, multicast, or reduction table based on the packet type. In the corresponding table, multicast in this example, the router looks up the table entry based on the index field in the packet. The multicast table entry has slots for all of the six possible fan-outs.

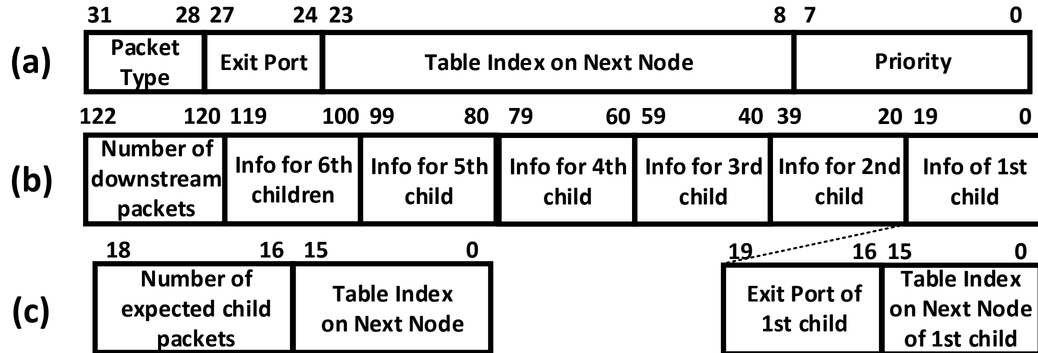

Figure 8·2: Packet Format for Static-Scheduled Routing

Figure 8·2 shows the data formats for the various types of table entries. In this example, the multicast packet has two fan-outs. For the first fan-out, the multicast table entry shows that it is a unicast packet, that it should be routed to the X+ port, and that the table index for the next node is 2. The router then generates a unicast-type packet and routes it to the X+ port, whence it enters the X- port on node B. On node B, since it is a unicast packet, the router looks up the entry in the unicast rather than the multicast table. The table entry shows that this packet should be routed to the Y+ port of node B. In the same manner, the router on Node B sends the packet to the Y- port of Node D, at which point it is ejected. Similarly, for the second branch, the packet is routed to the Y- port of Node C, where it is ejected.

8.1.2 Switch Architecture

Our static-scheduled router adopted the VC-based router (Dally, 1992), the architecture of which is shown in Figure 8·3.

To support the routing table lookup function, we extend the original four-staged

Figure 8·3: Router pipeline: (a) classic four-stage VC-based flow control; (b) proposed seven-staged pipeline

pipeline into seven stages (see Figure 8·4). The original switch has four stages: routing computation (RC), virtual channel allocation (VA), switch allocation (SA), switch traversal (ST). The first difference is that we divide the RC stage into three stages: input buffer consumption (IC), routing table lookup (RL), and multicast table lookup (ML). When a packet is injected into the switch input buffer, during the IC stage, we spend one cycle to fetch the flit from the buffer. We examine its header, to obtain its routing table index; this is used during RL to find the routing table entry. At that point, the switch knows whether the packet is multicast or not. If so, then during ML we look up its multicast table entry based on the index from the routing table entry. The VA stage allocates VCs to all the multicast children that are generated during

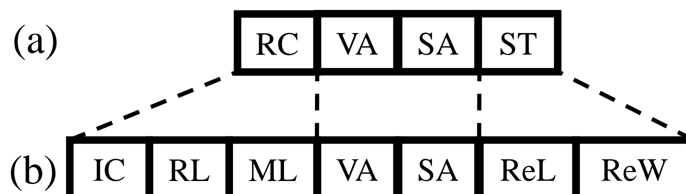

Figure 8·4: Switch architecture: (a) The switch is connected by seven input and seven output handlers. (b) The input handler has four stages: input buffer consumption, routing table lookup, multicast table lookup, and virtual channel allocation. (c) The output handler has three stages: switch allocation, reduction table lookup, and reduction table write-back.

ML. If the VA fails, the switch generates back pressure to stall the pipeline. In the SA stage, there might be multiple packets (up to 7) contending for the same output port. The packet with the highest priority, which is determined during the RL stage, wins the arbitration. Our current priority scheme is farthest-first.

Another difference between our switch and classical switch is that we divide the ST stage into two stages: reduction table lookup (ReL) and reduction table write-back (ReW). If the packet is not a reduction packet, it still traverses the last two stages (bypassing is undoubtedly an option). If the packet is a reduction packet, it is routed to the reduction unit. We allocate one entry in the reduction table for each reduction operation. During ReL, the reduction packet checks for its corresponding entry and whether the expected number of downstream packets have arrived. If not, then the reduction operation is executed and the reduction table entry updated. If all the expected downstream packets have arrived, the reduction unit dispatches a new packet and injects it into the upstream link.

8.1.3 Routing Algorithm

RPM (Wang et al., 2009) provides a solution to generate the multicast pattern recursively on a 2D-mesh network. Their goal is to build a multicast tree that tries to

reuse the network links as much as possible. However, we found that there are two obvious drawbacks in their algorithm, which is illustrated in Figure 8·5.

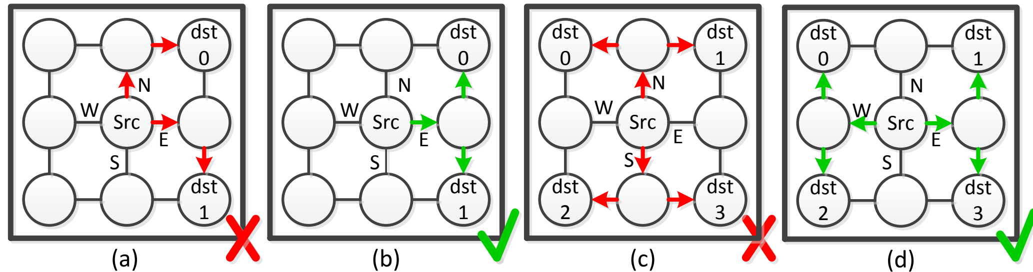

Figure 8·5: (a) and (c) is the routing decision made by RPM, (b) and (d) is the expected better routing decision. In (c) and (d), the north and south links are more congested than the west and east links

In Figure 8·5 (a) and (b), dst 0 is at the northeast of src, and dst 1 is at the southeast of src. The best multicast routing decision should first send the packet to the node on the east side and let this node distribute the packet to dst 0 and dst 1, as illustrated in (b). However, the RPM defines the North link always has the highest priority. The packet takes first north direction and then east direction to the dst 0 in RPM, which does not reuse the east link. In Figure (c) and (d), the two routing decisions have the same reusability of links. However, the north and south links are less congested than the west and east links, which means the routing decision made in (d) is better than the one in (c). The RPM makes decisions like (c) because the RPM does not take the congestion of links into account and the north link has higher priority.

Our new offline collective algorithm addresses the two drawbacks in RPM, and it is able to support the topology, including 3D- and 2D-torus and mesh. We call our offline collective routing algorithm as OCR. The pseudo-code of OCR is shown in Figure 8·6. The first step is to determine the space is a 1D, 2D, or 3D space. If it is a 1D space, the multicast routing is straightforward to implement. The algorithm for the 2D space is introduced later. If it is a 3D space, the next step is going to find

an optimal partition option. Three possible partition options corresponding to three dimensions. If space is a 3D torus, we can always partition it into three parts because every node can be viewed as the center of the space. As the example illustrated in Figure 8·7, we partition the entire space along three dimensions. Then we count the number of outbound links that goes out from the source to all the destinations for each kind of partition. In this example, partition along YZ plane requires only one outbound link, while partition along XZ plane and XY plane requires two and three links respectively. The partition along YZ plane is the best partition method in this example. If there is more than one partition method that has a minimal number of outbound links, we select the one that results in a smaller variance of the load on the six outbound links. So the load on the six links is more balanced. This step costs nothing because we know the application communication pattern as a prior so that we get the information of load on links for free. If the variances are still not able to distinguish them, we have a global round-robin pointer that keeps alternating among the three partitions. We use it as our selection. After we find the best partition, we partition the entire space into three parts: up space, middle plane, and down space. Also, we distribute the destination into the three subspaces depending on their coordinates. For the up and down space, we still call the 3D OCR algorithm again recursively. However, for the middle plane, we call the 2D OCR algorithm.

The 2D OCR algorithm is similar to RPM. As shown in Figure 8·8, we also partition the 2D space into 8 regions depending on where the source is. If space is a 2D-space, we can always find the 8 regions, because the source node could always be the center of the 2D torus. We call region 0,2,4,5 as corner regions and region 1,3,5,7 as side regions. In a 2D plane, one source has at most four fan-outs, which means we can have at most 4 partitions among the 8 regions. One corner region must merge with either of its two adjacent side regions. The first step is that we count the number

```
OCR_3D(src, dst_list, space){
   if space is a 1D space
      multicast src and dst_list in 1D space
   else if space is 2D space
      OCR_2D(src,dst_list,space)
   else if space is 3D space
     compare(partition_along_yz_plane, partition along xy_plane,
partition along_xz_plane)
     select best partition to get the new src dst_list and space for up
space, down space and plane space
     OCR_3D(src_up,dst_list_up,space_up)
     OCR_2D(src_plane,dst_list_plane,space_plane)
     OCR_3D(src_down,dst_list_down,space_down)
}
```

Figure 8·6: Pseudo code for proposed offline collective routing algorithm on 3D tours space

of nodes in all 8 regions. The next step is to determine whether we should enable the links in the north, south, west, and east directions and also determine which side region each corner region should merge. In Figure 8·9, the algorithm for the north link, south link, and region 0 is illustrated as an example. The merge direction of each corner region firstly depends on whether there are nodes in region 1, 2, 6, and 7, secondly depends on the balance of load on north and east links. In the next step, the plane and the destination list is partitioned into up to four parts. Moreover, the 2D OCR algorithm is called recursively for the four parts until they become 1D space.

