UNIVERSITY OF CALIFORNIA

Los Angeles

# Communication Optimization for Customizable Domain-Specific Computing

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Electrical Engineering

by

**Bingjun Xiao**

2015

Abstract of the Dissertation

# Communication Optimization for Customizable Domain-Specific Computing

by

**Bingjun Xiao**

Doctor of Philosophy in Electrical Engineering

University of California, Los Angeles, 2015

Professor Jingsheng Jason Cong, Chair

This dissertation investigates the communication optimization for customizable domain-specific computing at different levels in a customizable heterogeneous platform (CHP) to improve the system performance and energy efficiency.

*Fabric-level optimization driven by emerging devices.* Programmable fabrics (e.g., FPGAs) can be used to improve domain-specific computing by >10x in energy efficiency over CPUs since FPGAs can be customized to the application kernels in the target domain. But the programmable interconnects inside FPGAs occupy >50% of the FPGA area, delay and power. We propose a novel architecture of programmable interconnects based on resistive RAM (RRAM), a type of emerging device with high density and low power. We optimize the layout and the programming circuit of the new architecture. We also extend RRAM benefits to routing buffers. We observe the high defect rate in the emerging RRAM manufacturing and further develop a defect-aware communication mechanism. Conventional defect avoidance leaves a large portion of the chip in the new architecture unusable. So we propose new defect utilization methodologies by treating stuck-closed defects as shorting constraints in the routing of signals. We develop a scalable algorithm to perform timing-driven routing under these extra constraints and successfully suppress the impact of defects.

*Chip-level optimization driven by accelerator-centric architectures.* A chip can also be customized to an application domain by integrating a sea of accelerators designed for the frequently used kernels in the domain. The design of interconnects among customized accelerators and shared resources (e.g., shared memories) is a serious challenge in chip design. Accelerators run 100x faster than CPUs and post a high data demand on the communication infrastructure. To address this challenge, we develop a novel design of interconnects between accelerators and shared memories and exploit several optimization opportunities that emerge in accelerator-rich computing platforms. Experiments show that our design outperforms prior work that was optimized for CPU cores or signal routing. Another design challenge lies in the data reuse optimization within an accelerator to minimize its off-chip accesses and on-chip buffer usage. Since the fully pipelined computation kernel consumes large amounts of data every clock cycle, and the data access pattern is the major difference among applications, existing accelerators use ad hoc data reuse schemes that are carefully tuned per application to fit the data demand. To reduce the engineering cost of accelerator-rich architectures, we develop a data reuse infrastructure that is generalized for the stencil computation domain and can be instantiated to the optimal design for any application in the domain. We demonstrate the robustness of our method over a set of real-life benchmarks.

*Server-level and cluster-level optimization driven by big data.* In the era of big data, workloads can no longer fit into a single chip. Most data are stored in disks, and we can only load a small part of it into main memories during computation. Due to the low access speed of disks, our primary design goal becomes minimization of the data transfer between disks and the main memory. We select a popular big data application, convolutional neural network (CNN), as a case study. We analyze the linear algebraic properties of CNN, and propose algorithmic modifications to reduce the total computational workload and the disk access. Furthermore, when the application data become even larger, it needs to be distributed among a cluster of server nodes. This motivates us to develop an accelerator-centric computing cluster. We test two machine learning applications, logistic regression and artificial neural network (ANN), on our prototyping cluster and try to minimize the total data transfer incurred during the computation in this cluster. We select the distributed stochastic gradient descent (dSGD) as our training algorithm to eliminate the inter-node communication within a training iteration. We also deploy an in-memory cluster computing infrastructure, Spark, to eliminate the inter-node communication across training iterations. The baseline Spark only supports CPUs, and we develop a software layer to allow Spark tasks to offload their major computation to accelerators which are equipped by each server node. During the computation offloading, we group multiple tasks into a batch and transfer it to the target accelerator in one transaction to minimize the setup overhead of the data transfer between accelerators and host servers. We further realize accelerator data caching to eliminate the unnecessary data transfer of training data based on the properties of iterative machine learning applications.

The dissertation of Bingjun Xiao is approved.

Tyson Condie

Dejan Markovic

Mau-Chung Frank Chang

Jingsheng Jason Cong, Committee Chair

University of California, Los Angeles

2015

*To my family*

*for their constant source of*

*love, concern, support, and strength all these years.*

# Table of Contents

**1 Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1**

1.1 Customizable Domain-Specific Computing . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Communication Optimization for Customizable Domain-Specific Computing . . . 4

1.2.1 Programmable Interconnects Driven by Emerging Devices . . . . . . . . . 4

1.2.2 Defect-Aware Routing on Nanodevice-Based Programmable Interconnects 6

1.2.3 Optimized Interconnects Between Accelerators and Shared Memories . . . 8

1.2.4 Data Reuse Optimization for Stencil Access Patterns . . . . . . . . . . . . 10

1.2.5 Computation and Communication Optimization in Convolutional Neural Network Driven by Big Data . . . . . . . . . . . . . . . . . . . . . . . . 11

1.2.6 Accelerator-Centric Cluster Computing with Communication Optimization 12

**2 Programmable Interconnects Driven by Emerging Devices** . . . . . . . . . . . . . **16**

2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

2.2 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

2.2.1 Conventional FPGA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

2.2.2 FPGAs with Emerging NVMs . . . . . . . . . . . . . . . . . . . . . . . . 20

2.3 The FPGA-RPI Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

2.3.1 Choice of NVMs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

2.3.2 Overall Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

2.3.3 RRAM-Friendly Layout Design . . . . . . . . . . . . . . . . . . . . . . . 24

2.3.4 Programming Schematic . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

2.3.5 On-Demand Buffering Architecture . . . . . . . . . . . . . . . . . . . . . 33

2.4 Design Tool Support . . . . . 41

2.4.1 CAD Flow . . . . . 41

2.4.2 Equivalent Circuit Model for Interconnect . . . . . 42

2.4.3 On-Demand Buffer Allocation Algorithm . . . . . 43

2.4.4 VPR-RPI: the P&R Tool for FPGA-RPI . . . . . 44

2.5 Simulation Results . . . . . 46

2.5.1 Settings . . . . . 46

2.5.2 Optimize the On-Demand Buffering Architecture . . . . . 47

2.5.3 Evaluation of FPGA-RPI . . . . . 48

2.5.4 Impact on LB Architectural Design . . . . . 52

2.5.5 Sensitivity Analysis . . . . . 52

2.6 Interconnect Improvement in LBs . . . . . 54

2.7 Conclusion and Future Work . . . . . 56

**3 Defect-Aware Routing on Nanodevice-Based Programmable Interconnects . . . . . 57**

3.1 Introduction . . . . . 57

3.2 Quantitative Defect Analysis . . . . . 60

3.2.1 Impact on Routing . . . . . 60

3.2.2 Impact on Placement . . . . . 63

3.3 Defect-Tolerant Routing . . . . . 64

3.3.1 Enforcement of Shorting Constraints . . . . . 65

3.3.2 Prune Invalid Solutions Before Routing . . . . . 67

3.3.3 Smart Mapping of ES-Clusters to Nets . . . . . 68

3.3.4 Implementation . . . . . 71

3.4 Defect-Tolerant Placement . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
3.4.1 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
3.4.2 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . 75
3.5 Simulation Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
3.5.1 Settings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
3.5.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
3.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83

**4 Optimized Interconnects Between Accelerators and Shared Memories . . . . . . . 84**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
4.2 Preliminary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
4.2.1 Interconnect Requirement Between Accelerators and Shared Memories . . 86
4.2.2 Configurable Crossbar to Meet Accelerator Demand . . . . . . . . . . . 87
4.2.3 Problem Formulation . . . . . . . . . . . . . . . . . . . . . . . . 88
4.3 Optimization Inspired by Accelerators with Multiple Ports . . . . . . . . . . . 89
4.3.1 Identification of Unnecessary Switches . . . . . . . . . . . . . . . . . 89
4.3.2 Minimum Cost of Accelerator Sub-Crossbar . . . . . . . . . . . . . . . 90
4.3.3 Sub-Crossbar Design for Routing Flexibility . . . . . . . . . . . . . . . 91
4.3.4 Virtualized Interface for Global Optimization Among Accelerators . . . . . 93
4.4 Optimization Inspired by Accelerators in Dark Silicon Age . . . . . . . . . . . 94
4.4.1 Minimal Design with 100% Routability . . . . . . . . . . . . . . . . . 94
4.4.2 Design with 100% Routability when Optimality is Unreachable . . . . . . 98
4.4.3 Smaller Design with 95% Routability . . . . . . . . . . . . . . . . . . 102
4.5 Optimization Inspired by Accelerator Heterogeneity . . . . . . . . . . . . . . . . . 105

4.5.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105

4.5.2 Methodology and Effectiveness . . . . . . . . . . . . . . . . . . . 106

4.6 Verification and Comparisons . . . . . . . . . . . . . . . . . . . . . . 108

4.6.1 Verification on Commodity FPGA . . . . . . . . . . . . . . . . . 108

4.6.2 Comparison with Prior Work . . . . . . . . . . . . . . . . . . . . 109

4.7 Conclusion and Future Work . . . . . . . . . . . . . . . . . . . . . . 109

**5 Data Reuse Optimization for Stencil Access Patterns . . . . . . . . . . . . . . . . . 112**

5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112

5.2 Preliminary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113

5.2.1 Stencil Computation . . . . . . . . . . . . . . . . . . . . . . . . 113

5.2.2 Microarchitecture for Stencil Accesses . . . . . . . . . . . . . . . 115

5.2.3 Design Objectives . . . . . . . . . . . . . . . . . . . . . . . . . 117

5.2.4 Polyhedral Model . . . . . . . . . . . . . . . . . . . . . . . . . . 119

5.3 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123

5.3.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123

5.3.2 Denotations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124

5.3.3 Working Principle . . . . . . . . . . . . . . . . . . . . . . . . . 124

5.3.4 Insights Gained From RTL Simulation . . . . . . . . . . . . . . . 129

5.3.5 Other Design Issues . . . . . . . . . . . . . . . . . . . . . . . . 131

5.3.6 System Integration . . . . . . . . . . . . . . . . . . . . . . . . . 132

5.3.7 Enabling Bandwidth/Memory Trade-off . . . . . . . . . . . . . . . 134

5.4 Design Automation Flow . . . . . . . . . . . . . . . . . . . . . . . . . 136

5.5 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137

5.5.1 Experiment Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137

5.5.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 138

5.6 Conclusions and Future Work . . . . . . . . . . . . . . . . . . . . . . . . 140

**6 Computation and Communication Optimization in Convolutional Neural Networks for Big Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 142**

6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 142

6.2 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143

6.2.1 Algorithm Review of CNNs . . . . . . . . . . . . . . . . . . . . . . . 143

6.2.2 Architecture of Real-Life CNN . . . . . . . . . . . . . . . . . . . . . 145

6.2.3 Runtime Breakdown of Real-Life CNN . . . . . . . . . . . . . . . . . 146

6.3 Properties of CNN Computation . . . . . . . . . . . . . . . . . . . . . . . 146

6.3.1 Another View of CNN Computation . . . . . . . . . . . . . . . . . . . 146

6.3.2 Enabling New Optimization Opportunities . . . . . . . . . . . . . . . 148

6.3.3 Properties of Convolutional MM . . . . . . . . . . . . . . . . . . . . 148

6.4 Computation Optimization . . . . . . . . . . . . . . . . . . . . . . . . . . 149

6.4.1 Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149

6.4.2 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . . 151

6.5 Communication Optimization . . . . . . . . . . . . . . . . . . . . . . . . 153

6.5.1 Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153

6.5.2 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . . 156

6.5.3 Combining with Strassen Algorithm . . . . . . . . . . . . . . . . . . . 156

6.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158

**7 Accelerator-Centric Cluster Computing with Communication Optimization . . . . 160**

7.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 160
7.2 MapReduce System with FPGAs . . . . . . . . . . . . . . . . . . . . . . . . 163
7.2.1 Choice of Accelerators . . . . . . . . . . . . . . . . . . . . . . . . . . 163
7.2.2 Target Cluster Architecture . . . . . . . . . . . . . . . . . . . . . . . 165
7.2.3 Execution Flow Overview . . . . . . . . . . . . . . . . . . . . . . . . 166
7.2.4 FPGA Management . . . . . . . . . . . . . . . . . . . . . . . . . . . 167
7.3 Communication Optimization in Cluster Computing . . . . . . . . . . . . . . . . . 172
7.3.1 Application Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . 172
7.3.2 Reducing Inter-Node Communication . . . . . . . . . . . . . . . . . . . 173
7.3.3 Reducing File Accesses . . . . . . . . . . . . . . . . . . . . . . . . . 174
7.3.4 Reducing Data Transfers between CPUs and FPGAs . . . . . . . . . . . . 174
7.4 Prototyping and Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . 175
7.4.1 Prototyping Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . 175
7.4.2 System Validation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 176
7.4.3 Communication Optimization Breakdown . . . . . . . . . . . . . . . . . 177
7.4.4 Runtime Breakdown . . . . . . . . . . . . . . . . . . . . . . . . . . . 178
7.4.5 Resource Management for Multiple Jobs . . . . . . . . . . . . . . . . . 179
7.4.6 System Scalability . . . . . . . . . . . . . . . . . . . . . . . . . . . . 182
7.5 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 182

**8 Summary and Concluding Remarks . . . . . . . . . . . . . . . . . . . . . . . 184**
8.1 Technology Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 184
8.2 Accumulative Impact . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 185
8.3 Design Automation Tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . 188

8.4 Open Issues . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 191

**References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 192**

# List of Figures

1.1 Customizable heterogeneous platform for domain-specific computing. . . . . . . . 2

1.2 Rough energy costs for various operations in 45nm 0.9V, adopted from [1]. . . . . 3

2.1 A typical FPGA architecture. LB = logic block (or CLB). CB = connection block. SB = switch block. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

2.2 Area breakdown of different components in an FPGA based on an architectural model [2] that mimics the Xilinx Virtex7 FPGAs [3]. All three components (SRAM bits, routing switches and routing buffers) take up significant area in programmable interconnects. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

2.3 FPGA with emerging nonvolatile memories. . . . . . . . . . . . . . . . . . . . . . 20

2.4 In FPGA-RPI, switch blocks and connection blocks in transistor-less programmable interconnects are placed over logic blocks in the same die according to existing RRAM fabrication structures [4, 5, 6, 7]. . . . . . . . . . . . . . . . . . . . . 24

2.5 Overview of FPGA-RPI where the FPGA area is mainly contributed by logic blocks instead of programmable interconnects. . . . . . . . . . . . . . . . . . . . . 25

2.6 Circuit schematics of transistor-less programmable interconnects. . . . . . . . . . . 25

2.7 An RRAM-friendly layout design of the programmable interconnects in FPGA-RPI using the RRAM fabrication structure shown in Fig. 2.4. Here, each metal via (marked as a red point) at the intersection between wires of M9 and M5–M8 refers to a vertical connection between them through metal via(s). . . . . . . . . . . . 26

2.8 An illustration of how via blockages are avoided by metal wires in our layout design of switch blocks. (a) An abstract structure of FPGA-RPI switch blocks (M9 and M8 are omitted here for clarity). (b) 3D view to show how via blockages are avoided by metal wires. The RRAM layer is omitted here for clarity. For demonstration purposes, vertical connections are bounded in a plane to construct the case most challenging for via blockage avoidance. . . . . . . . . . . . . . . . . 27

2.9 Programming circuits for RRAMs in interconnects. The two programming transistors will apply programming voltages to program the RRAM between them. $V_p$ is the threshold voltage that switches the RRAM state. . . . . . . . . . . . . . . . . 29

2.10 In an existing work [8], sharing paths are blocked by transistors (marked as dotted lines). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

2.11 In our work, one programming transistor can be shared by RRAMs among switch blocks and connection blocks. . . . . . . . . . . . . . . . . . . . . . . . . . 31

2.12 Row/column addressing for the programming transistor array of RRAMs in interconnects. If two programming transistors in one array are selected simultaneously, two other transistors will be selected parasitically. . . . . . . . . . . . . . . . . . . . 32

2.13 Two solutions to solve parasitic selection: (a) double programming transistors at each programming node, and (b) partition programming transistors into five banks, where the two programming transistors at the terminals of any arbitrary RRAM lie in two different banks. . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

2.14 The circuit structure and layout of the on-demand buffering architecture in our FPGA-RPI. R = repeater. . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

2.15 Delay distribution of all paths in a typical application for mapping onto FPGA. Most paths can be relaxed for less use of buffers. . . . . . . . . . . . . . . . . . . 37

2.16 Performance comparison between the fixed buffer patten in conventional FPGAs and the on-demand buffering architecture in FPGA-RPI. To simplify, only three routings tracks per channel are shown for the conventional FPGA, and one track is shown for our FPGA-RPI. Note that the buffers in FPGA-RPI are also shared by other tracks in the same channel (not shown here but discussed in Section 2.3.5.1). ✓= buffer allocated. ✗= not allocated. . . . . . . . . . . . . . . . . . . . . . . . 38

2.17 Illustration of glitch elimination. Glitches $< 0.5t_d$ are filtered out by the RC network of wire segments before regenerative feedback repeaters. Our on-demand buffering architecture always guarantee sufficient RC delay of wires for this filtering. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

2.18 SPICE simulation results of the scenario in Fig. 2.17. Simulation settings can be found in Section 2.5.1. Wire segments are modeled as a distributed RC network. . 40

2.19 Customized CAD flow for FPGA-RPI. An enhanced P&R tools is developed for FPGA-RPI. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

2.20 Extracted equivalent circuit model of the FPGA-RPI routing structure and comparison with a conventional FPGA. . . . . . . . . . . . . . . . . . . . . . . . 42

2.21 Overview of the two tools, VPR and VPR-RPI. . . . . . . . . . . . . . . . . . . . 45

2.22 View of the buffer allocation result generated by VPR-RPI. . . . . . . . . . . . . 46

2.23 Exploration of the impact of buffer sizing and richness on the area-delay product. Buffer sizes and area-delay products are normalized. . . . . . . . . . . . . . . . . 48

2.24 Area comparison of programmable interconnects in a single tile. . . . . . . . . . 49

2.25 Comparison of the delays contributed by programmable interconnects in critical paths in FPGAs over benchmarks. . . . . . . . . . . . . . . . . . . . . . . . 50

2.26 Comparison of the leakage powers contributed by programmable interconnects in a single tile. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

2.27 Comparison of the dynamic powers contributed by programmable interconnects in FPGAs over benchmarks. To separate the impact of the clock period from the dynamic powers, we use pJ/cycle as the metric. . . . . . . . . . . . . . . . . . . . . 51

2.28 Comparisons of FPGA architectures with more complicated LBs and simpler LBs. 53

2.29 Sensitivity analysis of the RRAMs on/off ratio for FPGA-RPI. Each line corresponds with a fixed on/off ratio. There is a tradeoff between delay and power savings on each line. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

2.30 The internal structure of a state-of-art LB described in [2]. . . . . . . . . . . . . . 55

2.31 Area comparison of a single tile in an FPGA. The white parts belong to LBs. Each LB contains eight 6-LUTs. The number of LB inputs is 27, optimized by design experience in [9]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

3.1 Illustration of nanodevices. (a) Hysteresis characteristic of a two-terminal RRAM nanodevice. (b) Function as a routing switch in place of a pass transistor and its six-transistor SRAM cell. . . . . . . . . . . . . . . . . . . . . . . . . . . 57

3.2 Illustration of stuck-open and stuck-closed defects of nanodevice-based routing switches in programmable interconnects. LB $\Rightarrow$ logic block. . . . . . . . . . . . 61

3.3 Solve a stuck-closed defect by simple defect avoidance. All of the 15 switches shown in this figure need to be discarded due to a single stuck-closed switch. . . . 61

3.4 Impact comparison of the stuck-open and stuck-closed defects on routability. Delay going down to zero $\Rightarrow$ unroutable. $\sim$10x gap observed between the impacts of the two defect types. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

3.5 The number of logic blocks (LBs) with shorted pins over different stuck-closed defect rates and numbers of routing tracks in channel. . . . . . . . . . . . . . . . 64

3.6 Example of a routing graph with shorting constraints. Bold red arrows indicate shorting edges. Nodes $\{c, d, e, g, h\}$ shorted by the shorting edges form an electrically shorted cluster (ES-cluster) as highlighted. . . . . . . . . . . . . . . . . . . . 65

3.7 Our extension of the congestion concept for routing under shorting constraints. (a) Add node c to a routing tree. (b) Recursively add all the successor nodes and increase congestions. (c) Add node d to another routing tree. (d) Replace the congestion cost with the effective cost for precedent nodes, e.g., node e. . . . . . . 66

3.8 Example of fast pruning of invalid routing solutions. Any routing solution that maps track 1 to the net in set $\overline{\{i, j, k\}}$ can be pruned ahead of time. . . . . . . . . 67

3.9 To route a net from $s_2$ to $t_{21}$, there exist two shortest paths—one via node d and the other via node e as highlighted. We guide the router to route via node e for better utilization of resources. . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

3.10 A distribution of ES-clusters with different sizes in a defective nanodevice-based FPGA. Along with it is an example of the different potentials of small ES-clusters and large ES-clusters exposed to the same nets. . . . . . . . . . . . . . . . . . . . 69

3.11 Example of the challenge in the placement of logic blocks with shorted pins. (a) A netlist. (b) Logic blocks with shorted pins (checked pad ⇒ output pin). Though the connections of (b,d) and (b,c) in the netlist match the shorted pins of (B,D) and (B,C) respectively, the connection of (b,c,d) does not match (B,C,D) but matches (B,E,D). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

3.12 Distribution of logic blocks over different numbers of used inputs in a netlist after logic synthesis. Logic synthesis is performed by the Berkeley ABC tool [10] using a 4-LUT FPGA library. Pins are not fully utilized in many logic blocks. . . . . . 74

3.13 (a) Count of ES-clusters over different numbers of shorted pins. (b) Distribution of physical logic blocks (PLBs) over different numbers of active inputs (i.e., not disabled) after pin disabling. . . . . . . . . . . . . . . . . . . . . . . . . . . . 75

3.14 Area usage of benchmarks after placement. . . . . . . . . . . . . . . . . . . . . . 80

3.15 Routability of benchmarks using simple defect avoidance and our adaptive defect recovery. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80

3.16 Critical delay of benchmarks after routing. . . . . . . . . . . . . . . . . . . . . . . 81

3.17 A comparison of runtime complexity. The scales of benchmarks are evaluated as the number of nodes in the form of an and-inverter graph (AIG) of benchmarks. . . 82

3.18 Yield comparison over multiple defect rates. . . . . . . . . . . . . . . . . . . . . . 82

4.1 Difference between memory sharing among general-purpose CPU cores and among accelerators. (a) A simple interconnect for CPU cores. (b) Demanding interconnects for accelerators. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86

4.2 Interconnects designed as a configurable crossbar between accelerators and shared memories to keep data access cost small. ✓ : switch turned on. . . . . . . . . . . 88

4.3 As the multiple ports of the same accelerator are powered on/off together, it is not necessary to provide switches to connect them to the same memory banks. Dashed switches with (x,y) coordinates (1,1), (2,1), (4,2), (6,3), (5,4) can be removed. This shows the different impact of ports from the same accelerator and those from different accelerators on the crossbar design. . . . . . . . . . . . . . . . . . . . . 90

4.4 Our design with interleaving connections between an accelerator and its accessible memory range. (a) Switch positions in the sub-crossbar of the accelerator. (b) Connectivity graph. Any three continuous memory banks can be routed to the three data ports of the accelerator via separate paths. (c) Our interconnect optimization within the single accelerator can be virtualized as an interval (i.e., its accessible memory range) in which any segment with length of 3 (memory banks) can be routed to the accelerator. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92

4.5 The interface for global optimization among accelerators. (a) Global optimization only needs to consider the lengths and positions of accessible memory ranges of accelerators to reduce the total number of switches while keeping a high probability of finding segments without overlaps. (b) If the multiple data ports of an accelerator are grouped into a larger multi-channel port to converge the interconnect design to the CPU scenario, we will lose a large design space and get an inefficient design. 93

4.6 Minimal crossbar design that targets the case of six accelerators in total, and at most four accelerators powered on due to the power budget in dark silicon age. All the $C_6^4 = 15$ possible combinations of four accelerators out of the six can be routed to shared memories. Removal of any switch will break the 100% routability guarantee. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95

4.7 Crossbar design when the number of memory banks increases from the minimum eight in Fig. 4.6 to ten. Since the accessible memory ranges of accelerators are scattered, there are fewer conflicts on data paths. The number of switches can be reduced to 18 while still keeping the 100% routability. . . . . . . . . . . . . . . . 99

4.8 The average number of switches per accelerator port in crossbars generated by Algorithm 6. In this experiment, we initially set $k = 200$, $n = 10$, $c = 10$, and $m = 300$. (a) Sweep $m$ from 100 to 500. (b) Sweep $k$ from 200 to 2,000. . . . . . 100

4.9 Removal of two switches from Fig. 4.7 to sacrifice routability for switch savings. In this design, the accelerators 2–5 can still be routed to shared memories after switch removal. Out of all the $C_6^4 = 15$ possible combinations of four accelerators powered on, only the accelerator set $\{1, 2, 3, 4\}$ and $\{3, 4, 5, 6\}$ cannot be routed. . 103

4.10 Methodology of switch count reduction by horizontal transformation of accessible memory ranges of accelerators. The number of switches reduced in this process can be precisely calculated as $d \times k$ (the area difference between the outer parallelogram and inner parallelogram). . . . . . . . . . . . . . . . . . . . . . . . . . 104

4.11 Experimental results on routability after reduction of switch count. In this experiment, $k = 200$, $n = 10$, $c = 10$, and $m = 300$. . . . . . . . . . . . . . . . . . . . . 104

4.12 Optimization opportunities brought by accelerator heterogeneity. (a) Execution correlations among heterogeneous accelerators. Accelerators within the same application domain have a higher probability of being powered on or off together. (b) We can scatter the accelerators within the same application domain throughout the shared memories so that they have fewer overlaps in their accessible memory range and will be easier to route. . . . . . . . . . . . . . . . . . . . . . . . . . . 106

4.13 30% reduction of switch count at 95% routability after optimization of accelerator mapping based on accelerator heterogeneity. . . . . . . . . . . . . . . . . . . . . . 107

4.14 Comparison of our crossbar design with the design in [11]. Our crossbar design outperforms the work in [11] since the work in [11] performs optimization for general-purpose signal routing; our crossbar design fully exploits the optimization opportunities that emerge in accelerator-rich computing platforms as identified in this work. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110

5.1 Iteration domain of the example stencil computation in Listing 5.1. . . . . . . . . . 115

5.2 The overall architecture of our microarchitecture for stencil computation. It decouples stencil accesses from computation. . . . . . . . . . . . . . . . . . . . . . 116

5.3 Even for a constant stencil window in Fig. 5.1, the number of banks varies as the row size of the data grid changes [12]. . . . . . . . . . . . . . . . . . . . . . . . . 118

5.4 Example stencil windows where more banks are needed than the # of array references in [13, 14]. (a) 4-point stencil in ‘BICUBIC’ [15]. (b) 4-point stencil in ‘RICIAN’ [16]. (c) 19-point stencil in ‘SEGMENTATION_3D’ [14]. . . . . . . . . 119

5.5 The example circuit structure of our memory system generated for array $A$ in the stencil computation of Listing 5.1. . . . . . . . . . . . . . . . . . . . . . . . . . . 123

5.6 The dependency graph of data filters 'filter_$x$' and 'filter_$y$' where $x < y$, with polyhedral expressions of dependency relations appended. . . . . . . . . . . . . . 127

5.7 Non-rectangular iteration domain of an example application with dynamically changeable reuse distance. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131

5.8 Structure of the data filter which can be applied to general polyhedral data domains. 132

5.9 Integration of an accelerator with our microarchitecture in a system. . . . . . . . . 133

5.10 Breaking the chain at the largest reuse buffer when the number of off-chip accesses is increased. (a) An example of the original microarchitecture with one off-chip access per cycle. (b) The new microarchitecture with two off-chip accesses per cycle, but smaller total buffer size. . . . . . . . . . . . . . . . . . . . . . . . . 135

5.11 Graceful degradation of on-chip memory usage with the increase of off-chip bandwidth for (a) the example in Fig. 5.10, and (b) the 19-point stencil in SEGMENTATION in Fig. 5.4(c). . . . . . . . . . . . . . . . . . . . . . . . . . . . 135

5.12 Design automation flow of accelerator generation for stencil computation. . . . . . 136

6.1 An example of a convolutional neural network. . . . . . . . . . . . . . . . . . . . 143

6.2 Example loop-nest representing the computation in a convolution layer of a CNN. . 144

6.3 A real-life CNN that was used in the 2012 ImageNet contest [17]. . . . . . . . . . 145

6.4 Comparison of our convolutional MM with the normal MM in terms of the savings of GFLOPS by the Strassen algorithm. . . . . . . . . . . . . . . . . . . . . . . . 151

6.5 A sensitivity study on convolution kernel size and feature map size. . . . . . . . . 152

6.6 Partitioning of large matrices into smaller data blocks that can be loaded in the main memory for computation. . . . . . . . . . . . . . . . . . . . . . . . . . . 154

6.7 A comparison of disk accesses using the conventional uniform matrix partitioning and our optimal nonuniform matrix partitioning for CNNs. . . . . . . . . . . . . 156

6.8 The Strassen algorithm before communication optimization. The different dashed lines indicate the matrix partitioning af the different levels of the recursions. . . . . 157

6.9 The Strassen algorithm after our communication optimization. The solid lines indicate the matrix partitioning to optimize the ratio of matrix sizes. . . . . . . . . 157

6.10 A comparison of disk accesses using the Strassen algorithm with and without communication optimization. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158

7.1 Implementations of the sum of vector elements on (a) an FPGA, and (b) a GPU. . . 164

7.2 Our target cluster architecture. . . . . . . . . . . . . . . . . . . . . . . . . . . 165

7.3 The overall flow of our MapReduce system with FPGA accelerators. . . . . . . . . 166

7.4 The task queues maintained in our FPGA agent for resource scheduling. . . . . . . 168

7.5 The flowchart of our FPGA agent to process acceleration requests from FPGA clients. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 169

7.6 The flowchart of our FPGA agent to get the computation done as required by the kernel acceleration API called by users. . . . . . . . . . . . . . . . . . . . . . . 171

7.7 A photograph of our cluster with FPGA devices. . . . . . . . . . . . . . . . . . . . 175

7.8 The classification accuracy during the logistic regression and neural network training running on our MapReduce system with FPGA accelerators. . . . . . . . . . . 176

7.9 Communication optimization breakdown. . . . . . . . . . . . . . . . . . . . . . . 177

7.10 Runtime breakdown of an application running on our MapReduce system with FPGA accelerators. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 178

7.11 Latency of each job when we simultaneously run multiple jobs in our MapReduce system with FPGA accelerators. . . . . . . . . . . . . . . . . . . . . . . . . . . 179

7.12 Latency of each job when we simultaneously run multiple jobs with different priorities in our MapReduce system with FPGA accelerators. . . . . . . . . . . . . . 180

7.13 FPGA utilization of each job when we run parallel jobs in our system. . . . . . . . 181

7.14 The performance of our MapReduce system with FPGA accelerators scales with the number of nodes included in the cluster. . . . . . . . . . . . . . . . . . . . . . . 182

8.1 A generalized case study that can benefit from all the communication optimization technologies proposed in this thesis. . . . . . . . . . . . . . . . . . . . . . . . . . 185

8.2 Power measurement of our system. . . . . . . . . . . . . . . . . . . . . . . . . . . 186

8.3 Step-by-step optimization breakdown of the example in Fig. 8.1 by the different technologies proposed in this thesis. . . . . . . . . . . . . . . . . . . . . . . . . . 189

8.4 The general framework of our design automation tools with the integration of communication optimization. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 190

# List of Tables

2.1 Comparison of programming transistor counts in programmable interconnects based on CMOS, the schematic in [8], and this work. $N$ represents the routing channel width. $M$ represents the number of logic block pins accessible by one routing channel. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

3.1 Denotation of settings for defect impact analysis. . . . . . . . . . . . . . . . . . . . 62

3.2 Denotation table for defect-tolerant routing. . . . . . . . . . . . . . . . . . . . . . . . 71

4.1 Comparison of private memories, shared memories via buses [18], and our crossbar in FPGA implementation and testing. . . . . . . . . . . . . . . . . . . . . . . . . 108

5.1 Denotations used in the working principle of our memory system. . . . . . . . . . 124

5.2 Reuse FIFOs with nonuniform sizes calculated from maximum reuse distances of adjacent array references and mapped to different physical implementations (block memory, distributed memory, or register) if targeted an FPGA platform. . . . . . 126

5.3 The execution flow of our microarchitecture in the example of Listing 5.1. The latency among the data streams at different modules is ignored here for demonstration purpose only. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129

5.4 High-level partitioning results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 138

5.5 Synthesis experimental results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 139

6.1 Breakdown of CNN runtime in the recognition of 256 images. . . . . . . . . . . . 146

6.2 The FLOPS comparison of different operations in the normal MM and our convolutional MM. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 151

6.3 Workload reduction by extending the Strassen algorithm to the real-life CNN in Fig. 6.3 via our convolutional MM. . . . . . . . . . . . . . . . . . . . . . . . . 153

8.1 Summary of proposed communication optimizations for customizable domain-specific computing. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 184

8.2 Power numbers of different components at different working states measured by a power meter for the estimation of energy consumption. . . . . . . . . . . . . . . . . 187

## ACKNOWLEDGMENTS

Here I would like to offer my sincerest gratitude to my advisor, Professor Jason Cong. Professor Cong is a Chancellor's Professor at the Computer Science Department and Electrical Engineering Department of UCLA, Director of the Center for Customizable Domain-Specific Computing, and Co-Director of the UCLA VAST Lab. His insightful guidance, professional advice, and constant support helped me throughout the time of my PhD study at UCLA. I would not have reached so far without our weekly group meetings, offline discussions, and email exchanges which inspired my research ideas and pinpointed my focus. From him I learned not only ideas and ways to conduct research, but also many other things beyond research. It is my great fortune and honor to have him as my adviser.

I am also very grateful to my doctoral committee members, Professor M.-C. Frank Chang, Professor Dejan Markovic, and Professor Tyson Condie, for their time, interest, and insightful comments. Their rich experience in circuit design, architecture design, and system design brought further improvements to this thesis.

I would like to thank Janice Wheeler for her help in editing the language of my papers and this thesis. Whenever a conference submission deadline approaches, Janice receives many paper drafts to edit. There were several times when I sent her my paper drafts rather late. This left her with a tight time constraint, but she always worked very hard to ensure that my paper and the other papers were carefully proofread, with attention to details of word choice, spelling, and punctuation. Her great job prevented my paper submissions from losing scores—at least in the aspect of writing quality.

Moreover, I would like to thank all my fellow researchers who contributed in some way to this thesis. In Chapter 4, my interconnect design serves as a component of the CHP prototyping on FPGA devices. Yu-Ting Chen, Mohammad Ali Ghodrat, Chunyue Liu, Muhuan Huang and Zou Yi contributed to this prototyping effort. In Chapter 5, the proposed work initiated from a discussion with Peng Li and Peng Zhang. Peng Zhang also helped develop the compiler flow to analyze the data access patterns in user applications. In Chapter 7, the proposed work is part of a joint project for accelerator management in heterogeneous clusters. In this project, Muhuan Huang worked on the global resource management, and Hui Huang worked on the application resource management. Last, Karthik Gururaj, Sen Li and Di Wu maintained the servers in the VAST Lab and provided a stable platform for me to do most of the experiments.

I am also thankful to my fellow researchers and friends in the VAST Lab who made my graduate study in UCLA colorful and joyful. Moreover, I would like to thank all the faculty and the staff in the Electrical Engineering Department and Computer Science Department at UCLA for helping me get through the PhD program.

My love and thanks to my parents for their constant source of love, concern, support and strength all these years.

The work in this thesis is partially supported by the Center for Domain-Specific Computing (CDSC) which is funded by the NSF Expeditions in Computing Award CCF-0926127, CFAR (Center for Future Architecture Research, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA), and grants from Altera Corporation and Xilinx Inc.

## Vita

| | |
|---|---|
| 2010 | B.S. (Microelectronics), Peking University, China. |
| 2012 | M.S. (Electrical Engineering), University of California, Los Angeles. |
| 2010–2015 | Research Assistant, Department of Electrical Engineering, University of California, Los Angeles, U.S.A. |

## Publications

Jason Cong, Hui Huang, Muhuang Huang, Bingjun Xiao, Peng Zhang*, "CMOST: A System-Level FPGA Compilation Framework", *Design Automation Conference (DAC)*, 2015.

Cheng Zhang*, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, Jason Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks: An Analytical Approach based on Roofline Model", *International Symposium on FPGAs*, 2015.

Jason Cong, and Bingjun Xiao*, "Minimizing Computation in Convolutional Neural Networks", *International Conference on Artificial Neural Networks (ICANN)*, 2014.

Jason Cong, Peng Li, Bingjun Xiao*, and Peng Zhang, "An Optimal Microarchitecture for Stencil Computation Acceleration Based on Non-Uniform Partitioning of Data Reuse Buffers", *Design Automation Conference (DAC)*, 2014.

Jason Cong, and Bingjun Xiao*, "FPGA-RPI: A Novel FPGA Architecture With RRAM-Based Programmable Interconnects", *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 2014.

Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao*, and Peipei Zhou, "A Fully Pipelined and Dynamically Composable Architecture of CGRA", *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2014, pp. 9-16.

Jason Cong, and Bingjun Xiao*, "Optimization of Interconnects Between Accelerators and Shared Memories in Dark Silicon", *International Conference on Computer-Aided Design (ICCAD)*, 2013.

Yu-Ting Chen, Jason Cong, Mohammad, Ali Ghodrat, Muhuan Huang, Chunyue Liu, Bingjun Xiao*, and Yi Zou, "Accelerator-Rich CMPs: From Concept to Real Hardware", *International Conference on Computer Design (ICCD)*, 2013, pp. 169-176.

Jason Cong, Muhuan Huang, Sen Li, and Bingjun Xiao*, "Energy-Efficient Computing Using Adaptive Table Lookup Based on Nonvolatile Memories", *International Symposium on Low Power Electronics and Design (ISLPED)*, 2013.

Jason Cong, and Bingjun Xiao*, "Defect Tolerance in Nanodevice-Based Programmable Interconnects: Utilization Beyond Avoidance", *Design Automation Conference (DAC)*, 2013.

Jason Cong, Guojie Luo, Kelly Tsota* and Bingjun Xiao, "Optimizing Routability in Large-Scale Mixed-Size Placement", *Asia and South Pacific Design Automation Conference (ASPDAC)*, 2013.

Jason Cong, Karthik Gururaj, Muhuan Huang, Sen Li, Bingjun Xiao* and Yi Zou, "Domain-Specific Processor with 3D Integration for Medical Image Processing", *International Conference* *on Application-specific Systems, Architectures and Processor (ASAP)*, 2011, pp. 247-250.