8.2 Performance Evaluation on Static-Scheduled Router

8.2.1 Hardware Cost

We have implemented both the switch with online RPM routing logic and our switch with OCR routing on FPGA. The targeted network size is an 8×8×8 3D-torus. The synthesized results are demonstrated at Table 8.1. We can find that our offline router can save 5% of the entire chip area by eliminating the routing computation logic. We

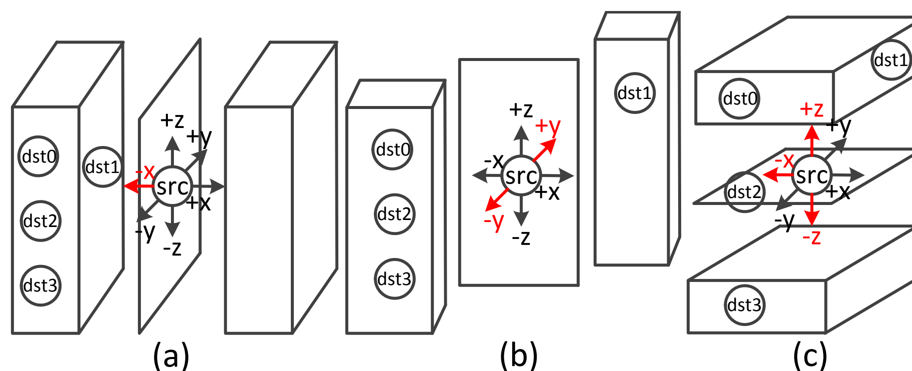

Figure 8·7: Partition evaluation example of OCR algorithm. (a) partition along yz plane, (b) partition along xz plane, (c) partition along xy plane

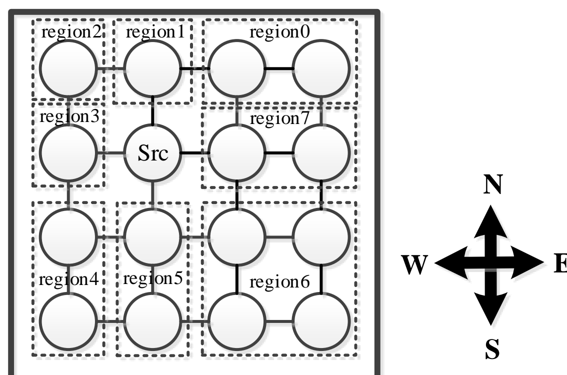

Figure 8·8: 8 regions on a 2D plane, Region 0, 2, 4 and 6 are called corner regions. Region 1, 3, 5 and 7 are called side regions.

measure the table sizes required by OCR algorithm for three typical collective patterns in Table 8.2. For all three cases, the routing table sizes of our offline OCR switch only consumes at most 1.17% of total on-chip memory more than online routing switch, which means our offline routing solution saves many logic elements on FPGA by only increasing a little bit more memory. One thing worth noticing is that the reduction operation requires much larger tables than multicast operation. This is because of that we need different table entries for different packets in reduction so that we could buffer the temporary reduction results.

Table 8.3 compares the empirically determined worst-case buffer sizes of online

```
initial north link and east  link as disabled
If dst in region1      enable north link
if dst in region 7     enable east link
if dst in region 0
  if north link is enabled and east link is disabled
    merge region0 to region 1
  else if north link is disabled and east link is enabled
    merge region 0 to region 7
  else if north link is enabled and east link is enabled
    if north link is more congested than east link
      merge region 0 to region 7
    else
      merge region 0 to region 1
  else //both north and east link are disabled
    if dst in region 2 and no dst in region 6
      enable north link, merge region 0 to region 1
    else if dst in region 6 and no dst in region 2
      enable east link, merge regoin0 to region 7
    else
      if north link is more congested than east link
        merge region 0 to region 7
      else
        merge region 0 to region 1
```

Figure 8·9: part of pseudo code of 2D OCR algorithm for region 0, north link and east link

and offline routing for the three patterns. The results show that offline routing can be expected to save around 20% to 30% input buffer size. Two other factors increase the advantage of the offline design. First, in production design, the online buffers would need to be somewhat larger to deal with worst-case scenarios. Second, while the offline buffer can be sized per application, the online account for the worst-case across all applications. An alternative for the online design is to use backpressure, but this substantially increases latency.

Another advantage of offline routing is the packet size. In online broadcast, the

Table 8.1: Logic elements utilization of RPM router and OCR router on 8×8×8 torus network

	RPM online router	OCR offline router
ALMs	56177	40895
utilization percent	21%	16%

Table 8.2: Memory consumption of routing tables (including multicast tables and reduction tables) of OCR algorithm on 4×4×4 torus network

Pattern	operation	table size(bits)	percentage
All-to-all	multicast	6968	0.013%
All-to-all	reduction	61.7K	1.17%
Bit Rotation	multicast	1122	0.002%
Bit Rotation	reduction	10K	0.19%
Nearest Neighbor	multicast	2928	0.005%
Nearest Neighbor	reduction	25.5K	0.485%

packet header has to contain the entire destination list, while the offline packet header only needs to carry a table index in the header. For a network has N nodes, the online routing header needs to have N bits, while the offline routing header needs only $\log_2 N$.

Table 8.3: Requirements of worst-case buffer size (depth) of online routing and offline routing for these three synthetic patterns, injection rate here is 1 packet per node per cycle

	operations	online routing	offline routing
All-to-all	multicast	1532	1132
All-to-all	reduction	367	288
Bit Rotation	multicast	91	57
Bit Rotation	reduction	1	1
Nearest neighbor	multicast	277	157
Nearest neighbor	reduction	14	9

8.2.2 Performance

We measure latency with respect to two types of loads, batch and continuous. For batch, each node transmits a fixed number of collective packets; latency is the time from when the first packet is sent until the last packet is received. For continuous, each node generates collective with a certain injection rate; latency is the average packet latency.

Figure 8·10 shows the results for batched experiments. We apply three typical benchmarks (all-to-all, nearest neighbor, and bit rotation) for two kinds of network size: $4\times4\times4$ and $8\times8\times8$. For the nearest neighbor pattern in $4\times4\times4$ network, each source node communicates to nearest 26 neighbors (3^3-1). For the nearest neighbor pattern in $8\times8\times8$ network, each source node communicates to nearest 124 neighbors (5^3-1). The patch size is set to 64 packets, and the injection rate is set to 1 packet per cycle per node. The results show that the for multicast operation, the latency of our offline routing solution is better than the online routing solution in most cases. For the reduction operation, the improvement is not as significant as multicast, because each node can at most inject one packet per cycle but can eject more than one packet per cycle. The reduction has much more injections than multicast. So the bottleneck of reduction is mostly the time spent on the injection.

The Figure 8·11 shows the results for continuous multicast experiments. Besides the previous three benchmarks, we apply two random patterns with different used nodes ratio. Compared with online routing, our offline routing solution not only provides the smaller average collective operation latency in almost all cases but also provides delayed saturation point in most cases. There are exceptions in the neighbor pattern. The gap between offline routing and online routing is minimal. This is because the nearest neighbor pattern is very symmetric and balanced. There is not much room for improvement.

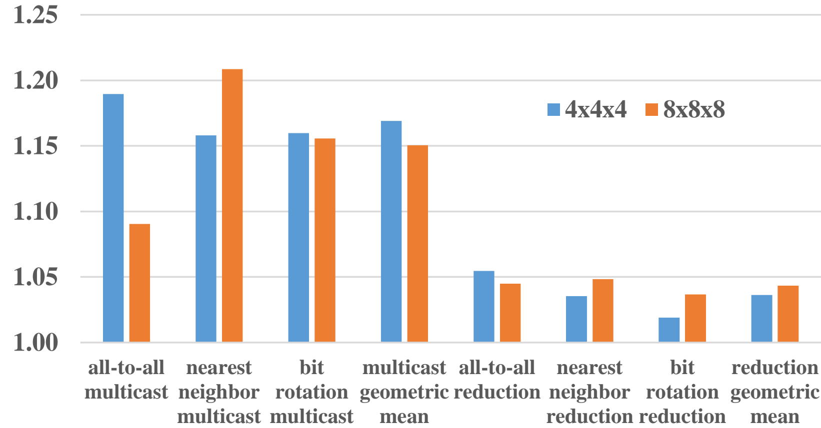

Figure 8·10: batched experiments with three typical benchmarks (all-to-all, nearest neighbor, and bit rotation) for two kinds of network size: $4 \times 4 \times 4$ and $8 \times 8 \times 8$.

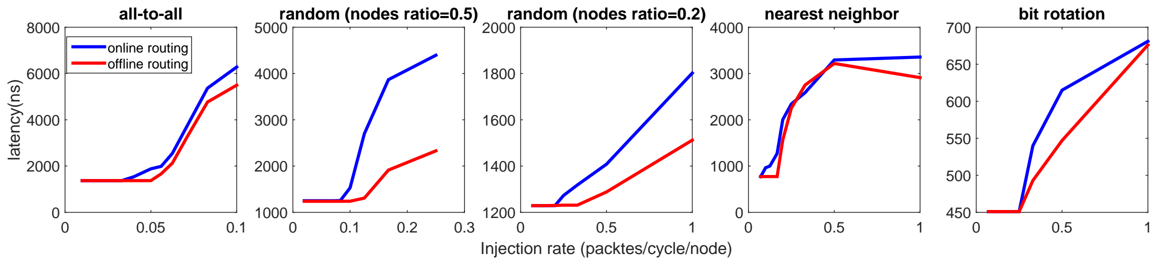

Figure 8·11: Average latency of multicast packets in $4 \times 4 \times 4$ network

8.3 Summary of Statically-Scheduled Router Design

In this chapter, we describe a complete communication infrastructure to support offline (statically scheduled) routing of collective communication on FCCs. We use table-based routing and new switch design. We propose a new offline collective routing algorithm (OCR) that takes advantage of the knowledge of communication patterns to load-balance network links and reduce congestion. Experiments show that this offline routing solution has significantly better performance and lower hardware cost than a state-of-the-art online routing solution. The OCR algorithm has not been proven to be optimal; however, the infrastructure described supports improvements as they are developed with no change in design.

Chapter 9

FPGA-Centric Cluster Router, Part 5: Topology Emulation

The Novo-G# is physically configured into a 3D-torus topology to take advantage of communication commonly used in many applications including physical simulations. Not all applications have a 3D topology. The cost of rewiring the cluster to match other applications' logical mapping is clearly impractical. In those cases, a simple alternative is to use an oblivious mapping scheme that can fit any applications onto our existing 3D-torus fixture.

In this chapter, we propose a generalized *fold-and-cut* mechanism that can automatically convert the node coordinates from 2D logical mapping onto a 3D-torus. This project is still work-in-progress, but we find the results are promising and thus introduce our preliminary findings in this chapter.

9.1 Logical Topology to Physical Topology Mapping

In this chapter, we are focusing on mapping applications with a lower dimension (2D) topology onto a higher dimension (3D). Based on different mapping schemes, we can map any 2D node coordinates to our 3D torus with new coordinates. For any 2D communication patterns, by converting its source and destination coordinates to the 3D coordinates, we can directly make use of our router infrastructure, with which, those 2D applications can make use of the two additional links provided by our FCC hardware and thus gain better performance.

9.1.1 Folding Mechanism

Our 2D to 3D mapping is performed in a "fold and cut" manner. The core idea of this implementation is to keep the neighboring connections inside logical mapping when making the conversion.

The first step is to roll and fold. Based on 2D logical topology size and our FCC size, the users can choose to the number of folds (Figure 9·1 shows a 3-fold). In general, we want to have fewer folds, thus better maintain the spatial locality inside each post-folding surface.

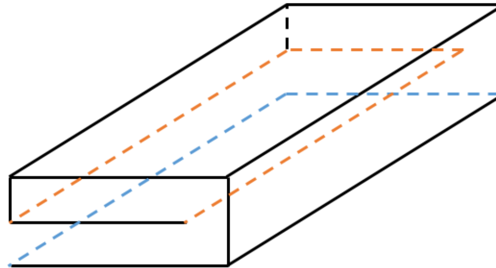

Figure 9·1: Step 1: Fold

The second step is to cut. Which evenly cut the folded mapping into more blocks (see Figure 9·2).

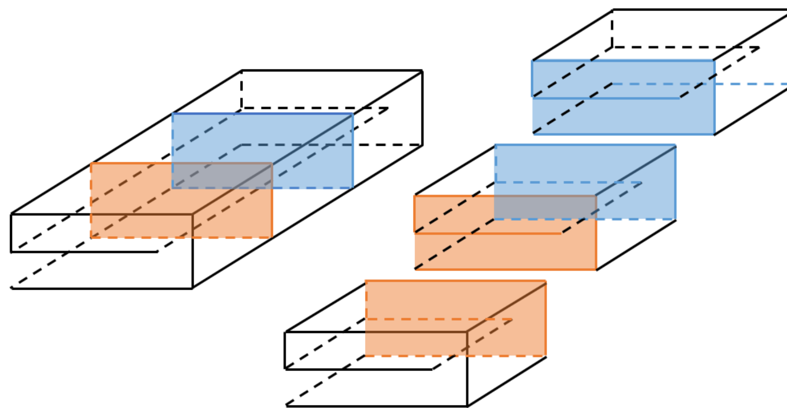

Figure 9·2: Step 2: Cut

The next step is to pile up those blocks. We Z-fold those blocks so that we are trying to keep original topology, keep the nearest neighbor still close enough(see Figure 9·3).

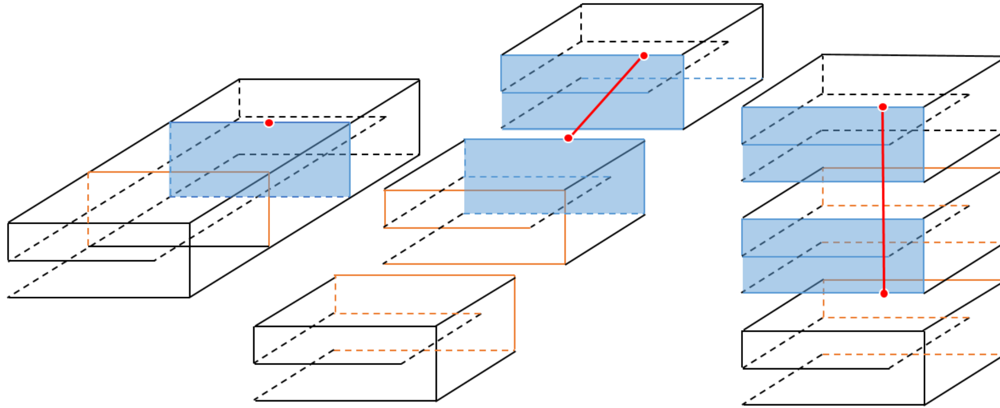

Figure 9·3: Step 3: Z-Fold

9.2 Topology Emulation Performance

We use this cut-and-fold to maintain the spatial locality from logical mapping as much as possible. Depending on the original 2D logical topology size and 3D torus hardware fixture size, the number of cuts and folds can vary. We use a 16×16 2D torus mapping onto $8 \times 8 \times 8$ 3D-torus hardware as an example. To make sure the transformed topology can fit on our hardware, we limited the fold value from 2 to 4, and cut value from 1 to 3.

9.2.1 Fold and Cut Performance

We use our application-aware framework to evaluate the performance of different folding and cutting schemes regarding different 2D communication patterns. The patterns we used are “all-to-all” and “square nearest-neighbor” in 2D logical mapping. After mapping onto the 3D torus hardware, the related source and destination is

transformed automatically.

The evaluated latency and throughput performance for the all-to-all pattern are shown in Figure 9·4 and 9·5. When running in batch mode, 4-fold and 1-cut gives the best performance across all three router designs. While in batch mode, 3-fold and 1-cut is better on Design 1 and 3, with 4-fold and 1-cut the best in Design 2.

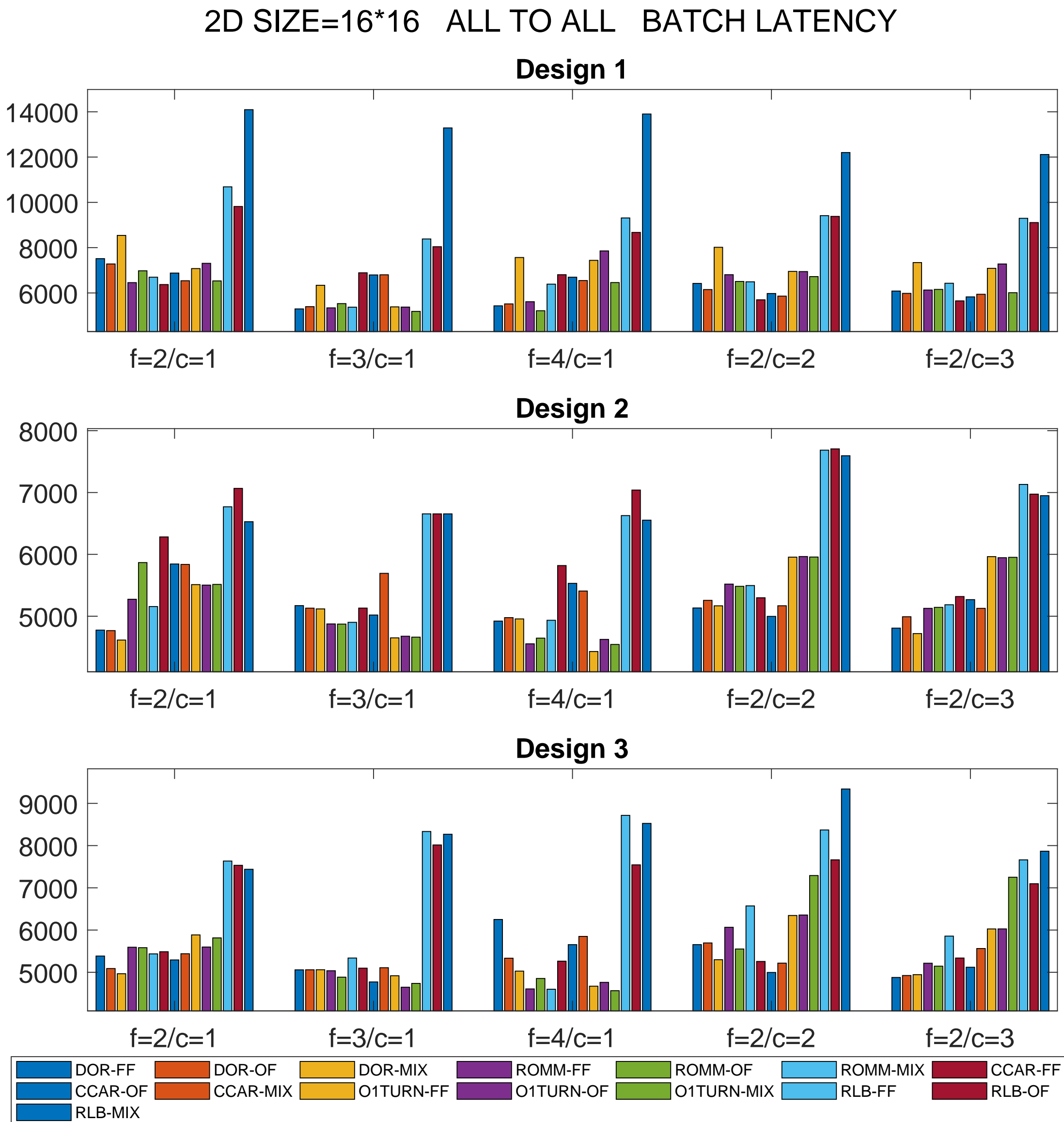

Figure 9·4: 2D to 3D Mapping Performance: All-to-All Batch Latency

For square neighbor, the performance is shown in Figure 9·6 and 9·7. In batch

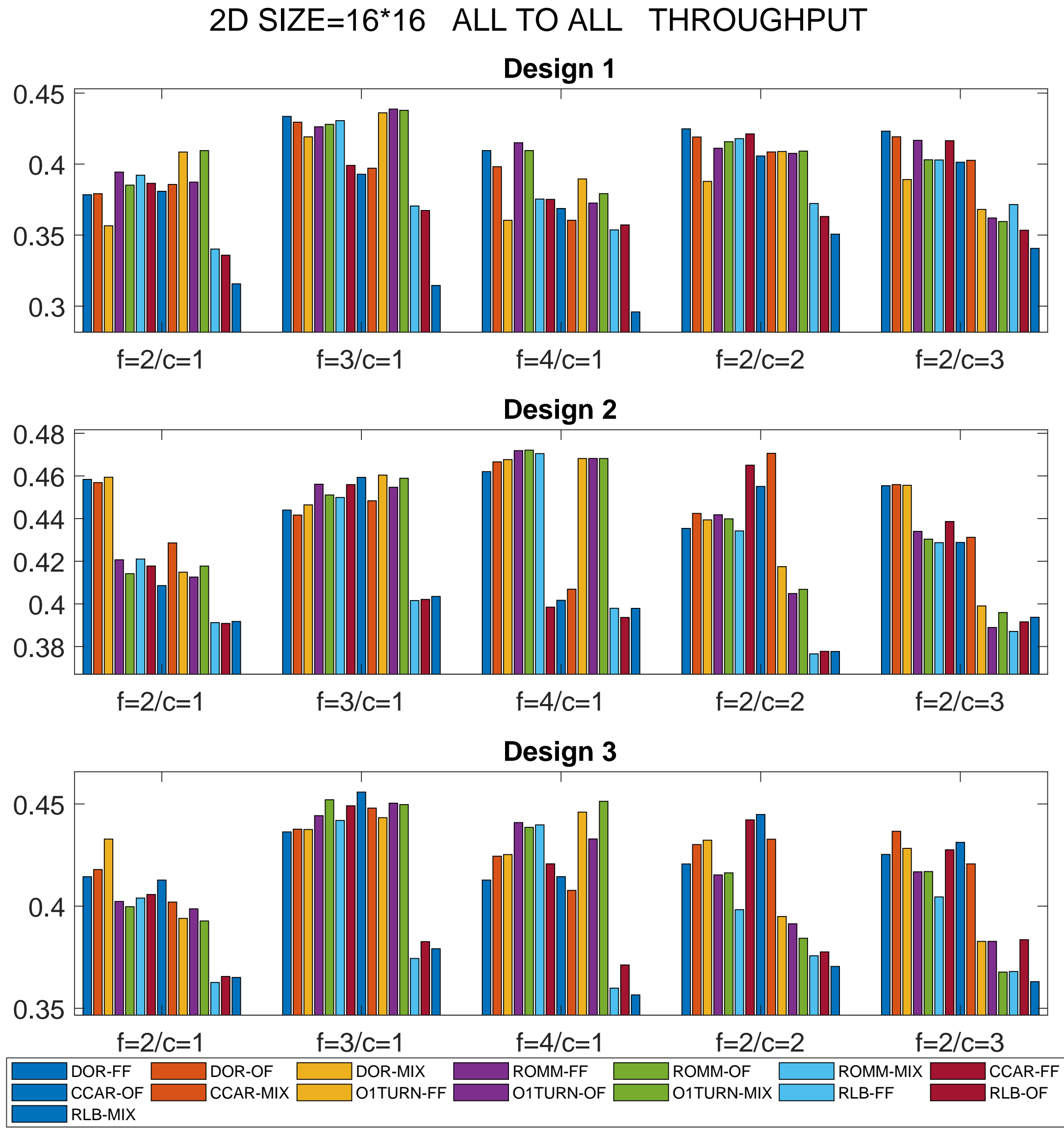

Figure 9·5: 2D to 3D Mapping Performance: All-to-All Throughput

mode, 2-fold and 1-cut always return the best-case performance while a similar trend shows in continuous mode as the all-to-all pattern.