Jason Cong, and Bingjun Xiao*, “mrFPGA: A Novel FPGA Architecture with Memristor-Based Reconfiguration”, *International Symposium on Nanoscale Architectures*, 2011, pp. 1-8.

# CHAPTER 1

# Introduction

## 1.1 Customizable Domain-Specific Computing

As pointed out in [19], in order to meet ever-increasing computing needs and overcome power density limitations, the computing industry has halted simple processor frequency scaling and entered the era of parallelization, with tens to hundreds of computing cores integrated in a single processor, and hundreds to thousands of computing servers connected in a warehouse-scale data center. However, such highly parallel, general-purpose computing systems still face serious challenges in terms of performance, power, heat dissipation, space, and cost. Recently the research focus has moved from parallelization to domain-specific customization as the next disruptive technology [19], based on three observations:

First, each user or enterprise typically has a high computing demand in one or a few selected application domains (e.g., graphics for game developers, circuit simulation for integrated circuit design houses, financial analytics for investment banks). Therefore, it is possible to develop a customizable computing platform where computing engines and communication infrastructures can be specialized to a particular application domain, thereby gaining significant improvements in power-performance efficiency compared to a general-purpose architecture.

Second, the performance gap between a totally customized solution (using an application-specific integrated circuit (ASIC)) and a general-purpose solution can be very large. A case study of the 128-bit key AES encryption algorithm was presented in [20]. An ASIC implementation in 0.18um CMOS achieves 3.86 Gbits/second at 350mW, while the same algorithm coded in Java and executed on an embedded SPARC processor yields 450bits/second at 120mW. This difference implies a performance/power efficiency (measured in Gbits/second/W) gap of roughly 3 million.

Last, it is extremely costly and impractical to implement each application in ASIC. The nonrecurring engineering cost of an ASIC design at the current 45nm CMOS technology is over $50M [21], and the design cycle can easily exceed a year. There is a strong need for a novel architecture platform that can be efficiently customized to a wide range of applications in a domain or a set of domains to bridge the huge performance/power gap between ASICs and general-purpose processors.

To realize the order-of-magnitude performance/power efficiency improvement via customization, yet still leverage economy of scale, Center for Customizable Domain-Specific Computing (CDSC) is developing a customizable heterogeneous platform (CHP), consisting of a heterogeneous set of adaptive computational resources with communication optimization at three different levels. Fig. 1.1 illustrates an example CHP configuration. A CHP-oriented data center includes

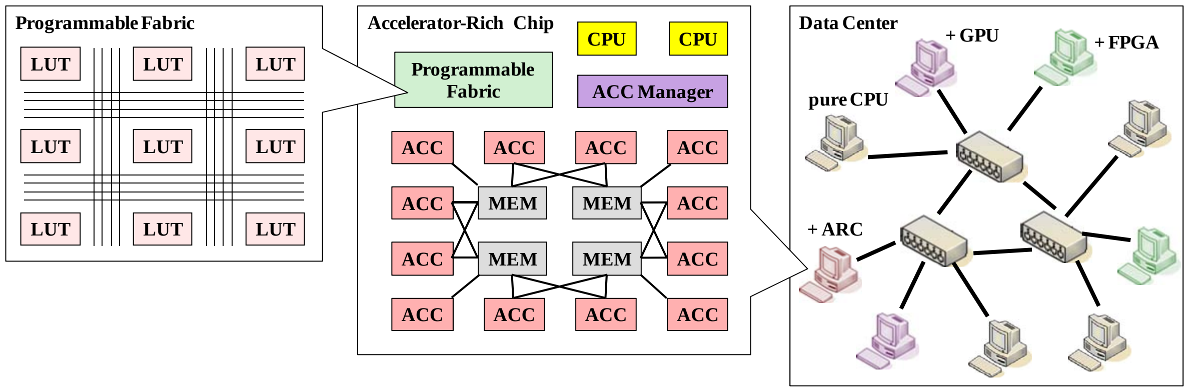


Figure 1.1: Customizable heterogeneous platform for domain-specific computing.

a sea of heterogeneous processor nodes equipped with CPUs, GPUs, FPGAs and accelerator-rich chips to serve different user tasks for high energy efficiency. All of these processor nodes communicate with each other for joint tasks decomposed from big data applications. A CHP-oriented accelerator-rich chip includes a sea of heterogeneous accelerators to execute different application kernels in user tasks for high energy efficiency. All of these accelerators communicate through an on-chip network which can be reconfigured to meet the data demand imposed by working accelerators. A CHP-oriented programmable fabric includes a sea of look-up tables (LUTs) which are programmed to different basic operations in user application kernels. All of these LUTs are connected by programmable interconnects which can be reconfigured to route the Boolean network among basic operations.

As we can see, each level of the CHP is composed of the computing units and the communication infrastructure. While we can easily increase the number of computing units, it is usually a big challenge to design a corresponding communication infrastructure to feed data into the huge amount of these computing units in a timely manner. In addition, the data access consumes much more energy than the arithmetic operation and has a high impact on the system energy efficiency. Fig. 1.2 adopted from [1] shows the data for various operations in a 45nm technology, and gives the energy breakdown of a simple in-order processor. While the architecture customization in

| Integer | |
|---|---|
| Add | |
| 8 bit | 0.03pJ |
| 32 bit | 0.1pJ |
| Mult | |
| 8 bit | 0.2pJ |
| 32 bit | 3.1pJ |

| FP | |
|---|---|
| FAdd | |
| 16 bit | 0.4pJ |
| 32 bit | 0.9pJ |
| FMult | |
| 16 bit | 1.1pJ |
| 32 bit | 3.7pJ |

| Data Access | |
|---|---|
| Cache | (64bit) |
| 8KB | 10pJ |
| 32KB | 20pJ |
| 1MB | 100pJ |
| DRAM | 1.3-2.6nJ |

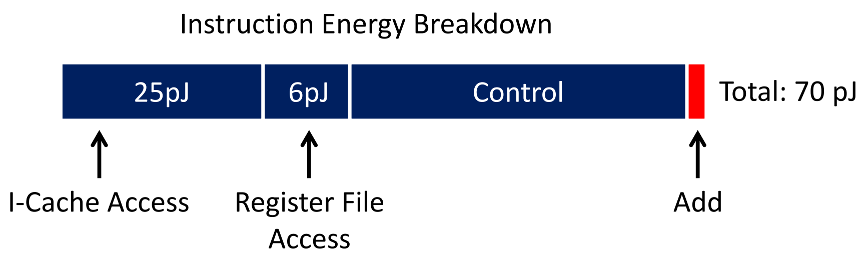


Figure 1.2: Rough energy costs for various operations in 45nm 0.9V, adopted from [1].

our CHP can save energy consumption via the control part in the professor, the data access still remains a big problem. This thesis focuses on the communication optimization in the CHP.

## 1.2 Communication Optimization for Customizable Domain-Specific Computing

In domain-specific computing, the communication pattern of an application can be obtained by an offline compiler or a profiling analysis. With this knowledge, in the CHP the communication infrastructure among the components can be customized to eliminate unnecessary contention and arbitration and achieve high energy efficiency. Based on this observation, this dissertation investigates the energy-efficient communication optimization for the CHP, at fabric level, chip level, and server/cluster level. Fabric-level communication optimization includes novel architecture of programmable interconnects driven by emerging devices, and defect-aware routing to tolerate the high defect rate in interconnects based on emerging technologies. Chip-level communication optimization includes novel interconnect designs between accelerators and shared memories, and a data reuse infrastructure to reduce offchip accesses. Server/cluster-level communication optimization includes algorithm redesigns of popular big data applications for disk access reduction, and accelerator-centric cluster computing with data transfer minimization. These communication optimizations are introduced in the following subsections. In each subsection, we discuss the requirement of the CHP on each level and the design challenges, and also give a brief overview of how we solved these challenges.

### 1.2.1 Programmable Interconnects Driven by Emerging Devices

Field programmable gate arrays (FPGAs) can be customized to user applications and provide $>$10x improvement in power-efficiency over CPUs [19]. The programmable interconnects inside FPGAs allow user applications with an arbitrary Boolean network to be mapped to FGPAs, but pay high overhead for such a flexibility. The programmable interconnects account for 50 to 90% of the total FPGA area [22, 23, 8], 70 to 80% of the total delay [24, 22, 25, 23, 8], and 60 to 85% of the total power consumption [26, 22, 25, 23, 8].

Emerging technologies, especially emerging nonvolatile memory (NVM) technologies, lead to new opportunities for design improvement. Popular emerging NVMs include spin-transfer torque RAM (STTRAM), phase-change RAM (PRAM), nanoelectromechanical (NEM) relay, and resistive RAM (RRAM). All of these technologies have demonstrated CMOS-compatible fabrication and can be integrated in metal layers over CMOS via a back-end-of-line (BEOL) process [27, 28, 29, 30]; this leads to opportunities for high-density circuit designs. These NVMs also have the desirable property of nonvolatility, which means that they can be turned off during standby to save power. Various semiconductor companies (i.e., Micronic, Panasonic, and Spansio) are manufacturing NVMs in mass volume [8]. Many studies focus on the memory applications of NVMs, and there have also been a number of works on the introduction of NVMs to FPGA architectures over the past few years [31, 32, 33, 34, 35, 36, 37, 38, 23, 39, 8, 40, 41, 42].

In Chapter 2 we present a novel FPGA architecture with RRAM-based programmable interconnects (FPGA-RPI), which are enabled by using the RRAM properties. Our RRAM-based programmable interconnects are composed of three disjoint structures: transistor-less programmable interconnects, a programming grid, and an on-demand buffering architecture. Specific optimizations are performed on each structure. The transistor-less programmable interconnects are built by RRAMs and metal wires alone, and are placed over CMOS transistors. An RRAM-friendly layout is designed for this structure. It meets the tight space constraints coming from the stacked structure and, at the same time, fits into the footprint of the CMOS transistors below. The programming transistors in the programming grid are heavily shared among RRAMs via the transistor-less programmable interconnects. The on-demand buffering architecture provides opportunities to allocate buffers in interconnects during the implementation phase. It allows utilization of the application information for a better allocation of buffers. Note that the feasibility of the disjoint structures, and the feasibility of all their improvements mentioned above, is based on the use of RRAMs as programmable switches. Simulation results show that our RRAM-based programmable interconnects achieve a significant reduction in area, delay, and power.

This work expands on the preliminary work in my master's thesis [43], with a deeper examination of NVM candidates, the addition of the programming scheme, a discussion of the benefits of using our on-demand buffering architecture, and additional evaluations of each architectural change. This work is published in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* [44].

### 1.2.2 Defect-Aware Routing on Nanodevice-Based Programmable Interconnects

A number of FPGAs based on emerging nanodevices have been explored in the past few years [45, 38, 23, 35, 25, 41]. The emerging nanodevices include resistive RAM (RRAM) [45], phase-change RAM (PCRAM) [38], nanoelectromechanical (NEM) relays [23, 35], and molecular switches [25, 41]. They can be generalized as bistable switches which can be programmed between the ”on” and ”off” states [46]. A single nanodevice can function as a routing switch in place of a pass transistor and its six-transistor SRAM cell in conventional FPGAs. Programmable interconnects of FPGAs can therefore be built from nanodevices and have smaller footprints. In addition, these nanodevices are fabricated among metal layers and do not contribute to the footprint of the CMOS transistors below them. Note that conventional FPGAs prefer multiplexers to pass transistors as the basic circuits in programmable interconnects for fewer configuration bits, though signals have to pass more levels of gates in multiplexers. However, nanodevices provide configuration bit storages along with signal paths. Nanodevice-based FPGAs switch back to pass transistors for higher performance [45, 38, 23, 25, 41]. These nanodevices are also nonvolatile devices and can save significant leakage power. To summarize, nanodevice-based FPGAs show a significant potential to save footprint, critical path delay and power consumption. For example, a NEM-relay FPGA in [23] achieves savings of 43%, 28% and 37% respectively. A RRAM-based FPGA proposed in [45] achieves savings of 80% , 56% and 39% respectively.

In nanodevice manufacturing, defects are a certainty, and reliability becomes a critical issue. The projected defect rate of nanodevices can be up to $10^{-1}$, which is much higher than the level of $10^{-9}$ to $10^{-12}$ in CMOS systems [47, 35]. A number of approaches have been proposed to improve the FPGA yield by leveraging its reconfigurable structures. Among them, we choose the component-specifc implementation [48, 49, 50, 35, 47, 41, 46] which works on a defect map obtained from testing as the basic framework of our defect tolerance techniques.

Defects in programmable nanodevices are manifested as losses of configurability. A defective nanodevice may be stuck at an either ”on” or ”off” state [47, 46]. If the nanodevice is used in a logic block, it leads to a stuck-at-1 or stuck-at-0 bit. If the nanodevice is used in programmable interconnects, it leads to a stuck-closed or stuck-open switch.

Defect tolerance in logic blocks has been explored in recent years [50, 35, 47]. However, in an FPGA chip, programmable interconnects usually occupy 2-4x more area than logic blocks [22, 23, 25], and are the dominant part. Tolerance of defects in programmable interconnects requires more attention than that in logic blocks. The stuck-open switches in interconnects can be easily solved by removing the broken edges from the routing graph [51, 41]. The authors of [41] showed that yield can remain nearly 100%, even at a defect rate of stuck-open switches as large as 50%. However, they ignored stuck-closed switches, which are much more challenging than stuck-open switches. We will show that stuck-close switches need to remove $>$10x routing resources than stuck-open switches when simple defect avoidance is used. However, the good thing is that a stuck-closed switch can still be used opportunistically if we can guarantee that the two nodes shorted by the switch are always mapped to the same net. They can be regarded as extra shorting constraints during the routing phase [46], just as done for logic blocks in [35, 47]. Along with huge resource savings, this method has two challenges: 1) Defects in programmable interconnects can propagate over the entire chip, and their tolerance has to be solved in a more global way, with scalability taken into account; 2) Existing FPGA routing algorithms work on a directed routing graph which assumes that all the edges can programmed to be open or closed, and shorting constraints break this assumption. The second challenge pushes some researchers to switch to algorithms that can easily deal with shorting constraints, but lead to poor scalability and solution quality. For example, the SAT-based method in [46] uses Boolean clauses to apply defect constraints, but has high time complexity due to the large search space and is unable to develop a timing-driven flow based on the satisfiability solver.

In Chapter 3 we first provide a complete defect analysis. Then we propose a scalable algorithm to perform timing-driven routing under shorting constraints. We start from the negotiation-based procedure [52], the state-of-art routing algorithm of FPGAs, to maintain the circuit performance and the tool runtime. We extend the idea of the resource negotiation to balance the goals of timing and routability under shorting constraints. We also observe that a routing node will be logically inconsistent with certain nets due to shorting edges. Therefore, we add a mechanism to achieve fast pruning before routing of each net. We also develop several techniques to guide the router to map the shorting clusters to those nets with more shared paths for better utilization of routing resources while automatically balancing it with circuit performance. In addition, we discovered that some logic blocks will become virtually unusable due to shorted pins found in the defect analysis. We enhance the placement algorithm to recover these logic blocks. This work is published in the *Proceedings of 2013 Design Automation Conference* [53].

### 1.2.3 Optimized Interconnects Between Accelerators and Shared Memories

Energy efficiency has become one of the primary design goals in the many-core era. A recent trend to address this is the use of on-chip accelerators as coprocessors [54, 55, 56, 57, 58]. Application-specific accelerators can fully explore parallelism and customization and provide a 100x improvement in energy efficiency over general-purpose processors [58, 20, 59]. As predicted by ITRS [21], future computing platforms may integrate hundreds or thousands of accelerators on a chip to improve their performance and energy efficiency. Accelerator-rich computing platforms also offer a good solution for overcoming the “utilization wall” as articulated in the recent study reported in [60]. As scaling down continues, there will be tens of billions of transistors on a single chip. However, it has been demonstrated that a 45nm chip filled with 64-bit operators will only have around 6.5% utilization (assuming a power budget of 80W) [60]. This means that an architecture with a massive number of homogeneous cores may not work since only a limited number of cores can work in parallel. However, we can implement tens of billions of transistors into a sea of heterogeneous accelerators and launch the corresponding accelerators upon user demand. This methodology does not expect all the accelerators to be used all the time, but expects most computation tasks to be executed by the most energy-efficient hardware.

Though the computation parts of accelerators are heterogeneous, the memory resources in accelerators are homogeneous and can be shared among accelerators [18, 61]. The transistors saved from memory sharing can be used to implement more accelerators to cover more application kernels.

One of the key challenges to realizing memory sharing among accelerators is how to design the interconnects between accelerators and shared memories. Since accelerators run $>$100x faster than CPU cores [59], accelerators perform many independent data accesses every clock cycle. This requires a high-speed, high-bandwidth interconnect design with many conflict-free data channels to prevent accelerators from starving for data. There are very few works on optimization of interconnects between accelerators and shared memories. The work in [62] uses shared buses as the interconnects. The work in [61] assumes a network-on-chip (NoC) that connects all the accelerators and shared memories together. The work in [18] uses interconnects similar to shared buses in local regions, and an NoC for global interconnects. All still treated the data ports of accelerators like ports of CPU cores. They will suffer either performance inferiority from lack of support to meet the high data demands of accelerators, or a huge transistor cost from hardware duplication to meet the accelerator demand.

In Chapter 4 we identify three optimization opportunities that emerge in accelerator-rich platforms, and exploit them to develop novel interconnects between accelerators and shared memories:

- An accelerator contains multiple data ports, and in the interconnect design the relations of the ports from the same accelerator should be handled differently compared to the ports from different accelerators. We propose a two-step optimization rather than optimizing all the ports of all accelerators globally in a single procedure. Many unnecessary connections associated with each individual accelerator are identified and removed before all the accelerators are optimized together.

- Due to the power budget in dark silicon, only a limited number of accelerators will be powered on in an accelerator-rich platform. The interconnects can be partially populated to just fit the data access demand limited by the power budget.

- Accelerators are heterogeneous. Some accelerators will have a higher chance of being turned on or off together if they belong to the same application domain. This kind of information can be used to customize the interconnect design to remove potential data path conflicts and use fewer transistors to achieve the same efficiency.

This work is published in the *Proceedings of 2013 International Conference on Computer-Aided Design* [63].

### 1.2.4 Data Reuse Optimization for Stencil Access Patterns

Accelerator-centric architectures can bring 10-100x energy efficiency by offloading computation from general-purpose CPU cores to application-specific accelerators [64]. The engineering cost of designing massive heterogeneous accelerators is high, but can be much reduced by raising their abstraction level beyond RTL to C by high-level synthesis (HLS) [65]. Data access optimization has a strong impact on HLS results. This significantly motivates recent work on data reuse [66, 67] and memory partitioning [12, 68, 13, 14, 69] in HLS.

External memory bandwidth is a significant bottleneck for system performance and power consumption [1, 70, 71]. Data reuse is an efficient technique for using on-chip memories to reduce external memory accesses. When an application contains a data array with multiple references, we can allocate a reuse buffer and keep each array element in the buffer from its first access until its last access. Then each array element needs to be fetched from the external memory only once, and the off-chip traffic is reduced to the minimum. Loop transformation can be applied to improve data locality and reduce the size of the data reuse buffer [66, 67].

When the loops in an application are fully pipelined, an accelerator needs to perform multiple load operations from the same reuse buffer every clock cycle. To avoid contention on memory ports, memory partitioning of the reuse buffer is required. Since the transistor count of memory control logics is proportional to the number of memory banks after partitioning, the optimization goal of memory partitioning is to minimize the number of memory banks. The constraint is that the multiple array elements to be loaded every clock cycle are always stored in different memory banks. The work in [12, 68] provides solid frameworks of memory partitioning. Further optimizations, including memory access rescheduling [13] and multidimensional arrays [14], are also proposed. But none of these can guarantee the optimal solution for a given case in terms of the number of banks. The reason is that their optimization space is limited to uniform memory partitioning; i.e., all the memory banks have to be of the same size. It is an unnecessary constraint which was assumed by commodity HLS tools, e.g., [72]. Other work partitions different fields of a single data *structure* into multiple memory banks for data parallelism based on profiling results [69]. It is orthogonal to the problems on multiple array references in [12, 68, 13, 14] and this work.

In Chapter 5 we go beyond the limitation of uniform memory partitioning, and propose a novel method based on nonuniform memory partitioning. As a result, we can achieve fewer memory banks than the optimal solutions in prior work [12, 68, 13, 14] which were limited to uniform memory partitioning. As an early-stage exploration of nonuniform memory partitioning, we focus on stencil computation, a popular communication-intensive application domain. We develop a microarchitecture with novel structures of memory systems which achieve the theoretical minimum number of memory banks for any stencil access patterns. Experimental results show that we can reduce $25 - 100\%$ of various resources, including BRAMs, logic slices, and DSPs, compared to prior work [14], along with slightly improved timing. This work is published in the *Proceedings of 2014 Design Automation Conference* [73].

### 1.2.5 Computation and Communication Optimization in Convolutional Neural Network Driven by Big Data

As the capability of data collection continuously increases, the data processing and analytics relies more on machine learning to automatically mine out more insights from the big data. Biologically inspired deep learning based on artificial neural networks (ANNs) has gained its popularity in recent years since the network complexity can be easily scaled up with the increasing data volume to capture a better data model. Among the different kinds of ANNs, convolutional neural networks (CNNs) have achieved good success, particularly in computer vision applications, e.g., the recognition of handwritten digits [74, 75], and the detection of faces [76, 77]. In the 2012 ImageNet contest [78], a CNN-based approach named SuperVision [17] outperformed all the other traditional image recognition algorithms. On one hand, CNNs keep the advantage of artificial neural networks which use a massive network of neurons and synapses to automatically extract features from data. On the other hand, CNNs further customize their synapse topologies for computer vision applications to exploit the feature locality in image data.

The success of CNNs promises wide use for many future platforms to recognize images, e.g., micro-robots, portable devices, and image search engines in data centers. It will be beneficial to improve the implementation of the CNN algorithm to reduce the computational cost. One direction is to improve the CNN algorithm using hardware accelerators, e.g., GPUs and field-programmable gate arrays (FPGAs) [79, 80, 81]. Another orthogonal direction is to reduce the theoretical number of basic operations needed in the CNN computation from the algorithmic aspect, as will be discussed in this thesis.

In Chapter 6 we first reveal the linear algebraic properties in the CNN computation. Based on these properties, we invent an efficient algorithm that can be applied to generic CNN architectures to reduce the arithmetic operations without any penalty on the image recognition quality or hardware cost. Furthermore, we develop a communication-avoidance algorithm to minimize the disk access. Part of this work was published in the *Proceedings of 2014 International Conference on Artificial Neural Networks* [82].

### 1.2.6 Accelerator-Centric Cluster Computing with Communication Optimization

As we enter the era of big data, the data volume to be timely processed becomes increasingly larger. Meanwhile, the performance improvement of a single CPU server has slowed down in recent years. To meet the increasing computation demand from big data, there are two major directions in computing solutions. One is horizontal scaling, i.e., add more server nodes to a system. The other is vertical scaling, i.e., improve the performance of a single node by hardware accelerators. There are two popular candidates for hardware accelerators: GPGPUs and FPGAs. In this thesis, we mainly consider FPGAs.

The use of FPGAs discussed above is often limited to single-node solutions. In the era of big data, user data are usually partitioned into many data blocks and stored on many servers in a cluster. It is important to exploit parallelism among data blocks by distributing computing tasks to these servers. This capability is missing in single-node solutions of FPGAs. To exploit both cluster parallelism and microarchitecture customization, FPGA-centric cluster computing solutions are a promising trend. Microsoft recently developed such a cluster, where all the servers in its cluster are equipped with an FPGA and can work together to accelerate a single application [83]. However, this development at Microsoft is limited to the single application of Bing search. All the proposed methodologies are coupled with the properties of this search application.

On the other hand, the existing methodologies for the pure-CPU clusters are general enough to cover most big data applications. One popular category among them is the MapReduce framework, e.g., Google MapReduce [84], Apache Hadoop [85] and Spark [86]. These systems accept the parallel programming model MapReduce as user inputs. The MapReduce model allows users to express their computation in terms of a *Map* function that processes all the data blocks in parallel, and a *Reduce* function that merges all the mapped outputs with the same keys to produce a final result. This parallel programming model has even been used to synthesize FPGA accelerator designs by the compilation flows in [87, 88]. These flows map the program parallelism which is explicitly expressed in the MapReduce programming model to the hardware duplication of the customizable pipelines in FPGAs. However, their scope is still limited to single-node FPGA solutions. An enhanced compilation flow in [89] further supports partitioning of jobs in users' MapReduce programs among the FPGAs, GPUs and CPUs in a cluster. After the job partitioning, the compiler generates an optimized MPI host program for the collaboration of the FPGAs, GPUs and CPUs in a cluster on the target application. However, while this flow may provide a good static solution for a standalone application, it cannot handle the situation when multiple applications are launched in the cluster.

Instead of using the MapReduce model for design optimization, we want to keep the capability of the existing MapReduce systems that automatically distribute map and reduce tasks from user applications to the servers in a cluster. If we enhance these MapReduce systems so that the map and reduce tasks can further get acceleration from the FPGA devices installed on the local servers of these tasks, we will be able to achieve both cluster parallelism and microarchitecture customization. There are some research projects along this direction. In [90], Hadoop is deployed on a server with NetFPGAs connected via an Ethernet switch. However, in this system MapReduce programmers need to perform low-level interactions with the FPGA device in their designs of *Map* and *Reduce* functions. In addition, no results are provided to show whether this system can be extended to a cluster of multiple servers. In [91], Hadoop is deployed on a cluster of Xilinx Zynq SoC devices [92] where CPU cores and programmable logics are fabricated on the same chip. However, this methodology is tightly coupled with the underlying Zynq platform and is hard to port to clusters with commodity CPU servers.

We observe that there are several challenges in building a general MapReduce system with FPGA accelerators:

1. The methodology to support FPGA accelerators in the MapReduce system should not be specific to a single application, but should allow multiple different jobs to share the system during the runtime.

2. The methodology should not be platform-specific, but should be feasible for clusters of commodity servers.

3. The methodology should not require MapReduce programmers to acquire knowledge about FPGAs, but should abstract away FPGA hardware details from them.

In Chapter 7 we build a MapReduce system extended from Spark [86] to support FPGA accelerators and solve the challenges listed above. We prototype our system in a Xeon cluster equipped with FPGAs, and run multiple machine learning applications on big data. Note that with hardware accelerators, the communication often becomes the system bottleneck. Our cluster computing solution also optimizes the data transfer at different levels as follows:

1. We redesign algorithms to minimize the inter-node communication within a training iteration.
2. We exploit the in-memory processing of Spark to minimize the inter-node communication across training iterations.
3. We group multiple tasks in a batch to offload to an FPGA device to minimize the setup overhead of the data transfer between FPGAs and host servers.
4. We analyze the properties of iterative machine learning applications and cache the training data in FPGA device memory to minimize the data transfer between FPGAs and host servers.

Experimental results validate the effectiveness and scalability of our approach.

# CHAPTER 2

# Programmable Interconnects Driven by Emerging Devices

## 2.1 Introduction

Field programmable gate arrays (FPGAs) can be customized to user applications and provide $>$10x improvement in power-efficiency over CPUs [19]. The programmable interconnects inside FPGAs allow user applications with an arbitrary Boolean network to be mapped to FGPAs, but pay high overhead for such a flexibility. The programmable interconnects account for 50 to 90% of the total FPGA area [22, 23, 8], 70 to 80% of the total delay [24, 22, 25, 23, 8], and 60 to 85% of the total power consumption [26, 22, 25, 23, 8].

Emerging technologies, especially emerging nonvolatile memory (NVM) technologies, lead to new opportunities for design improvement. Popular emerging NVMs include spin-transfer torque RAM (STTRAM), phase-change RAM (PRAM), nanoelectromechanical (NEM) relay, and resistive RAM (RRAM). All of these technologies have demonstrated CMOS-compatible fabrication and can be integrated in metal layers over CMOS via a back-end-of-line (BEOL) process [27, 28, 29, 30]; this leads to opportunities for high-density circuit designs. These NVMs also have the desirable property of nonvolatility, which means that they can be turned off during standby to save power. Various semiconductor companies (i.e., Micronic, Panasonic, and Spansio) are manufacturing NVMs in mass volume [8]. Many studies focus on the memory applications of NVMs, and there have also been a number of works on the introduction of NVMs to FPGA architectures over the past few years [31, 32, 33, 34, 35, 36, 37, 38, 23, 39, 8, 40, 41, 42].

In this chapter we present a novel FPGA architecture with RRAM-based programmable interconnects (FPGA-RPI), which are enabled by using the RRAM properties. Our RRAM-based programmable interconnects are composed of three disjoint structures: transistor-less programmable interconnects, a programming grid, and an on-demand buffering architecture. Specific optimizations are performed on each structure. The transistor-less programmable interconnects are built by RRAMs and metal wires alone, and are placed over CMOS transistors. An RRAM-friendly layout is designed for this structure. It meets the tight space constraints coming from the stacked structure and, at the same time, fits into the footprint of the CMOS transistors below. The programming transistors in the programming grid are heavily shared among RRAMs via the transistor-less programmable interconnects. The on-demand buffering architecture provides opportunities to allocate buffers in interconnects during the implementation phase. It allows utilization of the application information for a better allocation of buffers. Note that the feasibility of the disjoint structures, and the feasibility of all their improvements mentioned above, is based on the use of RRAMs as programmable switches (to be discussed in Section 2.2.2.2). Simulation results show that our RRAM-based programmable interconnects achieve a significant reduction in area, delay, and power.

This chapter expands on the preliminary work [45], with a deeper examination of NVM candidates, the addition of the programming scheme, a discussion of the benefits of using our on-demand buffering architecture, and additional evaluations of each architectural change.

## 2.2 Background

### 2.2.1 Conventional FPGA

Fig. 2.1 shows a conventional FPGA architecture. It is a typical island-based type, which is made up of an array of tiles. Each tile consists of one logic block (LB, also called configurable logic block, or CLB), two connection blocks (CB) and one switch block (SB). Each LB contains a cluster of basic logic elements (BLEs), typically look-up tables (LUTs), to provide customizable logic functions. Each LB also contains local routing multiplexers (MUXs) to provide connections

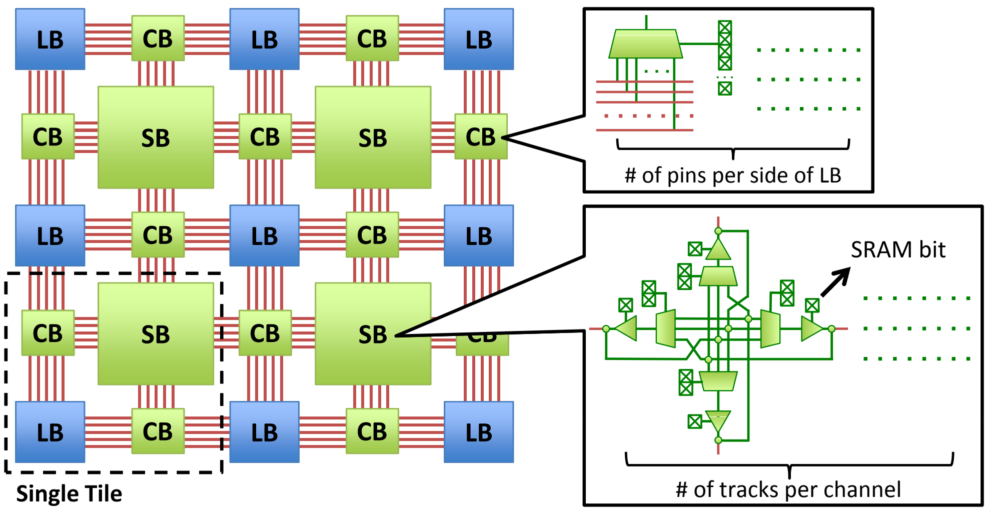


Figure 2.1: A typical FPGA architecture. LB = logic block (or CLB). CB = connection block. SB = switch block.

among BLEs. LBs are connected to the routing channels through CBs, and the segmented routing channels are connected with each other through SBs. Fig. 2.1 also depicts two typical circuit designs of CBs and SBs based on MUXs and buffers described in [93]. The selector pins of each MUX are connected to a group of storage cells that determine the connectivity of the MUX. The storage cells can be SRAMs, anti-fuses or flash cells [94]. SRAM-based FPGAs are currently popular due to their standard CMOS manufacturing processes [94]. In this work we focus our analysis on SRAM-based FPGA architectures. Note that the circuits presented in Fig. 2.1 have copies of up to the number of pins per side of LBs in a single CB, and up to the number of tracks per channel in a single SB. We see that CBs and SBs make up the interconnects of FPGAs, and pay a high logic complexity for flexibility.

The FPGA programmable interconnects usually contain three types of components: SRAM-based storage of configuration bits, MUX-based routing switches, and buffers. All three components are nontrivial parts in FPGAs, as shown in Fig. 2.2.[1] Though CMOS-based programmable

[1] This breakdown is based a recent commercial FPGA (details in Section 2.5.1) and shows a large percentage of LB area. The increasing overhead of programmable interconnects pushes FPGA vendors to larger and more complex LBs. In evaluations [22, 23, 8] in academia, the proportion of programmable interconnects is >75%, which provides sufficient improvement space for these interconnects. We will show in Section 2.5.4 that after the overhead of

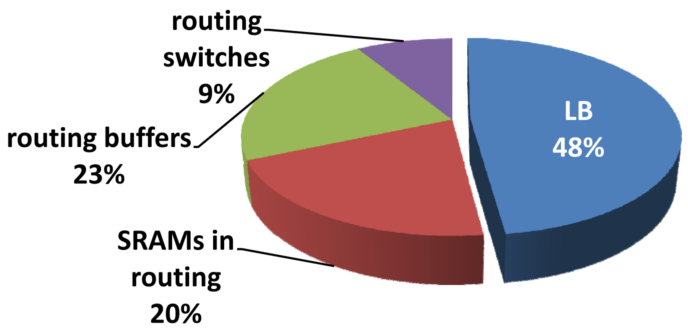


Figure 2.2: Area breakdown of different components in an FPGA based on an architectural model [2] that mimics the Xilinx Virtex7 FPGAs [3]. All three components (SRAM bits, routing switches and routing buffers) take up significant area in programmable interconnects.

interconnects in FPGAs have been optimized, there are fundamental limitations in all three components of programmable interconnects (shown in Fig. 2.2):

1. Most FPGAs use SRAMs to store programming bits [94]. Each SRAM cell uses more than six transistors to store only one bit. Modern FPGAs enable serial bitstream programming by storing configuration bits in latch nodes embedded in a shift register structure, with an area overhead no less than the SRAM-based storage. In addition, these storage media are volatile—this causes excessive power consumption during standby.

2. MUX-based routing switches in FPGAs are implemented in tree structures with serial pass transistors for less SRAM-based storage of select bits. Each pass transistor has to be wide enough to provide sufficient drive, and this leads to a large footprint.

3. In most cases, routing buffers in FPGAs exceed the buffering demand of a given application. The quantity and positions of routing buffers placed in each track of programmable interconnects are optimized to meet the worst-case demand of the track.

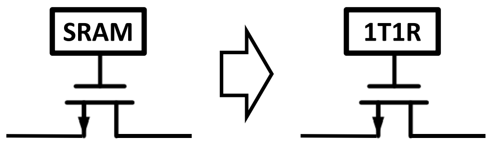
programmable interconnects is significantly reduced by our technologies, smaller LBs will be preferred.

### 2.2.2 FPGAs with Emerging NVMs

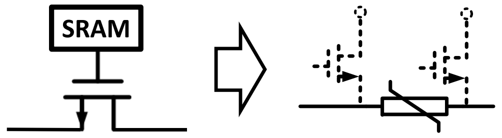
With the recent development of emerging nonvolatile memory (NVM) technologies, a number of novel FPGA architectures based on those technologies have been proposed.

#### 2.2.2.1 Replace SRAMs with NVMs

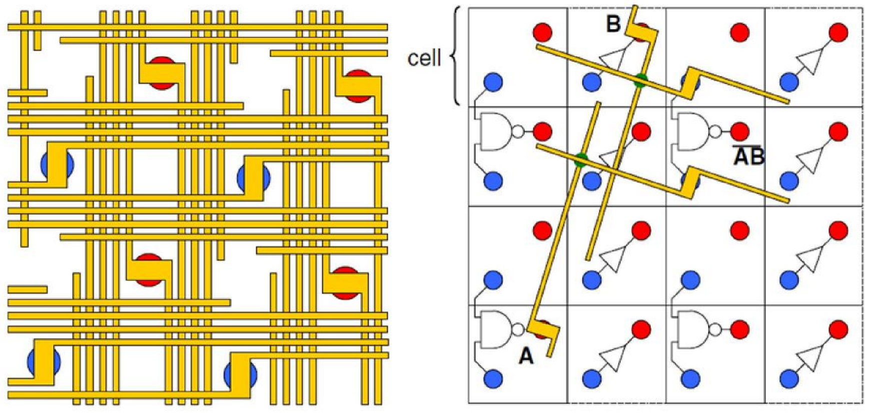
In [31, 32, 33, 34, 35, 36, 37] the SRAM-based configuration bits are moved to STTRAMs, PRAMs, NEM relays, or RRAMs, as shown in Fig. 2.3a. FPGA area is reduced since NVMs



(a) Replace SRAMs with emerging NVMs [31, 32, 33, 34, 35, 36, 37].



(b) Use NVMs as programmable switches [38, 23, 39, 8, 40].



(c) Integration of CMOS, NVMs and nanowire in field-programmable nanowire interconnect (FPNI) [41, 42].

Figure 2.3: FPGA with emerging nonvolatile memories.

have a 5 to 25x higher density than SRAMs [34] and/or can be placed over CMOS transistors [32, 33, 36]. The nonvolatility of NVMs also saves the excessive leakage power during standby and the long configuration loading time at boot-up [34]. The main disadvantage of NVMs is the poor performance of write operations in terms of latency, energy, and endurance, although their read operations are competitive with SRAMs [34]. These drawbacks in write operations are masked when they are used to store FPGA configuration bits; this is because there are write accesses to these bits only during FPGA programming [23]. The number of programming cycles is expected to be small (e.g. <500 [95]) for typical FPGA users. Note that in [31, 32, 33, 34, 35, 36, 37], NVMs are applied to not only the SRAMs in programmable interconnects, but also the SRAMs in logic blocks.

#### 2.2.2.2 Use NVMs as Programmable Switches

In [38, 23, 39, 8, 40], emerging NVMs, including PRAMs, NEM relays and RRAMs, are used more aggressively as programmable switches in place of SRAM-based pass transistors in conventional FPGA, as shown in Fig. 2.3b. This kind of use is enabled by a common property of these emerging NVMs. That is, the connection between two terminals of these devices can be programmed to turn on or turn off. By applying specific programming voltages, the resistance between the two terminals can be switched between the "on" state and the "off" state. The programmed resistance value can be kept either under operating voltages or without supply voltage due to nonvolatility. This kind of NVM use saves the area of SRAMs and also the pass transistors that build routing switches. The use of RRAM in our work belongs to this category.