In general, in batch mode, we notice that for a simple pattern (square neighbor), the less fold and cut returns better latency performance. Because that pattern only sends to its logic neighbors, where less fold and cut means more spatial locality is preserved. While for all-to-all patterns, since each node is sending to multiple nodes

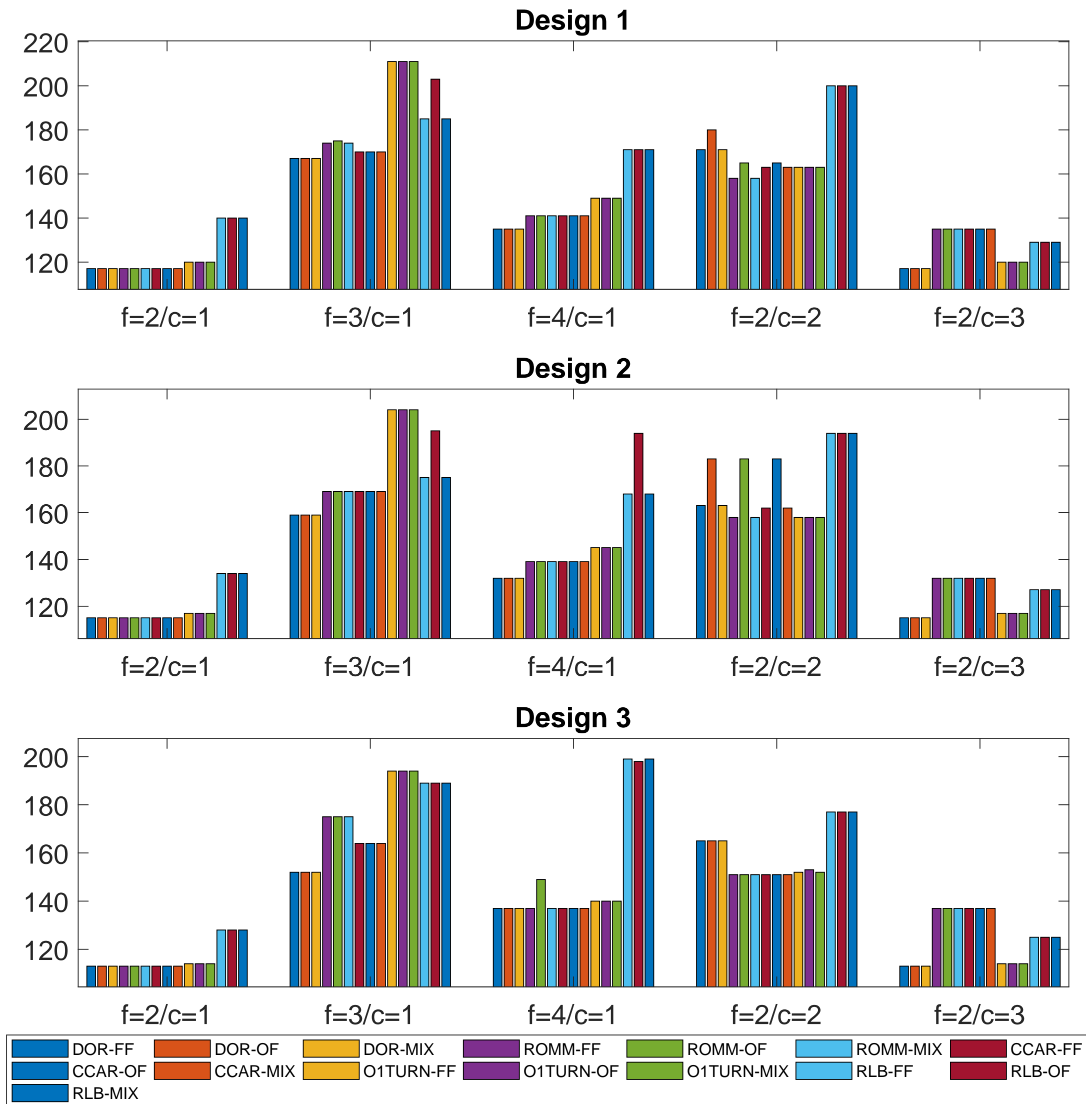

Figure 9·6: 2D to 3D Mapping Performance: Square Nearest Neighbor Batch Latency

in the network, the spatial locality is of less importance. More folds and cuts result in a more cube-like topology, which provides more vertical links for traffic to use. So this is precisely the reason why we are seeing 3-fold and 4-fold are returning better throughput performance in continuous. On the other hand, we cannot have unlimited folds and cuts. One of the reason being we need to fit the ending topology on our

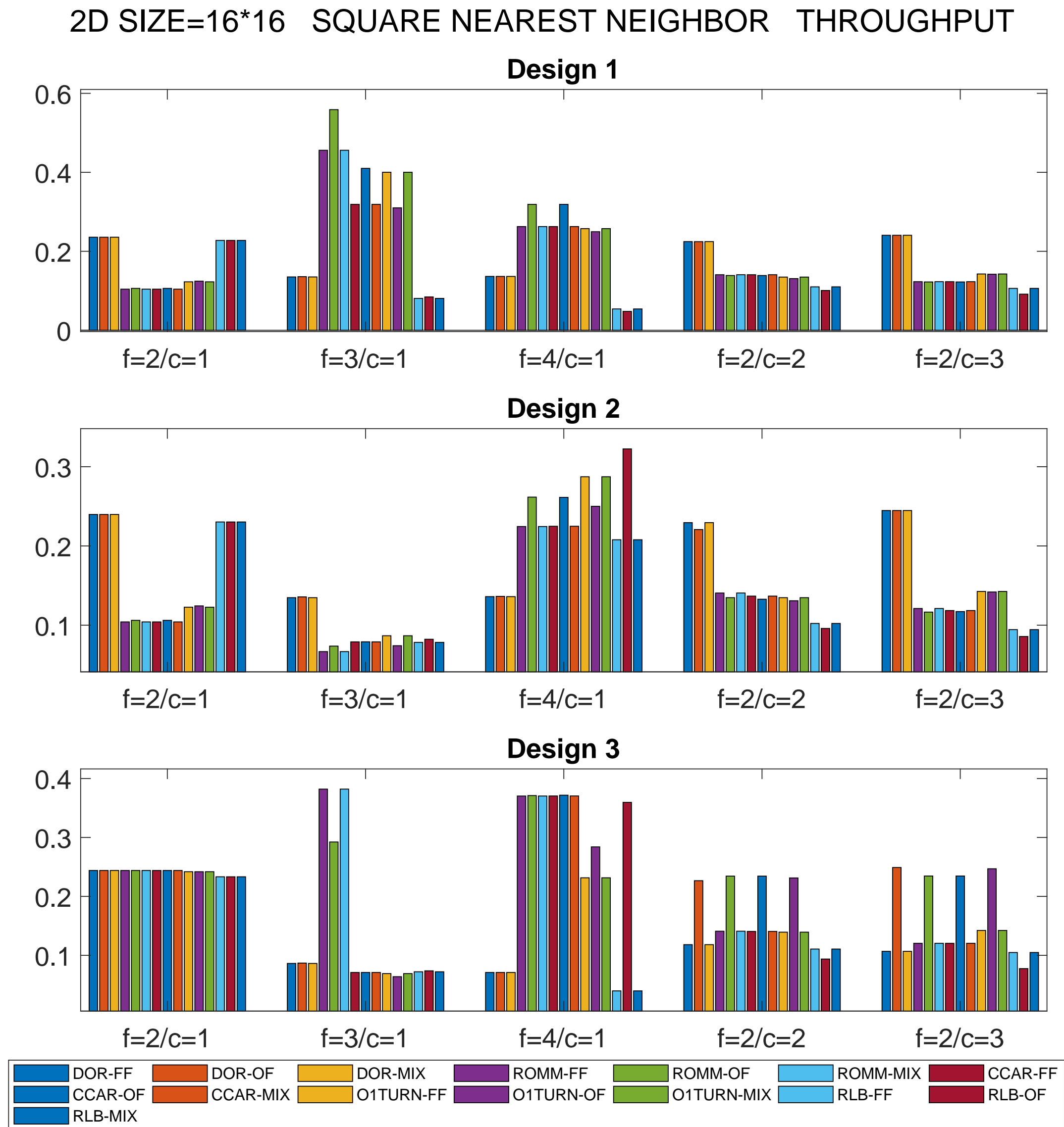

Figure 9·7: 2D to 3D Mapping Performance: Square Nearest Neighbor Throughput

$8 \times 8 \times 8$ hardware. Another reason being each time we cut and fold, we necessarily extend the distance between the neighboring nodes in the logical mapping (indicate by the red line in Figure 9·3).

9.2.2 Link Usage Imbalance

When increasing the topology dimension from 2D to 3D, each router provides two extra links. After the fold-and-cut, those two extra links can be used to provide more flexibility and extra bandwidth. However, the post-fold-and-cut routing pattern is no longer uniformly distributed. As a result, link usage may vary depending on the routing algorithm and patterns.

Figure 9·8 depicts our measure link usage status. For each pattern and each fold-and-cut configuration, we collect the average link usage status across all the links that are used at least once during the routing process. From the figure, we find that for patterns like all-to-all and nearest neighbor, the link usage is imbalanced.

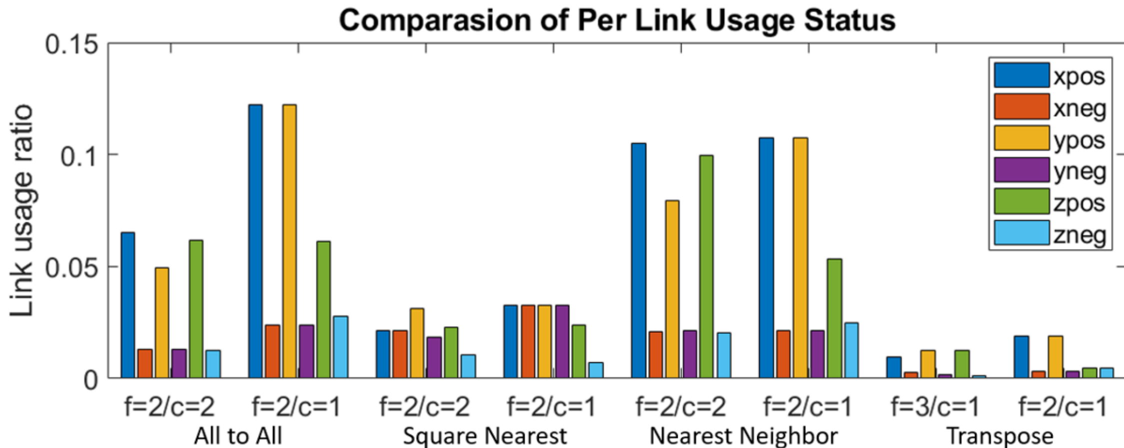

Figure 9·8: Link usage under different fold and cut configurations

Given the fact that there is a large variance in link usage, in the future, we can further optimize the router design by allocating fewer resources to those links with less usage and thus improve the resource usage efficiency.

9.3 Summary

In this chapter, we present a general approach for mapping a 2D logical topology onto a physical 3D torus. Evaluation results show different optimal fold and cut

configurations for different workloads. Another fact is the imbalanced workload on different links, which provides a chance to further optimize the router design by reallocating the buffer resources to links that are most heavily used. Although this is work-in-progress, preliminary results are promising and point to further exploration.

Chapter 10

Conclusion

In this chapter, a summary of this dissertation is provided, followed by proposed future work.

10.1 Conclusion

In this dissertation, we have explored the communication infrastructure design space target FCCs on three different levels.

On the application layer, we describe two applications: Dark Matter (DM) detection and Molecular Dynamics (MD) simulation. The DM project requires communication between eight FE FPGAs and a single BE FPGA and demonstrates the communication capability of FPGAs. MD is a compute-intensive application. In the initial implementation, by having all particle data and evaluation modules integrated on a single FPGA chip, we eliminate the communication overhead between the host processor and FPGA. The single-chip implementation achieves a comparable performance with respect to the latest GPU running a production MD package. Another significant point of MD is the potential of scaling onto multiple FPGAs. In this case multiple different communication patterns are used; these serve as a good model application for applying the application-aware communication framework.

On the physical link level, we examine the inter-FPGA link performance for two different link-layer protocols: Interlaken and SerialLite III. We measure the link variance on multiple levels: different lanes in the same link, different links on the same

FPGA, and different links on different FPGAs. One significant result is simply the measure of link latency, which fundamentally shaped the router designs. Also, by comparing the underlying hardware configurations between NoCs and FCCs, we formed the design philosophy of the router architecture on FCCs, which served as a guideline for all the proposed router architectures.

On the network layer, we propose three different router architectures, with two wormhole-style routers and a virtual cut-through router. We parameterize the router with regard to the number of VCs, buffer size, and various other attributes. We also adopted five different routing algorithms and three switch arbitration policies. These all contribute to a large design space. To find the router design that fit well to any communication patterns, we propose an application-aware framework. In this framework we provide a cycle-accurate simulator that is an exact match of the hardware design. With the help of the simulator, for a given communication pattern, the framework can not only provide an estimate of communication performance, but do so quickly. Based on the simulation result, we can determine the application-aware optimal design. Compared with a global optimal design, the application-aware design can lead to substantial improvement in performance or reduction in resource utilization. Also, we develop several modules and function units that can provide specific optimizations for certain types of communication workloads depending on the application is going to serve. Besides the dynamic router, we also propose a table-based statically-scheduled router. If the communication pattern of an application is known *a priori*, we can make the routing decisions beforehand, and upload the routing decisions into a table. With the help of routing tables, we can support complex multicast and reduction patterns. Compared with dynamic multicast support, the static-scheduled version shows good speedup.

10.2 Future Work

10.2.1 Future Work on Inter-FPGA Link

In the current implementation, both IP cores we used, Interlaken PHY and SerialLite III, have a similar level of link variances. To achieve better efficiency, the link variances should be minimized. In (Liu et al., 2014; Giordano and Aloisio, 2012), the authors design a custom IP core with the purpose of having a fixed latency. But they run on FPGAs manufactured by different vendors, and are not ready to use on the FPGA. One future project in this domain is adopting their design idea and redesigning the transceiver IP controller.

10.2.2 Future Work on Router Design

The design space now featuring three router designs, five routing algorithms, and three arbitration policies. But we can further expand the current design space on multiple levels.

First, we can introduce more routing algorithms and arbitration policies into the design. For algorithms, we can try PROM (Cho et al., 2009), which introduces a probability on top of the RMR routing algorithm. For arbitration policies, we can try others including Round-Robin, First Come First Served (FCFS), and Priority-based Rounding Robin (PBRR).

Second, in the current design, once the data is transmitted on the link, we assume that the flit will be delivered correctly. Even though the MGT link is reliable, there is still a chance that link error occurs. In order to address this, an error correction mechanism is needed. This requires the sender to hold the packets that are sent out for a given period of time just in case the retransmission is required. Since this design significantly increases resource usage, and also requires an acknowledge mechanism, this could be an application-specific add-on, which only serves those applications have

high requirements on robustness. Another function we can add to the application-specific library is the DRAM controller, to satisfy the needed for applications with large storage requirements.

Third, in the previous work, we allocate the same amount of resources for each input/output port. However, the evaluation of topology emulation has demonstrated that different fold-and-cut configurations and different patterns lead to link usage imbalance. Thus, we can further optimize the router resource usage by reallocating the routing resources to those links that are heavily used.

10.2.3 Future Work on Model Applications

For the MD project, the immediate next step is to extend the current single-chip MD simulation system onto an FPGA-cluster. In this implementation, the distribution of workloads and particle data will have a large impact on communication patterns. A selection of communication patterns includes all-to-all and scatter-gather as required by multi-chip 3D-FFT and cube nearest neighbor as required by RL force evaluations. Using the communication framework presented in this thesis, we can find an MD-specific optimal design.

References

A. M. Caulfield et al. (2016). A cloud-scale acceleration architecture. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13.

Abad, P., Puente, V., and Gregorio, J. (2009). Mrr: Enabling fully adaptive multicast routing for cmp interconnection networks. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 355–366. IEEE.

Ade, P. et al. (2016). Planck 2015 results. XIII. Cosmological parameters. *Astronomy & Astrophysics*, 594:A13.

Akerib, D., Araujo, H., Bai, X., Bailey, A., Balajthy, J., Beltrame, P., Bernard, E., Bernstein, A., and et al. (2016). FPGA-based Trigger System for the LUX Dark Matter Experiment. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 818:57–67.

Alam, S., Agarwal, P., Smith, M., Vetter, J., and Caliga, D. (2007). Using FPGA devices to accelerate biomolecular simulations. *Computer*, 40(3):66–73.

Altera (2014a). Cyclone V Device Handbook vol. 2: Transceivers.

Altera (2014b). Cyclone V GT FPGA Development Board Reference Manual.

Altera (2014c). *Stratix V Device Handbook volume2: Transceivers*. Altera.

Altera (2015a). Altera SDK for OpenCL: Best Practices Guide.

Altera (2015b). Altera Transceiver PHY IP Core User Guide.

Alverson, B., Froese, E., Kaplan, L., and Roweth, D. (2012). Cray xc series network.

Amber (2018). Amber18: pmemd.cuda performance information. `http://ambermd.org/GPUPerformance.php#RCWBenchmarks`.

Azizi, N., Kuon, I., Egier, A., Darabiha, A., and Chow, P. (2004). Reconfigurable molecular dynamics simulator. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, pages 197–206.

Badr, H. and Podar, S. (1989). An optimal shortest-path routing policy for network computers with regular mesh-connected topologies. *IEEE Transactions on Computers*, 38(10):1362–1371.

Baran, P. (1964). On distributed communications networks. *IEEE transactions on Communications Systems*, 12(1):1–9.

Bertone, G., Hooper, D., and Silk, J. (2005). Particle dark matter: Evidence, candidates and constraints. *Physics Reports*, 405:279–390.

Birrittella, M., Debbage, M., Huggahalli, R., Kunz, J., Lovett, T., Rimmer, T., Underwood, K., and Zak, R. (2015). Intel Omni-path Architecture: Enabling Scalable, High Performance Fabrics. In *High-Performance Interconnects (HOTI)*, pages 1–9.

Brightwell, R., Pedretti, K. T., Underwood, K. D., and Hudson, T. (2006). Seastar interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3):41–57.

Brown, W., Wang, P., Plimpton, S., and Tharrington, A. (2011). Implementing molecular dynamics on hybrid high performance computers–short range forces. *Computer Physics Communications*, 182(4):898–911.

Bunker, T. and Swanson, S. (2013). Latency-Optimized Networks for Clustering FPGAs. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Case, D., Ben-Shalom, I., Brozell, S., Cerutti, D., Cheatham, T., et al. (2018). Amber18. `http://ambermd.org/doc12/Amber18.pdf`.

Case, D., Cheatham III, T., Darden, T., Gohlke, H., Luo, R., Merz, Jr., K., Onufriev, A., Simmerling, C., Wang, B., and Woods, R. (2005). The Amber biomolecular simulation programs. *Journal of Computational Chemistry*, 26:1668–1688.

Cerf, V. and Kahn, R. (1974). A protocol for packet network intercommunication. *IEEE Transactions on Communications*, (5):637–648.

Chiu, M. (2011). *Accelerating Molecular Dynamics Simulations with High Performance Reconfigurable Systems*. PhD thesis, Department of Electrical and Computer Engineering, Boston University.

Chiu, M. and Herbordt, M. (2009). Efficient filtering for molecular dynamics simulations. In *Proceedings of IEEE Conference on Field Programmable Logic and Applications (FPL)*.

Chiu, M. and Herbordt, M. (2010a). Molecular dynamics simulations on high performance reconfigurable computing systems. *ACM Transcation Reconfigurable Techology and Systems (TRETS)*, 3(4):1–37.