There are still many challenges to using NVMs as programmable switches in FPGAs. One of the key challenges is how to solve the tight space constraints in a layout where CMOS, NVMs and metal wires are stacked together. Another key problem is the programmability of NVMs integrated in interconnects.[2] NVMs, such as PRAMs and RRAMs, have only two terminals. When they are used as routing switches, the two terminals are shared between the programming and signal paths. Their programming circuits need careful design; otherwise there will be interference between the two modes of operation. Furthermore, all of the work referenced above focuses on the replacement

[2]NEM relays are exempted from this issue. An NEM relay can have four terminals—two for the reconfigurable connection and two for programming. Programming circuits of this kind of NVM in the application of routing switches can remain the same as those used in the memory application [23]. The main problem with NEM relays is their large cell sizes (to be discussed in Section 2.3.1).

of routing switches with NVMs. However, as the author of [23] concludes, routing buffers become a bottleneck after this replacement and thus need further improvement.

#### 2.2.2.3 Integrate NVMs with Nanowire Crossbars

A field-programmable nanowire interconnect (FPNI) [41, 42] is an attempt to further introduce nanowire crossbars to FPGAs. As shown in Fig. 2.3c, the programmable interconnects are implemented by nanowire crossbars and RRAM cross-points over CMOS logics. The structure is quite different from the typical island-based FPGA architecture. It shows significant area reduction but requires that the feature size of nanowire crossbars be much smaller than that of CMOS logics to achieve higher RRAM density. Another problem is that in the path between two cells, at least two long nanowires have to be driven, even if the two cells are adjacent. Due to the large capacitance of these long nanowires and fine granularity of logic cells, the FPGA performance decreases by 30%, as reported in [41]; the dynamic power increases by 17.5%, as reported in [25].

## 2.3 The FPGA-RPI Architecture

We focus on the FPGA programmable interconnects, which are the dominant components in FPGA. As discussed in Section 2.2.2.1, much work already exists that introduces emerging technologies to logic blocks [31, 32, 33, 34, 35, 36]. Our work can be combined with these studies for further improvement. As analyzed in Section 2.2.2.2, using NVMs as programmable switches is clearly a good idea. Although there are still problems with this method, we will show that they can be solved by the methodologies proposed in this work.

### 2.3.1 Choice of NVMs

First, we need to decide which type of NVM to use in our work. Different NVMs have their own advantages/shortcomings when used as programmable switches. For example, STTRAM usually has an on/off ratio below 10. This ratio is sufficient for a memory application but far from enough for a routing switch. The NEM relay has the highest on/off ratio since it shows almost zero off-current [96]. However, it has a complex structure with a large fabrication size of $92.5F^2$ ($F$ is the feature size) and three metal layers [23]. The area of NEM relays can easily exceed that of the CMOS transistors below them, especially in cases where the area of an FPGA tile is much reduced by emerging technologies. In contrast, one RRAM device can be fabricated within an $F^2$ region and one metal layer [7]. The cell area of a PRAM can also be as small as that of an RRAM. However, the programming of PRAMs relies on temperature and is hard to control. In contrast, RRAMs can be set/reset by applying high positive/negative voltages [6, 7]. In addition, RRAMs provide more freedom in tuning their device properties than PRAMs. We can choose some fabrication technologies to manufacture RRAMs with a very small write latency ($<$5ns [5]) or a very high endurance ($>10^{12}$ [97]) for the memory application. We can also choose some other fabrication technologies to manufacture RRAMs with a very high on/off ratio ($\sim 10^6$ [6]) for the application of routing switches. In contrast, the on/off ratio of PRAMs fabricated by different technologies usually remains around $10^3$ [98]. There will be problems with both leakage and signal integrity if PRAMs are used as routing switches. Therefore we choose the RRAMs described in [6] to act as the routing switches in our work.

### 2.3.2 Overall Architecture

In FPGA-RPI, the programmable interconnects are composed of three disjoint structures:

- Transistor-less programmable interconnects
- A programming grid
- An on-demand buffering architecture

This composition takes routing buffers from programmable interconnects and puts them in a separate architecture. It enables an RRAM-friendly layout, sharing of more programming transistors, and on-demand buffer allocation (to be discussed in Section 2.3.3, Section 2.3.4, and Section 2.3.5 respectively). The transistor-less programmable interconnects correspond to SRAM-based configuration bits and MUX-based routing switches in conventional FPGAs. In FPGA-RPI, they are built by RRAMs and metal wires alone and are placed over CMOS transistors, as shown in Fig. 2.4. The

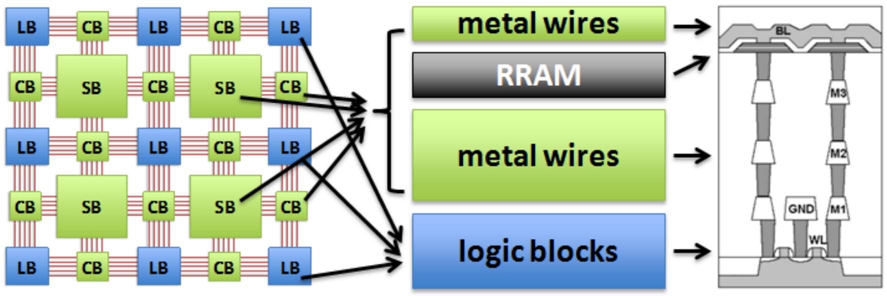


Figure 2.4: In FPGA-RPI, switch blocks and connection blocks in transistor-less programmable interconnects are placed over logic blocks in the same die according to existing RRAM fabrication structures [4, 5, 6, 7].

programming grid and the on-demand buffering architecture will be optimized to consume much fewer CMOS transistors than logic blocks. FPGA-RPI then becomes a highly compact array, as shown in Fig. 2.5. The area of FPGA-RPI is almost solely determined by logic blocks, which take only 10 to 50% of the conventional FPGA area [22, 23, 8].

### 2.3.3 RRAM-Friendly Layout Design

In the structure of the transistor-less programmable interconnects, RRAMs and metal wires are stacked over CMOS transistors. The layout will be very different from that of conventional FPGAs and will be applied with very tight space constraints. In this section we provide an RRAM-friendly layout design which solves these constraints and at the same time fits into the footprint of the CMOS transistors below. First we give the circuit schematic of the transistor-less programmable interconnects in Fig. 2.6. We use a universal type switch block [99] for the RRAM-friendly layout design. We are aware that the universal type needs 1.26% more routing tracks than the Wilton type [100]. However, the area of programmable interconnects in our FPGA-RPI will be significantly

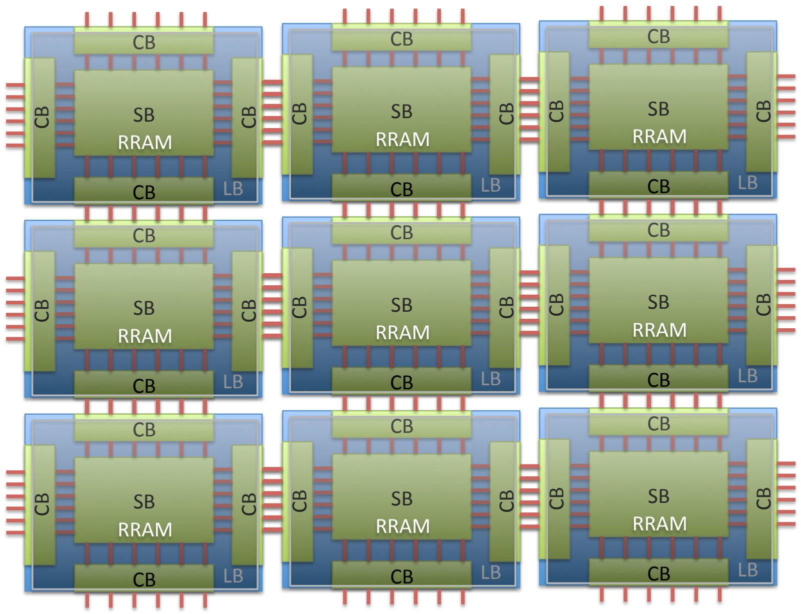
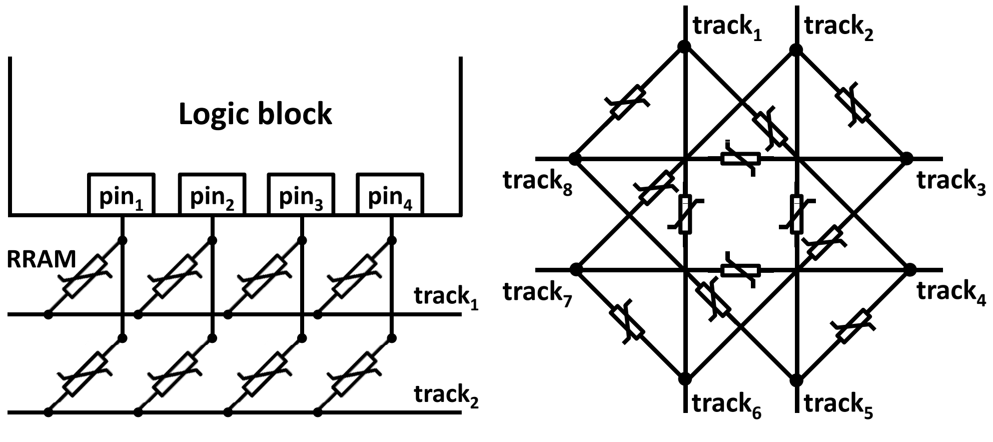


Figure 2.5: Overview of FPGA-RPI where the FPGA area is mainly contributed by logic blocks instead of programmable interconnects.



(a) An RRAM-based connection block. (b) An RRAM-based switch block.

Figure 2.6: Circuit schematics of transistor-less programmable interconnects.

reduced. Our FPGA-RPI can afford the increase in routing channel width. The layout design for Fig. 2.6 is shown in Fig. 2.7 (the number of tracks in Fig. 2.7 is increased to six to show scalability). It addresses the following design issues:

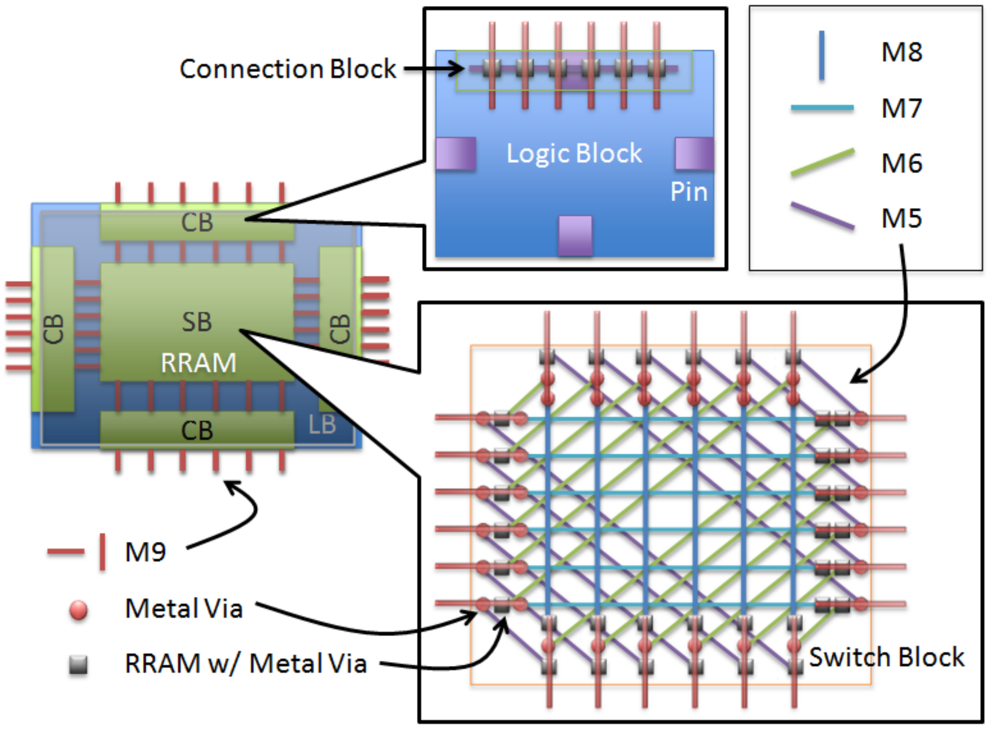


Figure 2.7: An RRAM-friendly layout design of the programmable interconnects in FPGA-RPI using the RRAM fabrication structure shown in Fig. 2.4. Here, each metal via (marked as a red point) at the intersection between wires of M9 and M5–M8 refers to a vertical connection between them through metal via(s).

1. To ease the FPGA-RPI fabrication, the RRAM layer is designed to be located between the M9 and M8 layers. It is close to the top, which is the same as the RRAM fabrication structure shown in the far right part of Fig. 2.4.
2. The placement of all metal wires is designed to avoid any blockage caused by a metal via. This blockage issue occurs only in switch blocks where there are overlaps of metal wires from at least three layers. Fig. 2.8 shows how we address this issue. In the abstract structure

of FPGA-RPI switch blocks, as shown in Fig. 2.8a, the location of any via that connects a metal layer is limited to the perimeter of the rectangular box assigned to that layer. Then, in the 3D view in Fig. 2.8b, the via blockages caused by vertical connections to the bottom metal layer are located at the most outside box, and vice versa. With the application of this principle, metal wires will naturally avoid all via blockages.

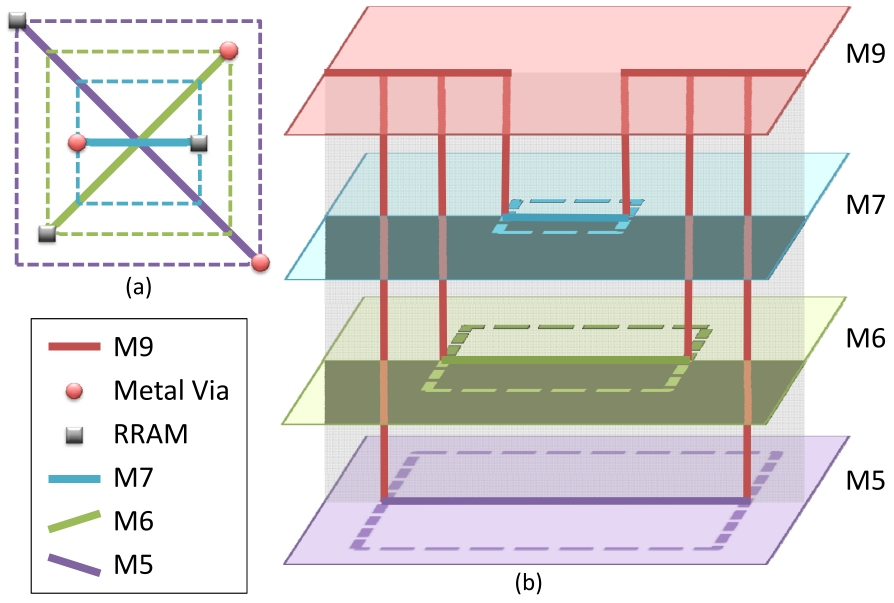


Figure 2.8: An illustration of how via blockages are avoided by metal wires in our layout design of switch blocks. (a) An abstract structure of FPGA-RPI switch blocks (M9 and M8 are omitted here for clarity). (b) 3D view to show how via blockages are avoided by metal wires. The RRAM layer is omitted here for clarity. For demonstration purposes, vertical connections are bounded in a plane to construct the case most challenging for via blockage avoidance.

3. RRAMs can easily fit into the layout design in Fig. 2.7 without extra area overhead. As stated in Section 2.3.1, the size of an RRAM cell can be close to or even smaller than the width of a metal wire [7].

4. Each metal layer in M5 to M9 consists of a set of parallel metal wires without any turns.

This reduces fabrication complexity and avoids the high resistance of turning points.

5. The metal wires in the metal layers M1 to M4 within the logic block boundary in Fig. 2.7 are left for the routing within logic blocks. They are sufficient for practical FPGA logic blocks. We verify this by producing a compact layout of the configurable logic blocks used in the Xilinx Virtex6 and Virtex7 families according to their datasheets [101, 3].

6. Though our design uses multiple metal layers, a signal rarely goes through many metal layers. While a signal is transmitted from one logic block to another, the straight paths in switch blocks are used most frequently. To save extra latency on metal via, we place these paths in M8 and M7, which are close to the RRAM layer. Only when a signal reaches a logic block, does it need to access metal layers at lower levels, i.e., M1–M4, from the RRAM layer.

7. The metal wires in the top metal layer M9 in the middle of the switch block in Fig. 2.7 are left for the power and clock grids.

8. The CMOS and metal wire resources in the gaps between logic block boundaries in Fig. 2.5 are left for the programming grid and the on-demand buffering architecture.

9. The RRAMs in the middle of the switch block in Fig. 2.7 can be used to implement basic logic elements and local interconnects in logic blocks for further improvement (to be discussed in Section 2.6).

### 2.3.4 Programming Schematic

#### 2.3.4.1 Basic Programming Schematic

To program the RRAMs integrated in programmable interconnects, there must be a programming transistor at each of the two terminals of an RRAM, as shown in Fig. 2.9. The RRAM can be switched between the “on” and “off” states by turning on both the programming transistors and applying the programming voltages (paired as $(V_p, 0)$ or $(0, V_p)$), as shown in Fig. 2.9. The values

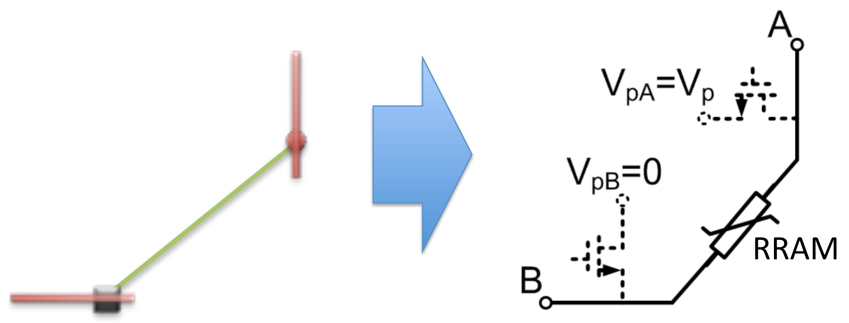


Figure 2.9: Programming circuits for RRAMs in interconnects. The two programming transistors will apply programming voltages to program the RRAM between them. $V_p$ is the threshold voltage that switches the RRAM state.

of programming voltages may be the same as those in [8]. The authors of [38] propose a different schematic where some PRAMs are not connected directly to any programming transistors.[3] The paths for programming currents going to these NVMs need to contain some other NVMs that have direct accesses to programming transistors. When we want to program the NVMs to the "off" state in the path, some NVMs will be turned off before the other NVMs and will block the programming current for the other NVMs. Therefore, we only consider the design of a programming grid which can guarantee that each of the two terminals of any RRAM connects to a programming transistor. A basic programming schematic is to allocate two programming transistors for each RRAM.

#### 2.3.4.2 Programming Transistor Sharing

The problem with the basic programming grid is the large area of two programming transistors compared to a single RRAM cell which undermines the area savings brought by the high density of RRAMs. We built a programming grid to allow RRAMs to share programming transistors. In the connection block in Fig. 2.6a, there are six nodes (four logic block pins and two routing tracks). We allocate only six programming transistors, one per node. Then each terminal of the eight RRAMs connects to a programming transistor. In this case, each programming transistor

[3] Though [38] is based on PRAMs, the programming schematics of RRAMs and PRAMs could be instructive to each other since both of these two NVMs are two-terminal devices, as discussed in Section 2.3.1.

is shared by all the RRAMs in the same row/column. This kind of sharing in connection blocks has already been proposed in [8]. However the authors of [8] did not apply the sharing to switch blocks, in which the area is 4x larger than connection blocks. The reason is that in their work, the sharing paths are blocked by the transistors in switch blocks, as shown in Fig. 2.10. This problem

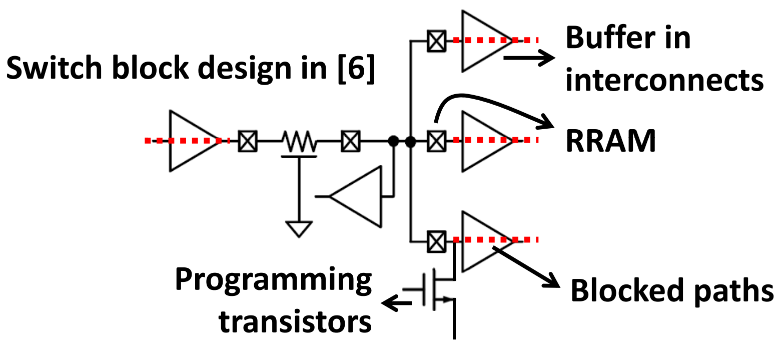


Figure 2.10: In an existing work [8], sharing paths are blocked by transistors (marked as dotted lines).

does not exist in our transistor-less programmable interconnects, where buffers are taken away and put in a separate structure. As shown in Fig. 2.11, a programming transistor in our architecture can be shared by RRAMs, not only in one switch block but also across adjacent switch blocks and connection blocks. In our transistor-less programmable interconnects, any RRAM is between either a track in a routing channel or a pin of a logic block, and it never touches an intermediate node in the interconnects. We allocate programming transistors only at these nodes, and then there will be programming transistors at the two terminals of all RRAMs. Table 2.1 shows a comparison of programming transistor counts in programmable interconnects based on CMOS, the schematic in [8], and this work. We can achieve a 12x programming transistor reduction in switch blocks compared to [8]. Note that this improvement also stems from the 50% reduction of RRAMs in our transistor-less switch blocks. In [8] two unidirectional buffers are used for the connection between two tracks, and each buffer needs one RRAM to control.[4] In our transistor-less switch blocks, only one RRAM is needed to connect two tracks since there are no unidirectional buffers.

[4] The switch block design in [8] is being replaced by directional switches [102] and is not used in our work either.

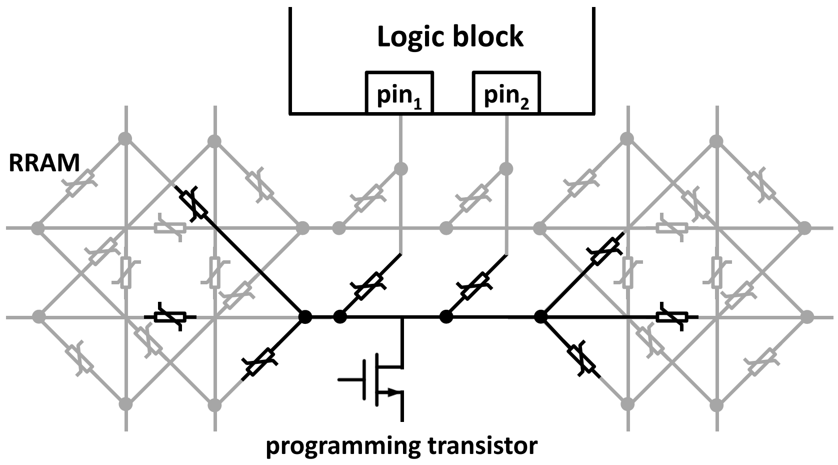


Figure 2.11: In our work, one programming transistor can be shared by RRAMs among switch blocks and connection blocks.

Table 2.1: Comparison of programming transistor counts in programmable interconnects based on CMOS, the schematic in [8], and this work. $N$ represents the routing channel width. $M$ represents the number of logic block pins accessible by one routing channel.

| | CMOS | Work in [8] | This work |
|---|---|---|---|
| $N \times M$ MUX in CB | $M\left(N + 13\sqrt{N}\right)$ | $N + M$ | $N + M$ |
| $N \times N$ SB | $64N$ | $24N$ | $2N$ |
| $100 \times 14$ MUX in CB | 3220 | 114 | 114 |
| $100 \times 100$ SB | 6400 | 2400 | 200 |
| one tile | 12840 | 2628 | 228 |

#### 2.3.4.3 Programming Transistor Selection

An important problem that has been ignored in previous work is the selecting circuit of programming transistors. In a memory system made up of 1T1C DRAM cells or 1T1R PCRAM (or RRAM cells), the row-column structure is often used to select a programming transistor, similar to Fig. 2.12. Contemporary FPGAs can contain $10^8$ configuration bits [103]. The selecting

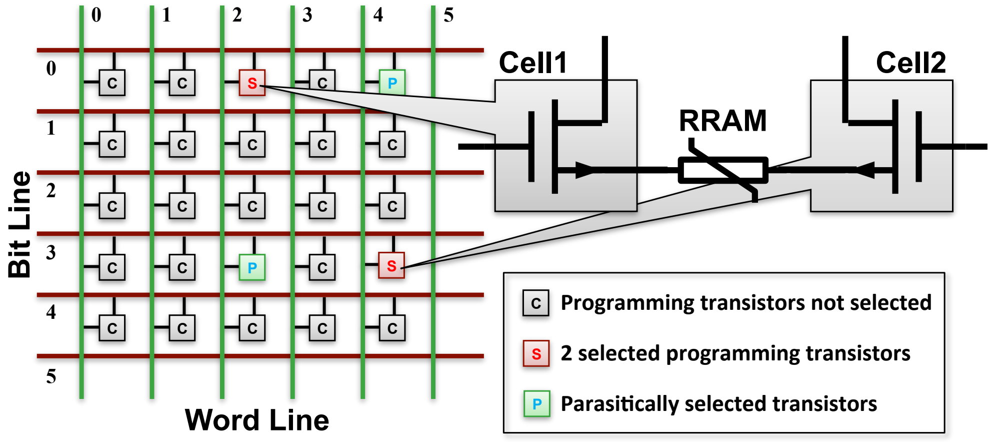

0
1
2
3
4
5
Bit Line
Word Line
Cell1
Cell2
RRAM
C
S
P
Programming transistors not selected
2 selected programming transistors
Parasitically selected transistors

Figure 2.12: Row/column addressing for the programming transistor array of RRAMs in interconnects. If two programming transistors in one array are selected simultaneously, two other transistors will be selected parasitically.

circuit will require only $\sim 2 \times 10^4$ drivers (one per row/column) for the entire FPGA. It will not significantly increase the area of the programming grid which is dominated by the $O(10^8)$ programming transistors for all RRAMs (or for SRAMs in conventional FPGAs). Note that in the memory application of RRAMs, an RRAM is dedicated to only one programming transistor. To program an RRAM which is used as a routing switch in programmable interconnects, we need to select two programming transistors, as discussed in Section 2.3.4.1. This is equivalent to a dual-port random access memory. This kind of access may cause problems in the structure designed for single-port access, as shown in Fig. 2.12.

To realize the dual-port random access without malfunction, one choice is to follow a simple solution in the memory application, where the programming transistors are doubled in all cells, as shown in Fig. 2.13a.

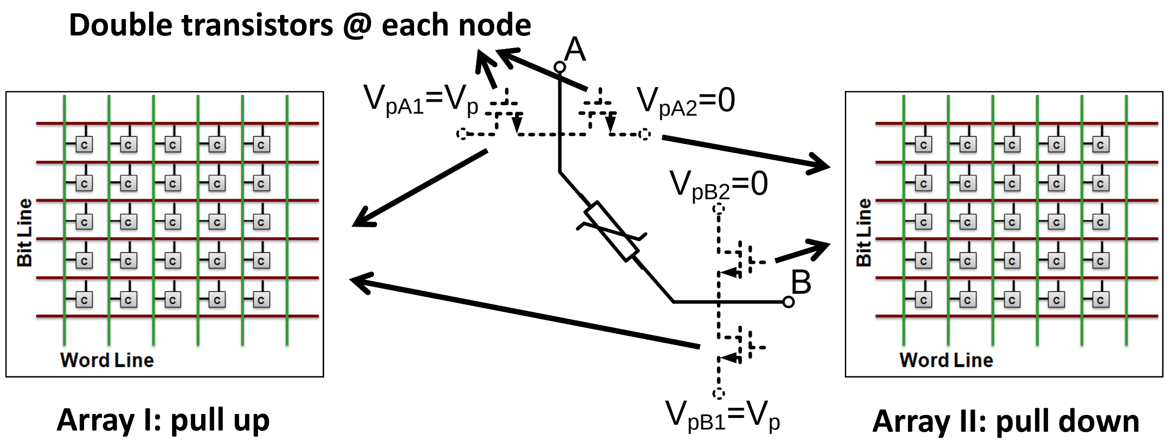
However, we could have a better choice that utilizes the structure of our programming grid to completely eliminate the extra increase in programming transistor counts. In the memory application, pseudo dual-port accesses are supported in a single-port memory if there are multiple memory banks and the memory controller can guarantee that the two simultaneous requests always access two different banks. For FPGA-RPI, we can also partition the programming transistors into different banks. The partitioning must guarantee that the two programming transistors of any arbitrary RRAM belong to different banks. We create a feasible partition of five banks based on the structure of our programming grid, as shown in in Fig. 2.13b. Recall that in Section 2.3.4.2, we allocate programming transistors at all pins of logic blocks and all tracks in routing channels. The programming transistors in y-directional routing channels are assigned to two banks named channel 0 and channel 1 by a parity of channel indices; so are the programming transistors in x-directional routing channels. The programming transistors at the logic block pins are assigned to a separate bank.

### 2.3.5 On-Demand Buffering Architecture

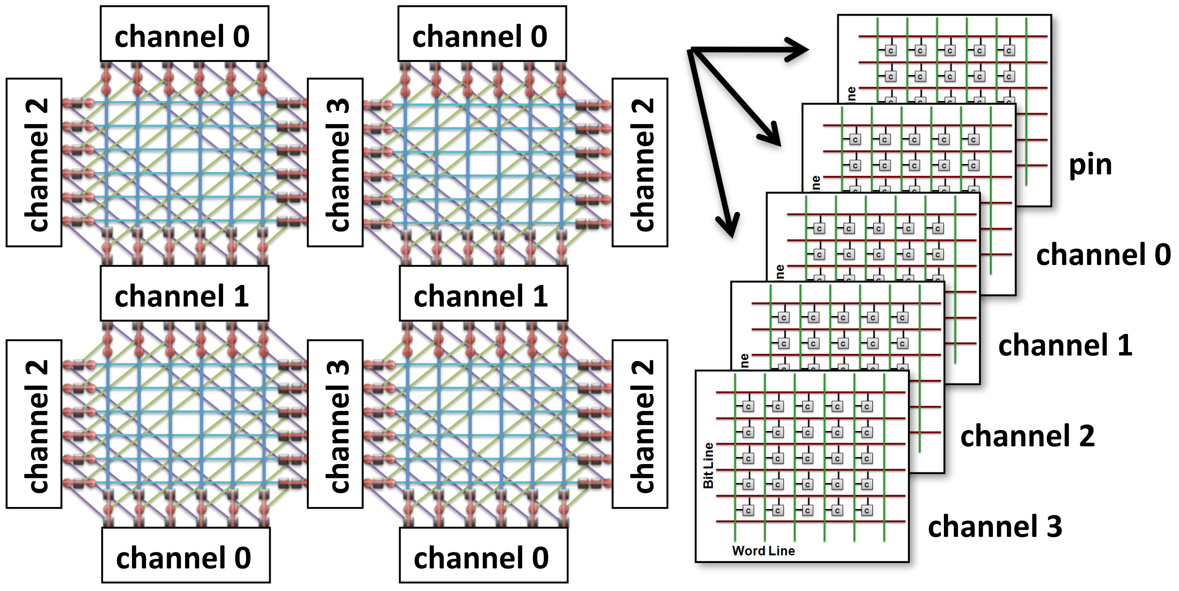
In our FPGA-RPI, buffers are taken away from programmable interconnects and put in the on-demand buffering architecture. This section will discuss the structure, the prerequisite, and the benefits of this architecture.

#### 2.3.5.1 Circuit Schematic and Layout

Fig. 2.14 shows the circuit structure and layout of the on-demand buffering architecture in our FPGA-RPI. A limited number of buffers are prefabricated in routing channels. They can be connected to the tracks in channels via RRAMs. Buffers are shared among tracks in the same channel. Only a track with a high demand for a buffer will be programmed to use a buffer. The demand depends on the routing paths of the user application that is implemented on FPGA-RPI. This



(a)



(b)

Figure 2.13: Two solutions to solve parasitic selection: (a) double programming transistors at each programming node, and (b) partition programming transistors into five banks, where the two programming transistors at the terminals of any arbitrary RRAM lie in two different banks.

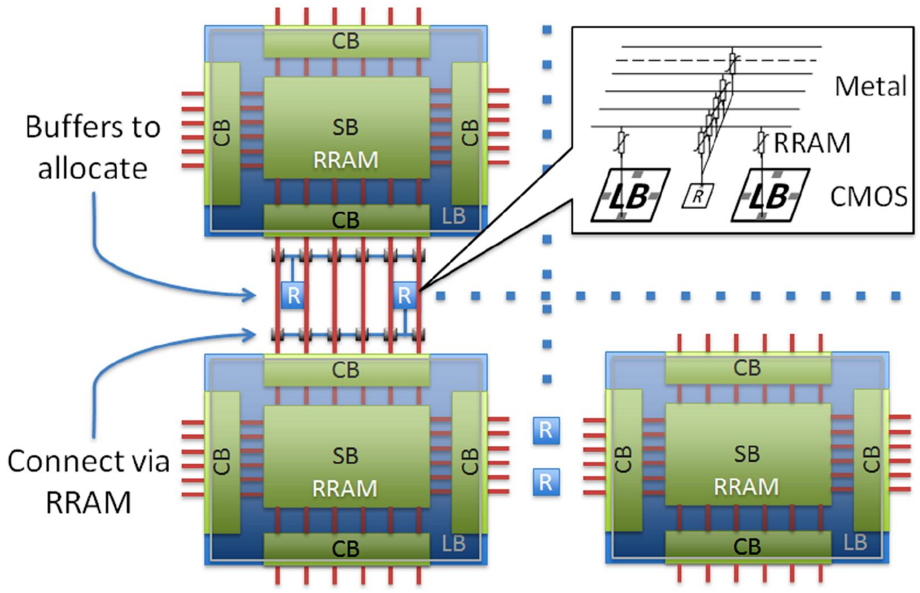


Figure 2.14: The circuit structure and layout of the on-demand buffering architecture in our FPGA-RPI. R = repeater.

mechanism brings benefits of both area and performance; these will be analyzed in the following sections.

To simply the interconnects between shared buffers and routing tracks, we attach each buffer to a routing track via one single RRAM. Therefore we choose the regenerative feedback repeater described in [104] as our buffer cell. This cell design has one signal terminal and can provide drive in both signal directions, as opposed to the conventional unidirectional buffers. As a result, it saves at least half of the buffers by serving the function of two complementary unidirectional buffers with only one repeater.

#### 2.3.5.2 Overhead and Prerequisite

The area overheads of the on-demand buffering architecture are composed of two parts. First, the buffers themselves consume the CMOS area. Second, we also need to allocate programming transistors to program the RRAM-based connections between the buffers and the transistor-less programmable interconnects. The connections are the same as RRAM-based multiplexers used in connection blocks in Fig. 2.6a, and their overhead is small as discussed in Section 2.3.4.2. Recall the comparison in Table 2.1 and assume every channel has 100 tracks and 10 buffers. The connection between the tracks and buffers in one channel is implemented by a $100 \times 10$ multiplexer which costs 110 programming transistors. Among them, 100 programming transistors are placed at the 100 routing tracks and can be shared with switch blocks (similar to the sharing between switch blocks and connect blocks discussed in Section 2.3.4.2). Therefore, the exact area overhead of the programming part of the on-demand buffering is only 10 transistors per routing channel.

The on-demand buffering architecture can also be implemented in conventional CMOS technology, but the area overhead of the multiplexers between buffers and the rest of the interconnects is much higher. According to the cost of the $N \times M$ MUX in Table 2.1, the area overhead is $2300$ extra transistors. This large overhead undermines the benefits of the on-demand buffering architecture and prevents this technology from being applied to conventional FPGAs. In summary, the on-demand buffering architecture requires efficient programmable connections between buffers and routing tracks enabled by emerging technologies like RRAMs.

#### 2.3.5.3 Savings of Unnecessary Buffers

In conventional FPGAs, the locations of buffers in programmable interconnects are predetermined during FPGA design. For example, as shown in Fig. 2.1, each switch from one track segment to another in a switch block (SB) contains a routing buffer . The motivation for this conservative over-buffering is to avoid a potential large RC delay caused by an unbuffered connection between two long unbuffered track segments. But this may not be always necessary, especially in short routing paths. Buffers are needed only in the limited number of routing paths that are long enough to exhibit a quadratic increase in RC delay.

Buffers with fixed locations also result in unnecessary buffers in non-critical paths. Fig. 2.15 shows a typical delay distribution of all paths in a typical MCNC benchmark ‘tseng’ mapped onto a conventional FPGA at 32nm technology node. The paths with less criticality (light color in Fig. 2.15) usually go from an FF through a few logic blocks to another FF, and can be relaxed for

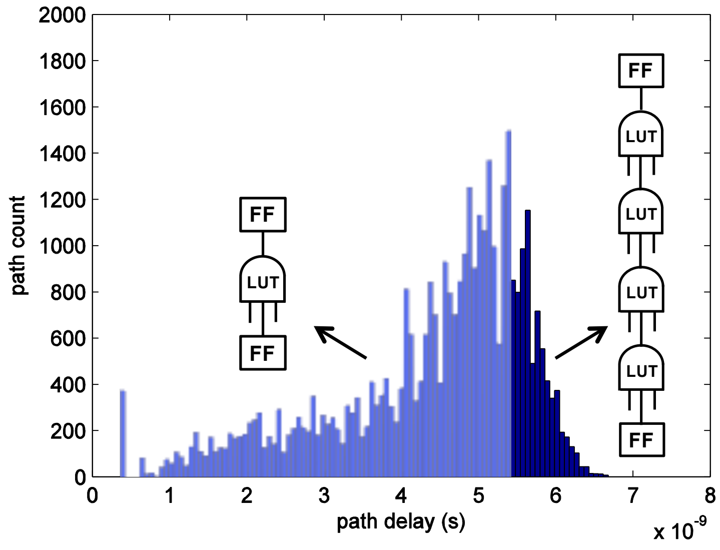


Figure 2.15: Delay distribution of all paths in a typical application for mapping onto FPGA. Most paths can be relaxed for less use of buffers.

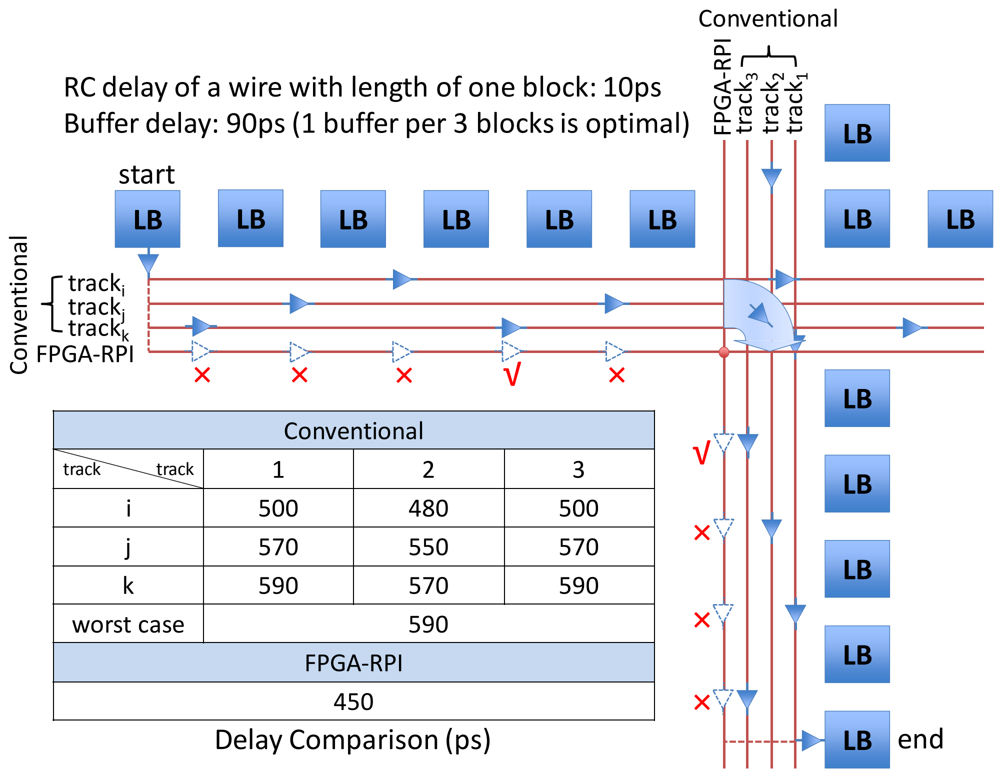
less use of buffers. In conjunction with the paragraph above, we conclude that a routing path needs buffers only if it is a long and critical path.

### 2.3.5.4 Performance Benefit

The fixed locations of buffers in conventional FPGAs also lead to a deviation from optimal timing results. A case study in Fig. 2.16 shows the timing benefit offered by the on-demand buffering architecture in FPGA-RPI when compared to the fixed buffer pattern in a conventional FPGA. To simplify the problem, we assume in this case that the RC delay of a wire with the length of one block is $0.5R_{\text{wire}}C_{\text{wire}} = 10$ps, and that the buffers in interconnects are considered ideal buffers with infinite drive, no input/output capacitance, and a fixed intrinsic delay of 90ps. The optimal length of a wire between two adjacent buffers would be

$$k = \sqrt{\frac{T_{\text{buffer}}}{0.5R_{\text{wire}}C_{\text{wire}}}} = 3$$

In conventional FPGAs, we suppose that the patterns of pre-fabricated buffers in interconnects already follow this optimal result to distribute buffers evenly with a distance of three blocks (as



| Conventional | | | |
|---|---|---|---|
| track \ track | 1 | 2 | 3 |
| i | 500 | 480 | 500 |
| j | 570 | 550 | 570 |
| k | 590 | 570 | 590 |
| worst case | 590 | | |
| FPGA-RPI | | | |
| 450 | | | |

Figure 2.16: Performance comparison between the fixed buffer patten in conventional FPGAs and the on-demand buffering architecture in FPGA-RPI. To simplify, only three routings tracks per channel are shown for the conventional FPGA, and one track is shown for our FPGA-RPI. Note that the buffers in FPGA-RPI are also shared by other tracks in the same channel (not shown here but discussed in Section 2.3.5.1). ✓= buffer allocated. ✗= not allocated.

shown in Fig. 2.16).[5] Since the starting point of each net is unknown before fabrication (the offset can be 0, 1 or 2), the patterns are staggered among different tracks in the same channel, as shown in Fig. 2.16. When a net is driven by routing congestion to a specific track, it is forced to use all the buffers in the track. The table in Fig. 2.16 lists all the possible delays of a net from the logic block “start” to the logic block “end” according to the settings in the conventional FPGA discussed above. The buffers at the logic block pins are not counted in the delay calculation. As shown in Fig. 2.16, the worst-case delay in a conventional FPGA is 590ps. On the contrary, the on-demand buffering architecture in our FPGA-RPI will not suffer from the deviation from the optimal buffer placement in interconnects as does the conventional FPGA. Buffers are allocated exactly one per three blocks, resulting in a total delay of only 450ps in Fig. 2.16.

#### 2.3.5.5 Glitch Hazard and Elimination

The regenerative feedback repeater proposed in [104] is chosen as the buffer cell in our buffering architecture. It is vulnerable to glitches which are shorter than half of the repeater’s delay $t_d$. The authors of [104] concluded that the use of the repeater was limited to FPGA architectures that use self-timing or clocking to eliminate glitches. But the experiments in [104] were based on 1.2$\mu$m which is more than 10 generations away from the current technology node. We revisit this glitch issue at the 32nm technology node and find that this hazard is much alleviated. As shown in Fig. 2.17, when there is a glitch from combinational logics within a logic block (LB), the glitch will go through the RC network formed by routing wires before reaching a regenerative feedback repeater. The RC network will act as a low pass filter and will put a lower bound on the width of a signal that can pass the network. This lower bound can be estimated by the RC delay of the network. According to ITRS [105], the RC delay of the metal wire with a constant length increases by 50% with every generation advancement. Meanwhile, the delay of gate decreases as scaling down continuous. These two trends significantly reduce the probability that glitches

[5] In fact it is not always the case in conventional FPGAs. The distribution of buffers is not usually designed for optimal delay for the sake of area reduction and power consumption savings.

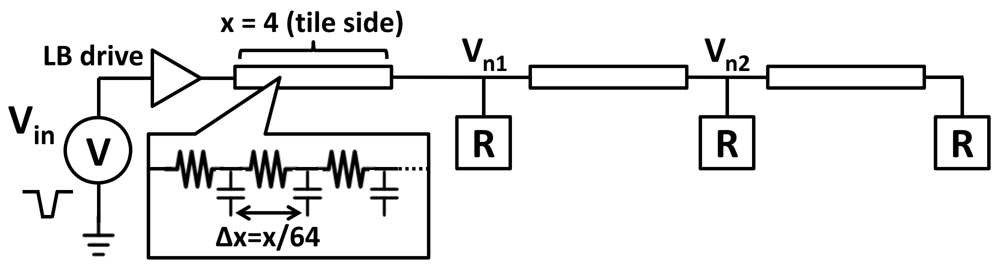


Figure 2.17: Illustration of glitch elimination. Glitches $< 0.5t_d$ are filtered out by the RC network of wire segments before regenerative feedback repeaters. Our on-demand buffering architecture always guarantee sufficient RC delay of wires for this filtering.

exposed to a repeater are shorter than $0.5t_d$. In addition, our on-demand buffering architecture guarantees that the wire segments before any repeater are long enough to filter out all glitches that are shorter than $0.5t_d$. The timing optimization in Section 2.3.5.4 is achieved when buffers in routing paths are allocated with an RC delay equal to $t_d$ per segment. Fig. 2.18 shows the simulation results of the scenario in Fig. 2.17. We set the width of the glitch coming from the

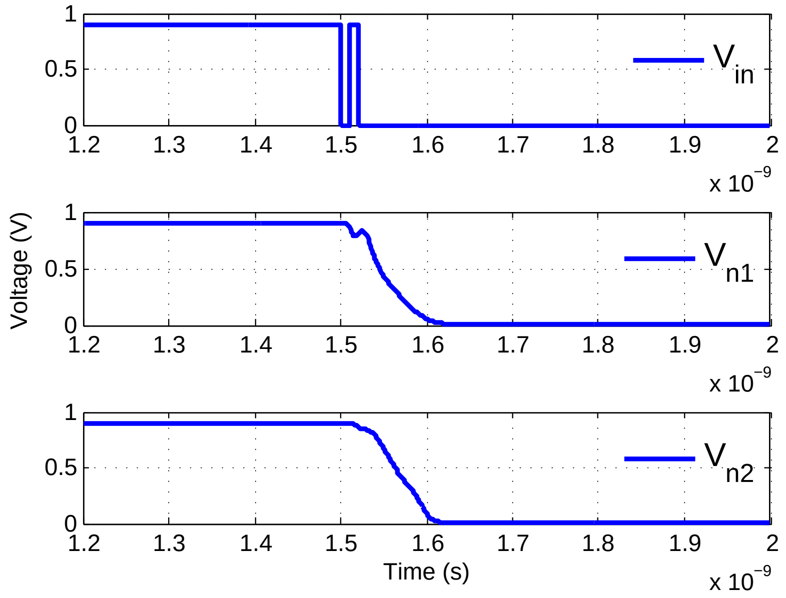


Figure 2.18: SPICE simulation results of the scenario in Fig. 2.17. Simulation settings can be found in Section 2.5.1. Wire segments are modeled as a distributed RC network.

logic block to be $0.4t_d$. The buffer cells in our architecture can still work correctly. Note that in conventional FPGAs without our on-demand buffering architecture, $<0.5t_d$ glitches may still be exposed to regenerative feedback repeaters due to fixed repeater locations in interconnects. For example, the leftmost repeater in track$_k$ in Fig. 2.16 may have glitch hazard since it immediately follows its predecessor without protection from wires.

## 2.4 Design Tool Support

### 2.4.1 CAD Flow

To evaluate performance and energy consumption of FPGA-RPI via widely used benchmarks, such as MCNC benchmarks, we need a CAD flow that is able to accept these benchmarks as input. The CAD flow for conventional FPGAs has been extensively studied [2]. We adopt it and modify it into a flow for our FPGA-RPI, as shown in Fig. 2.19. The input of the flow is the user application to

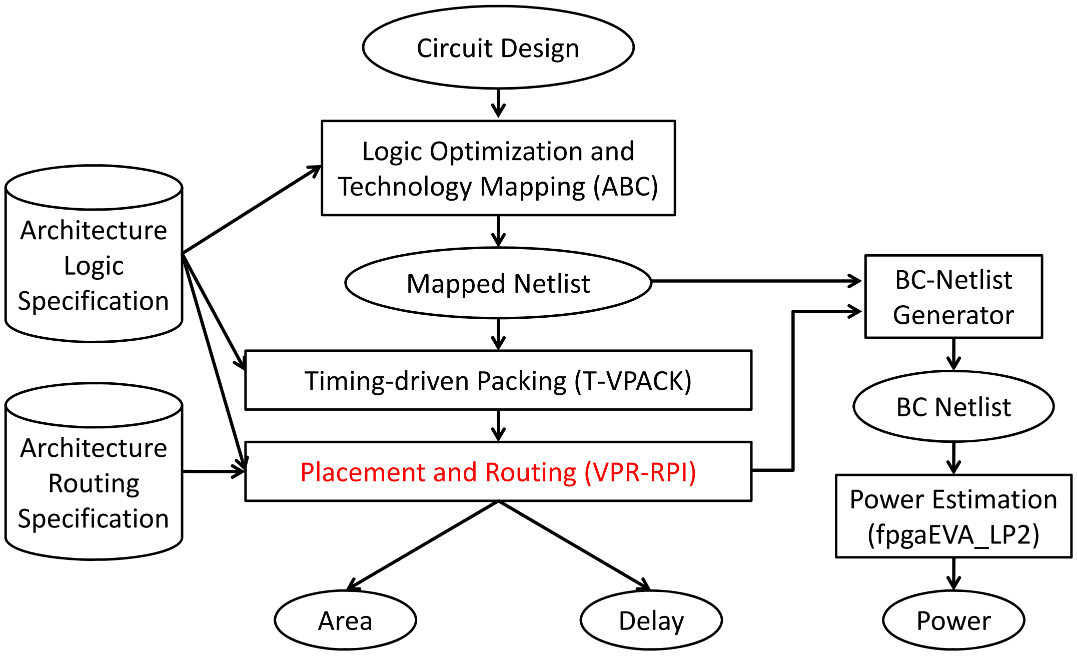

Circuit Design
Logic Optimization and Technology Mapping (ABC)
Architecture Logic Specification
Mapped Netlist
BC-Netlist Generator
Timing-driven Packing (T-VPACK)
BC Netlist
Architecture Routing Specification
Placement and Routing (VPR-RPI)
Power Estimation (fpgaEVA_LP2)
Area
Delay
Power

Figure 2.19: Customized CAD flow for FPGA-RPI. An enhanced P&R tools is developed for FPGA-RPI.

be implemented on FPGA-RPI. The output includes the placement and routing (P&R) result used for programming the application on FPGA-RPI, as well as an estimation of area, delay, and power consumption. The main changes of FPGA-RPI compared to the conventional FPGA are in the programmable interconnects, and we develop a new P&R tool VPR-RPI to adapt to the changes. We will introduce the features of this tool in the following Sections 2.4.2, 2.4.3, and 2.4.4.

### 2.4.2 Equivalent Circuit Model for Interconnect

To perform the timing and power analysis for FPGA-RPI, we first develop an equivalent circuit model for the FPGA-RPI routing structure. Fig. 2.20 shows the equivalent circuit of a representative routing path from the output pin of a logic block to the input pin of another logic block in FPGA-RPI and makes a comparison with that of a conventional FPGA. In the circuit model, the

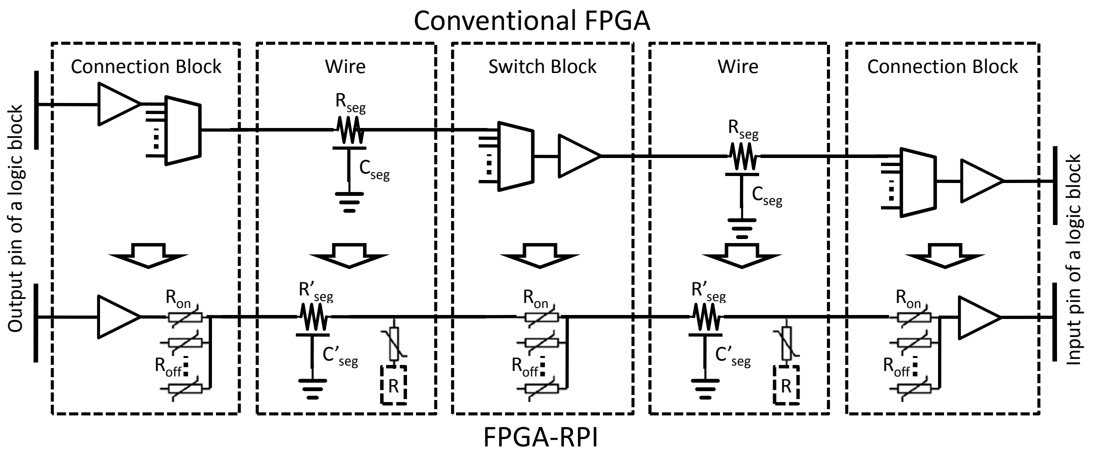


Figure 2.20: Extracted equivalent circuit model of the FPGA-RPI routing structure and comparison with a conventional FPGA.

MUXs in connection blocks and switch blocks are replaced by the RRAMs, of which one is the “on” state and the others are at the “off” state. The wire segments are still modeled as distributed RC lines, but with buffers if allocated. The RC values of wire segments are updated to reflect the tile area reduction in FPGA-RPI.

### 2.4.3 On-Demand Buffer Allocation Algorithm

#### 2.4.3.1 Role in the Flow

To fully utilize the benefits of the on-demand buffering architecture in FPGA-RPI, we develop an algorithm to choose the suitable positions for buffers that will be allocated in programmable interconnects. This will be performed after FPGA routing. Given all the nets routed in programmable interconnects, the goal is to find a buffer option that minimizes the critical delay. The constraint is the total number of prefabricated buffers in each channel. Since the on-demand buffering architecture in FPGA-RPI allows a buffer allocation which is similar to that in ASIC, we considered implanting into FPGA-RPI several state-of-art methods of ASIC buffer allocation [106, 107, 108]. After comparison of their optimization scenarios with ours, we decided to create a new algorithm that borrows the concepts of buffer placement for minimal delay in ASICs [106] and the negotiation-based iterative approach for FPGA routing [52].

#### 2.4.3.2 Optimization Without Resource Constraints

We first focus on the delay minimization without constraints of buffer resource. As analyzed in Section 2.3.5.4, more buffers do not necessarily lead to better timing. We extend a method of buffer placement in ASIC [106] to FPGA-RPI. Buffers are placed in suitable positions in each routed net in FPGA-RPI that connects an output pin and multiple input pins of logic blocks together through programmable interconnects. Each net is equivalent to a routing tree with one source and multiple sinks. We perform a depth-first search on the routing tree to construct a set of delay/$C_{\text{downstream}}$ pairs that correspond to different buffer options for every subtree ($C_{\text{downstream}}$ refers to downstream capacitance). During the combination of buffer options from two subtrees in the search, if both delay and $C_{\text{downstream}}$ of a buffer option are larger than those of another option, the former one will be pruned. The complexity of the algorithm is $O(B^2)$, where $B$ is the total number of legal buffer positions in a routing tree.

#### 2.4.3.3 Consideration of Resource Constraints

If the number of buffer allocated in Section 2.4.3.2 exceeds that of available buffers in some regions, we need to consider the constraint of buffer resource during delay minimization. The method that we use is inspired by the negotiation-based routing iteration procedure in [52]. That procedure minimizes the critical delay under the constraint of track resource in every channel, which is similar to our problem. In our case, we add the buffer overuse cost to the delay of a buffer option as one of the metrics for pruning buffer options (the other metric is still $C_{\text{downstream}}$). The total cost of a buffer option $y$ for a subtree $i$ of a routing tree is expressed as

$$\begin{aligned}\text{Cost}(y) = &\text{Crit}(i) \cdot \text{delay}(y) \\ &+ [1 - \text{Crit}(i)] \cdot \text{DNF} \cdot h(y) \cdot p(y)\end{aligned} \tag{2.1}$$

Crit$(i)$ is the largest criticality among all the paths through which subtree $i$ goes. Criticality of a timing-critical path is close to one, and criticality of a non-critical path is close to zero. DNF is the delay normalization factor. $h(y)$ and $p(y)$ are the historical and present overuse of buffer resources, respectively, in buffer option $y$. $p(y)$ will grow as iteration continues, and the buffers in uncritical paths will be pushed away from congested regions or even be removed. A brief outline of our methodology is provided in Algorithm 1.

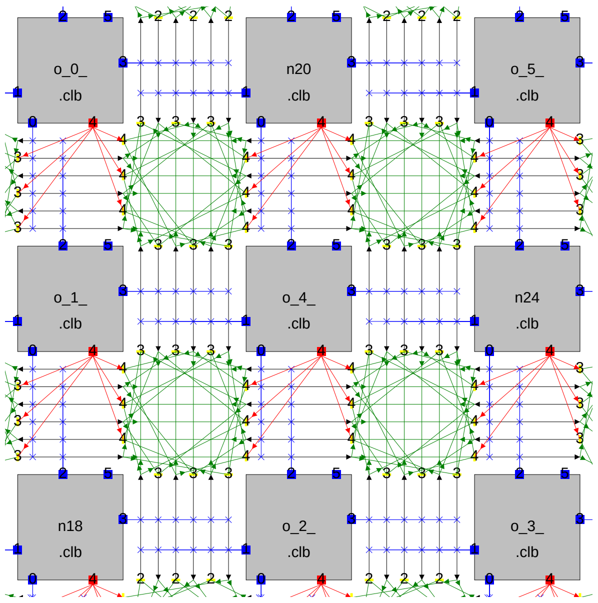
### 2.4.4 VPR-RPI: the P&R Tool for FPGA-RPI

To deal with the novel routing structure of FPGA-RPI in the P&R step of CAD flow, we developed an advanced tool named VPR-RPI (VPR for RRAM-based programmable interconnects) on the base of the state-of-art FPGA P&R tool in academia, VPR [2, 109]. The main contributions of this tool are as follows:

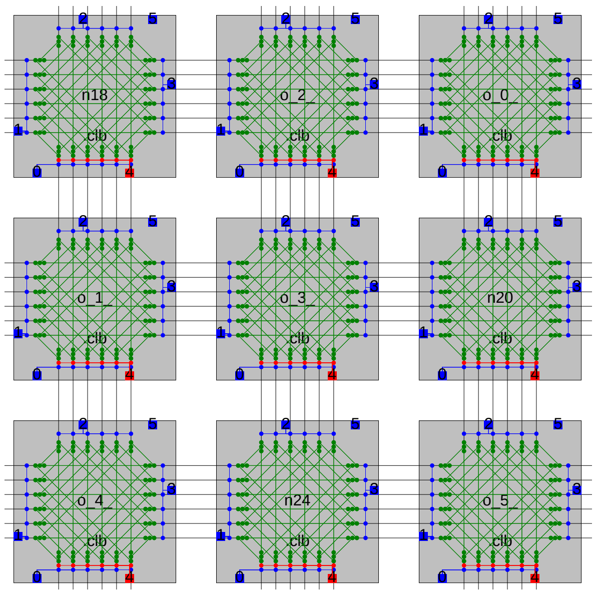
1. VPR-RPI can deal with the routing graph of FPGA-RPI as shown in Fig. 2.21b. In this figure the gray blocks are I/O blocks and logic blocks of FPGA, and the diagonal wires are the routing paths available in switch blocks. The switch blocks and connection blocks in

**Algorithm 1:** A brief outline of the buffer allocation algorithm under the constraint of buffer resources.

```
Input  : Routing trees $RT_1, RT_2, RT_3, ..., RT_n$
Output: A buffer option $Y = \{y_1, y_2, ..., y_n\}$ that defines buffer allocation in each $RT_i$
1 while ∃ overused buffers do
2     foreach $RT_i$ do
3         rip-up $y_i$ and update buffer congestion $p(y)$;
4         $y_i$ = AllocBuf ($RT_i$) in [52] with Cost($y$) in Eq. 2.1 ;
5         update buffer congestion $p(y)$;
6     end
7     update historical buffer congestion $h(y)$;
8 end
```



(a) VPR for conventional FPGA

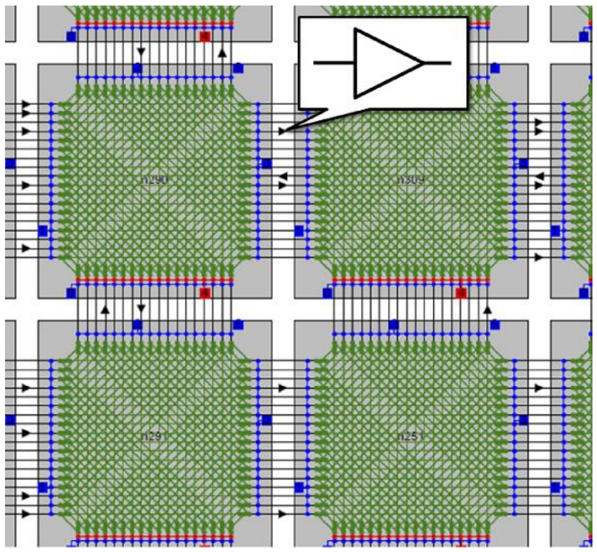


(b) Our VPR-RPI for FPGA-RPI

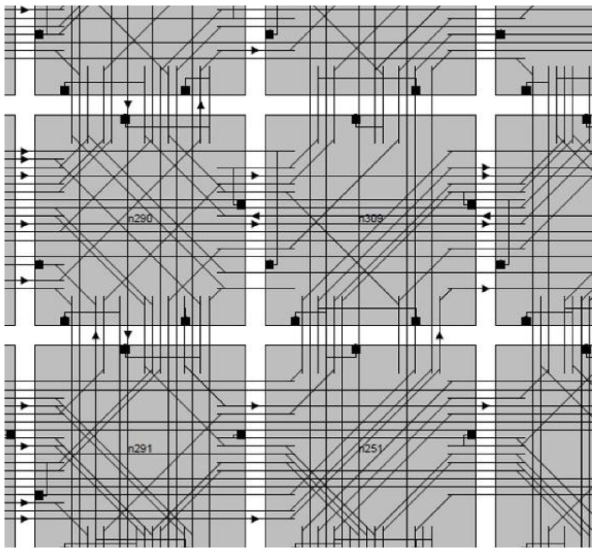
Figure 2.21: Overview of the two tools, VPR and VPR-RPI.

VPR-RPI are placed over logic blocks and provide connections for logic blocks nearby in a different way than that of the conventional FPGA shown in Fig. 2.21a.

2. VPR-RPI is integrated with the algorithm to perform the on-demand buffer allocation described in Section 2.4.3. The tool can display where buffers are needed for allocation in the final routing result, as shown in Fig. 2.22. The positions for the allocation of buffers are marked with a black delta shape. The interconnect view of the routing result in Fig. 2.22b is quite similar to that of ASIC. We describe the good performance of FPGA-RPI in Section 2.5.



(a) Routing resource view.



(b) Routing result view

Figure 2.22: View of the buffer allocation result generated by VPR-RPI.

## 2.5 Simulation Results

### 2.5.1 Settings

We tried to mimic the architecture of the commercial FPGA, the Xilinx Virtex7 family, for meaningful evaluations. It uses 28 nm processes [3]. Due to lack of technology libraries, we are unable to choose the exact 28nm models for our experiments. Instead, we choose the nearby 32nm technology node. We use the settings of eight 6-input look-up tables (6-LUTs) per logic block (LB); these are used starting from Virtex5 to the most recent Virtex7 [3]. The routing channel width is obtained with the help of the FPGA Editor in the Xilinx ISE environment. We refer to the intelligent FPGA Architecture Repository (iFAR) [9] for other less important architectural settings that are not provided by the Virtex7 datasheets [3]. We estimate the timing parameters using the lookup tables in ITRS 2011 [105] and SPICE simulation with PTM device models [110, 111]. The RRAM model is extracted from the measurement results of the RRAMs fabricated by the process in [6]. Its "on" resistance is $\sim 10^3$ohm and its "off" resistance is $\sim 10^9$ohm. We also perform a sensitivity analysis on the RRAM parameters. The 20 largest MCNC benchmark circuits [112] are used as the input of the CAD flow for conventional FPGA and FPGA-RPI, and comparisons are made on the output of the CAD flow from the aspects of area, delay and power.

### 2.5.2 Optimize the On-Demand Buffering Architecture

Compared to a conventional FPGA, FPGA-RPI has a unique on-demand buffering architecture. Before evaluation, we need to explore the new parameters related to this architecture. We found that a uniform distribution of buffers over routing channels best serves the buffer demands of different applications, similar to the uniform distribution of routing channel widths arrived at [2]. Then we need to decide how many buffers to prefabricate per routing channel. In addition, the size of the buffers needs to be optimized again since the buffering solution has changed as compared to a conventional FPGA. Based on the observation that a smaller buffer size will increase the number of buffers demanded by each channel, we explore these two parameters simultaneously. Note that both of the parameters have an impact on both the area and performance of FPGA-RPI. Therefore, we use the area-delay product as the optimization goal. Fig. 2.23 shows the result. Here, buffer size is normalized to the optimal size of the conventional FPGA in iFAR[9]. The area-delay product is calculated from the geometric mean of the results over the 20 benchmarks and normalized to the value of the case without any buffer allocated. The drop of the area-delay product curve in Fig. 2.23

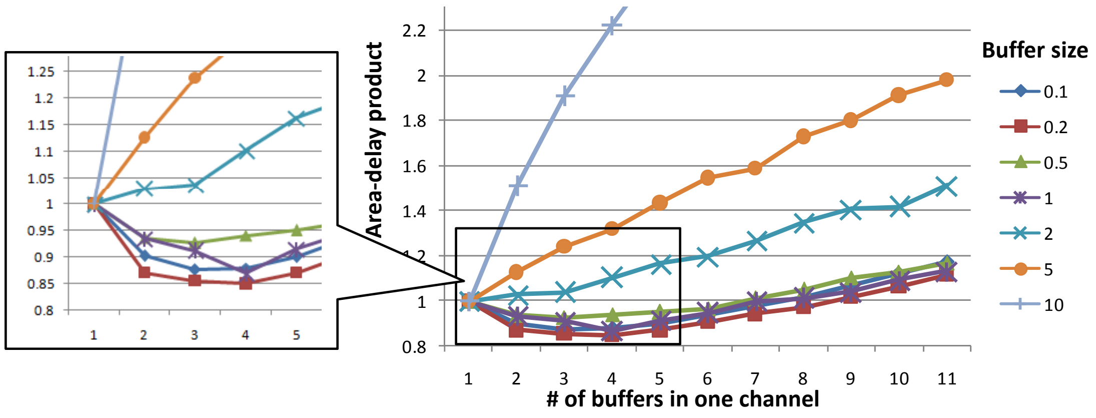


Figure 2.23: Exploration of the impact of buffer sizing and richness on the area-delay product. Buffer sizes and area-delay products are normalized.

is a result of the timing improvement when buffer demand is not fully satisfied. The rise of the curve is a result of the larger area with more buffers. Based on the enlarged portion of Fig. 2.23, we determine that in our FPGA-RPI the number of buffers per channel is four, and the buffer size is 0.2 times that used in a conventional FPGA (based on the optimal settings in iFAR).

### 2.5.3 Evaluation of FPGA-RPI

#### 2.5.3.1 Footprint

Fig. 2.24 provides an area comparison of programmable interconnects in different architecture settings. It shows the area reduction breakdown of the FPGA programmable interconnects achieved by different technologies proposed in this work. Starting from the baseline FPGA, we first replace the routing switches and their SRAMs in programmable interconnects with RRAM-based switches (as discussed in Section 2.2.2.2 and Section 2.3.2). These RRAMs are placed over CMOS transistors, and their footprint fits into that of the CMOS transistors below; this is guaranteed by our RRAM-friendly layout design in Section 2.3.3. So we can safely skip the area of RRAMs and metal wires in this evaluation. The contribution of RRAMs to the CMOS footprint will be their programming transistors shown in Fig. 2.24. Then we apply our programming transistor sharing

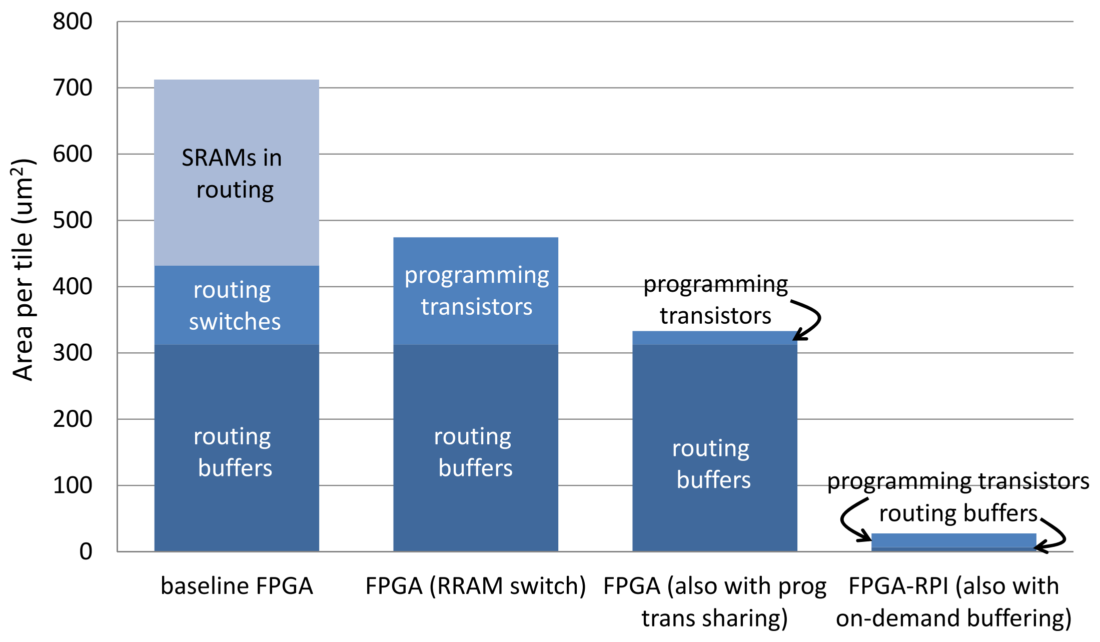


Figure 2.24: Area comparison of programmable interconnects in a single tile.

(as discussed in Section 2.3.4.2). Last, we implement our on-demand buffering architecture (as discussed in Section 2.3.5.1). The final area savings are 96% of the programmable interconnects, which corresponds to 50% of the total FPGA area (recall the FPGA area breakdown in Fig. 2.2).

#### 2.5.3.2 Performance

Fig 2.25 shows a performance comparison of programmable interconnects in different architecture settings.[6] The speedup of FPGAs with the three technologies applied step by step mainly stems from the shorter signal paths due to the area reduction. This observation corresponds with that of the 3D FPGAs [22]. In addition, the on-demand buffering architecture also drives the buffer solutions closer to the optimal. The geometric mean of the delay reduction achieved by our FPGA-RPI is 55%.

[6] The improvement trend is not always consistent for every benchmark due to certain delay noise in VPR [113].

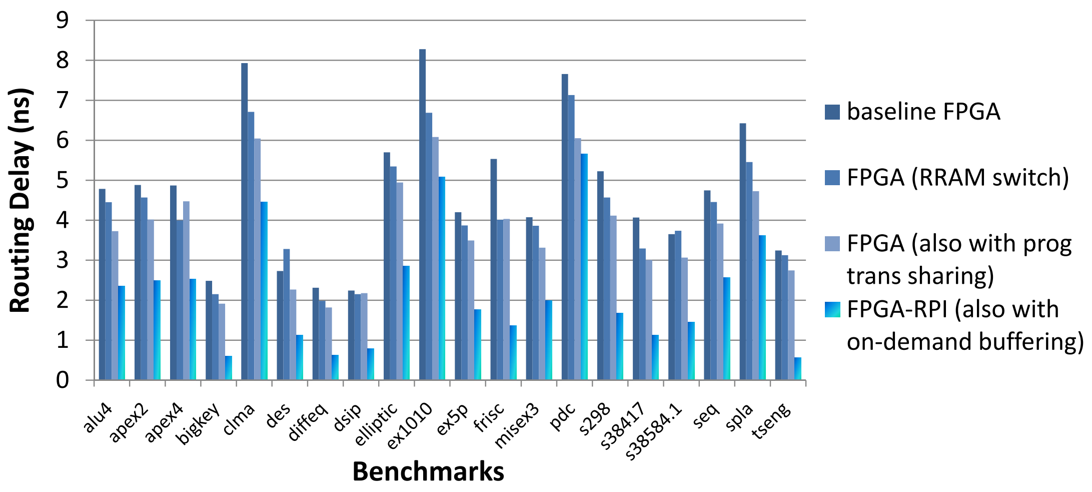


Figure 2.25: Comparison of the delays contributed by programmable interconnects in critical paths in FPGAs over benchmarks.

#### 2.5.3.3 Power Consumption

Fig. 2.26 shows a leakage power comparison of programmable interconnects in different architecture settings. Note that the programming transistors of RRAMs are not counted in the evaluations of leakage power since the programming grids will be completely turned off during the operation of FPGAs. Our FPGA-RPI reduces the leakage power of programmable interconnects by 79%. Note that RRAM is also nonvolatile. Power gating can be applied to RRAM-based FPGAs to further reduce the leakage power during standby. Fig. 2.27 shows a dynamic power comparison of programmable interconnects in different architecture settings. All of the three technologies applied step by step lead to area reduction and thus smaller capacitances of shorter wire interconnects. In addition, the on-demand buffering architecture removes many unnecessary buffers in both short paths and uncritical paths. The geometric mean of the dynamic power savings achieved by our FPGA-RPI is 56%.

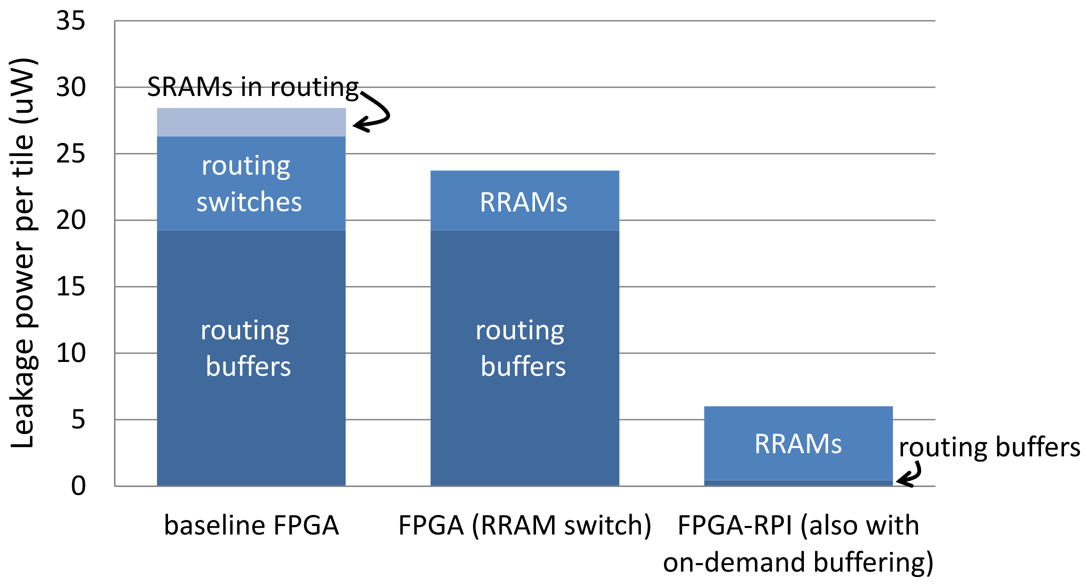


Figure 2.26: Comparison of the leakage powers contributed by programmable interconnects in a single tile.