Chiu, M. and Herbordt, M. (2010b). Towards production FPGA-accelerated molecular dynamics: Progress and challenges. In *Proceedings of High Performance Reconfigurable Technology and Applications*.

Chiu, M., Herbordt, M., and Langhammer, M. (2008). Performance potential of molecular dynamics simulations on high performance reconfigurable computing systems. In *Proceedings High Performance Reconfigurable Technology and Applications*.

Chiu, M., Khan, M., and Herbordt, M. (2011). Efficient calculation of pairwise nonbonded forces. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Cho, M., Lis, M., Shim, K., Kinsy, M., and Devadas, S. (2009). Path-based, randomized, oblivious, minimal routing. In *Proceedings of the 2nd International Workshop on Network on Chip Architectures*, pages 23–28. ACM.

Cong, J., Fang, Z., Kianinejad, H., and Wei, P. (2016). Revisiting FPGA Acceleration of Molecular Dynamics Simulation with Dynamic Data Flow Behavior in High-Level Synthesis. *arXiv preprint arXiv:1611.04474*.

Conti, A., VanCourt, T., and Herbordt, M. (2004). Flexible FPGA acceleration of dynamic programming string processing. In *Proceedings of IEEE Conference on Field Programmable Logic and Applications (FPL)*.

Cray (2010). The gemini network, rev 1.1.

Cray (2019). Slingshot: The interconnect for the exascale era.

D'Alberto, P., Milder, P., Sandryhaila, A., Franchetti, F., Hoe, J., Moura, J., Pueschel, M., and Johnson, J. (2007). Generating FPGA-Accelerated DFT Libraries. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Dally, W. and Aoki, H. (1993). Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE transactions on Parallel and Distributed Systems (TPDS)*, 4(4):466–475.

Dally, W., Carvey, P., and Dennison, L. (1998). The Avici Terabit Switch/Router. In *Proc. Sixth Symp. Hot Interconnects*, pages 41–50.

Dally, W. and Song, P. (1987). Design of a self-timed vlsi multicomputer communication controller. Technical report, Massachusetts Institute of Technology Cambridge Artificial Intelligence Lab.

Dally, W. and Towles, B. (2001). Route packets, not wires: on-chip inteconnection networks. In *Proceedings of the 38th annual Design Automation Conference*, pages 684–689. ACM.

Dally, W. and Towles, B. (2004). *Principles and Practices of Interconnection Networks*. Elsevier.

Dally, W. J. (1990). Performance analysis of k-ary n-cube interconnection networks. *IEEE Transaction on Computers (TC)*, C-39(6):775–785.

Dally, W. J. (1992). Virtual channel flow control. *IEEE Transaction on Parallel and Distributed Systems (TPDS)*, 3(2):194–205.

Dally, W. J. and Seitz, C. L. (1986). The Torus Routing Chip. *Distributed Computing*, 1:187–196.

Dally, W. J. and Seitz, C. L. (1987). Deadlock free routing in multiprocessor interconnection networks. *IEEE Transaction on Computers (TC)*, C-36(5).

Dershowitz, B., LaPointe, P., Eiben, T., Wei, L., et al. (1998). Integration of discrete feature network methods with conventional simulator approaches. In *SPE Annual Technical Conference and Exhibition*. Society of Petroleum Engineers.

Dick, C. (1998). Computing Multidimensional DFTs Using Xilinx FPGAs. In *8th Int. Conf. Signal Processing Applications and Technology*.

Draper, J. and Ghosh, J. (1994). A comprehensive analytical model for wormhole routing in multicomputer systems. *Journal of Parallel and Distributed Computing*, 23(2):202–214.

Duato, J., Yalamanchili, S., and Ni, L. (1997). *Interconnection Networks: An Engineering Approach*. IEEE Computer Society Press.

Fallin, C., Craik, C., and Mutlu, O. (2011). Chipper: A low-complexity bufferless deflection router. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 144–155. IEEE.

Feng, W. and Shin, K. (1997). Impact of selection functions on routing algorithm performance in multicomputer networks. In *Proceedings of International Conference on Supercomputing (SC)*, pages 132–139. ACM.

Fick, D., DeOrio, A., Chen, G., Bertacco, V., Sylvester, D., and Blaauw, D. (2009). A highly resilient routing algorithm for fault-tolerant nocs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 21–26. European Design and Automation Association.

Fukushige, T., Taiji, M., Makino, J., Ebisuzaki, T., and Sugimoto, D. (1996). A highly parallelized special-purpose computer for many-body simulations with an arbitrary central force: MD-GRAPE. *Astrophysical Journal*, 468:51–61.

Gao, S., Schmidt, A. G., and Sass, R. (2009). Hardware implementation of MPI barrier on an FPGA cluster. In *FPL 09: 19th International Conference on Field Programmable Logic and Applications*, pages 12–17.

Garg, T., Wasly, S., Pellizzoni, R., and Kapre, N. (2019). Hoplitebuf: Fpga nocs with provably stall-free fifos. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 222–231. ACM.

Geng, T., Wang, T., Sanaullah, A., Yang, C., Xuy, R., Patel, R., and Herbordt, M. (2018a). FPDeep: A Framework for CNN Training Acceleration on FPGA Clusters. In *Proceedings of IEEE Conference on Field Programmable Logic and Applications (FPL)*.

Geng, T., Wang, T., Sanaullah, A., Yang, C., Xuy, R., Patel, R., and Herbordt, M. (2018b). FPDeep: Acceleration and Load Balancing of CNN Training on FPGA Clusters. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Geng, T., Wang, T., Wu, C., Yang, C., Li, A., Song, S., and Herbordt, M. (2019a). LP-BNN: Ultra-low-Latency BNN Inference with Layer Parallelism. In *Proceedings of International Conference on Application Specific Systems, Architectures, and Processors (ASAP)*.

Geng, T., Wang, T., Wu, C., Yang, C., Wu, W., Li, A., and Herbordt, M. (2019b). O3BNN: An Out-Of-Order Architecture for High-Performance Binarized Neural Network Inference with Fine-Grained Pruning. In *Proceedings of International Conference on Supercomputing (ICS)*.

George, A. (2010). Novo-G Overview. Presentation at CHREC: NSF Center for High-Performance Reconfigurable Computing, 16 June 2010, http://www.chrec.org / george/Novo-G.pdf.

George, A., Herbordt, M., Lam, H., Lawande, A., Sheng, J., and Yang, C. (2016). Novo-G#: A Community Resource for Exploring Large-Scale Reconfigurable Computing Through Direct and Programmable Interconnects. In *Proceedings of IEEE High Performance Extreme Computing Conference (HPEC)*.

George, A., Lam, H., and Stitt, G. (2011). Novo-G: At the Forefront of Scalable Reconfigurable Computing. *Computing in Science and Engineering*, 13(1).

Gidel (2010). PROCStar III.

Giordano, R. and Aloisio, A. (2012). Protocol-independent, fixed latency links with fpga-embedded serdeses. *Journal of Instrumentation*, 7(1).

Glass, C. and Ni, L. (1994). The turn model for adaptive routing. *Journal of the ACM (JACM)*, 41(5):874–902.

Grossman, J., Towles, B., Greskamp, B., and Shaw, D. (2015). Filtering, reductions and synchronization in the Anton 2 network. In *Proc. International Parallel and Distributed Processing Symposium*, pages 860 – 870.

Grubmüller, H., Heller, H., Windemuth, A., and Schulten, K. (1991). Generalized Verlet algorithm for efficient molecular dynamics simulations with long-range interactions. *Molecular Simulation*, 6(1-3):121–142.

Gu, Y. and Herbordt, M. (2007a). Amenability of multigrid computations to FPGA-based acceleration. In *Proceedings of High Performance Embedded Computing Workshop*.

Gu, Y. and Herbordt, M. (2007b). FPGA-based multigrid computations for molecular dynamics simulations. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, pages 117–126.

Gu, Y., VanCourt, T., and Herbordt, M. (2006a). Accelerating molecular dynamics simulations with configurable circuits. *IEE Proceedings on Computers and Digital Technology*, 153(3):189–195.

Gu, Y., VanCourt, T., and Herbordt, M. (2006b). Improved interpolation and system integration for FPGA-based molecular dynamics simulations. In *Proceedings of IEEE Conference on Field Programmable Logic and Applications (FPL)*, pages 21–28.

Gu, Y., VanCourt, T., and Herbordt, M. (2006c). Integrating FPGA acceleration into the ProtoMol molecular dynamics code: Preliminary report. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Gu, Y., VanCourt, T., and Herbordt, M. (2008). Explicit design of FPGA-based coprocessors for short-range force computation in molecular dynamics simulations. *Parallel Computing*, 34(4-5):261–271.

Haile, J. (1997). *Molecular Dynamics Simulation*. Wiley, New York, NY.

Hamada, T. and Nakasato, N. (2005). Massively parallel processors generator for reconfigurable system. *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM).*

Herbordt, M. (2013). Architecture/algorithm codesign of molecular dynamics processors. In *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, pages 1442–1446.

Herbordt, M., Gu, Y., VanCourt, T., Model, J., Sukhwani, B., and Chiu, M. (2008a). Computing models for FPGA-based accelerators with case studies in molecular modeling. *Computing in Science and Engineering*, 10(6):35–45.

Herbordt, M., Kosie, F., and Model, J. (2008b). An efficient $O(1)$ priority queue for large FPGA-based discrete event simulations of molecular dynamics. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, pages 248–257.

Herbordt, M., Model, J., Sukhwani, B., Gu, Y., and VanCourt, T. (2006). Single pass, BLAST-like, approximate string matching on FPGAs. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM).*

Herbordt, M., Model, J., Sukhwani, B., Gu, Y., and VanCourt, T. (2007a). Single pass streaming BLAST on FPGAs. *Parallel Computing*, 33(10-11):741–756.

Herbordt, M. and VanCourt, T. (2005). System and method for programmable logic acceleration of data processing applications and compiler therefore. United States Patent Application Publication. Pub. no.: US2007/0277161 A1.

Herbordt, M., VanCourt, T., Gu, Y., Sukhwani, B., Conti, A., Model, J., and DiSabello, D. (2007b). Achieving high performance with FPGA-based computing. *IEEE Computer*, 40(3):42–49.

Huan, Y. and DeHon, A. (2012). FPGA optimized packet-switched NoC using split and merge primitives. *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pages 47–52.

Humphries, B., Zhang, H., Sheng, J., Landaverde, R., and Herbordt, M. (2014). 3D FFT on a Single FPGA. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM).*

Infiniband (2015). Infiniband Architecture Specification Volume 1, Release 1.3.

Intel (2017). FFT IP Core User Guide.

Intel (2018a). Intel Stratix 10 Device Datasheet.