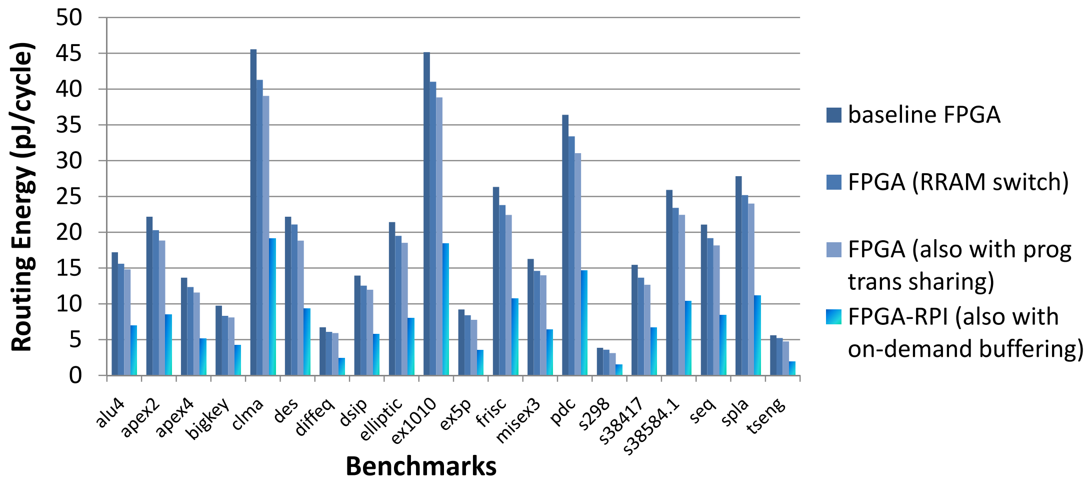


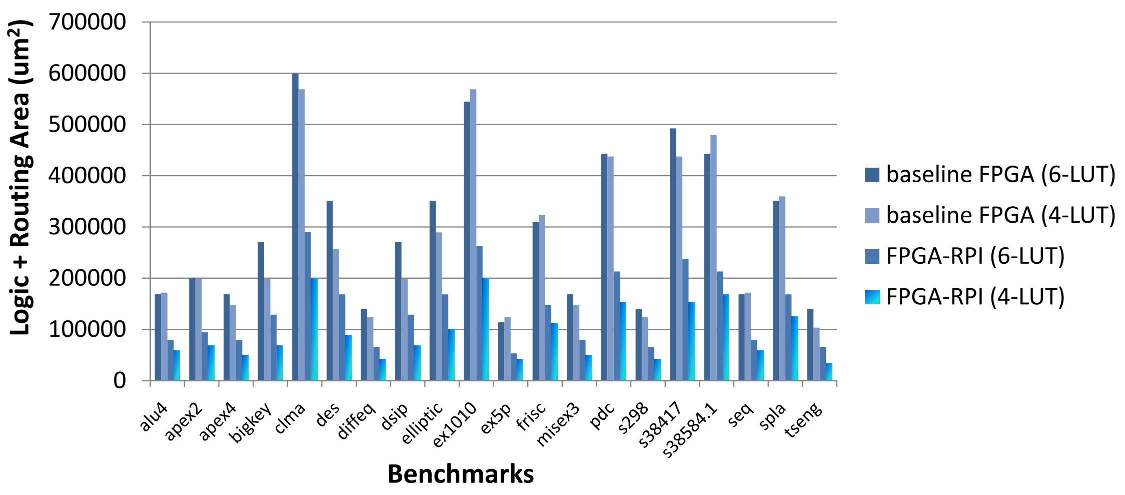
Figure 2.27: Comparison of the dynamic powers contributed by programmable interconnects in FPGAs over benchmarks. To separate the impact of the clock period from the dynamic powers, we use pJ/cycle as the metric.

### 2.5.4 Impact on LB Architectural Design

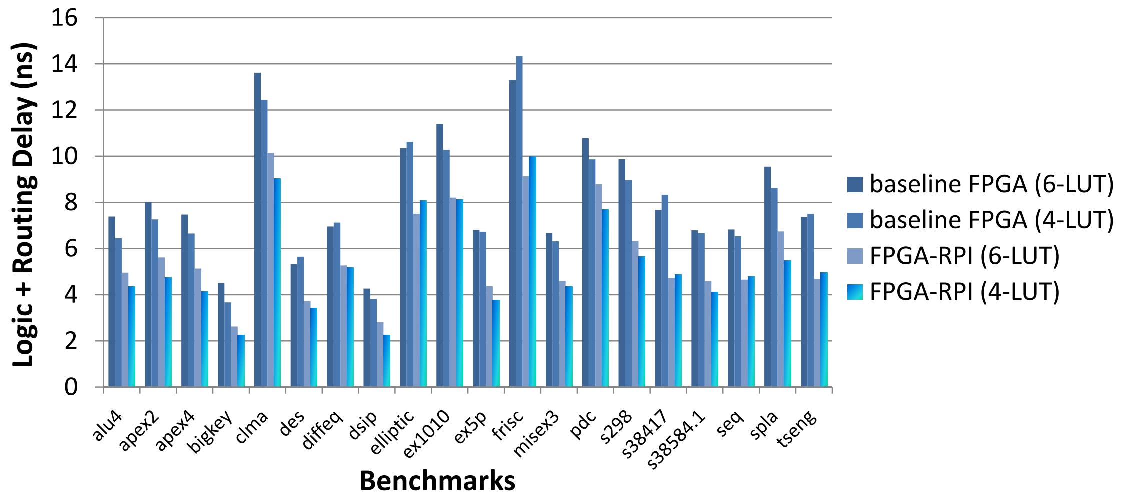
The increasing overhead of programmable interconnects pushes FPGA vendors to larger and more complicated logic blocks (LBs). For example, the Xilinx FPGAs in the Virtex family before Virtex5 put eight 4-input look-up-tables (4-LUTs) in one LB [114]. The FPGAs starting from Virtex5 to the most recent Virtex7 switched to eight 6-LUTs in one LB [3]. The proportion of LBs in today's FPGAs is close to 50%, and this large proportion masks the benefits coming from the improved programmable interconnects. In literature [22, 23, 8], researchers intentionally chose smaller LBs as their architecture settings to highlight the benefits from programmable interconnects. Here, we provide a more complete study to see which LBs will lead to a smaller footprint, higher speed, and lower power consumption for both logic and routing parts. Fig. 2.28 shows the overall footprints, delays, and power consumptions in the FPGA architectures with LBs made up of 6-LUTs and 4-LUTs. Before we apply our RRAM-based programmable interconnects, the FPGA architecture with more complicated LBs is generally comparable or even better than that with simpler LBs. After we improve the programmable interconnects with our FPGA-RPI, the FPGA architecture with simpler LBs is better since it receives more benefits from our techniques. This architectural change further reduces the total area, delay, and power consumption by averages of 34%, 7.6% and 25%, respectively.

### 2.5.5 Sensitivity Analysis

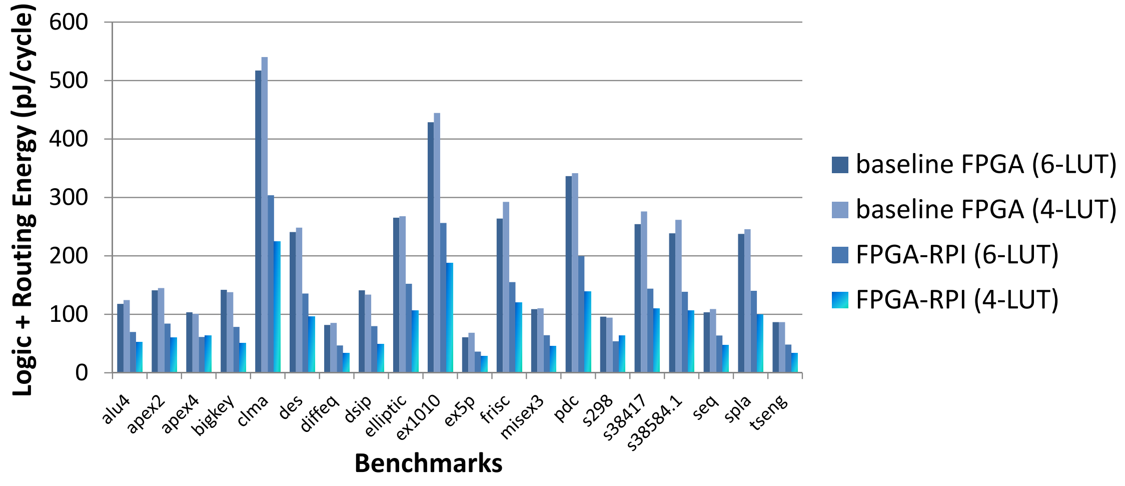
Since the RRAM is an emerging technology, its device parameters tend to be hard to control. This motivates us to conduct a sensitivity analysis for RRAM parameters. The key parameter of RRAMs is the on/off ratio. We sweep the on/off ratio by two orders of magnitude, and Fig. 2.29 shows the impact. The analysis is performed on the FPGA-RPI with 4-LUT, and the results are shown in the form of gains compared to the conventional FPGA with 6-LUT. On each curve in Fig. 2.29, the on/off ratio is fixed and there is a tradeoff between the "on" resistance $R_{\text{on}}$ and the "off" resistance $R_{\text{off}}$. Larger $R_{\text{off}}$ leads to smaller leakage power and larger energy savings. Larger $R_{\text{on}}$ leads to



(a) Total footprints.



(b) Total delays.



(c) Total energy consumptions per clock cycle (including both leakge and dynamic, assumed to operate at highest speed).

Figure 2.28: Comparisons of FPGA architectures with more complicated LBs and simpler LBs.

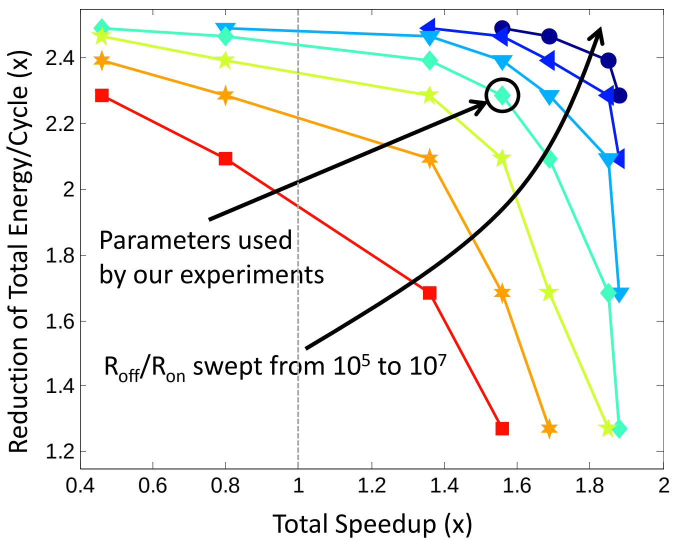


Figure 2.29: Sensitivity analysis of the RRAMs on/off ratio for FPGA-RPI. Each line corresponds with a fixed on/off ratio. There is a tradeoff between delay and power savings on each line.

larger resistance on signal paths and smaller speedup. We suggest trading off energy savings for larger speedup since we can apply power gating to save more energy by using the nonvolatility of RRAMs. When we switch from a line of a smaller on/off ratio to one of a larger ratio in Fig. 2.29, we will achieve improvement in both performance and power savings.

## 2.6 Interconnect Improvement in LBs

There are local interconnects in logic blocks (LBs). We can improve the local interconnects as we do for the global interconnects. For the sake of simplicity, we consider a basic LB model taken from [2] and shown in Fig. 2.30. Note that there are large MUXes between the input pins of LBs and those of basic logic elements (BLEs). They allow each of the BLE inputs to connect to any of the LB inputs or any of the BLE outputs. This feature of being *fully-connected* simplifies the design of CAD tools and is implemented in many commercial FPGAs [2]. The structure of these local interconnects is similar to the $N \times M$ MUXes used in connection blocks in Fig. 2.6a. We can also build them by RRAMs and apply our programming transistor sharing as discussed

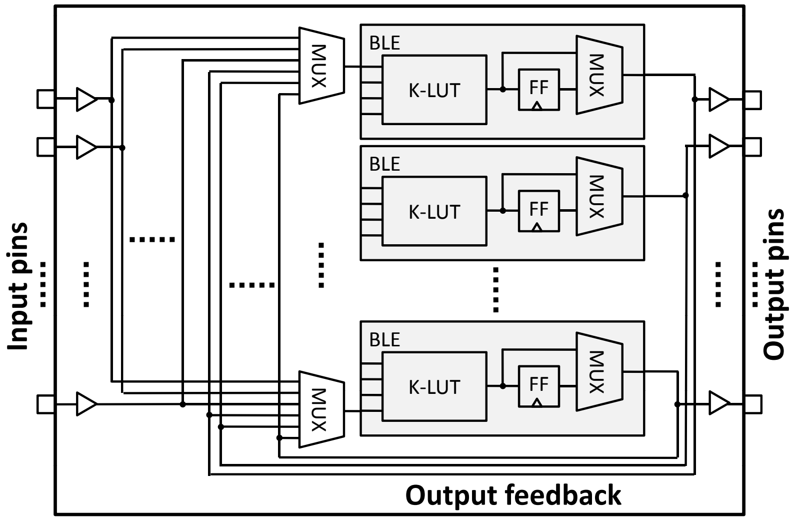
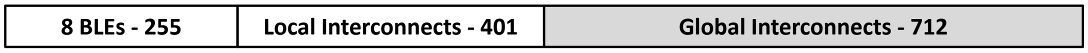
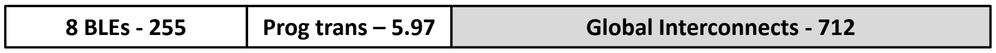


Figure 2.30: The internal structure of a state-of-art LB described in [2].

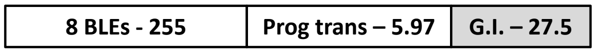
in Section 2.3.2 and Section 2.3.4.2. Fig. 2.31 shows the area reduction achieved by RRAM-



(a) Baseline FPGA. Total area = 1370 um$^2$.



(b) FPGA with RRAM-based local interconnects. Total area = 973 um$^2$.



(c) FPGA-RPI also with our renovated global interconnects. Total area = 289 um$^2$.

Figure 2.31: Area comparison of a single tile in an FPGA. The white parts belong to LBs. Each LB contains eight 6-LUTs. The number of LB inputs is 27, optimized by design experience in [9].

based local interconnects. The reduction of the LB area due to RRAM-based local interconnects is 60%. It further increases the overall FPGA area reduction enabled by RRAMs. It also leads to extra benefits in performance and power savings due to shorter signal paths, as discussed in Section 2.5.3 and [22]. Since there are many different kinds of LB structures in modern FPGAs, we only use Fig. 2.30 as a case study and expect that the area reduction will vary for different types of LB designs.

## 2.7 Conclusion and Future Work

This chapter presents FPGA-RPI, a novel FPGA architecture with RRAM-based programmable interconnects. Programmable interconnects are the dominant part of conventional FPGA. We use RRAMs to build programmable interconnects and optimize their structures for the RRAM properties. A customized CAD flow is provided for FPGA-RPI, with an advanced P&R tool named VPR-RPI that was developed for FPGA-RPI to deal with its novel structures. Results show that the programmable interconnects of FPGA-RPI have a 96% smaller footprint, 55% higher performance, and 79% lower power consumptions compared to other FPGA counterparts. Note that no die-stacking is needed to achieve this degree of area reduction and speedup. If 3D integration technology is introduced in the future to stack several FPGA-RPIs together, at least 2x more improvement in density and speedup can be expected according to the experimental results on 3D architectures in [22].

Though this work shows the significant benefits of NVMs that can be brought to FPGAs and offers solutions to important technical challenges, there are still problems to solve to make NVM-based FPGAs ready for manufacturing. A common problem in NVM-based FPGAs [31, 32, 33, 34, 35, 36, 37, 38, 23, 39, 8, 40, 41, 42] is the degraded NVM stability. During FPGA operations, NVMs have to continuously being monitored to control signal paths in programmable interconnects and suffer from constant voltage stresses. When NVMs are used as routing switches in FPGA interconnects [38, 23, 39, 8, 40, 41, 42], the reliability of CMOS gates might also degrade. When voltages are applied to the two terminals of an RRAM for programming, the CMOS gates in logic cells which are directly connected to the RRAM will be exposed to this high voltage and can suffer from pulse stresses. Given the high benefits of NVM-based FPGAs, it is worthwhile to solve these reliability problems in order to make real products from this technology. Chapter 3 in this thesis is an example of such efforts.

# CHAPTER 3

# Defect-Aware Routing on Nanodevice-Based Programmable Interconnects

## 3.1 Introduction

As discussed in the previous chapter, it is beneficial to introduce nanodevices to FPGAs. In fact, a number of FPGAs based on emerging nanodevices have been explored in the past few years [45, 38, 23, 35, 25, 41]. The emerging nanodevices include resistive RAM (RRAM) [45], phase-change RAM (PCRAM) [38], nanoelectromechanical (NEM) relays [23, 35], and molecular switches [25, 41]. They can be generalized as bistable switches which can be programmed between the ”on” and ”off” states, as shown in Fig. 3.1a [46]. A single nanodevice can function as a routing switch

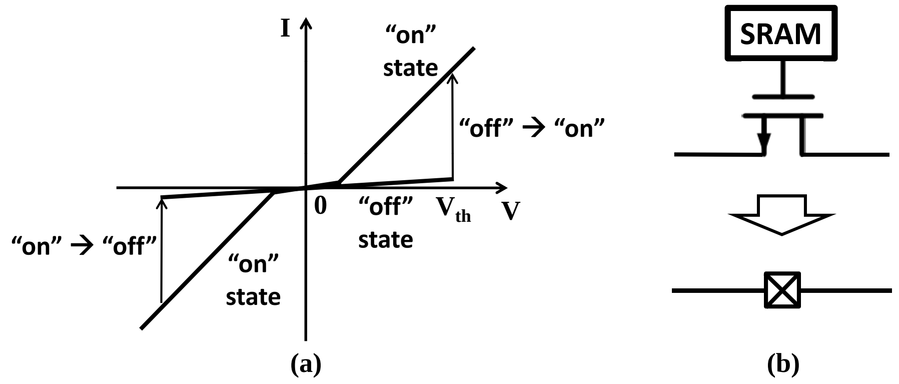


Figure 3.1: Illustration of nanodevices. (a) Hysteresis characteristic of a two-terminal RRAM nanodevice. (b) Function as a routing switch in place of a pass transistor and its six-transistor SRAM cell.

in place of a pass transistor and its six-transistor SRAM cell in conventional FPGAs, as shown in Fig. 3.1b. Programmable interconnects of FPGAs can therefore be built from nanodevices and have smaller footprints. In addition, these nanodevices are fabricated among metal layers and do not contribute to the footprint of the CMOS transistors below them. Note that conventional FPGAs prefer multiplexers to pass transistors as the basic circuits in programmable interconnects for fewer configuration bits, though signals have to pass more levels of gates in multiplexers. However, nanodevices provide configuration bit storages along with signal paths. Nanodevice-based FPGAs switch back to pass transistors for higher performance [45, 38, 23, 25, 41]. These nanodevices are also nonvolatile devices and can save significant leakage power. To summarize, nanodevice-based FPGAs show a significant potential to save footprint, critical path delay and power consumption. For example, a NEM-relay FPGA in [23] achieves savings of 43%, 28% and 37% respectively. A RRAM-based FPGA proposed in [45] achieves savings of 80% , 56% and 39% respectively.

In nanodevice manufacturing, defects are a certainty, and reliability becomes a critical issue. The projected defect rate of nanodevices can be up to $10^{-1}$, which is much higher than the level of $10^{-9}$ to $10^{-12}$ in CMOS systems [47, 35]. A number of approaches have been proposed to improve the FPGA yield by leveraging its reconfigurable structures. They can be categorized into several groups, including component-specific implementation [48, 49, 50, 35, 47, 41, 46], design-specific testing [115, 116, 117], and adding redundancy to FPGA architecture [118, 119]. Among them, component-specific implementation provides the highest level of defect tolerance at a modest area overhead. That's because it provides implementations adaptive to each defect. Defect-tolerant CAD tools are needed to configure FPGAs to work around all detected faults. Component-specific implementation proves beneficial for alleviating process variation as well, which is another main issue as feature sizes scale toward atomic limits [120]. The overhead of component-specific implementation is that the manufacturer needs to try programming every nanodevice in an FPGA chip to obtain the defect map. Then the defect-tolerant CAD flow needs to be performed for every defective chip. Given the high defect rate of nanodevices, all the approaches that target at defect tolerance in the nano era choose component-specific implementation [48, 49, 50, 35, 47, 41, 46]. Our work also belongs to this group.

Defects in programmable nanodevices are manifested as losses of configurability. A defective nanodevice may be stuck at an either ”on” or ”off” state [47, 46]. If the nanodevice is used in a logic block, it leads to a stuck-at-1 or stuck-at-0 bit. If the nanodevice is used in programmable interconnects, it leads to a stuck-closed or stuck-open switch.

Defect tolerance in logic blocks has been explored in recent years [50, 35, 47]. However, in an FPGA chip, programmable interconnects usually occupy 2-4x more area than logic blocks [22, 23, 25], and are the dominant part. Tolerance of defects in programmable interconnects requires more attention than that in logic blocks. The stuck-open switches in interconnects can be easily solved by removing the broken edges from the routing graph [51, 41]. The authors of [41] showed that yield can remain nearly 100%, even at a defect rate of stuck-open switches as large as 50%. However, they ignored stuck-closed switches, which are much more challenging than stuck-open switches. In Section 3.2, we will show that stuck-close switches need to remove $>$10x routing resources than stuck-open switches when simple defect avoidance is used. However, the good thing is that a stuck-closed switch can still be used opportunistically if we can guarantee that the two nodes shorted by the switch are always mapped to the same net. They can be regarded as extra shorting constraints during the routing phase [46], just as done for logic blocks in [35, 47]. Along with huge resource savings, this method has two challenges: 1) Defects in programmable interconnects can propagate over the entire chip, and their tolerance has to be solved in a more global way, with scalability taken into account; 2) Existing FPGA routing algorithms work on a directed routing graph which assumes that all the edges can programmed to be open or closed, and shorting constraints break this assumption. The second challenge pushes some researchers to switch to algorithms that can easily deal with shorting constraints, but lead to poor scalability and solution quality. For example, the SAT-based method in [46] uses Boolean clauses to apply defect constraints, but has high time complexity due to the large search space and is unable to develop a timing-driven flow based on the satisfiability solver.

The contributions of this chapter are as follows. First it provides a complete defect analysis. Then we propose a scalable algorithm to perform timing-driven routing under shorting constraints.

We start from the negotiation-based procedure [52], the state-of-art routing algorithm of FPGAs, to maintain the circuit performance and the tool runtime. We extend the idea of the resource negotiation to balance the goals of timing and routability under shorting constraints. We also observe that a routing node will be logically inconsistent with certain nets due to shorting edges. Therefore, we add a mechanism to achieve fast pruning before routing of each net. We also develop several techniques to guide the router to map the shorting clusters to those nets with more shared paths for better utilization of routing resources while automatically balancing it with circuit performance. In addition, we discovered that some logic blocks will become virtually unusable due to shorted pins found in the defect analysis. We enhance the placement algorithm to recover these logic blocks.

## 3.2 Quantitative Defect Analysis

This section provides a complete impact analysis of both stuck-open and stuck-close defects in programmable interconnects. To the best knowledge of the authors, this is the first work that systematically evaluates the impacts of the two defect types and observes new phenomenon.

### 3.2.1 Impact on Routing

As mentioned in Section 3.1, defects in programmable nanodevices are manifested as losses of configurability [47, 46]. When these nanodevices are used as routing switches in programmable interconnects, the defects can be categorized into two types. The stuck-open defect indicates that the connection between two nodes cannot be used. The stuck-closed defect indicates that the two nodes on the two sides of the switch will always be shorted. Fig. 3.2 shows an example of the two types of defects.[1] To solve a stuck-open defect, e.g., the switch between routing track A and B in Fig. 3.2, we can avoid using it during the routing process. The impact of removing a single edge from the routing graph is very limited since there are always many alternative paths between two

[1] For demonstration purpose, only one routing track per channel is shown in the figure. Routing buffers are also omitted. FPGAs have more complex structures and our tool works on a generalized structure.

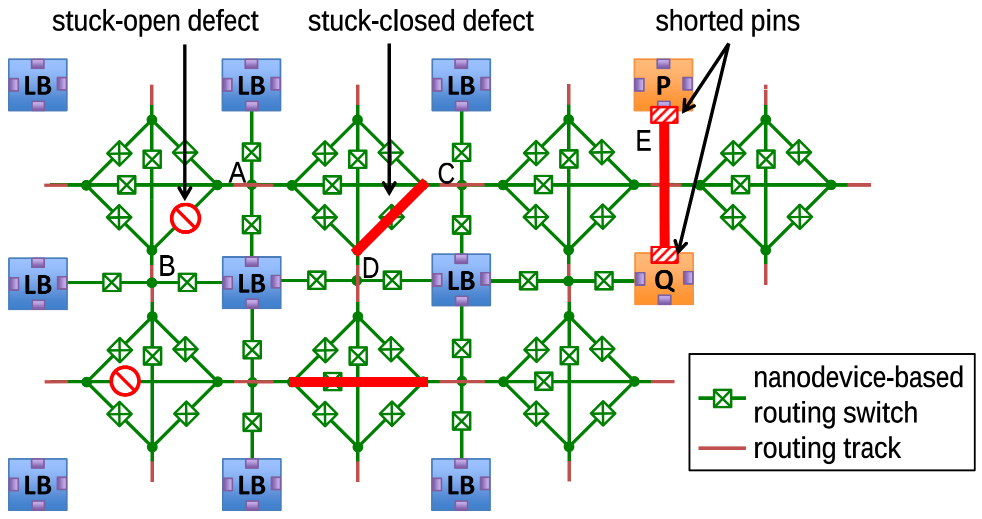


Figure 3.2: Illustration of stuck-open and stuck-closed defects of nanodevice-based routing switches in programmable interconnects. LB ⇒ logic block.

arbitrary nodes in programmable interconnects.

However a stuck-closed defect, e.g., the switch between routing track C and D in Fig. 3.2, has a much higher impact. When track C and D are mapped to two different nets during routing, a logic conflict will occur due to the stuck-closed switch between the two tracks. A simple solution to guarantee logic consistence is to avoid using the two routing tracks shorted by any stuck-closed switch (as shown in Fig. 3.3). This is equivalent to avoidance of all the routing switches connected

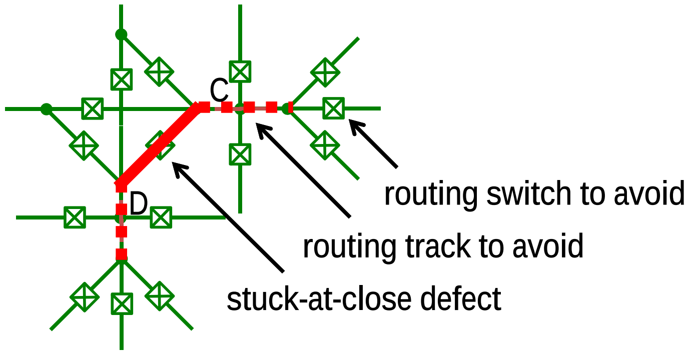


Figure 3.3: Solve a stuck-closed defect by simple defect avoidance. All of the 15 switches shown in this figure need to be discarded due to a single stuck-closed switch.

to the two routing tracks. In this case, all of the 15 switches shown in Fig. 3.3 need to be avoided. The overhead of solving a stuck-closed defect can be 15x that of solving a stuck-open defect when simple defect avoidance is used by the defect-tolerant CAD tool. To quantify the impact of a stuck-closed defect, we develop an approximate model based on probability. Let's use denotation in Table 3.1. Then the probability of a routing track $a$ that is connected with at least one stuck-

| symbol | meaning |
|---|---|
| $r$ | defect rate of stuck-closed switches |
| $n$ | number of switches that a routing track is connected with |

Table 3.1: Denotation of settings for defect impact analysis.

closed switch is

$$\mathrm{P}(a) = 1 - (1-r)^n \approx nr \quad \text{for } r \ll 1.$$

It is also the probability of this track to be disabled due to the stuck-closed switch(es). Every routing switch is connected with two routing tracks. A switch $s$ will be avoided if either of the two routing tracks is disabled, at probability

$$\mathrm{P}(s) = 1 - (1-\mathrm{P}(a))^2 \approx 2nr - n^2r^2 \approx 2nr \quad \text{for } r \ll 1. \tag{3.1}$$

This indicates that for stuck-closed defects, the effective defect rate is enlarged by $2n$ times. Depending on the structure of programmable interconnects, $n$ could be 6–100 [2, 41, 45]. We perform simulations to verify our analytic model using a typical MCNC benchmark [112] mapped onto the RRAM-based FPGA architecture ($n = 6$) in [45] and also a heavily modified version of VPR.[2]. Fig. 3.4 shows an impact comparison between the stuck-open and stuck-closed defects. The tolerance level of stuck-closed defects is 10x lower than that of stuck-open defects when simple defect avoidance is used. Eq. (3.1) also leads to a dilemma. When we want to improve routability by adding more switches, i.e., by increasing $n$, we may not achieve desirable results due to the deteriorating impact of stuck-closed defects. To overcome these difficulties, we need to utilize

[2] VPR is a state-of-art FPGA CAD tool in academia [2, 109]

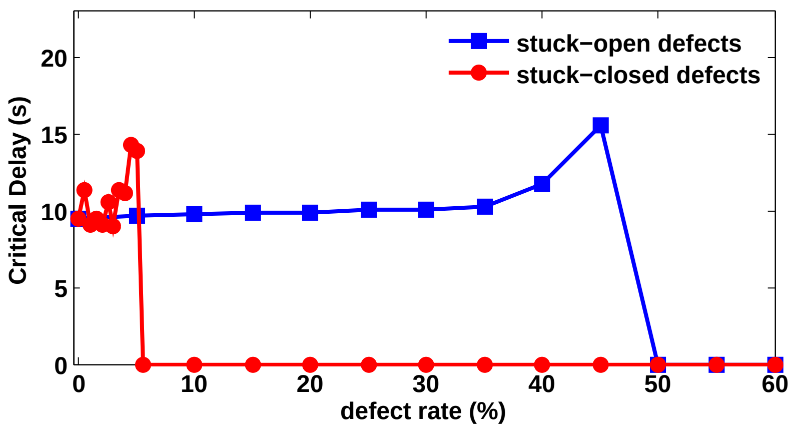
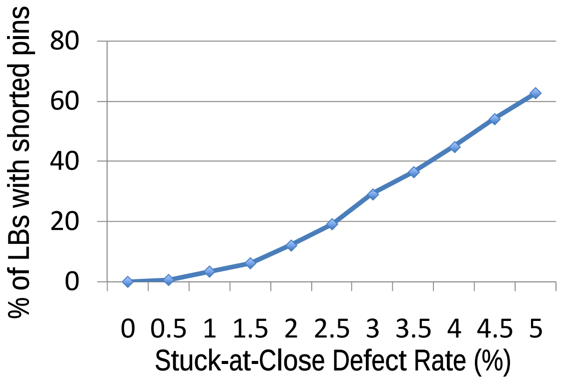


Figure 3.4: Impact comparison of the stuck-open and stuck-closed defects on routability. Delay going down to zero $\Rightarrow$ unroutable. $\sim$10x gap observed between the impacts of the two defect types.

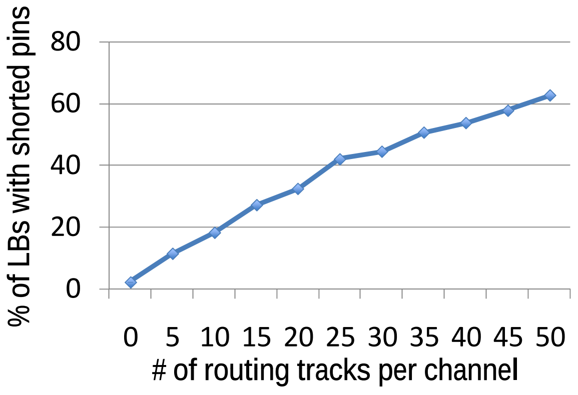
stuck-closed switches by treating them as shorting constraints during routing instead of simple avoidance.

### 3.2.2 Impact on Placement

Another problem brought on by stuck-closed defects is shorted pins of logic blocks. As shown in Fig. 3.2, consecutive stuck-closed switches can form shorting paths. Some shorting paths may happen to connect pins of logic blocks together. Take the two physical logic blocks P and Q with shorted pins in Fig. 3.2 for example. If the two netlist logic blocks which are placed at P and Q do not share common nets in their inputs or outputs, logic inconsistency will be found during routing. Considering the large number of paths between two arbitrary pins of logic blocks, the expectation of the number of logic blocks with shorted pins is not trivial. Again we perform simulations to evaluate this impact using the same settings in Section 3.2.1. Fig. 3.5a shows that $>60\%$ logic blocks are involved with shorted pins at a stuck-closed defect rate of 5%. It indicates that though the logic inconsistency of placed logic blocks like P and Q can be solved by rejecting all the physical logic blocks with shorted pins during placement, it is not practical to reject $>60\%$ of logic blocks. Fig. 3.5b further shows that the number of logic blocks with shorted pins will also increase as the number of routing tracks per channel increases. This leads to another dilemma.



(a)



(b)

Figure 3.5: The number of logic blocks (LBs) with shorted pins over different stuck-closed defect rates and numbers of routing tracks in channel.

When we want to improve routability by adding more routing tracks, more paths will be created between pins of logic blocks, and more blocks will be involved with shorted pins. To overcome these difficulties, we need to recover the logic blocks with shorted pins via an enhanced placement algorithm.

## 3.3 Defect-Tolerant Routing

The FPGA routing resources and their connections are represented as a directed graph $G = (V, E)$, as show in Fig. 3.6. The node set $V$ corresponds to input/output pins of logic blocks as well as routing tracks, the edge set $E$ to routing switches (with buffers). The edges of stuck-open defects will be removed from the graph before routing. The edges of stuck-closed defects will be marked as shorting edges, e.g., $e_s = (v_k \rightarrow v_l)$. Nodes shorted by the shorting edges will form an electrically shorted cluster (ES-cluster), e.g., $\{c, d, e, g, h\}$ in Fig. 3.6. Associated with each node $v$ is a constant delay $d(v)$ and a congestion cost $c(v)$ determined by the competition among signals for $v$. Each net $i$ in a netlist to be mapped onto the FPGA will place a source node $s_i$ and multiple

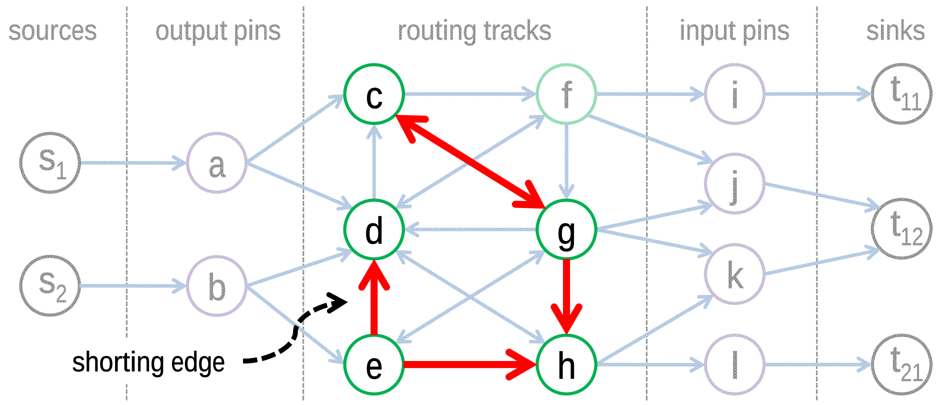


Figure 3.6: Example of a routing graph with shorting constraints. Bold red arrows indicate shorting edges. Nodes $\{c, d, e, g, h\}$ shorted by the shorting edges form an electrically shorted cluster (ES-cluster) as highlighted.

sink nodes $t_{ij}$ in the graph. A routing problem is to find a routing tree $RT_i = (V_i, E_i) \subseteq G$ to connect $s_i$ to all $t_{ij}$, for every $i$ and $j$. A valid routing solution requires that every node is included in only one routing tree, i.e., $\forall v \in V, c(v) \leq 1$. If the shorting constraints are applied to the routing solution, it also requires that a successor node is included in the same tree as its precedent node, i.e.,

$$\left.\begin{array}{l} \exists e_s = (v_k \rightarrow v_l) \\ v_k \in V_i \text{ of } RT_i \end{array}\right\} \Rightarrow v_l \in V_i \tag{3.2}$$

This section include several technologies to enhance routing under shorting constraints.

### 3.3.1 Enforcement of Shorting Constraints

The basic idea of the enforcement of shorting constraints in our tool is that we do not immediately remove the ES-cluster from the routing graph if any node in the cluster is used by a routing tree. For better solution quality, at the first few routing iterations, we allow multiple routing trees to use the nodes in the same ES-cluster and put circuit performance as the primary optimization goal. Then we gradually increase the penalty on the violation of shorting constraints to guarantee that the final routing solution complies with all the shorting constraints. The negotiation-based routing

[52] balances circuit performance and resource overuse via the concept of node congestion. We extend the congestion concept so that negotiation can be performed between circuit performance and shorting constraints as well. Fig. 3.7 is an illustration of our method. When we add one

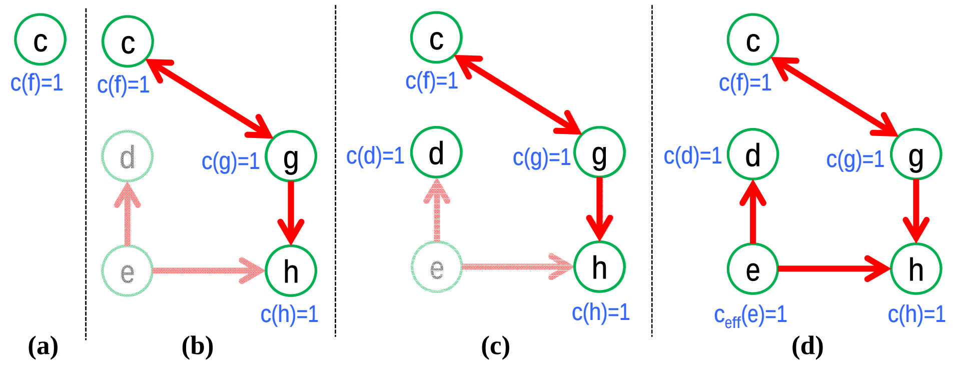


Figure 3.7: Our extension of the congestion concept for routing under shorting constraints. (a) Add node c to a routing tree. (b) Recursively add all the successor nodes and increase congestions. (c) Add node d to another routing tree. (d) Replace the congestion cost with the effective cost for precedent nodes, e.g., node e.

node to a routing tree, e.g., node c in Fig. 3.7(a), we recursively add all the successor nodes in the ES-cluster to the tree, e.g., nodes g and h in Fig. 3.7(b), and increase all of their congestions by one. Other routing tress can compete for node g and h as well as node c, but with the penalty of their congestion costs. As the routing iteration continues, the router will exponentially increase the weight of the congestion costs in the node costs to eliminate any constraint violation. Other routing trees can still use node d freely since it does not violate any shorting constraint, as shown in Fig. 3.7(c). But adding node e to other routing trees will incur a violation though its congestion has never been increased, as shown in Fig. 3.7(d). That's because node e is a predecessor node of other used nodes in the ES-cluster. To apply shorting constraints, we replace the congestion cost with an effective cost:

$$c_{\text{eff}}(v_l) = \max\{c(v_l), c(v_{l1}), c(v_{l2}), \cdots, c(v_{ln})\} \tag{3.3}$$

where $v_{lk}$ for $1 \leq k \leq n$ are all the sink nodes in an ES-cluster of routing nodes that are reachable from $v_k$. Here sink nodes refer to the nodes without any outgoing shorting edges. In Fig. 3.7(d), the qualified sink nodes for node e are node c and d. By our extension of the congestion concept, the route could utilize all the nodes in ES-clusters as much as it can while balancing with circuit performance.

### 3.3.2 Prune Invalid Solutions Before Routing

We also observe that there are nodes that will always be logically inconsistent with certain nets due to shorting edges. It takes many iterations for the router to figure out the inconsistency via the increasing congestion of these nodes. Therefore we add a mechanism to quickly prune these invalid routing solutions by analysis of shorting edges. Fig. 3.8 shows an example. Track 1 is

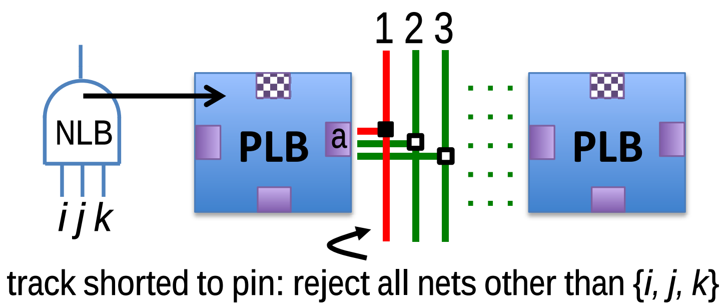


Figure 3.8: Example of fast pruning of invalid routing solutions. Any routing solution that maps track 1 to the net in set $\overline{\{i, j, k\}}$ can be pruned ahead of time.

shorted to pin a of the PLB. It should only be used by net i, j or k since the netlist logic block (NLB) placed in the PLB contains only net i, j and k as inputs. We mark track 1 incompatible with all the nets in set $\overline{\{i, j, k\}}$. We will calculate the incompatible node set for every net before routing. We temporarily remove all the incompatible nodes from the routing graph during the routing of a net to reduce the solution space.

### 3.3.3 Smart Mapping of ES-Clusters to Nets

#### 3.3.3.1 Motivation

Since the shorting constraints are new to the FPGA router, we want to develop some techniques to help the router converge at a solution that maps the ES-clusters to suitable nets for better utilization of routing resources. Fig. 3.9 shows a motivation example. To route a net from $s_2$ to $t_{21}$, there exist

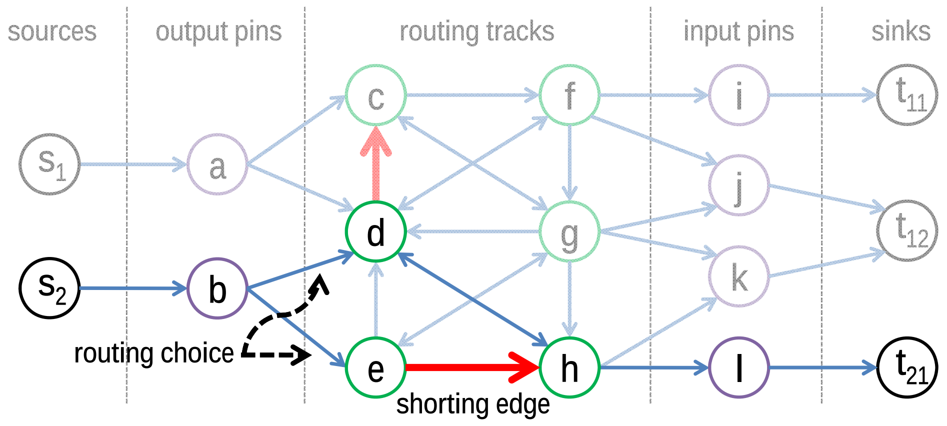


Figure 3.9: To route a net from $s_2$ to $t_{21}$, there exist two shortest paths—one via node d and the other via node e as highlighted. We guide the router to route via node e for better utilization of resources.

two shortest paths, one via node d and the other via node e as highlighted. Conventional FPGA routers treat nodes d and e equally. However the shorting edge from node d to c leads to a waste of node c. On the other hand, the shorting edge from node e to h saves resources. It motivates us to guide the router to map ES-clusters to those nets in more shared paths.

#### 3.3.3.2 Categorization of ES-Clusters

We discover that different techniques should be applied to large ES-clusters and small ES-clusters respectively. As shown in Fig. 3.10, while the small ES-cluster can be fully utilized by both net 1 and net 2, the large ES-cluster can be fully utilized only by net 2. This indicates that the benefits of using small ES-clusters can be judged locally during the routing of a net, since smaller ES-clusters have simple topologies and are usually fully utilized as long as they have paths shared with the net. The mapping of large ES-clusters needs to be planned globally before routing since partial

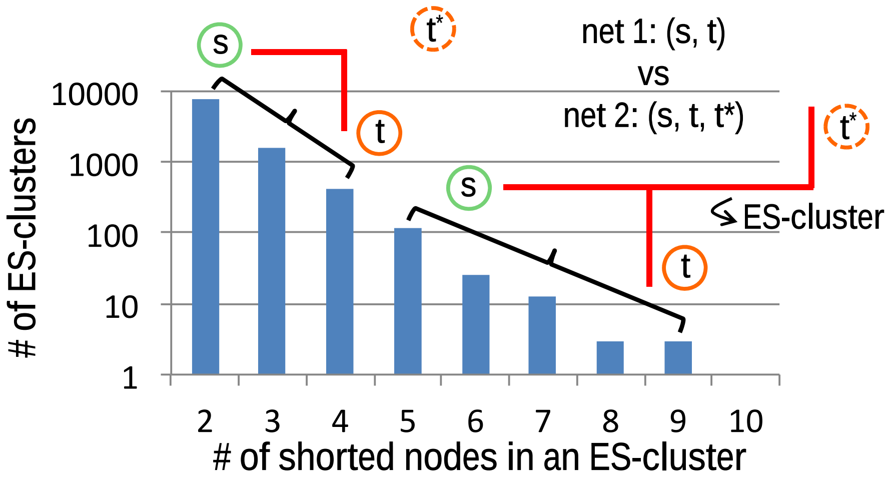


Figure 3.10: A distribution of ES-clusters with different sizes in a defective nanodevice-based FPGA. Along with it is an example of the different potentials of small ES-clusters and large ES-clusters exposed to the same nets.

utilization is a more common case, and we want to maximize the utilization ratio over more net candidates. The global and local strategies for large ES-clusters and small ES-clusters are also determined by the exponential relationship between cluster size and cluster amount, as shown in Fig. 3.10.

#### 3.3.3.3 Global Planning of Large ES-Clusters

We formulate the global planning of large ES-clusters to nets as a search for a subset of edges in a weighted bipartite graph $(Q, C, E)$, with net set $Q$, cluster set $C$ and edge set $E$. The weight of an edge $e = (q, c) \in E$ refers to the length of the shared path between an ES-cluster $c$ and a net $q$ (with reference to its source and sink locations). The goal is to find an edge subset $E' \subseteq E$ to attain the upper bound of $\sum_{e \in E'} w(e)$. The constraint is that no two edges in the subset $E'$ share a common cluster. The optimal solution can be obtained by a greedy algorithm which selects the edge of each ES-cluster $c_i$ to the net with the largest $w(q, c_i)$ among $\forall q \in Q$. Let us denote such $q$ that maximizes the weight for $c_i$ as $q(c_i)$. The optimality of the greedy algorithm can be proved as follows:

*Proof.* Since

$$w(q, c) \geq 0, \forall q \in Q, c \in C$$

we have

$$\forall c \in C, \exists (q, c) \in E'$$

otherwise, we can always add $(q, c)$ to $E'$ to make $\sum_{e \in E'} w(e)$ larger. Then

$$\max \sum_{e \in E'} w(e) = \max \sum_{i=1}^{|C|} w(q, c_i)$$

Since

$$\forall c_i \in C, w(q, c_i) \leq w\left(q(c_i), c_i\right)$$

we have

$$\sum_{e \in E'} w(e) \leq \sum_{i=1}^{|C|} w\left(q(c_i), c_i\right)$$

□

Here we assume that a net can be assigned with multiple ES-clusters. In practice we find that due to the limited connectivity of the routing graph, a net is usually able to use only one ES-cluster out of all the clusters assigned to it. To eliminate the waste of clusters, we add the constraint that no two edges share a common net. Now the problem becomes the maximum weighted bipartite graph matching. The optimal solution can be obtained by using the augmenting path algorithm.

#### 3.3.3.4 Runtime Mapping of Small ES-Clusters

When the router routes a net and has multiple routing node candidates to traverse towards the sink of the net, we guide the router to prefer the node which is connected to an ES-cluster with its path towards the sink, on the condition that this bias does not hurt timing. We enhance the cost function of a node candidate $v$ into

$$\text{Cost}(v) = \text{Crit}(RT_i) \cdot d(v) + [1 - \text{Crit}(RT_i)] \cdot \text{D} \cdot c_{\text{eff}}(v)/s(v) \tag{3.4}$$

$s(v)$ is a factor added by us to guide the router. It is equal to one plus the distance of the path shared between the involved ES-cluster and the net towards its sink. The other parts in the formula remain unchanged. Crit$(RT_i)$ is the largest timing criticality among all the paths in the routing $RT_i$ tree to route net $i$ and ranges between 0–1. $d(v)$ is the delay. D is the delay normalization factor. $c_{\text{eff}}(v)$ is the effective congestion cost. In this enhanced cost function, $s(v)$ plays role only when Crit$(RT_i)$ is small and $c_{\text{eff}}(v)$ is large, i.e., applied only to an uncritical path under a tight budget of routing resources. The proposed cost function enables our use of ES-clusters to automatically balance circuit performance and routability.

### 3.3.4 Implementation

Given the denotations in Table 3.2, Algorithm 2 shows how we implement our defect-tolerant routing into the negotiation-based routing procedure [52].

| Denotations | Meanings |
|---|---|
| $v, u$ | routing nodes |
| $s_i$ | the source node of net $i$ |
| $t_{ij}$ | the $j$th sink of $s_i$ |
| $RT_i$ | routing tree of $s_i$ |
| $e = (v_k, v_l)$ | reconfigurable edge that connects $v_k$ to $v_l$ in routing graph |
| $e_s = (v_k, v_l)$ | shorting edge that connects $v_k$ to $v_l$ in routing graph |
| $c(v)$ | congestion of $v$ which records how many routing trees use this node |

Table 3.2: Denotation table for defect-tolerant routing.

Step 1 of Algorithm 2, discussed in Section 3.3.3.3, maps large ES-clusters to more suitable

**Algorithm 2:** Implementation of our defect-tolerant routing in Pathfinder.

```
Input  : source nodes $s_i$ for each net $i$ and their corresponding sink nodes $t_{ij}$
Output: $RT_i = (V_i, E_i) \in G$ for $k \rightarrow \forall i$,
          s.t. $\forall e_s = (v_l \rightarrow v_k), v_k \in V_i$ of $RT_i \Rightarrow v_l \in V_i$

1  global planning of large ES-clusters;
2  while $\exists$ overused resources do
3      foreach $s_i$ do
4          rip-up $RT_i$ and $\forall v \in V_i$ of $RT_i$, update $c_{\text{eff}}(v)$ in eq. (3.3);
5          set $RT_i := s_i$;
6          foreach $t_{ij}$ of $s_i$ do
7              Set priority queue $PQ := RT_i$ with
               $\mathrm{PathCost}(v) := \text{Crit}(RT_i) \cdot \text{delay}(v), \forall v \in RT_i$;
8              while $t_{ij}$ not found do
9                  pop lowest cost node $v$ from $PQ$;
10                 foreach $u := \mathrm{fanout}(v)$ do
11                     add $u$ to $PQ$ with $\mathrm{PathCost}(u) := \mathrm{PathCost}(v) + \mathrm{Cost}(u)$ shown in
                       eq. (3.4);
12                 end
13             end
14             foreach node $v$ in path from $RT_i$ to $t_{ij}$ do
15                 update $c_{\text{eff}}(v)$ in eq. (3.3);
16                 RecursiveAdd($v$, $RT_i$);
17             end
18         end
19     end
20     $\forall v$, update historical congestion based on $c_{\text{eff}}(v)$ in eq. (3.3);
21     perform timing analysis and update Crit($RT_i$);
22 end
23 RecursiveAdd($v$, $RT_i$) begin
24     foreach $u$ where $\exists e_s = (v \rightarrow u)$ do
25         RecursiveAdd($u$, $RT_i$);
26     end
27 end
```

nets. The updates of congestion in Step 4, Step 15 and Step 20, discussed in Section 3.3.1, apply constraints to successor nodes in the routing. The calculation of node cost $\mathrm{Cost}(u)$ in Step 11, discussed in Section 3.3.3.4, maps small ES-clusters to more suitable nets. The recursive addition of nodes to routing trees in Step 16, discussed in Section 3.3.1, applies constraints to predecessor nodes in the routing.

## 3.4 Defect-Tolerant Placement

### 3.4.1 Methodology

In this section we enhance the placement algorithm to recover the logic blocks which become virtually unusable due to shorted pins. First we show that though it sounds like a good idea to place two netlist logic blocks (NLBs) with shared nets in two physical logic blocks (PLBs) with shorted pins, it is not practical. We discover that even the check of logic consistency on the placement of a group of PLBs with shorted pins cannot be divided into sub-problems and therefore is hard to solve. As shown in Fig. 3.11, NLB pair (b,d) in the netlist has a shared input and output which match the shorted pins of PLB (B,D). NLB pair (b,c) has shared inputs which match the shorted pins of PLB (B,C). However the connection of (b,c,d) does not match the shorted pins of (B,C,D) but matches (B,E,D). In addition, it is not desirable that the placement constraint of a PLB depends on the NLB placed in another PLB—e.g. PLB B and C in Fig. 3.11. Here we propose a cost-effective way to decorrelate the placement of multiple logic blocks. We reduce the ES-cluster size of shorted pins by disabling pins so that there is only one pin left in each cluster. All the other pins disabled in each cluster will be mapped to don't-cares for their logic blocks. Then we do not need to consider logic relationships among logic blocks and only need to focus on placement of logic blocks with different numbers of pins. This method works based on two of our observations. The first observation is that in a netlist there are many logic blocks that do not fully utilize their pins, as shown in Fig. 3.12. The second observation is that most ES-clusters contain only two shorted pins

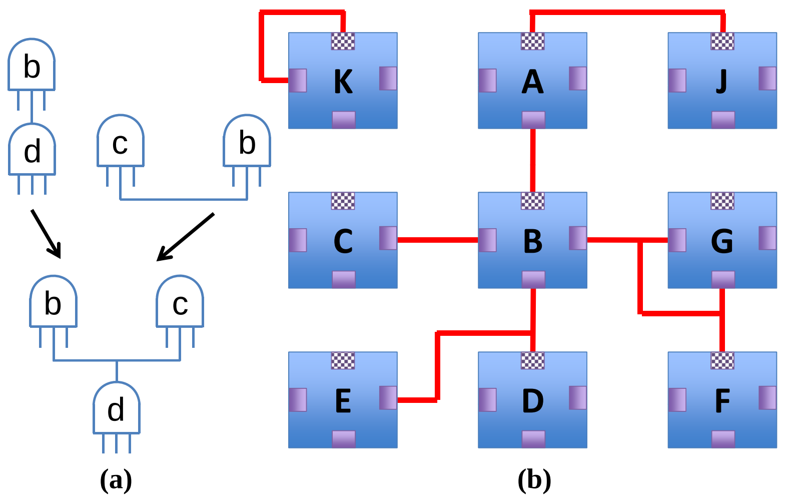


Figure 3.11: Example of the challenge in the placement of logic blocks with shorted pins. (a) A netlist. (b) Logic blocks with shorted pins (checked pad ⇒ output pin). Though the connections of (b,d) and (b,c) in the netlist match the shorted pins of (B,D) and (B,C) respectively, the connection of (b,c,d) does not match (B,C,D) but matches (B,E,D).

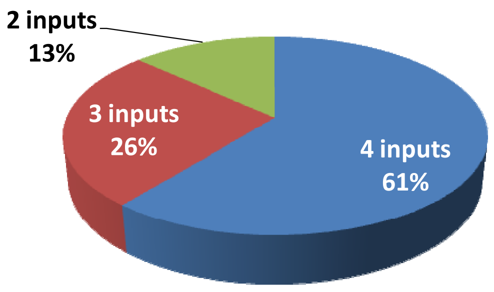
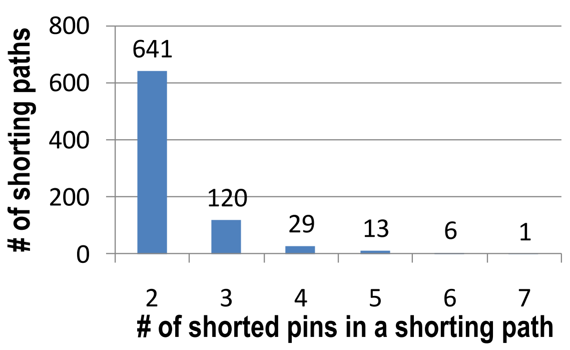
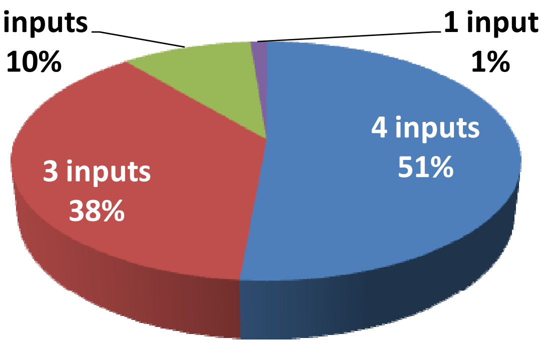


Figure 3.12: Distribution of logic blocks over different numbers of used inputs in a netlist after logic synthesis. Logic synthesis is performed by the Berkeley ABC tool [10] using a 4-LUT FPGA library. Pins are not fully utilized in many logic blocks.

and only half of them need to be disabled, as shown in Fig. 3.13a.[3] Therefore many logic blocks remain with the full pin set, as shown in Fig. 3.13b.



(a)



(b)

Figure 3.13: (a) Count of ES-clusters over different numbers of shorted pins. (b) Distribution of physical logic blocks (PLBs) over different numbers of active inputs (i.e., not disabled) after pin disabling.

### 3.4.2 Implementation

#### 3.4.2.1 Simulated Annealing

Simulated annealing [121] serves as the placement engine in VPR [2, 109], a state-of-art CAD tool in academia. Algorithm 3 shows how we implement our defect-tolerant placement in simulated annealing.

In step 1 of Algorithm 3, we first check hard violation for shorted pins of logic blocks against circuit rules. For example, in Fig. 3.11, the output pins of PLB A and J are shorted. In this case, we have to disable one of them — let's say PLB J, to avoid logic conflict.

In step 2 of Algorithm 3, we disable shorted pins in each ES-cluster to decorrelate the placement of multiple NLBs, as discussed in Section 3.4.

[3] See Section 3.5.1 for simulation settings.

**Algorithm 3:** Integration of our defect-tolerant placement in simulated annealing.

```
Input  : PLB set $P$, NLB set $V$.
Denote : $|p|$ := # of inputs of a PLB $p \in P$
         $|v|$ := # of inputs of a NLB $v \in V$
Output : An injective function $f : V \rightarrow P$,
         s.t. $\forall v \in V$, $f(v) = p \in P$ and $|v| \leq |p|$.

1  Check and disable PLBs with hard violation;
2  Check and disable part of shorted pins;
3  Initial random placement, s.t. $f(v) = p \Rightarrow |v| \leq |p|$;
4  while Simulated annealing continues to improve timing do
5      $\cdots$
6      Randomly select two PLBs $p_1$ and $p_2$;
7      foreach $(i, j) \in \{(1,2),(2,1)\}$ do
8          if $|f^{-1}(p_i)| \leq |p_j|$ or $f^{-1}(p_i) = null$ then
9              Swap: $\{f^{-1}(p_1), f^{-1}(p_2)\} \rightarrow \{f^{-1}(p_2), f^{-1}(p_1)\}$ ;
10         end
11     end
12     $\cdots$
13 end
```

In step 3 of Algorithm 3, we need to generate an initial placement of NLBs at PLBs as the starting point of simulated annealing. For every NLB, we randomly search for a PLB with sufficient active pins. Note that the NLBs with more inputs have fewer PLB candidates. To maximize the number of NLBs that can be placed at PLBs in the given FPGA, we place the NLBs in decreasing order in terms of the number of inputs.

In step 9 of Algorithm 3, during the random swap of the two NLBs placed at two PLBs, we also need to check whether active pins suffice the swapped case. It is also possible that a PLB is not placed with any NLB. In this case, the check can be exempted.

#### 3.4.2.2 Analytical Placement

Recent trends indicate that analytical placement is taking the place of simulated annealing as the mainstream placement method for FPGAs. For example, Xilinx released the Vivado®tool suite to replace its ISE®tool suite which was used for decades [122]. The Vivado®tool suite adopts analytical placement, and its runtime becomes 4x faster than its simulated annealing baseline. Our defect-tolerant placement can also be easily migrated into analytical placement. Enhancement is needed only in the legalization step in analytical placement. In this step, legalization is applied to each type of logic block sequentially. Each type of NLBs will be spread from unaligned locations optimized by an analytical solver for minimum cost function to nearby slots of PLBs with the same type. We can limit the spread of each NLB within those PLBs with sufficient active pins. Algorithm 4 shows how we enhance the legalization. This legalization flow will be applied to each type of logic block. Note that we call the function $\mathrm{OriginalLegalization}(V, P)$ used in the original analytical placement. The task of this function is to spread NLBs in set $V$ from unaligned locations to PLBs in set $P$. We apply this function to NLBs in set $V$ in a decreasing order of input pins since NLBs with more pins have fewer PLB candidates. By doing so, this flow could maximize the utilization of PLBs.

**Algorithm 4:** Implementation of our defect-tolerant placement in the legalization step of analytical placement.

```
Input  : PLB set $P$, NLB set $V$.
Denote : $|p| :=$ # of inputs of a PLB $p \in P$, $P_i := \{p| \, |p| = i\}$
         $|v| :=$ # of inputs of a NLB $v \in V$, $V_i := \{v| \, |v| = i\}$
Output : An injective function $f : V \rightarrow P$,
         s.t. $\forall v \in V$, $f(v) = p \in P$ and $|v| \leq |p$.
$P_c := \emptyset$;
$n := \max(i)$;
$f := null$;
for $i = n, n-1, n-2, \cdots, 1$ do
    $P_c := P_c \cup P_i$;
    $f_i : V_i \rightarrow P_c := \text{OriginalLegalization}(V_i, P_c)$;
    add $f_i$ to $f$;
end
```

## 3.5 Simulation Results

### 3.5.1 Settings

We implemented our defect tolerance methods in mrVPR [45], a modified version of the state-of-art VPR tool in FPGA CAD society [2, 109]. We chose the RRAM-based FPGA architecture in [45] as our experiment platform.[4] The technology node is 45nm as in [45]. The channel width is fixed when routability is evaluated. The nanodevice defect rate is set to 10% as reported in the papers published in recent years [47, 35], and stuck-open type and stuck-closed type account for this rate in equal proportion. We also provide sensitivity analysis on the defect rate. All experiments are performed on the 20 largest MCNC benchmark circuits [112] and also on two relatively large ($>$ 10k LUTs) circuits from the QUIP benchmark design set [123, 124]. Three different tool settings are used and compared: 1) conventional tool setting for defect-free circuits, 2) simple avoidance of logic blocks and routing resources affected by defects as discussed in Section 3.2, and 3) our method with defect utilization beyond avoidance (namely adaptive defect recovery). Note that we are unable to apply the SAT-based method in [46] to these benchmarks for comparison due to its impractical runtime for large designs. Comparisons of area and timing are made on the results of the three settings above.

### 3.5.2 Results

First we verify our defect-tolerant placement. Fig. 3.14 shows the area usage of benchmarks after placement. While simple defect avoidance has an average of 2.01x area compared to the defect-free case, our adaptive defect recovery has only 1.14x area. That's because we can recover most of logic blocks which become virtually unusable due to shorted pins. The result can be made even better if we improve logic synthesis to match the distribution in Fig. 3.12 to that in Fig. 3.13b.

[4] VPR does not provide opportunities to manipulate local interconnects within clustered logic blocks (CLBs). To evaluate and tolerate defects in these parts, we move these parts outside of CLBs by setting the CLB structure to a single logic block.

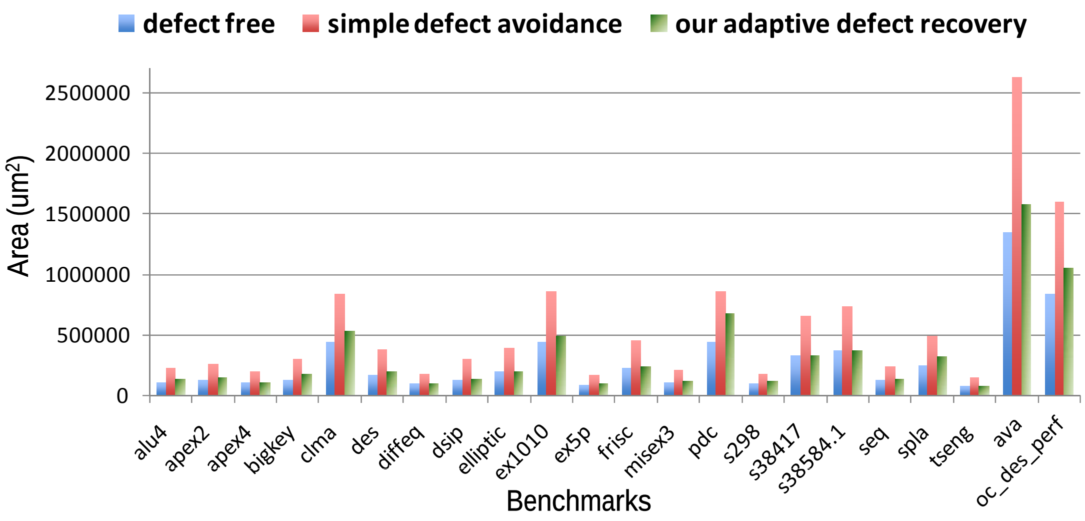


Figure 3.14: Area usage of benchmarks after placement.

Next, we verify our defect-tolerant routing. Fig. 3.15 shows the routability of benchmarks using the two defect-tolerant methods. Since simple defect avoidance wastes too many routing

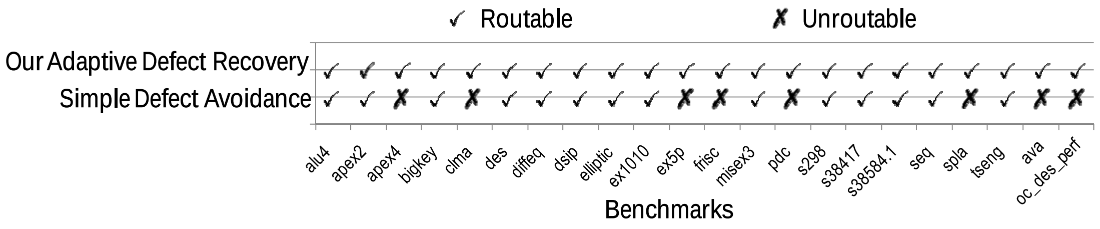


Figure 3.15: Routability of benchmarks using simple defect avoidance and our adaptive defect recovery.

resources, 37% of the benchmarks become unroutable. Our adaptive defect recovery can still keep 100% routability since we successfully use stuck-closed switches. We also perform experiments to justify the effect of smart mapping of ES-clusters to nets. We find that the routability is improved from 90.9% to 100%. Fig. 3.16 shows the critical delay of benchmarks after routing. This comparison is made on only those benchmarks routable in the case of the simple defect avoidance in Fig. 3.15. Simple defect avoidance shows an average of 1.43x critical delay compared to the defect-free case. That's because the tight budget of routing resources caused by simple defect

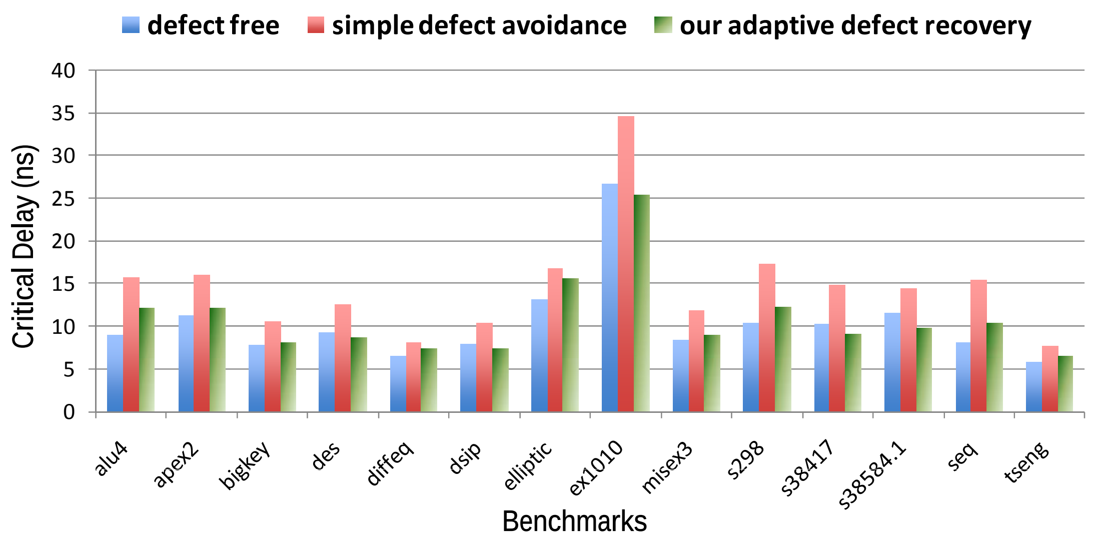


Figure 3.16: Critical delay of benchmarks after routing.

avoidance leads to deviation of routing results from the optimal. Our adaptive defect recovery has only an average of 1.07x critical delay (some benchmarks show even better timing than the defect-free cases due to VPR routing noise [113]). This proves that our method balances circuit performance and routability under shorting constraints.

Fig. 3.17 is a comparison of runtime complexity between the SAT-based method with defect utilization in [46] and our method. The SAT-based method in [46] shows a high runtime complexity due to the large search solution. In contrast, our method shows complexity similar to the conventional CAD tool for the defect-free case.

To verify the benefits of our method over different defect rates, we also perform a sensitivity analysis. Fig. 3.18 shows the overall yield of all the benchmarks over multiple defect rates. Yield here is defined as the success rate of a circuit benchmark to fit into the logic resources in the given FPGA chip and be routable under given routing resources. Logic resource constraints are set to 1.2x of the defect-free case in this experiment.[5] Our method gains significantly higher yield over simple defect avoidance. Moreover, the improvement on yield is most significant when the defect

[5] In practice, the maximum utilization of logic blocks on an FPGA chip is usually less than 80% for the sake of routability.

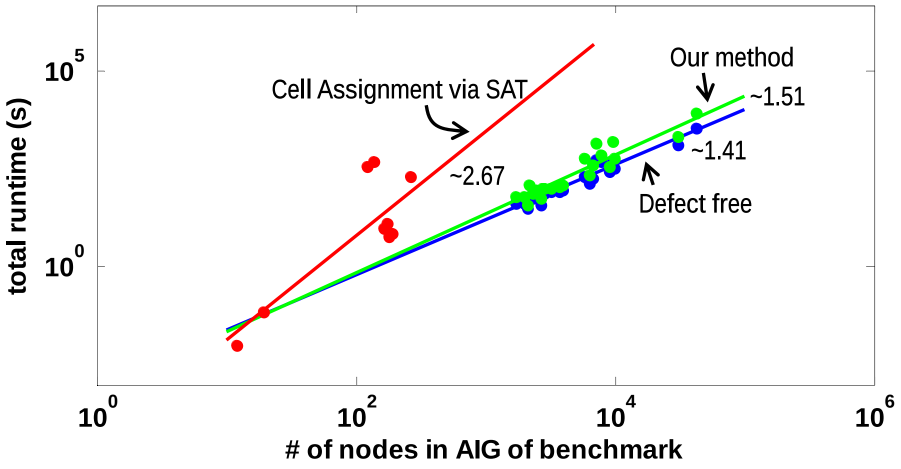


Figure 3.17: A comparison of runtime complexity. The scales of benchmarks are evaluated as the number of nodes in the form of an and-inverter graph (AIG) of benchmarks.

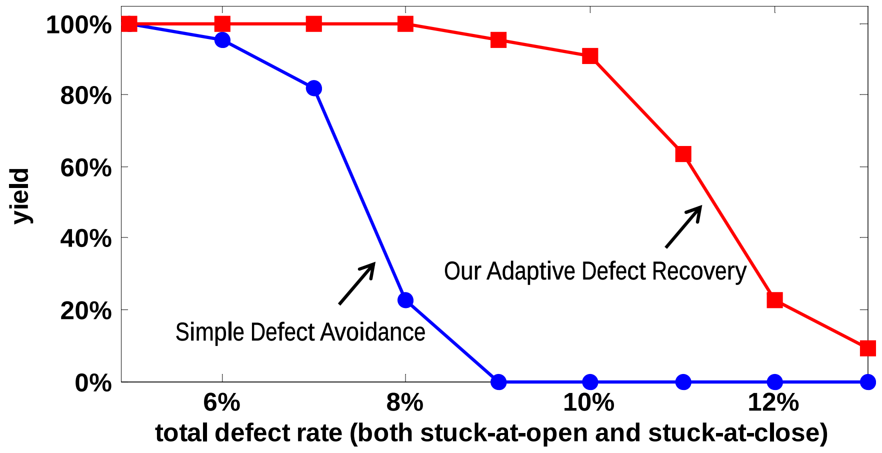


Figure 3.18: Yield comparison over multiple defect rates.

rate is high. This indicates the capability of our adaptive defect recovery to find more successful implementations in the ”difficult region.”

## 3.6 Conclusion

This chapter focuses on defect tolerance for nanodevice-based programmable interconnects of FPGAs. First, we observe that the stuck-closed defects of nanodevices incur much higher impact than the stuck-open defects. Instead of simply avoiding the stuck-closed defects, we use them by treating them as shorting constraints in the routing. We develop a scalable algorithm to perform timing-driven routing under these extra constraints. We also enhance the placement algorithm to recover logic blocks which become virtually unusable due to shorted pins. Simulation results show that our method is effective for defect tolerance of nanodevice-based FPGAs.

# CHAPTER 4

# Optimized Interconnects Between Accelerators and Shared Memories

## 4.1 Introduction

After discussion about the fabric-level communication optimization in the previous two chapters, we go up to the chip-level design in the chapter. Energy efficiency has become one of the primary design goals in the many-core era. A recent trend to address this is the use of on-chip accelerators as coprocessors [54, 55, 56, 57, 58]. Application-specific accelerators can fully explore parallelism and customization and provide a 100x improvement in energy efficiency over general-purpose processors [58, 20, 59]. As predicted by ITRS [21], future computing platforms may integrate hundreds or thousands of accelerators on a chip to improve their performance and energy efficiency. Accelerator-rich computing platforms also offer a good solution for overcoming the "utilization wall" as articulated in the recent study reported in [60]. As scaling down continues, there will be tens of billions of transistors on a single chip. However, it has been demonstrated that a 45nm chip filled with 64-bit operators will only have around 6.5% utilization (assuming a power budget of 80W) [60]. This means that an architecture with a massive number of homogeneous cores may not work since only a limited number of cores can work in parallel. However, we can implement tens of billions of transistors into a sea of heterogeneous accelerators and launch the corresponding accelerators upon user demand. This methodology does not expect all the accelerators to be used all the time, but expects most computation tasks to be executed by the most energy-efficient hardware.

Though the computation parts of accelerators are heterogeneous, the memory resources in accelerators are homogeneous and can be shared among accelerators [18, 61]. The transistors saved from memory sharing can be used to implement more accelerators to cover more application kernels.

One of the key challenges to realizing memory sharing among accelerators is how to design the interconnects between accelerators and shared memories. Since accelerators run $>$100x faster than CPU cores [59], accelerators perform many independent data accesses every clock cycle. This requires a high-speed, high-bandwidth interconnect design with many conflict-free data channels to prevent accelerators from starving for data. There are very few works on optimization of interconnects between accelerators and shared memories. The work in [62] uses shared buses as the interconnects. The work in [61] assumes a network-on-chip (NoC) that connects all the accelerators and shared memories together. The work in [18] uses interconnects similar to shared buses in local regions, and an NoC for global interconnects. All still treated the data ports of accelerators like ports of CPU cores. They will suffer either performance inferiority from lack of support to meet the high data demands of accelerators, or a huge transistor cost from hardware duplication to meet the accelerator demand.

In this chapter we identify three optimization opportunities that emerge in accelerator-rich platforms, and exploit them to develop novel interconnects between accelerators and shared memories:

- An accelerator contains multiple data ports, and in the interconnect design the relations of the ports from the same accelerator should be handled differently compared to the ports from different accelerators. We propose a two-step optimization rather than optimizing all the ports of all accelerators globally in a single procedure. Many unnecessary connections associated with each individual accelerator are identified and removed before all the accelerators are optimized together.

- Due to the power budget in dark silicon, only a limited number of accelerators will be powered on in an accelerator-rich platform. The interconnects can be partially populated to just fit the data access demand limited by the power budget.

- Accelerators are heterogeneous. Some accelerators will have a higher chance of being turned on or off together if they belong to the same application domain. This kind of information can be used to customize the interconnect design to remove potential data path conflicts and use fewer transistors to achieve the same efficiency.

## 4.2 Preliminary

### 4.2.1 Interconnect Requirement Between Accelerators and Shared Memories

Interconnects between CPU cores and shared memories (L2 cache) have been the subject of much attention over recent years [125, 126, 127, 128]. When there are not many CPU cores on the chip, shared bus is commonly used as the design choice for interconnects [125]. When the number of CPUs scaled up in recent years, designers changed to network-on-chip (NoC) [126, 127] or a combination of buses and NoC [128]. The effectiveness of these interconnects is based on the assumption that each CPU core performs a load/store every few clock cycles. Therefore, a simple interconnect, e.g., Fig. 4.1(a), can arbitrate the data channel alternatively among CPU cores without reducing much of their performance. In contrast, accelerators run >100x faster than

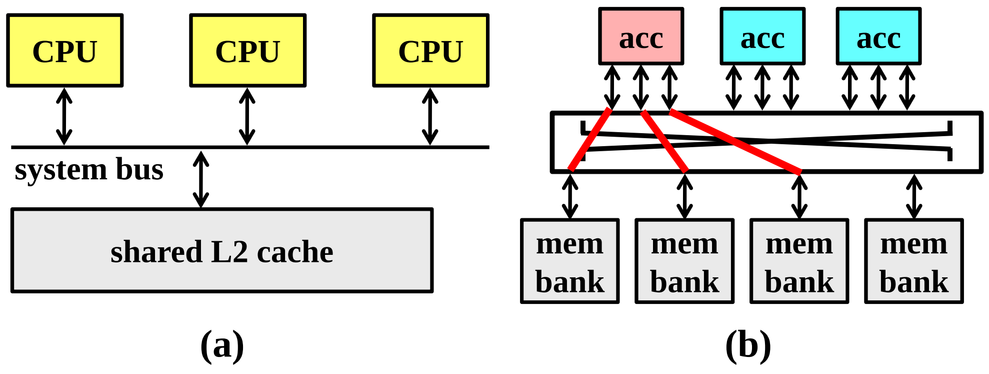


Figure 4.1: Difference between memory sharing among general-purpose CPU cores and among accelerators. (a) A simple interconnect for CPU cores. (b) Demanding interconnects for accelerators.