Intel (2018b). Intel Stratix 10 Embedded Memory User Guide.

Intel (2019). Intel Stratix 10 L- and H-Tile Transceiver PHY User Guide.

Jerger, N., Peh, L., and Lipasti, M. (2008). Virtual circuit tree multicasting: A case for on-chip hardware multicast support. In *2008 International Symposium on Computer Architecture*, pages 229–240. IEEE.

Kapre, N. and Gray, J. (2015). Hoplite: building austere overlay NoCs for FPGAs. *Proceedings of the 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8.

Kapre, N. and Gray, J. (2017). Hoplite: A deflection-routed directional torus noc for fpgas. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 10(2):14.

Kermani, P. and Kleinrock, L. (1979). Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3:267–286.

Kessler, R. E. and Schwarzmeier, J. L. (1993). Cray t3d: A new dimension for cray research. In *Digest of Papers. COMPCON Spring*, pages 176–182. IEEE.

Khan, M., Chiu, M., and Herbordt, M. (2013). Fpga-accelerated molecular dynamics. In Benkrid, K. and Vanderbauwhede, W., editors, *High Performance Computing Using FPGAs*, pages 105–135. Springer Verlag.

Khan, M. and Herbordt, M. (2011). Parallel discrete event simulation of molecular dynamics with speculation and in-order commitment. *Journal of Computational Physics (JCP)*, 230(17):6563–6582.

Khan, M. and Herbordt, M. (2012). Communication requirements for FPGA-centric molecular dynamics. In *Symposium on Application Accelerators for High Performance Computing*.

Kim, J., Dally, W. J., Scott, S., and Abts, D. (2008). Technology-driven, highly-scalable dragonfly topology. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 77–88. IEEE.

Kim, J., Park, D., Theocharides, T., Vijaykrishnan, N., and Das, C. (2005). A low latency router supporting adaptivity for on-chip interconnects. In *Proceedings of the Design Automation Conference*, pages 559–564.

Kindratenko, V. and Pointer, D. (2006). A case study in porting a production scientific supercomputing application to a reconfigurable computer. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, pages 13–22.

Kinsy, M., Cho, M., Shim, K., and Lis, M. (2013a). Optimal and Heuristic Application-Aware Oblivious Routing. *IEEE Transaction on Computers*, 62(1):59–73.

Kinsy, M., Cho, M. H., Wen, T., Suh, E., van Dijk, M., and Devadas, S. (2009). Application-aware deadlock-free oblivious routing. *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 208–219.

Kinsy, M. A., Pellauer, M., and Devadas, S. (2013b). Heracles: A Tool for Fast RTL-based Design Space Exploration of Multicore Processors. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 125–134. ACM.

Komeiji, Y., Uebayasi, M., Takata, R., Shimizu, A., Itsukashi, K., and Taiji, M. (1997). Fast and accurate molecular dynamics simulation of a protein using a special-purpose computer. *Journal of Computational Chemistry*, 18(12):1546–1563.

Lam, S. (1976). Store-and-forward buffer requirements in a packet switching network. *IEEE Transactions on Communications*, 24(4):394–403.

Lawande, A., George, A., and Lam, H. (2016). Novo-G#: a multidimensional torus-based reconfigurable cluster for molecular dynamics. *Concurrency and Computation: Practice and Experience*.

Leiserson, C. E. (1985). Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers*, 100(10):892–901.

Lin, X. and Ni, L. M. (1991). Deadlock-free multicast wormhole routing in multicomputer networks. *ACM SIGARCH Computer Architecture News*, 19(3):116–125.

Liu, X., Deng, Q., and Wang, Z. (2014). Design and fpga implementation of high-speed, fixed-latency serial transceivers. *IEEE Transaction on Nuclear Science*, 61(1).

Mahram, A. and Herbordt, M. (2010). Fast and Accurate NCBI BLASTP: Acceleration with Multiphase FPGA-Based Prefiltering. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 73–82.

Mahram, A. and Herbordt, M. (2012a). CAAD BLASTP 2.0: NCBI BLASTP accelerated with pipelined filters. In *Proceedings of 22nd International Conference on Field Programmable Logic and Applications*, pages 217–223.

Mahram, A. and Herbordt, M. (2012b). FMSA: FPGA-Accelerated ClustalW-Based Multiple Sequence Alignment through Pipelined Prefiltering. In *Proceedings of 20th International Symposium on Field Programmable Custom Computing Machines*, pages 177–183.

Mahram, A. and Herbordt, M. (2016). NCBI BLASTP on High Performance Reconfigurable Computing Systems. *ACM Transcation Reconfigurable Techology and Systems (TRETS)*, 15(4):6.1–6.20.

Malkhi, D., Naor, M., and Ratajczak, D. (2002). Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 183–192. ACM.

Marx, J. N. and Nygren, D. R. (1978). The Time Projection Chamber. *Physics Today*, 31N10:46–53.

Matthey, T. (2004). ProtoMol, an object-oriented framework for prototyping novel algorithms for molecular dynamics. *ACM Transactions on Mechanical Software*, 30(3):237–265.

Mellanox (2003). Introduction to infiniband.

Mellanox (2018). *Introducing 200G HDR InfiniBand Solutions*. Mellanox.

Mello, A., Tedesco, L., Calazans, N., and Moraes, F. (2005). Virtual channels in networks on chip: implementation and evaluation on hermes noc. In *Proceedings of the 18th Annual Symposium on Integrated Circuits and System Design*, pages 178–183. ACM.

Model, J. and Herbordt, M. (2007). Discrete event simulation of molecular dynamics with configurable logic. In *Proceedings of IEEE Conference on Field Programmable Logic and Applications (FPL)*, pages 151–158.

Mohapatra, P. (1998). Wormhole routing techniques for directly connected multicomputer systems. *ACM Computing Surveys (CSUR)*, 30(3):374–410.

Moorthy, P. and Kapre, N. (2015). Zedwulf: Power-Performance Tradeoffs of a 32-Node Zynq SoC Cluster. In *Proceedings of Field-Programmable Custom Computing Machines (FCCM)*, pages 68–75.

Moscibroda, T. and Mutlu, O. (2009). A case for bufferless routing in on-chip networks. *ACM SIGARCH Computer Architecture News*, 37(3):196–207.

NAMD (Accessed 6/2017). *Molecular Modeling in the Cloud*. Theoretical and Computational Biophysics Group, University of Illinois at Urbana-Champaign, http://www.ks.uiuc.edu/Research/cloud/.

Nesson, T. and Johnsson, S. (1995). ROMM routing on mesh and torus networks. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 275–287.

Ni, L. and McKinley, P. (1993). A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76.

Orito, R., Sasaki, O., Kubo, H., Miuchi, K., Nagayoshi, T., Okada, Y., Takada, A., Takeda, A., Tanimori, T., and Ueno, M. (2004). Development of an ASD IC for the micro pixel chamber. *IEEE Transaction on Nuclear Science*, 51(4):1337–1342.

Papamichael, M. and Hoe, J. (2012). CONNECT: re-examining conventional wisdom for designing nocs in the context of FPGAs. *Proceedings of the ACM/SIGDA international symposium on the Field Programmable Gate Arrays (FPGA)*, pages 37–46.

Papamichael, M. and Hoe, J. (2015). The CONNECT Network-on-Chip Generator. *Computer*, 48(12):72–79.

Park, J., Qiu, Y., and Herbordt, M. (2009). CAAD BLASTP: NCBI BLASTP Accelerated with FPGA-Based Pre-Filtering. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, pages 81–87.

Park, J., Qui, Y., and Herbordt, M. (2010). Caad blastn: Accelerated ncbi blastn with fpga prefiltering. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 3797–3800.

Pascoe, C., Lawande, A., Lam, H., George, A., Sun, Y., Farmerie, W., and Herbordt, M. (2010). Reconfigurable supercomputing with scalable systolic arrays and in-stream control for wavefront genomics processing. In *Proceedings of Symposium on Application Accelerators for High Performance Computing.*

Patel, A., Madill, C., Saldana, M., Comis, C., Pomes, R., and Chow, P. (2006). A Scalable FPGA-Based Multiprocessor. In *Proc. Field-Programmable Custom Computing Machines*, pages 111–120.

Peng, Y., Madill, C. A., Na, M. S., Zou, X., and Chow, P. (2014). Benefits of Adding Hardware Support for Broadcast and Reduce Operations in MPSoC Applications. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3).

Pfister, G. F. (2001). An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632.

Porter, G., Strong, R., Farrington, N., Forencich, A., Chen-Sun, P., Rosing, T., Fainman, Y., Papen, G., and Vahdat, A. (2013). Integrating microsecond circuit switching into the data center. *ACM SIGCOMM Computer Communication Review*, 43(4):447–458.

Putnam, A. et al. (2014). A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 13–24.

Reflex (2018). Xpressgxs10-fh20xgt board.

Richer, J., Bourrion, O., Bosson, G., Guillaudin, O., Mayet, F., and Santos, D. (2011). Development and validation of a 64 channel front end ASIC for 3D directional detection for MIMAC. *Journal of Instrumentation*, 6(11):C11016.

Salomon-Ferrer, R., Case, D., and Walker, R. (2013). An overview of the amber biomolecular simulation package. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 3(2):198–210.

Sanaullah, A. and Herbordt, M. (2017). OpenCL for HPC/FPGAs: Case Study with 3D FFT. In *Proceedings of Heterogeneous High Performance Reconfigurable Computing (H2RC)*.

Sanaullah, A. and Herbordt, M. (2018a). An Empirically Guided Optimization Framework for FPGA OpenCL. In *Proceedings of IEEE Conference on Field Programmable Technology (FPT)*.

Sanaullah, A. and Herbordt, M. (2018b). FPGA HPC using OpenCL: Case Study in 3D FFT. In *Proceedings of International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*.

Sanaullah, A. and Herbordt, M. (2018c). Unlocking Performance-Programmability by Penetrating the Intel FPGA OpenCL Toolflow. In *Proceedings of IEEE High Performance Extreme Computing Conference (HPEC)*.

Sanaullah, A., Khoshparvar, A., and Herbordt, M. (2016a). FPGA-Accelerated Particle-Grid Mapping. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Sanaullah, A., Lewis, K., and Herbordt, M. (2016b). Accelerated Particle-Grid Mapping. In *Proceedings of ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

Sanaullah, A., Sachdeva, V., and Herbordt, M. (2018a). SimBSP: Enabling RTL Simulation for Intel FPGA OpenCL Kernels. In *Proceedings of Heterogeneous High Performance Reconfigurable Computing (H2RC)*.

Sanaullah, A., Yang, C., Alexeev, Y., Yoshii, K., and Herbordt, M. (2018b). Real-Time Data Analysis for Medical Diagnosis using FPGA Accelerated Neural Networks. *BMC Bioinformatics*, 19 Supplement 18.