CPUs [59], and each accelerator needs to perform several loads/stores every clock cycle. The interconnects between accelerators and shared memories need to be high-speed, high-bandwidth and contain many conflict-free data channels to prevent accelerators from starving for data, as shown in Fig. 4.1(b). An accelerator needs to have at least $n$ ports if it wants to fetch $n$ data every cycle. The $n$ ports of the accelerator need to be connected to $n$ memory banks via $n$ conflict-free data paths in the interconnects.

Another problem with conventional interconnect designs is that the interconnect arbitration among requesting accelerators is performed upon each data access. NoC designs even perform multiple arbitrations in a single data access as the data packet goes through multiple routers. Since accelerators implement computation efficiently but have no way of reducing the number of necessary data accesses, the extra energy consumed by the interconnect arbitration during each data access will become a major concern. The arbitration upon each data access also leads to a large and unexpected latency for each access. Since accelerators aggressively schedule many operations (computation and data accesses) into every time slot [65], any late response of data access will stall many operations and lead to significant performance loss. Many accelerator designs prefer small and fixed latencies to keep their scheduled performance, and because of this many accelerators [18] have to give up memory sharing.

Though the works in [62, 18, 61] are on memory sharing among accelerators, they still use either shared buses or meshed-based NoC which were designed and optimized for the data demand of CPUs.

### 4.2.2 Configurable Crossbar to Meet Accelerator Demand

In this work we design the interconnects into a configurable crossbar as shown in Fig. 4.2. The configuration of the crossbar is performed only upon accelerator launch. Each memory bank will be configured to connect to only one accelerator (only one of the switches that connect the memory bank will be turned on). When accelerators are working, they can access their connected memory banks just like private memories, and no more arbitration is performed on their data paths. Fig. 4.2

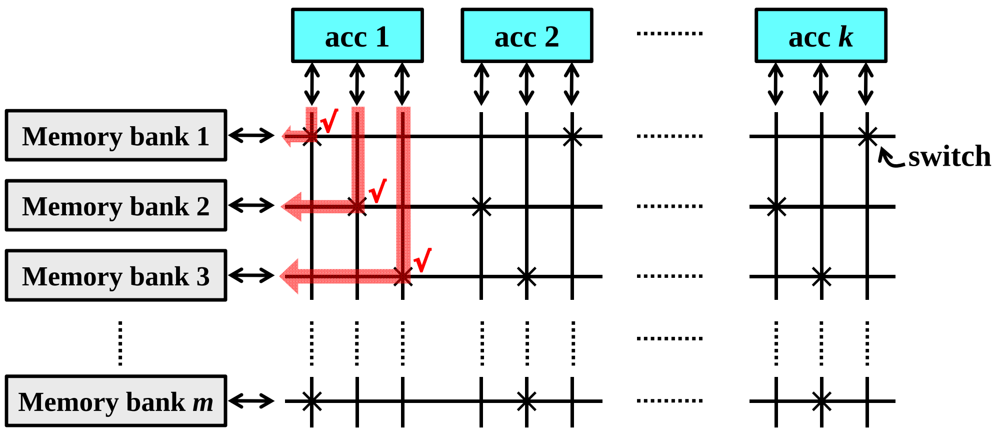


Figure 4.2: Interconnects designed as a configurable crossbar between accelerators and shared memories to keep data access cost small. ✓ : switch turned on.

shows that acc 1, which contains three data ports (i.e., demands three data accesses every cycle), is configured to connect memory banks 1–3. We are aware that there are other design choices for the configurable network between accelerators and shared memories. However the crossbar design contains the fewest logics (only one switch) in the path from an accelerator port to a memory bank, and thus helps minimize the data access latency. In Section 4.6.1 we will show that we managed to achieve access latency of only one clock cycle in an FPGA implementation so that the timing of access to shared memories acts exactly as private memories within accelerators.

### 4.2.3 Problem Formulation

The primary goal of the crossbar design is that for any set of accelerators that are powered on in an accelerator-rich platform and require $t$ memory banks in total, a feasible configuration of the crossbar can be found to route the $t$ data ports of the accelerators to $t$ memory banks. A full crossbar that connects every accelerator port to every memory bank provides a trivial solution. However, it is extremely area-consuming. We would like to find a sparsely-populated crossbar (e.g., Fig. 4.2) to achieve high routability. We give the definition of routability as follows:

**Definition 1.** *Suppose the total number of accelerators in an accelerator-rich platform is $k$; the* *number of data ports of the accelerators is $n$;*[1] *the number of memory banks in the platform is $m$; and the maximum number of accelerators that can be powered on under power budget in dark silicon is $c$. The routability of the crossbar is defined as: the probability that the a randomly selected workload of $c$ accelerators out of the total $k$ accelerators can be routed to $c \times n$ memory banks via $c \times n$ separate data paths in the crossbar.*

The goal of this work is to *optimize the crossbar for the fewest switches while keeping high routability*.

## 4.3 Optimization Inspired by Accelerators with Multiple Ports

### 4.3.1 Identification of Unnecessary Switches

A fundamental difference between CPU cores and accelerators is that one accelerator needs multiple data ports to perform multiple loads/stores every cycle. This difference appears as a challenge if interconnect designers choose to duplicate hardware that were optimized for single-port cores to match multi-port accelerators. This difference can also be used as an optimization opportunity. In our crossbar design, though every data port of every accelerator is an individual input at one side of the crossbar, the relations of the ports from the same accelerator are handled differently compared to the ports from difference accelerators. Fig. 4.3 shows an example of this different handling in the crossbar optimization. It is meaningful to provide connections from accelerator ports 1 and 5 to the same memory bank 2 since accelerator 1 may be powered off and may leave memory bank 2 for accelerator 2 to use. However it is not necessary to place switches to connect from accelerator ports 1–3 to the same memory bank 1, as ports 1–3 belong to the same accelerator 1 and will be either routed together or turned off together. We will always route them to different memory banks and can eliminate their unnecessary competition for shared resources during the crossbar design.[2]

[1] *For simplicity, we assume that all the accelerators have the same number of data ports. More discussions about it can be found in Section 4.7.*

[2] If an accelerator requires two of its data ports to connect to the same memory bank, it means that the accelerator is over-designed, and its internal structure can be modified to combine these two ports into one.

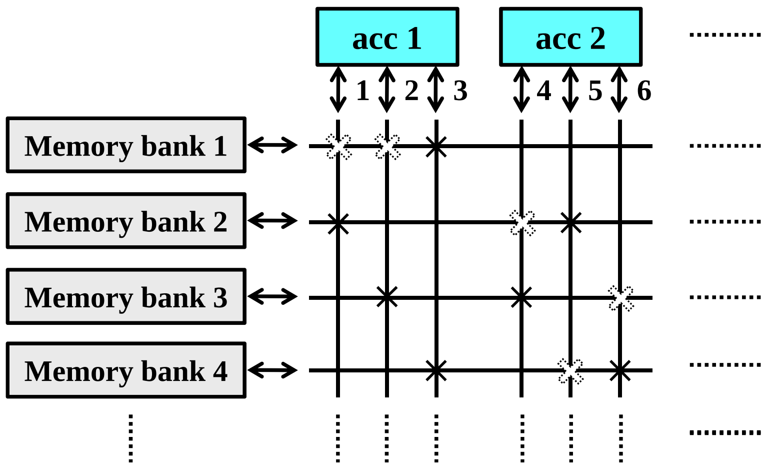


Figure 4.3: As the multiple ports of the same accelerator are powered on/off together, it is not necessary to provide switches to connect them to the same memory banks. Dashed switches with (x,y) coordinates (1,1), (2,1), (4,2), (6,3), (5,4) can be removed. This shows the different impact of ports from the same accelerator and those from different accelerators on the crossbar design.

The dashed switches in Fig. 4.3 depict an example of how we remove unnecessary switches for resource savings. After switch removal, both accelerators 1 and 2 still keep the full access of the four memory banks.

### 4.3.2 Minimum Cost of Accelerator Sub-Crossbar

Motivated by the optimization opportunity described above, we first perform optimization of switches in the sub-crossbar associated with each single accelerator before going to global optimization among accelerators. Suppose an accelerator $a_i$ has $n$ data ports, and has an accessible memory range $R(a_i)$ of $r = |R(a_i)|$ banks in an initial crossbar design.

**Definition 2.** *The accessible memory range $R(a_i)$ of accelerator $a_i$ is defined as the maximum set of memory banks such that for any memory bank $t \in R(a_i)$, there exists at least one crossbar switch to connect $t$ to any of the $n$ data ports of the accelerator.*

For any memory bank, all the switches that connect this memory bank to the data ports of the accelerator will be removed except one switch. Each memory bank will be connected with only one switch for each accelerator, and there will be only $r$ switches in the sub-crossbar associated with each accelerator. Note that $r$ is the minimum switch cost of the sub-crossbar that connects an accelerator to its accessible memory range with $r = |R(a_i)|$ banks. Removal of any switch will lead to the reduction of the accessible memory range.

### 4.3.3 Sub-Crossbar Design for Routing Flexibility

Now the remaining problem is how to place the $r$ switches to keep high flexibility of routing. Fig. 4.4 is an example of how we place these switches in the sub-crossbar of an accelerator with three data ports and seven accessible memory banks. We alternate the switch positions as shown in Fig. 4.4(a) and get an interleaving connection graph in Fig. 4.4(b). This design holds an important property: for any three continuous memory banks in the accelerator's accessible range (e.g., $[j+2, j+4] \in [j, j+6]$ in the figure), there always exists a feasible routing solution to connect the three data ports of the accelerator to the three memory banks. This kind of design can be generalized as follows. In the sub-crossbar of an accelerator with $n$ data ports and an accessible memory range $[j, j+l-1]$ ($j, l \in N$, $l \geq n$), the only switch connected with memory bank $s \in [j, j+l-1]$ is placed at the crosspoint that connects the $p$th data port of the accelerator where

$$p = s \mod n \tag{4.1}$$

Then we can prove

**Theorem 1.** *For the sub-crossbar generated by Eq.*(4.1) *for the accelerator $a_i$ with an accessible memory range $[j, j+l-1]$, any $n$ continuous memory banks $[t, t+n-1] \in [j, j+l-1]$ can be routed to the $n$ data ports of $a_i$ in $n$ separate data paths.*

*Proof.* For any $n$ continuous memory banks $[t, t+n-1]$, their remainders divided by $n$ will cover $1, 2, ..., n$, and therefore all the $n$ data ports of the accelerator will be connected to at least one of the $n$ memory banks. Since each memory bank has only one switch to connect to the accelerator data port, the connections are one to one and go through $n$ different data paths. $\square$

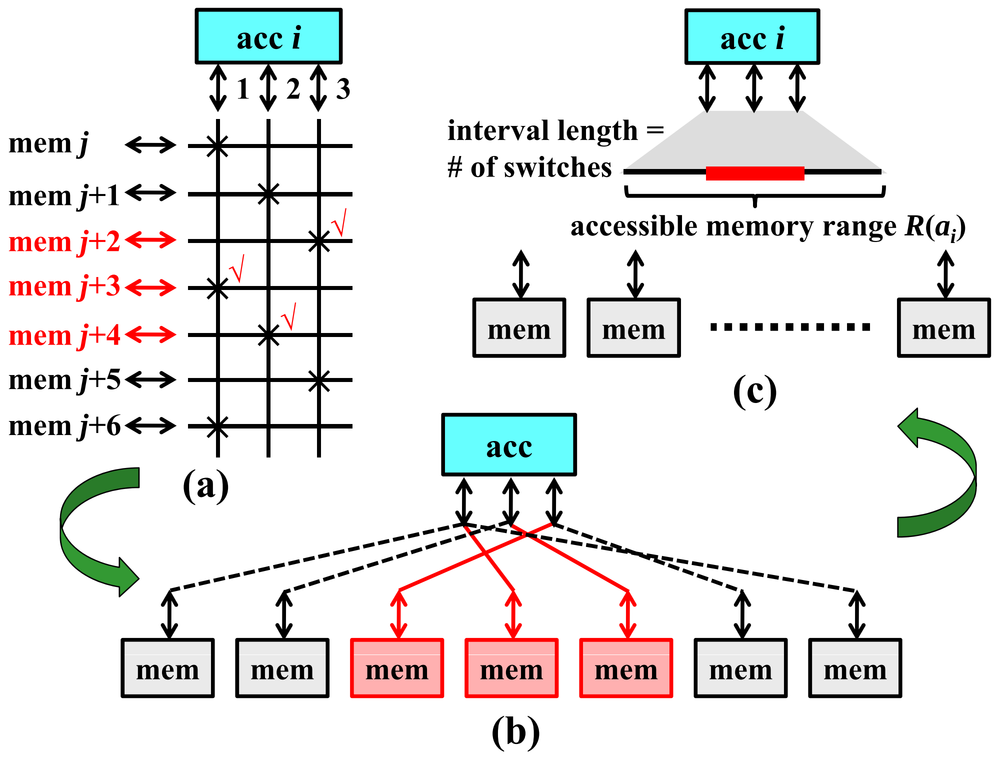


Figure 4.4: Our design with interleaving connections between an accelerator and its accessible memory range. (a) Switch positions in the sub-crossbar of the accelerator. (b) Connectivity graph. Any three continuous memory banks can be routed to the three data ports of the accelerator via separate paths. (c) Our interconnect optimization within the single accelerator can be virtualized as an interval (i.e., its accessible memory range) in which any segment with length of 3 (memory banks) can be routed to the accelerator.

### 4.3.4 Virtualized Interface for Global Optimization Among Accelerators

Our interconnect optimization within the single accelerator with $n$ data ports can be virtualized as an interval (i.e., its accessible memory range) in which any segment with length of $n$ can be routed to the accelerator, as shown in Fig. 4.4(c). The interval length is equal to the number of switches in the sub-crossbar of the accelerator. The global optimization among accelerators will focus on the length and position of the accessible memory range of each accelerator over the whole range of shared memories in the accelerator-rich platform, as shown in Fig. 4.5(a). This can be formulated

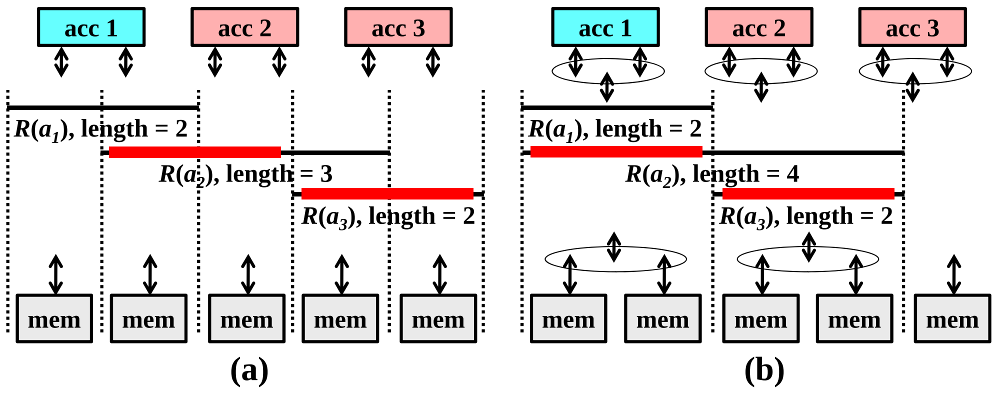


Figure 4.5: The interface for global optimization among accelerators. (a) Global optimization only needs to consider the lengths and positions of accessible memory ranges of accelerators to reduce the total number of switches while keeping a high probability of finding segments without overlaps. (b) If the multiple data ports of an accelerator are grouped into a larger multi-channel port to converge the interconnect design to the CPU scenario, we will lose a large design space and get an inefficient design.

as a well-defined problem: *reduce the sum of interval lengths while keeping high routability*. The routability means the probability that in the accessible memory ranges of a set of working accelerators, e.g., accelerators 2 and 3 in Fig. 4.5(a), a segment can be found in each memory range without overlaps (bold red segments in Fig. 4.5(a)). Its mathematical expression is as follows. Given any $k$ accelerators $\{a_1, a_2, \cdots, a_k\}$ and their accessible memory ranges $R(a_i) = [s_i, e_i], i = 1..k$, we can find a set of *non-overlapping* intervals $\{[s'(a_{i_1}), s'(a_{i_1}) + n - 1], [s'(a_{i_2}), s'(a_{i_2}) + n - 1], ...\}$

associated with each accelerator powered on such that

$$\forall a_i, [s'(a_{i_j}), s'(a_{i_j}) + n - 1] \subseteq R(a_i) = [s_{i_j}, e_{i_j}]$$

Note that though the multiple data ports of an accelerator can be grouped into a larger multi-channel port to converge the interconnect design to the CPU scenario, we will lose a large design space. As shown in Fig. 4.5(b), after grouping the two data ports of each accelerator into one (shared memories also need to do so to be matched in interconnects), the design loses at least three degrees of freedom:

- The accessible memory range of an accelerator has to be divisible by the number of data ports of the accelerator.
- The position of the accessible memory range has to be aligned to the boundary divisible by the number of data ports of the accelerator.
- The number of shared memory banks has to be divisible by the number of data ports of the accelerator.

As a result, Fig. 4.5(b) has to use more switches (larger accessible memory range) to achieve the same routability as Fig. 4.5(a).

## 4.4 Optimization Inspired by Accelerators in Dark Silicon Age

### 4.4.1 Minimal Design with 100% Routability

#### 4.4.1.1 Example Design

In a computing platform based on many CPU cores, the interconnects need to satisfy the data demand when all the CPU cores are working in parallel. In an accelerator-rich computing platform, however, only a limited number of accelerators will be powered on due to the power budget of dark silicon as described in Section 4.1. Inspired by this, we can have a partial population of accessible memory ranges, as shown in Fig. 4.6. This design targets the case where there are

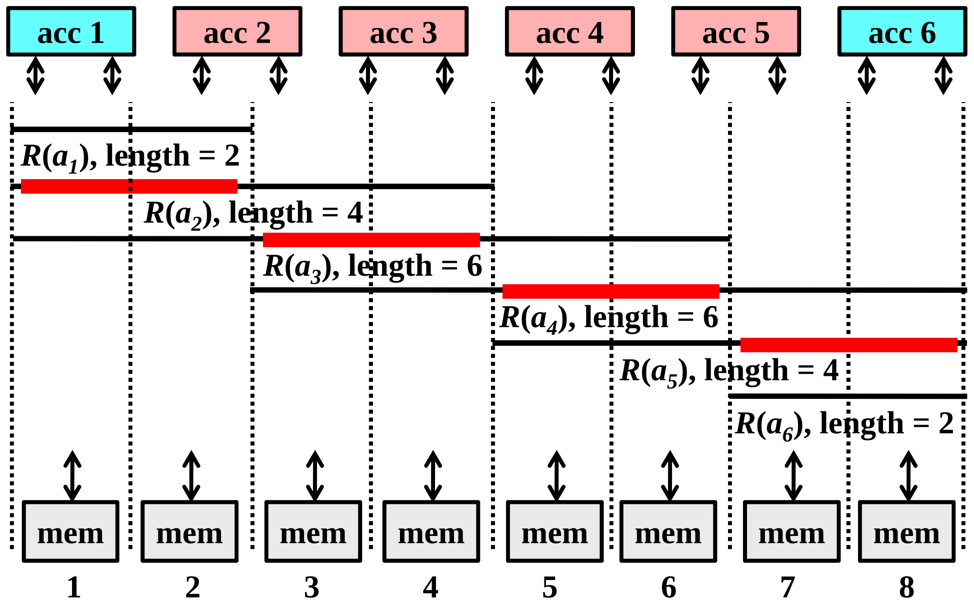


Figure 4.6: Minimal crossbar design that targets the case of six accelerators in total, and at most four accelerators powered on due to the power budget in dark silicon age. All the $C_6^4 = 15$ possible combinations of four accelerators out of the six can be routed to shared memories. Removal of any switch will break the 100% routability guarantee.

six accelerators in the platform and when the power budget limits at most four accelerators to be powered on simultaneously (as an initial attempt, we assume that the limit of a working accelerator count is independent of the types of accelerators powered on). The total number of switches used in Fig. 4.6 is 24, which is calculated as the sum of interval lengths (recall Section 4.3.4). When accelerators 2–5 are powered on, we can find a segment in the accessible memory range of each of the four accelerators so that there are no overlaps of the four segments (as marked in bold and red in Fig. 4.6). Readers can easily figure out that for all the $C_6^4 = 15$ possible combinations of four accelerators out of the six, there always exists a feasible routing solution like this case. It means that even though the crossbar is partially populated, it still guarantees a 100% routability.

#### 4.4.1.2 Optimality Analysis

The design in Fig. 4.6 is also minimal in two aspects:

- The platform has the minimum number of memory banks, i.e., $4 \times 2 = 8$ banks.
- The crossbar uses the minimum number of switches to achieve the 100% routability. Removal of any switch from the crossbar will break the 100% routability guarantee.

Though the second item is not intuitive, we can prove it in a theoretical way. Denote the total number of accelerators in the platform as $k$, the number of data ports of an accelerator as $n$, the maximum number of working accelerators due to dark silicon as $c$, and the total number of memory banks in the platform as $m$. We prove the following theorem.

**Theorem 2.** *The lower bound of switches to guarantee a 100% routability is $(k-c+1)m$, when the number of memory banks $m$ is set to be the lower bound $cn$.*

*Proof.* If a memory bank is connected with fewer than $(k-c+1)$ switches, it is accessible by fewer than $(k-c+1)$ accelerators. It means that there are at least $k-(k-c)=c$ accelerators that cannot access this memory bank due to the lack of switches. However, when we power on these $c$ accelerators, they need to be routed to all the $m=cn$ memory banks in the platform since each of them requires $n$ memory banks, which contradicts the last statement. Therefore, for each memory bank, it must contain at least $(k-c+1)$ switches to connect to at least $(k-c+1)$ accelerators. This results in a total of at least $(k-c+1)m$ switches. □

We can apply Theorem 2 to the case of Fig. 4.6 and calculates the minimum number of switches as $(6-4+1)\times 8 = 24$ which is exactly the number of switches used in Fig. 4.6

There could be minimal designs other than Fig. 4.6. The reason why we choose the design in Fig. 4.6 is that it is a locality-driven design that leads to the easier timing closure. Though Fig. 4.6 is just an illustration, rather than the real layout, we can see that each accelerator can only access the memory banks nearby. Accelerator 1 will not jump to connect to memory bank 8. This kind of design is layout-friendly and facilitates placement in post-synthesis to achieve smaller wire length and latency (verified in our experiments in commodity FPGA in Section 4.6.1).

#### 4.4.1.3 Algorithm for Generating Minimal Design

The methodology used to generate the minimal design in Fig. 4.6 is generalized as follows. For the accelerator set $\{a_1, a_2, \cdots, a_k\}$ in an accelerator-rich platform, the accessible memory range of $a_i$ is $R(a_i) = [s_i, e_i]$. It can be generated by Algorithm 5. Readers can check that the total number

**Algorithm 5:** Algorithm for generating the minimal crossbar with 100% routability.

```
Input  : the settings of the target accelerator-rich platform $(k, c, n)$ defined in Definition 1
Output: accessible memory ranges of $k$ accelerators $\{[s_1, e_1], [s_2, e_2], ..., [s_k, e_k]\}$
1 for $i = 1..k$ do
2     if $i \leq k - c + 1$ then $s_i = 1$ ;
3     else $s_i = (i + c - k - 1)n + 1$ ;
4     if $i \leq c$ then $e_i = i \times n$ ;
5     else $e_i = m$ ;
6 end
```

of switches is equal to $(k - l + 1)m$ by calculating the sum of these interval lengths.

#### 4.4.1.4 Proof of 100% Routability

Then we prove

**Theorem 3.** *The crossbar generated by Algorithm 5 guarantees the 100% routability for any $c$ accelerators $\{a_{i_1}, a_{i_2}, ..., a_{i_c}\}$ powered on.*

*Proof.* Without loss of generality, we assume that $\{a_{i_1}, a_{i_2}, \cdots, a_{i_c}\}$ has been sorted in the ascending order, i.e., $i_1 < i_2 < \cdots < i_c$. Since there are only $m = cn$ memory banks in the platform, we divide the memory banks into $c$ groups and distribute them among the $c$ accelerators. That is, the memory banks assigned to the $j$th working accelerator $a_{i_j}$ are $[(j-1)n+1, jn]$. We need to prove that these memory banks are covered by the accelerator's accessible memory range $R(a_{i_j}) = [s_{i_j}, e_{i_j}]$. Since $i_1 < i_2 < \cdots < i_c$, we have $i_j \geq j$ and $i_j \leq k - c + j$. They indicate

$$e_{i_j} \geq \min\left(i_j \times n, m\right) \geq \min\left(j \times n, m\right) \geq jn$$

and

$$\begin{aligned} s_{i_j} &\leq \max\left(1, \left(i_j + c - k - 1\right) n + 1\right) \\ &\leq \max\left(1, \left(\left(k - c + j\right) + c - k - 1\right) n + 1\right) \\ &\leq (j-1)n + 1 \end{aligned}$$

respectively. Therefore

$$[(j-1)n+1, jn] \subseteq R(a_{i_j}) = [s_{i_j}, e_{i_j}]$$

$\square$

### 4.4.2 Design with 100% Routability when Optimality is Unreachable

#### 4.4.2.1 Example Design

When the number of memory banks $m$ is larger than the minimum requirement $cn$, there is no explicit mathematical equation to get the tight lower-bound on the number of switches. Even when there is only one data port in each accelerator and the design case is reduced to a classical crossbar that routes signals in circuits, there is no solution known to be optimal. The work in [11] shows that a crossbar that is designed to route any $c$ out of $k$ input signals to $m$ output signals, i.e., the classical $(k, n, c)$ concentrator, only has a loose lower-bound of the number of switches. It will be harder to compose a minimal crossbar design in the emerging scenario of accelerators and their shared memories in an accelerator-rich platform. What we can do is to design a crossbar that exploits any opportunity to reduce the number of switches and is able to converge to the minimal design when the number of memory banks $m$ approaches $cn$. Fig. 4.7 shows our design extended

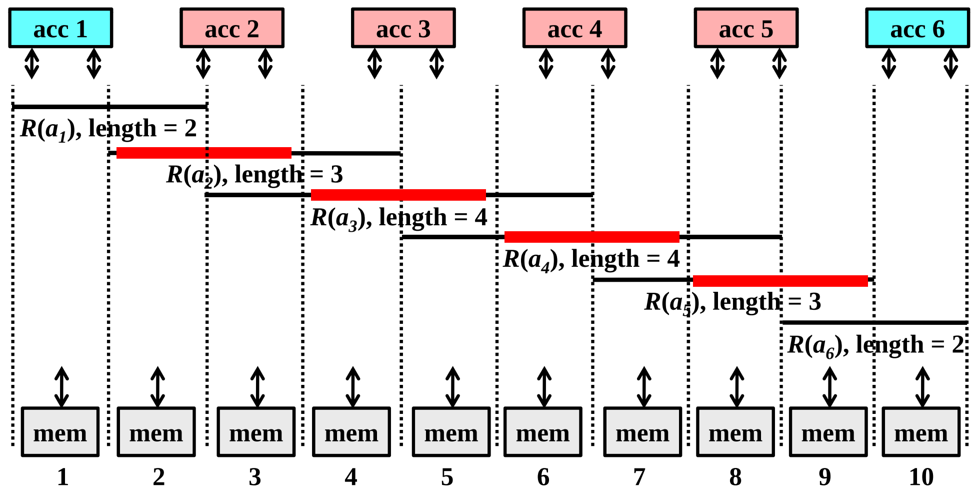


Figure 4.7: Crossbar design when the number of memory banks increases from the minimum eight in Fig. 4.6 to ten. Since the accessible memory ranges of accelerators are scattered, there are fewer conflicts on data paths. The number of switches can be reduced to 18 while still keeping the 100% routability.

from Fig. 4.6 when the number of memory banks increases from the minimum eight to ten. Since the accessible memory ranges of accelerators are scattered compared to Fig. 4.6, there are fewer conflicts on data paths. While keeping the 100% routability of any four accelerators out of the six, the total number of switches is reduced from 24 to 18.

#### 4.4.2.2 Algorithm for Partial Crossbar Generation with 100% Routability Guarantee

The methodology to generate the design in Fig. 4.7 is generalized as follows. For the accelerator set $\{a_1, a_2, \cdots, a_k\}$ in an accelerator-rich platform with the accessible memory range $R(a_i) = [s_i, e_i]$, we first evenly distribute $\{s_1, s_2, ..., s_k\}$ over the $m$ memory banks. Then we pass $\{s_1, s_2, ..., s_k\}$ to Algorithm 6 to obtain $\{e_1, e_2, ..., e_k\}$. Instead of giving a complex expression of the number of switches generated by Algorithm 6, we plot the average number of switchers per accelerator port in crossbars generated by Algorithm 6 in Fig. 4.8. As shown in Fig. 4.8(a), as the number of memory banks approaches the minimal design condition $m = cn = 100$, the number of switches

**Algorithm 6:** Algorithm to generate crossbar with 100% routability.

```
Input  : starting points of accessible memory ranges of $k$ accelerators $\{s_1, s_2, ..., s_k\}$
Output: accessible memory ranges of $k$ accelerators $\{[s_1, e_1], [s_2, e_2], ..., [s_k, e_k]\}$
1 $a = \{\underbrace{0, 0, ..., 0}_{k \text{ zeros}}\}$;
2 for $i = 1..k$ do
3     for $j = 1..c$ do
4         $a(j) = \max(a(j), s_i + jn - 1)$;
5     end
6     $e_i = a(1)$;
7     $a = \{a[2..c], 0\}$;
8 end
```

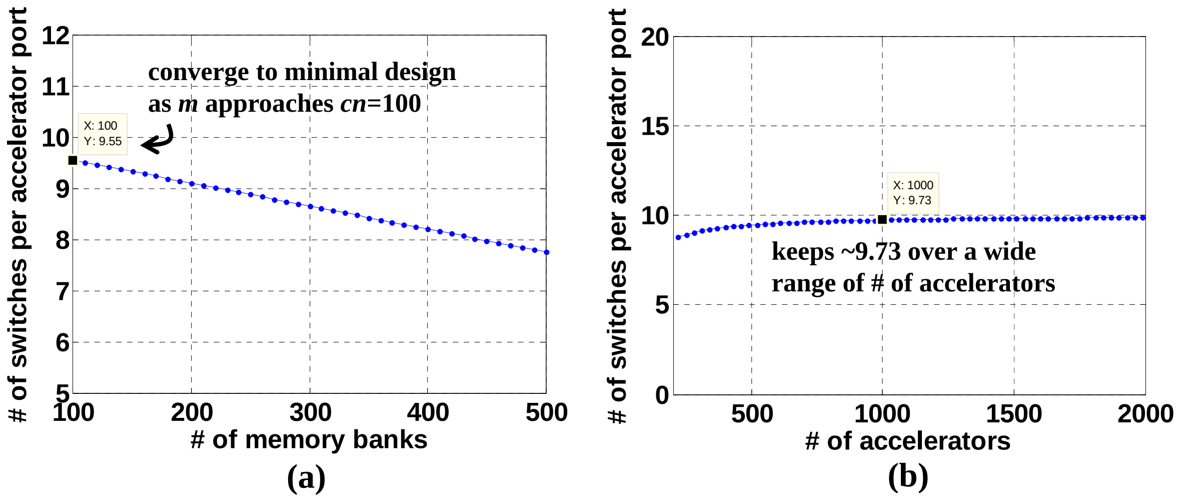


Figure 4.8: The average number of switches per accelerator port in crossbars generated by Algorithm 6. In this experiment, we initially set $k = 200$, $n = 10$, $c = 10$, and $m = 300$. (a) Sweep $m$ from 100 to 500. (b) Sweep $k$ from 200 to 2,000.

converges to the minimum $(k-c+1)m = 19100$, i.e., an average of 9.55 switches per accelerator port. As shown in Fig. 4.8(b), as the number of accelerators in the platform is swept over a wide range from 200 to 2,000, the average number of switchers per accelerator port is kept around 9.73, which shows the scalability of our crossbar design.

#### 4.4.2.3 Proof of 100% Routability

Then we prove

**Theorem 4.** *The crossbar generated by Algorithm 6 guarantees the 100% routability for any $c$ accelerators $\{a_{i_1}, a_{i_2}, ..., a_{i_c}\}$ powered on.*

*Proof.* Without loss of generality, we assume that $\{a_{i_1}, a_{i_2}, \cdots, a_{i_c}\}$ has been sorted in the ascending order, i.e., $i_1 < i_2 < \cdots < i_c$. The end points generated by Algorithm 6 satisfy

$$e_{i_j} = \max_{t=1..c}(s_{i_j-t+1} + tn - 1) \tag{4.2}$$

For the given $c$ accelerators $\{a_{i_1}, a_{i_2}, ..., a_{i_c}\}$, we can assign $cn$ memory banks to these accelerators by determining the starting point $s'(a_{i_j})$ of each assignment recursively as follows:

$$s'(a_{i_j}) = \max(s_{i_j}, s'(a_{i_{j-1}}) + n)$$

Then we have

$$s'(a_{i_j}) \geq s_{i_j}$$

and by induction we can prove

$$s'(a_{i_j}) \leq \max_{t=1..c}(s_{i_{j-t+1}} + (t-1)n)$$

By using the property of the sorted set $\{a_{i_1}, a_{i_2}, ..., a_{i_c}\}$ that

$$i_{j-t} \leq i_j - t$$

and

$$j \leq p \Rightarrow s_{i_j} \leq s_{i_p}$$

we further have

$$s'(a_{i_j}) \leq \max_{t=1..c}(s_{i_{j-t+1}} + (t-1)n) \leq \max_{t=1..c}(s_{i_j - t+1} + (t-1)n)$$

Combined with Eq. (4.2), we have

$$s'(a_{i_j}) + n - 1 \leq e_{i_j}$$

and

$$[s'(a_{i_j}), s'(a_{i_j}) + n - 1] \subseteq [s_{a_{i_j}}, e_{a_{i_j}}]$$

□

By further removing some starting segments in the last $c$ accelerators by taking advantage of their last positions and correspondingly fewer conflicts, we can get the crossbar design similar to Fig. 4.7.

### 4.4.3 Smaller Design with 95% Routability

#### 4.4.3.1 Example Design

All of the designs described above guarantee the 100% routability of any $c$ accelerators powered on. To keep the 100% routability, these designs use many switches to handle the limited number of difficult cases. Next, we explore the opportunities found in sacrificing certain routability to further reduce the number of switches. Fig. 4.9 gives an example of removing two extra switches from the design in Fig. 4.7. The accelerators 2–5 can still be routed to shared memories after switch removal. Out of all the $C_6^4 = 15$ possible combinations of four accelerators powered on, only the accelerator set $\{1, 2, 3, 4\}$ and $\{3, 4, 5, 6\}$ cannot be routed. These unroutable cases usually contain many accelerators with adjacent IDs and large overlaps among their accessible memory ranges. In most cases, it is more likely that the $c$ working accelerators have distributed IDs, and the routability will not suffer from overlap congestion of their accessible memory ranges.

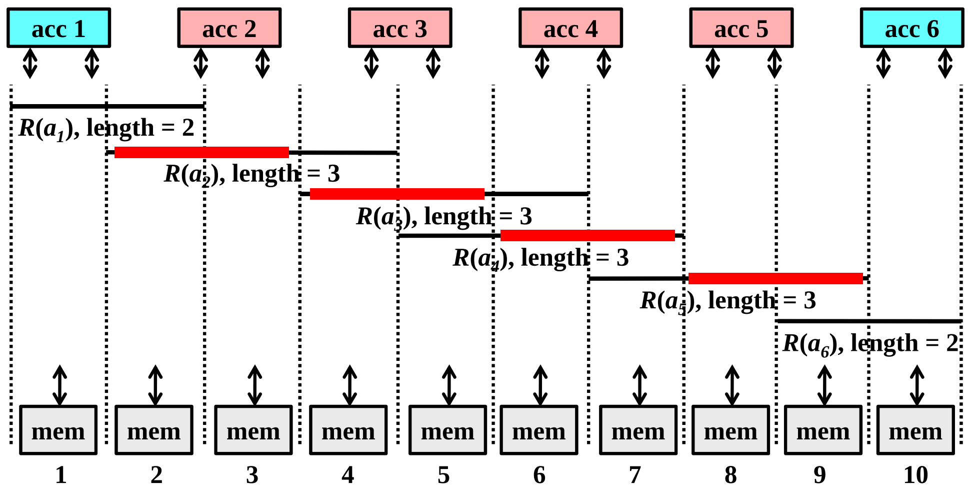


Figure 4.9: Removal of two switches from Fig. 4.7 to sacrifice routability for switch savings. In this design, the accelerators 2–5 can still be routed to shared memories after switch removal. Out of all the $C_6^4 = 15$ possible combinations of four accelerators powered on, only the accelerator set $\{1, 2, 3, 4\}$ and $\{3, 4, 5, 6\}$ cannot be routed.

#### 4.4.3.2 Methodology and Effectiveness

The shape of the accessible memory ranges of all the $k$ accelerators generated by Algorithm 6 looks like the outer parallelogram in Fig. 4.10. The area of the parallelogram is the sum of all the accessible memory ranges, which is also the total number of switches. To reduce the number of switches, we apply horizontal transformation to the parallelogram to make it slim, and correspondingly with smaller area, as shown in Fig. 4.10. The area reduction can be easily calculated as $d \times k$, which enables us to precisely control the number of switches in this reduction process. Fig. 4.11 shows the effectiveness of our methodology; it displays the experimental results on the routability after reduction of switch count. The routability of the crossbar is measured using a Monte Carlo test. 10,000 random test vectors are generated. Each test vector contains 10 working accelerators randomly selected out of the total 200 and is tested as to whether the 10 accelerators can be routed to shared memories or not. The routability of the crossbar is estimated as the percentage of test vectors which can be successfully routed. As shown in Fig. 4.11, $\sim 70\%$ of switch savings can be

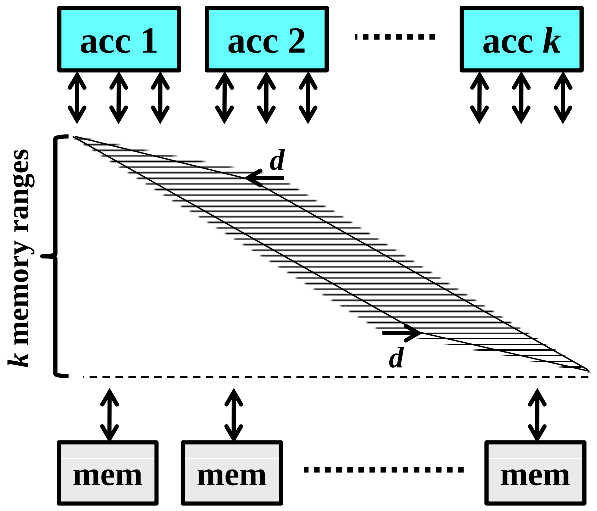


Figure 4.10: Methodology of switch count reduction by horizontal transformation of accessible memory ranges of accelerators. The number of switches reduced in this process can be precisely calculated as $d \times k$ (the area difference between the outer parallelogram and inner parallelogram).