Sass, R. et al. (2007). Reconfigurable computing cluster (RCC) project: Investigating the feasibility of FPGA-based petascale computing. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, pages 127–138.

Schmidt, A., Kritikos, W., Gao, S., and Sass, R. (2012). An evaluation of an integrated on-chip/off-chip network for high-performance reconfigurable computing. *International Journal of Reconfigurable Computing (IJRC)*, 2012.

Schmidt, A., Kritikos, W., Sharma, R., and Sass, R. (2009). Airen: A novel integration of on-chip and off-chip fpga networks. In *Proceedings of the IEEE International Symposium on Field Programmable Custom Computing Machines*, pages 271–274. IEEE.

Scott, S. (1996). Synchronization and communication in the T3E multiprocessor. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 26–36.

Scott, S., Abts, D., Kim, J., and Dally, W. (2006). The blackwidow high-radix clos network. *ACM SIGARCH Computer Architecture News*, 34(2):16–28.

Scott, S. and Thorson, G. (1994). Optimized routing in the Cray T3D. In *Proc. Work. Parallel Computer Routing and Communication*, pages 281–294.

Scrofano, R., Gokhale, M., Trouw, F., and Prasanna, V. (2006). A hardware/software approach to molecular dynamics on reconfigurable computers. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, pages 23–32.

Scrofano, R. and Prasanna, V. (2006). Preliminary investigation of advanced electrostatics in molecular dynamics on reconfigurable computers. In *Proceedings of ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 90–102.

Seo, D., Ali, A., Lim, W., Rafique, N., and Thottethodi, M. (2005). Near-optimal worst-case throughput routing for two-dimensional mesh networks. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 432–443.

Sheng, J. (2017). *High Performance Communication on FPGA-Centric Clusters.* PhD thesis, Department of Electrical and Computer Engineering, Boston University, https://open.bu.edu/handle/2144/27045.

Sheng, J., Humphries, B., Zhang, H., and Herbordt, M. (2014). Design of 3D FFTs with FPGA Clusters. In *Proceedings of IEEE High Performance Extreme Computing Conference (HPEC).*

Sheng, J., Xiong, Q., Yang, C., and Herbordt, M. (2015a). Hardware-Efficient Compressed Sensing Encoder Designs for WBSNs. In *Proceedings of IEEE High Performance Extreme Computing Conference (HPEC).*

Sheng, J., Xiong, Q., Yang, C., and Herbordt, M. (2016a). Collective Communication on FPGA Clusters with Static Scheduling. *Computer Architecture News*, 44(4).

Sheng, J., Yang, C., Caulfield, A., Papamichael, M., and Herbordt, M. (2017). HPC on FPGA Clouds: 3D FFTs and Implications for Molecular Dynamics. In *Proceedings of IEEE Conference on Field Programmable Logic and Applications (FPL)*.

Sheng, J., Yang, C., and Herbordt, M. (2015b). Towards Low-Latency Communication on FPGA Clusters with 3D FFT Case Study. In *Proceedings of International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*.

Sheng, J., Yang, C., and Herbordt, M. (2016b). Application-Aware Collective Communication on FPGA Clusters. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines*.

Sheng, J., Yang, C., and Herbordt, M. (2018a). High Performance Dynamic Communication on Reconfigurable Clusters. In *Proceedings of IEEE Conference on Field Programmable Logic and Applications (FPL)*.

Sheng, J., Yang, C., and Herbordt, M. (2018b). High Performance Dynamic Communication on Reconfigurable Clusters (Extended Abstract). In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Shi, Z. and Burns, A. (2008). Real-time communication analysis for on-chip networks with wormhole switching. In *Second ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, pages 161–170. IEEE.

Singh, A., Dally, W., Gupta, A., and Towles, B. (2003). GOAL: a load-balanced adaptive routing algorithm for torus networks. *Proceedings of International Symposium on Computer Architecture (ISCA)*, 31(2):194–205.

Singh, A., Dally, W., Towles, B., and Gupta, A. (2002). Locality-preserving randomized oblivious routing on torus networks. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, pages 9–13.

Spergel, D. N. (1988). The motion of the Earth and the detection of WIMPs. *Physical Review*, D37:1353.

Stanford Concurrent VLSI Architecture Group (2014). Open Source Networkon-Chip Router RTL.

Stern, J., Xiong, Q., Sheng, J., Skjellum, A., and Herbordt, M. (2017). Accelerating MPI_Reduce with FPGAs in the Network. In *Proceedings of Workshop on Exascale MPI*.

Stern, J., Xiong, Q., Skjellum, A., and Herbordt, M. (2018). A Novel Approach to Supporting Communicators for In-Switch Processing of MPI Collectives. In *Proceedings of Workshop on Exascale MPI*.

Stuecheli, J., Blaner, B., Johns, C., and Siegel, M. (2015). Capi: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 59(1):1–7.

Stunkel, C., Denneau, M., Nathanson, B., Shea, D., Hochschild, P., Tsao, M., Abali, B., Joseph, D., and Varker, P. (1994). Architecture and implementation of vulcan. In *Proceedings of 8th International Parallel Processing Symposium*, pages 268–274. IEEE.

Stunkel, C.B., et al. (1995). The SP2 high-performance switch. *IBM Systems Journal*, 34(2):185–204.

Sukhwani, B. and Herbordt, M. (2008). Acceleration of a Production Rigid Molecule Docking Code. In *Proceedings of IEEE Conference on Field Programmable Logic and Applications (FPL)*, pages 341–346.

Sukhwani, B. and Herbordt, M. (2010). FPGA Acceleration of Rigid Molecule Docking Codes. *IET Computers and Digital Techniques*, 4(3):184–195.

Sullivan, H. and Bashkow, T. R. (1977). A large scale, homogeneous, fully distributed parallel machine, i. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 105–117. ACM.

Top500 (2019). Top500 interconnect family.

Valiant, L. (1982). A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361.

VanCourt, T., Gu, Y., and Herbordt, M. (2004a). FPGA acceleration of rigid molecule interactions. In *Proceedings of IEEE Conference on Field Programmable Logic and Applications (FPL)*.

VanCourt, T. and Herbordt, M. (2004). Families of FPGA-based algorithms for approximate string matching. In *Proceedings of International Conference on Application Specific Systems, Architectures, and Processors (ASAP)*, pages 354–364.

VanCourt, T. and Herbordt, M. (2005a). LAMP: A tool suite for families of FPGA-based application accelerators. In *Proceedings of IEEE Conference on Field Programmable Logic and Applications (FPL)*.

VanCourt, T. and Herbordt, M. (2005b). Three dimensional template correlation: Object recognition in 3D voxel data. In *Proceedings of Computer Architectures for Machine Perception*, pages 153–158.

VanCourt, T. and Herbordt, M. (2006a). Application-dependent memory interleaving enables high performance in FPGA-based grid computations. In *Proceedings of IEEE Conference on Field Programmable Logic and Applications (FPL)*, pages 395–401.

VanCourt, T. and Herbordt, M. (2006b). Rigid molecule docking: FPGA reconfiguration for alternative force laws. *Journal on Applied Signal Processing*, v2006:1–10.

VanCourt, T. and Herbordt, M. (2006c). Sizing of processing arrays for FPGA-based computation. In *Proceedings of IEEE Conference on Field Programmable Logic and Applications (FPL)*, pages 755–760.

VanCourt, T. and Herbordt, M. (2007). Families of FPGA-based accelerators for approximate string matching. *Microprocessors and Microsystems*, 31(2):135–145.

VanCourt, T. and Herbordt, M. (2009). Elements of high performance reconfigurable computing. In Zelkowitz, M., editor, *Advances in Computers*, volume v75, pages 113–157. Elsevier.

VanCourt, T., Herbordt, M., and Barton, R. (2003). Case study of a functional genomics application for an FPGA-based coprocessor. In *Proceedings of IEEE Conference on Field Programmable Logic and Applications (FPL)*, pages 365–374.

VanCourt, T., Herbordt, M., and Barton, R. (2004b). Microarray data analysis using an FPGA-based coprocessor. *Microprocessors and Microsystems*, 28(4):213–222.

Wang, L., Jin, Y., Kim, H., and Kim, E. (2009). Recursive partitioning multicast: A bandwidth-efficient routing for networks-on-chip. In *Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, pages 64–73. IEEE Computer Society.

Wang, T., Geng, T., Jin, X., and Herbordt, M. (2019a). Accelerating AP3M-Based Computational Astrophysics Simulations with Reconfigurable Clusters. In *Proceedings of International Conference on Application Specific Systems, Architectures, and Processors (ASAP)*.

Wang, T., Geng, T., Jin, X., and Herbordt, M. (2019b). FP-AMR: A Reconfigurable Fabric Framework for Block-Structured Adaptive Mesh Refinement Applications. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Xilinx (2009). *Virtex-5 FPGA RocketIO GTP Transceiver*. Xilinx.

Xilinx (2018). *Vivado Design Suite User Guide: High-Level Synthesis*. Xilinx.

Xiong, Q., Bangalore, P., Skjellum, A., and Herbordt, M. (2018a). MPI Derived Datatypes: Performance and Portability Issues. In *Proceedings of the EuroMPI Conference*.

Xiong, Q. and Herbordt, M. (2017). Bonded Force Computations on FPGAs. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Xiong, Q., Skjellum, A., and Herbordt, M. (2018b). Accelerating MPI Message Matching Through FPGA Offload. In *Proceedings of IEEE Conference on Field Programmable Logic and Applications (FPL)*.

Xiong, Q., Yang, C., Patel, R., Geng, T., Skjellum, A., and Herbordt, M. (2019). GhostSZ: A Transparent SZ Lossy Compression Framework with FPGAs. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Yang, C., Geng, T., Wang, T., Patel, R., Xiong, Q., Sanaullah, A., Lin, C., Sachdeva, V., Sherman, W., and Herbordt, M. (2019a). FPGA Molecular Dynamics Simulations. In *Proceedings of ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

Yang, C., Geng, T., Wang, T., Sheng, J., Lin, C., Sachdeva, V., Sherman, W., and Herbordt, M. (2019b). Molecular Dynamics Range-Limited Force Evaluation Optimized for FPGA. In *Proceedings of International Conference on Application Specific Systems, Architectures, and Processors (ASAP)*.

Yang, C., Sheng, J., Patel, R., Sanaullah, A., Sachdeva, V., and Herbordt, M. (2017). OpenCL for HPC with FPGAs: Case Study in Molecular Electrostatics. In *Proceedings of IEEE High Performance Extreme Computing Conference (HPEC)*.

Young, C., Bank, J., Dror, R., Grossman, J., Salmon, J., and Shaw, D. (2009). A 32x32x32, spatially distributed 3D FFT in four microseconds on Anton. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pages 1–11.

CURRICULUM VITAE