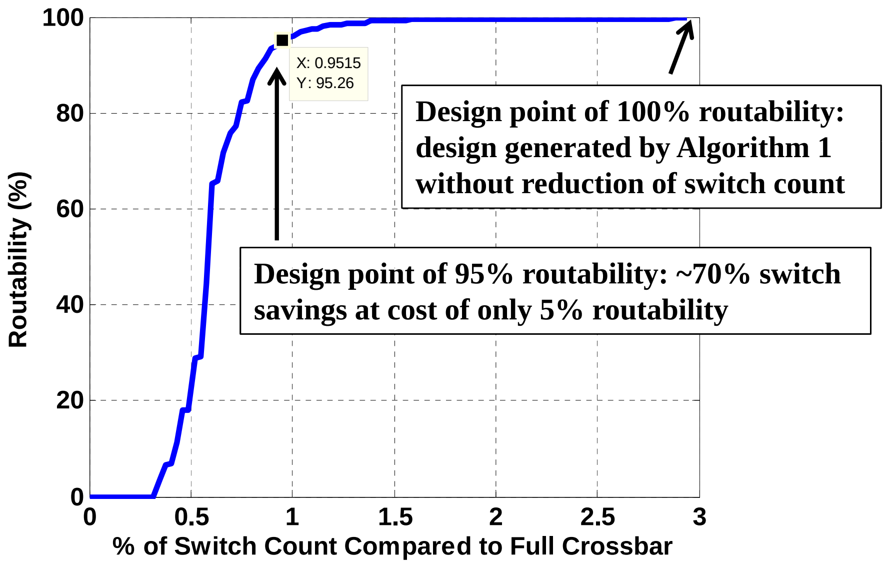


Figure 4.11: Experimental results on routability after reduction of switch count. In this experiment, $k = 200$, $n = 10$, $c = 10$, and $m = 300$.

achieved by sacrificing $5\%$ of routability. For the test vectors where not all of the 10 accelerators can be routed together, the worst case is to schedule part of the 10 accelerators to run first, and then the other part to run later. The upper-bound of the overall runtime of the accelerator-rich computing platform to execute these 10,000 test vectors will become $(95\% \times 1 + 5\% \times 2) = 1.05$x, which is a very small overhead. We also checked the 474 unroutable test vectors and found that 446 test vectors could be routed by removing only one accelerator from the task list. In practice, the accelerator-rich computing platform will not keep full workloads all the time, and we can reschedule the removed accelerator to fill an empty slot when the platform uses fewer than 10 accelerators. This creates new research opportunities for the accelerator manager in an accelerator-rich platform [58].

## 4.5 Optimization Inspired by Accelerator Heterogeneity

### 4.5.1 Background

Different from homogeneous CPUs, heterogeneous accelerators are designed for different applications. While the executions of $k$ CPU cores can be modeled as $k$ independent random variables, the executions of $k$ accelerators can be modeled as $k$ random variables with a covariance matrix. The accelerators that belong to the same application domain will have a higher probability of being powered on or off together. For example, the medical imaging domain [19, 129] has accelerators "denoise," "gaussianblur," "segmentation," "registration," "reconstruction," etc. When the user of an accelerator-rich platform works on some tasks related to medical imaging, the medical imaging accelerators will be likely powered on together. Fig. 4.12(a) shows an example of the execution correlations among heterogeneous accelerators. Here, we still assume that there are 200 accelerators in the platform, as in Section 4.4. We generate 10,000 random test vectors, and each test vector contains 10 working accelerators selected out of the total 200 accelerators. We divide the 200 accelerators into 20 application domains as $\{a_1..a_9\}, \{a_{10}..a_{19}\}, ..., \{a_{190}..a_{199}\}$. The covariance of accelerators within the same application domain is set to 0.9, and that from different domains is set

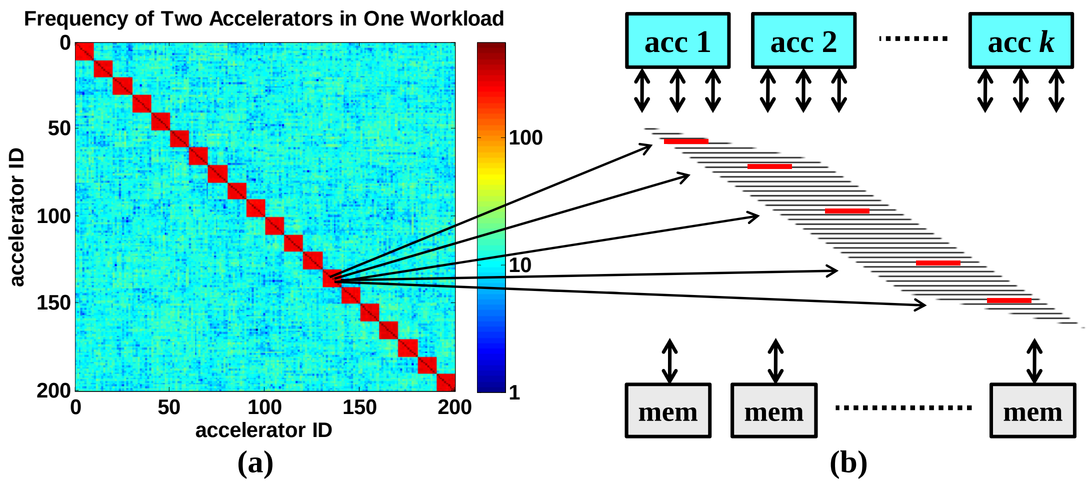


Figure 4.12: Optimization opportunities brought by accelerator heterogeneity. (a) Execution correlations among heterogeneous accelerators. Accelerators within the same application domain have a higher probability of being powered on or off together. (b) We can scatter the accelerators within the same application domain throughout the shared memories so that they have fewer overlaps in their accessible memory range and will be easier to route.

to 0 for the generator of the random test vectors. These settings reflect the fact that the executions of accelerators within the same domain are strongly correlated, and those from different domains are independent random variables. Each crosspoint $(a_i, a_j)$ in Fig. 4.12(a) shows the frequency of the two accelerators $a_i$ and $a_j$ appearing in the same test vector with workloads of ten accelerators. Fig. 4.12(a) shows that accelerators within a domain have a higher probability of appearing in the same set of workloads.

### 4.5.2 Methodology and Effectiveness

The heterogeneity of accelerators described above can be used to further optimize our crossbar. As mentioned in Section 4.4.3.1, the test cases that contain many accelerators with adjacent IDs and large overlaps among their accessible memory ranges will be difficult to route. Since we have the information about which accelerators have a higher probability of being powered on together, we can scatter these accelerators throughout the shared memories to increase their routability, as shown in Fig. 4.12(b).

To exploit this optimization opportunity, we add an extra layer between the accelerators and crossbar topology: physical accelerator IDs $p_1, p_2, ..., p_k$. The $k$ accelerators in the platform $\{a_1, a_2, ...a_k\}$ will be mapped to $\{p_1, p_2, ..., p_k\}$ so that the accelerators with high correlations are scattered in $\{p_1, p_2, ..., p_k\}$. The crossbar topology is now determined by $R(p_1), R(p_2), ..., R(p_k)$. The overlaps of accessible memory ranges $R(p_i)$ and $R(p_j)$ can be precalculated and stored in a matrix $\{D_{ij}\}$. Our task is to find an optimal mapping $P : \{a_1, a_2, ...a_k\} \leftarrow \{p_1, p_2, ..., p_k\}$, such that the cost function

$$\text{cost}(P) = \sum_{i,j=1..k, i\neq j} \text{Cov}(a_i, a_j) D(p_i, p_j)$$

is minimized. An exhaustive search of such an optimal mapping needs to try all the possible $k!$ candidates and is impractical. We implemented the simulated annealing algorithm to solve this optimization problem. Fig. 4.13 shows the effectiveness of our optimization on the accelerator mapping. For the crossbar with the same number of switches, the routability is increased by

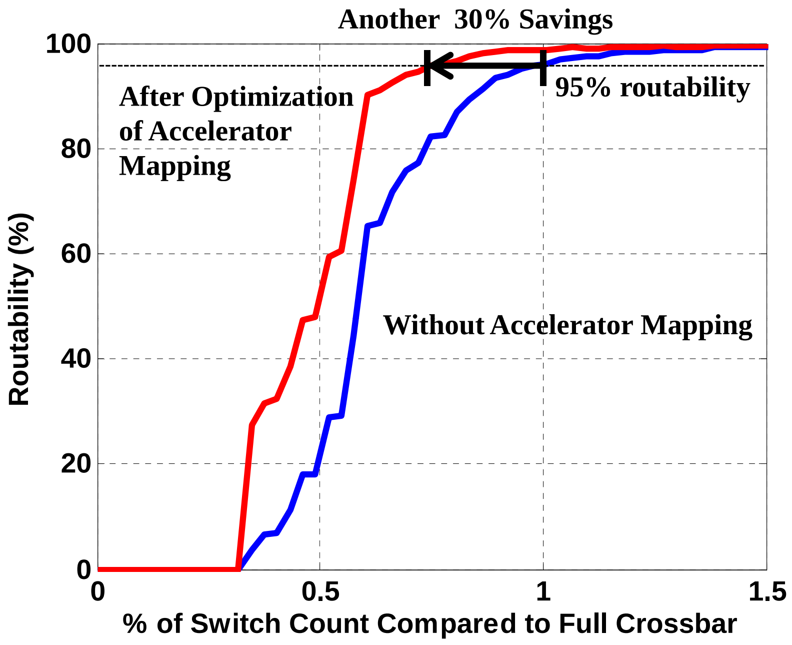


Figure 4.13: 30% reduction of switch count at 95% routability after optimization of accelerator mapping based on accelerator heterogeneity.

$\sim 30\%$ after we optimize accelerator mapping to scatter highly correlated accelerators to reduce their conflicts on data paths. In other words, we have more routability to trade off for a smaller number of switches to be used in a crossbar. While keeping the 95% routability, the number of switches is further reduced by $\sim 30\%$, as shown in Fig. 4.13

## 4.6 Verification and Comparisons

### 4.6.1 Verification on Commodity FPGA

Table 4.1: Comparison of private memories, shared memories via buses [18], and our crossbar in FPGA implementation and testing.

| | memory usage | interconnect cost in # of LUTs | accelerator subtask runtime |
|---|---|---|---|
| private memories | 3328KB (177%) | 0 | 10.3us |
| shared memories via buses similar to [18] | 768KB (41%) | 50043 (33%) | 117us |
| shared memories via our crossbar | 768KB (41%) | 3169 (2%) | 10.3us |

We implement an automatic tool to generate the hardware of our crossbar design. We create an accelerator-rich platform [58] in a Xilinx®Virtex6 FPGA chip in a ML605 board [64] and integrate our crossbar to bridge the accelerators and shared memories in the platform. We also implement two other versions for comparison. The first one is accelerators using private memories without sharing. The second one is accelerators accessing shared memories via shared buses similar to that in [18]. The bus hardware is provided by the Xilinx AXI4 bus IP. Table 4.1 shows the experimental results. Due to limited FPGA resources, we implement only six accelerators in the accelerator-rich platform under test. The primary goal of this experiment is to verify whether our automatically generated crossbar works in real hardware and meets timing. The percentages in Table 4.1 are in terms of the total FPGA resources available in the Virtex6 chip. Table 4.1 shows that memory sharing leads to significant memory savings. It also shows that our partial crossbar between accelerators and shared memories consumes little area but achieves the same accelerator performance as private buffers. That is because, unlike bus interconnect, there is no arbitration on the crossbar interconnects for each data access. Instead, arbitration is performed when the accelerator manager in our platform schedules workloads to a set of accelerators and configures the crossbar to build data paths. In this experiment, our crossbar is implemented in combinational logic which guarantees that the clock cycle count of accelerator performance is the same as that of private buffers. The locality-driven design of our crossbar described in Section 4.4 guarantees that the data paths through the crossbar can meet timing constraints. As a result, accelerators using our crossbar have 11.3x better performance than those using a traditional network.

### 4.6.2 Comparison with Prior Work

Since this work is the first attempt to optimize interconnects between accelerators and shared memories, there is no prior work for us to use as a comparison. However, we choose a recent work on the crossbar design that routes signals in PLDs [11] for comparison. The work in [11] optimized the crossbar topology to maximize the difference of the positions of switches connected by different input signals. Fig. 4.14 shows the comparison of our crossbar design with the design in [11]. In the experiments in [11], the number of crossbar inputs was 168, the number of outputs was 24, and the number of signals to route was 12. To match their settings, we set $k = 28$, $n = 6$, $c = 2$, and $m = 24$ in our experiments. As shown in Fig. 4.14, our crossbar design outperforms the work in [11]. This is because the work in [11] performs optimization for general-purpose signal routing, while our crossbar design fully exploits the optimization opportunities that emerge in accelerator-rich computing platforms as identified in this work.

## 4.7 Conclusion and Future Work

This chapter provides several optimization technologies on the interconnects between accelerators and shared memories. It intensively exploits the optimization opportunities that emerge in accelerator-rich computing platforms: 1) The multiple data ports of the same accelerators are powered on/off together, and the competition for shared resources among these ports can be eliminated

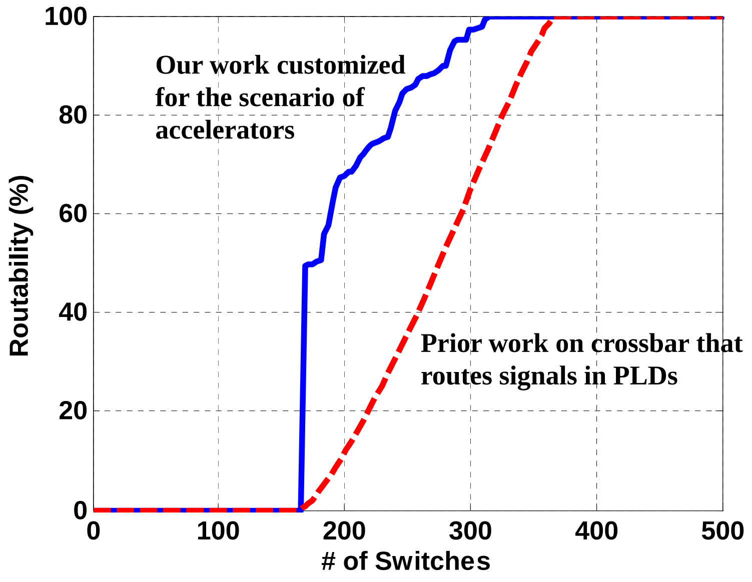


Figure 4.14: Comparison of our crossbar design with the design in [11]. Our crossbar design outperforms the work in [11] since the work in [11] performs optimization for general-purpose signal routing; our crossbar design fully exploits the optimization opportunities that emerge in accelerator-rich computing platforms as identified in this work.

to save interconnect transistor cost; 2) In dark silicon, the number of active accelerators in an accelerator-rich platform is usually limited, and the interconnects can be partially populated to just fit the data access demand limited by the power budget; 3) The heterogeneity of accelerators leads to execution patterns among accelerators and, based on the probability analysis to identify these patterns, interconnects can be optimized for the expected utilization. Experiments show that our interconnect design outperforms prior work that was optimized for CPU cores or signal routing.

As this is the first work on optimization of interconnects in the scenario of accelerators in dark silicon, our primary goal is to identify new research opportunities and potential benefits as presented in this work. Many opportunities exist for further improvement. For example, to focus on the demonstration of the benefits achieved by the technologies proposed in this work, we simply assume that all the accelerators have $n$ data ports. When each accelerator has a different number of data ports, a new design space is opened. Another example is that in this work, we mainly focus on the routability optimization in the simplified case where the $c$ accelerators are powered on at the same time. In practice, the thermal design power (TDP) constraint is considered in dark silicon, and different types of accelerators contribute different power consumptions. Also, computation tasks will come in one by one, and accelerators will be powered on and off successively. All of these are interesting research problems that need further investigation. We believe that accelerator-rich computing platforms are promising computer architectures and will encourage more research opportunities than the interconnect design.

# CHAPTER 5

# Data Reuse Optimization for Stencil Access Patterns

## 5.1 Introduction

Accelerator-centric architectures can bring 10-100x energy efficiency by offloading computation from general-purpose CPU cores to application-specific accelerators [64]. The previous chapter covers the global communication optimization in the accelerator-centric architectures. In this chapter, we focus on the communication within each accelerator. The engineering cost of designing massive heterogeneous accelerators is high, but can be much reduced by raising their abstraction level beyond RTL to C by high-level synthesis (HLS) [65]. Data access optimization has a strong impact on HLS results. This significantly motivates recent work on data reuse [66, 67] and memory partitioning [12, 68, 13, 14, 69] in HLS.

External memory bandwidth is a significant bottleneck for system performance and power consumption [1, 70, 71]. Data reuse is an efficient technique for using on-chip memories to reduce external memory accesses. When an application contains a data array with multiple references, we can allocate a reuse buffer and keep each array element in the buffer from its first access until its last access. Then each array element needs to be fetched from the external memory only once, and the off-chip traffic is reduced to the minimum. Loop transformation can be applied to improve data locality and reduce the size of the data reuse buffer [66, 67].

When the loops in an application are fully pipelined, an accelerator needs to perform multiple load operations from the same reuse buffer every clock cycle. To avoid contention on memory ports, memory partitioning of the reuse buffer is required. Since the transistor count of memory control logics is proportional to the number of memory banks after partitioning, the optimization goal of memory partitioning is to minimize the number of memory banks. The constraint is that the multiple array elements to be loaded every clock cycle are always stored in different memory banks. The work in [12, 68] provides solid frameworks of memory partitioning. Further optimizations, including memory access rescheduling [13] and multidimensional arrays [14], are also proposed. But none of these can guarantee the optimal solution for a given case in terms of the number of banks. The reason is that their optimization space is limited to uniform memory partitioning; i.e., all the memory banks have to be of the same size. It is an unnecessary constraint which was assumed by commodity HLS tools, e.g., [72]. Other work partitions different fields of a single data *structure* into multiple memory banks for data parallelism based on profiling results [69]. It is orthogonal to the problems on multiple array references in [12, 68, 13, 14] and this work.

In this chapter we go beyond the limitation of uniform memory partitioning, and propose a novel method based on nonuniform memory partitioning. As a result, we can achieve fewer memory banks than the optimal solutions in prior work [12, 68, 13, 14] which were limited to uniform memory partitioning. As an early-stage exploration of nonuniform memory partitioning, we focus on stencil computation, a popular communication-intensive application domain. We develop a microarchitecture with novel structures of memory systems which achieve the theoretical minimum number of memory banks for any stencil access patterns. Experimental results show that we can reduce $25 - 100\%$ of various resources, including BRAMs, logic slices, and DSPs, compared to prior work [14], along with slightly improved timing.

## 5.2 Preliminary

### 5.2.1 Stencil Computation

Stencil computation comprises an important class of kernels in many application domains, such as image processing, constituent kernels in multigrid methods, and partial differential equation solvers. These kernels often contribute to most workloads in these applications. Even in recent technologies on memory partitioning [13, 14] which were developed for general applications, all the benchmarks used are in fact stencil computation.

The data elements accessed in stencil computation are on a large multidimensional grid which usually exceeds on-chip memory capacity. The computation is iterated as a stencil window slides over the grid. In each iteration, the computation kernel accesses all the data points in the stencil window to calculate an output. Both the grid shape and the stencil window can be arbitrary, as specified by the given stencil applications. A precise definition of stencil computation can be found in [130] and Section 5.2.4.

```
void denoise2D( float A[768][1024],
                float B[768][1024] )
{
    for( int i = 1; i < 767; i++ )
        for( int j = 1; j < 1023; j++ )
            B[i][j] =
                pow(A[i][j] - A[i][j-1], 2) +
                pow(A[i][j] - A[i][j+1], 2) +
                pow(A[i][j] - A[i-1][j], 2) +
                pow(A[i][j] - A[i+1][j], 2);
}
```

Listing 5.1: Example C code of a typical stencil computation (5-point stencil window in the kernel ‘DENOISE’ in medical imaging [19]).

Listing 5.1 shows an example stencil computation in the kernel ‘DENOISE’ in medical imaging [19]. Its grid shape is a $768 \times 1024$ rectangle, and its stencil window contains five points, as shown in Fig. 5.1. Five data elements need to be accessed in each iteration. In addition, many data elements will be repeatedly accessed among these iterations. For example, $A[2][2]$ will be accessed five times, when $(i, j) \in \{(1, 2), (2, 1), (2, 2), (2, 3), (3, 2)\}$. This leads to high on-chip memory port contention and off-chip traffic, especially when the stencil window is large (e.g., after loop

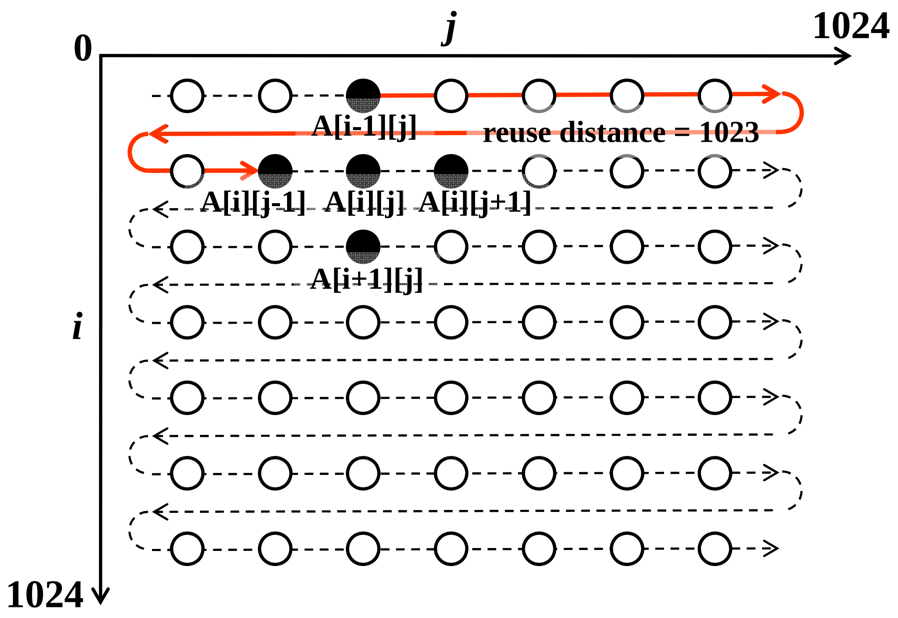


Figure 5.1: Iteration domain of the example stencil computation in Listing 5.1.

fusion of stencil applications for computation reduction as proposed in [131]). Therefore, during the hardware development of a stencil application, a large portion of engineering effort is spent on data reuse and memory partitioning optimization.

### 5.2.2 Microarchitecture for Stencil Accesses

We develop a microarchitecture to decouple the stencil access patterns from the computation, as shown in Fig. 5.2. The microarchitecture contains multiple memory systems, and each is optimized to a data array with stencil accesses. Since there are no reuse opportunities among different data arrays, the memory systems for different arrays are independent of each other. Each memory system receives a single data stream which iterates on a multi-dimensional grid without any repeated external access. Each memory system contains data reuse buffers, memory controllers and interconnects that have been customized for the access patterns of the data array used in the target stencil computation. A memory system sends data to the computation kernel via each data port associated with each array reference in the original user code. If all the data are consumed by the

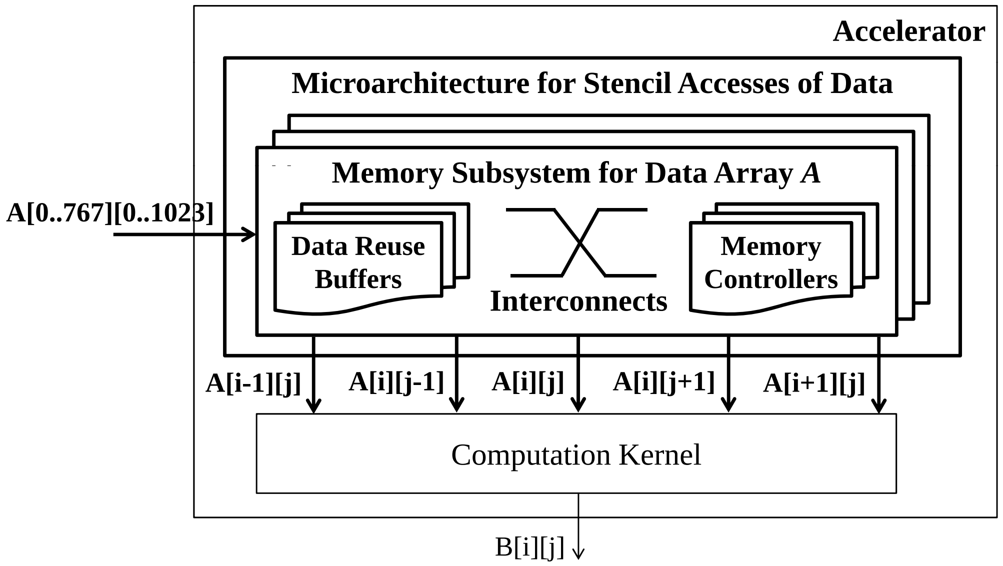


Figure 5.2: The overall architecture of our microarchitecture for stencil computation. It decouples stencil accesses from computation.

computation kernel, the memory system will immediately prepare the data used in the next cycle to feed into the fully pipelined computation kernel.

```
void denoise2D_kernel( volatile float* a0_ptr,
                       volatile float* a1_ptr,
                       volatile float* a2_ptr,
                       volatile float* a3_ptr,
                       volatile float* a4_ptr,
                       volatile float* b_ptr )
{
    for( int i = 1; i < 767; i++ )
        for( int j = 1; j < 1023; j++ )
        {
#pragma AP pipeline II=1
            float a0 = *a0_ptr;
            *b_ptr =
                    pow(a0 - *a1_ptr, 2) +
                    pow(a0 - *a2_ptr, 2) +
                    pow(a0 - *a3_ptr, 2) +
                    pow(a0 - *a4_ptr, 2);
        }
}
```

Listing 5.2: Example code of the computation kernel where all the memory accesses are offloaded to our microarchitecture. The keyword 'volatile' in the code informs HLS tools of potential data change after access. The pragma 'pipeline is used in Xilinx Vivado HLS [72] to pipeline the innermost loop.

With our microarchitecture, the C code of the computation kernel can be simplified to Listing 5.2, where users no longer need to optimize memory accesses, which is offloaded to our microarchitecture. Users can assume that each data access to the points should get the same data from our accelerator microarchitecture as the original load operation. The C code of the computation kernel can be compiled by HLS tools for a fully pipelined hardware implementation with the most efficient resource usage.

### 5.2.3 Design Objectives

We have three design objectives for the proposed microarchitecture:

1. *Full pipelining*. As the stencil window slides every clock cycle, the microarchitecture is able to send out all the data in the stencil window to the computation kernel and get all the data ready for the consecutive accesses in the next cycle.

2. *Minimum data reuse buffer size*. When a data element is fully reused for each access, it stays in on-chip memories from its first access until its last access. Meanwhile, other elements in the array are loaded from external memory every clock cycle. Therefore, the theoretical minimum size of the reuse buffer for a data array is equal to the maximum lifetime of any element in the array.

In the example in Fig. 5.1, $A[2][2]$ is accessed now by the array reference $A[i+1][j]$ for the first time, and is accessed 2048 cycles later by the reference $A[i-1][j]$ for the last time. Therefore, the minimum size of the data reuse buffer for array $A$ will be 2048.

3. *Minimum number of reuse buffer banks*. Suppose the stencil window of an input array contains $n$ points, i.e., there are $n$ data references to the array. It means that each clock cycle, $n$ data elements need to be read out, either from reuse buffers or from the external memory. Suppose we use dual-port memories to implement reuse buffers. One port of a buffer bank is occupied by the replacement of an expired data element with a new element from the external memory every cycle. There is only one port left in each memory bank for us to read the $n$ elements needed by the stencil window. Suppose one of the $n$ element happens to be the new element from the external memory in a certain smart data reuse mechanism. There are still $n-1$ data elements to read, and we would need at least $n-1$ memory banks. In the example of Fig. 5.1, $n = 5$ indicates that we need at least four memory banks. As the stencil window slides, all five data elements in the window should never be in the same bank. This is a tough constraint to satisfy, and prior work [12, 68, 13, 14] had to use more banks to eliminate bank conflicts in difficult cases. Fig. 5.3 shows

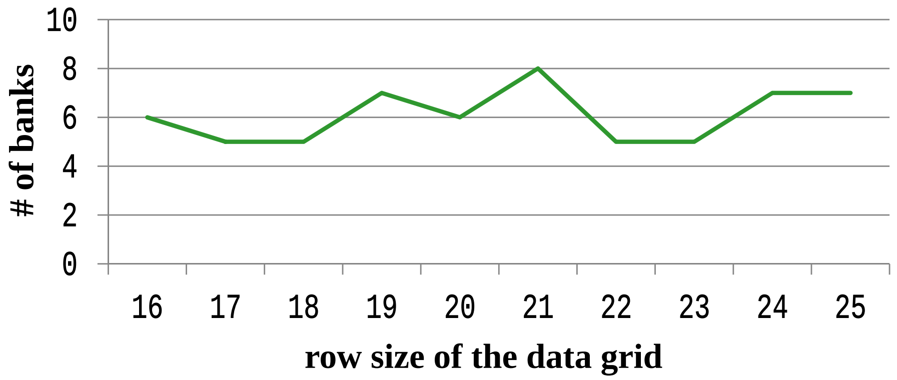


Figure 5.3: Even for a constant stencil window in Fig. 5.1, the number of banks varies as the row size of the data grid changes [12].

that the number of banks ranges from five to eight in [12] as the row size of the data grid changes, even if the stencil window keeps the constant shape in Fig. 5.1. The technologies proposed in [13, 14] can keep the number of banks consistently to be five in the case of the stencil window shown in Fig. 5.1. However when the stencil window changes to some other shape in other

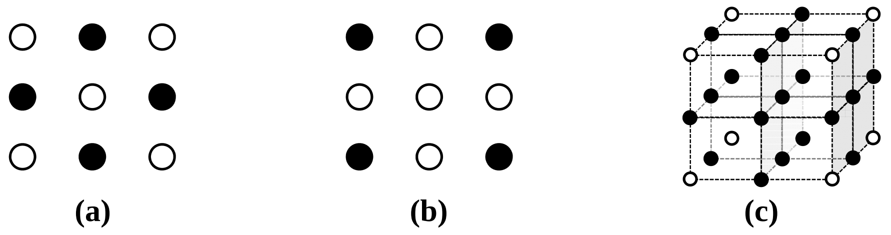


Figure 5.4: Example stencil windows where more banks are needed than the # of array references in [13, 14]. (a) 4-point stencil in 'BICUBIC' [15]. (b) 4-point stencil in 'RICIAN' [16]. (c) 19-point stencil in 'SEGMENTATION_3D' [14].

applications, e.g., the ones shown in Fig. 5.4, the methods in [13, 14] will need 5, 5, 20 banks respectively, which are larger than the minimum values.

In this chapter we will present a generalized microarchitecture that can simultaneously achieve these optimal design objectives for any application that falls in the category of stencil computation.

### 5.2.4 Polyhedral Model

We support general stencil accesses where neither the data grid nor the stencil window needs to be rectangular. It is often the case when stencil computation is applied with the loop transformation based on polyhedral model [66, 67, 132] before memory access optimization. Our work is also built on top of polyhedral model for generality as below. Several properties of stencil computation are identified under the framework of polyhedral model as well.

**Definition 3** (Iteration Domain [133]). *The iteration vector of a multi-level loop nest over a $m$-dimensional grid is a vector of iteration variable, $\vec{i} = (i_0, i_1, ..., i_{m-1})^T$, where $i_0, ...i_{m-1}$ are the iteration variables from outermost to innermost loop. Iteration domain $\mathcal{D} \subseteq \mathbb{Z}^m$ is the set of iteration vectors of the loop nest, and is expressed by a set of linear inequalities $\mathcal{D} = \left\{\vec{i} \middle| \mathbf{P}\vec{i} \geq \vec{b}\right\}$.*

**Example 1.** *Consider the loop nest execution in Listing 5.1:*

$$\mathcal{D} = \{(i_0, i_1) \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i_0 \\ i_1 \end{pmatrix} \geq \begin{pmatrix} 1 \\ 767 \\ 1 \\ 1023 \end{pmatrix}, (i_0, i_1) \subseteq \mathbb{Z}^2\}.$$

This definition of iteration domain can describe any polyhedral shape on a multi-dimensional data grid, e.g., rectangular, triangular, diamond.

**Definition 4** (Lexicographic Order [133]). *Lexicographic order relation $\succ_l$ of two iteration vectors $\vec{i}$ and $\vec{j}$ is defined as:*

$$\begin{aligned} \vec{i} \succ_l \vec{j} \;\; \Leftrightarrow \;\; & (i_0 > j_0) \vee (i_0 = j_0 \wedge i_1 > j_1) \vee \\ & (i_0 = j_0 \wedge i_1 = j_1 \wedge i_2 > j_2) \vee \\ & ... \vee (i_0 = j_0 \wedge ... \wedge i_{m-2} = j_{m-2} \wedge i_{m-1} > j_{m-1}) \end{aligned}$$

**Definition 5** (Access Function [133]). *For a k-dimensional array reference, its access function $\mathcal{H} \subseteq \mathbb{Z}^m \rightarrow \mathbb{Z}^k$ is the mapping from iteration vector $\vec{i}$ to the access index $\vec{h}$:*

$$\mathcal{H} = \left\{ \vec{i} \rightarrow \vec{h} \,\middle|\, \vec{h} = \mathbf{H}_{k \times m} \vec{i} + \vec{f} \right\}$$

**Example 2.** *The access function of the array reference $A[i][j+1]$ in Listing 5.1 is*

$$\vec{h} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \vec{i} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

**Definition 6** (Stencil Computation). *Under the framework of polyhedral model, stencil computation is defined as: the access function of each array reference satisfies*

$$\begin{cases} k = m \\ \mathbf{H}_{k \times m} = \mathbf{I}_{m \times m} \\ \left|\vec{f}\right| \ll \left|\vec{b}\right|, \end{cases}$$

*where* $\mathbf{I}$ *is the identify matrix.*

It means that for any reference $A_x$ of a data array $A$,

$$\vec{h}_x = \vec{i} + \vec{f}_x. \tag{5.1}$$

**Definition 7** (Data Domain). *For an array reference $A_x$, data domain of $\mathcal{D}_{A_x}$ is the set of data elements accessed by $A_x$, and is expressed by a set of linear inequalities*

$$\mathcal{D}_{A_x} = \left\{ \vec{h} \middle| \mathbf{P} \left( \vec{h} - \vec{f}_x \right) \geq \vec{b} \right\}$$

**Example 3.** *The data domain of $A[i][j+1]$ is*

$$\{(i,j) \,|2 \leq i \leq 767, 1 \leq j \leq 1022\}.$$

Followed by Eq.(5.1), we have the following property:

**Property 1** (Lexicographic Access Pattern). *In stencil computation, data elements in $D_{A_x}$ are accessed by the array reference $A_x$ in the lexicographic order.*

**Definition 8** (Input Data Domain). *For an data array $A$ with $n$ references $A_0$, $A_1$,...,$A_{n-1}$, the input data domain of $A$ is the union set of data elements that need to be fetched from external memory, defined as*

$$\mathcal{D}_A = \mathcal{D}_{A_0} \cup \mathcal{D}_{A_1} \cup ... \cup \mathcal{D}_{A_{n-1}}$$

**Example 4.** *The input data domain of array $A$ is*

$$\{(i,j) \,|0 \leq i \leq 767, 0 \leq j \leq 1023\} -$$
$$\{(0,0), (0,1023), (767,0), (767,1023)\}.$$

In this thesis, we use $A[0..767][0..1023]$ to represent this input data domain.

**Definition 9** (Reuse Distance Vector [133]). *Suppose $A_x$ and $A_y$ are two read references of the same data array. If the data element accessed by $A_x$ in loop iteration $\vec{i}$ is accessed again by $A_y$ in iteration $\vec{j}$, the reuse distance of vector from $A_x$ to $A_y$ is $\vec{r}_{A_x \to A_y} = \vec{j} - \vec{i}$.*

By Definition 9, we have $\mathbf{H}_x(\vec{i}) = \mathbf{H}_y(\vec{j})$. By Definition 6, we have $\mathbf{H}(\vec{i}) = \vec{i} + \vec{f}$. Combining these two, we have the following property:

**Property 2.** *In stencil computation, reuse distance vector $\vec{r}_{A_x \to A_y} = \vec{f}_x - \vec{f}_y$ is a constant vector.*

**Example 5.** *The reuse distance vector from $A[i-1][j]$ to $A[i+1][j]$ in Listing 5.1 is $(2,0)^T$.*

**Definition 10** (Reuse Distance)**.** *Reuse distance $\left|\vec{r}_{A_x \to A_y}\right|$ from array reference $A_x$ to $A_y$ is the number of data elements between $\vec{h}$ accessed by $A_x$ and $\vec{h} + \vec{r}$ accessed by $A_y$, defined as*

$$\left|\vec{r}_{A_x \to A_y}\right|\Big|_{\vec{h}} = \left|\left\{\vec{g}\,\middle|\,\vec{h} + \vec{r} \succ_l \vec{g} \succeq_l \vec{h}, \vec{g} \in \mathcal{D}_A\right\}\right|$$

**Example 6.** *The reuse distance between $A[i-1][j]$ and $A[i+1][j]$ in Listing 5.1 is a constant value 2048.*

**Definition 11** (Maximum Reuse Distance)**.** *The maximum reuse distance from array reference $A_x$ to $A_y$ is defined as*

$$\left\|r_{A_x \vec{\to} A_y}\right\| = \max_{\vec{h} \in \mathcal{D}_{A_x}} \left|\vec{r}_{A_x \to A_y}\right|_{\vec{h}}$$

**Example 7.** *The* maximum *reuse distance between $A[i-1][j]$ and $A[i+1][j]$ in Listing 5.1 is 2048.*

Suppose three data array references $A_x$, $A_y$ and $A_z$ satisfy

$$\vec{r}_{A_x \to A_y} \succ_l 0, \vec{r}_{A_y \to A_z} \succ_l 0$$

Followed by Definition 10, we have

$$\left|\vec{r}_{A_x \to A_z}\right|_{\vec{h}} = \left|\vec{r}_{A_x \to A_y}\right|_{\vec{h}} + \left|\vec{r}_{A_y \to A_z}\right|_{\vec{h} + \vec{r}_{A_x \to A_y}}$$

Followed by Definition 7 and Property 2, we have

$$\vec{h} \in \mathcal{D}_{A_x} \Rightarrow \left(\vec{h} + +\vec{r}_{A_x \to A_y}\right) \in \mathcal{D}_{A_y}$$

Combining these two with Definition 11, we have the following property:

**Property 3** (Linearity of Maximum Reuse Distance)**.** *In stencil computation,*

$$\left\|\vec{r}_{A_x \to A_z}\right\| = \left\|\vec{r}_{A_x \to A_y}\right\| + \left\|\vec{r}_{A_y \to A_z}\right\|$$

## 5.3 Methodology

### 5.3.1 Overview

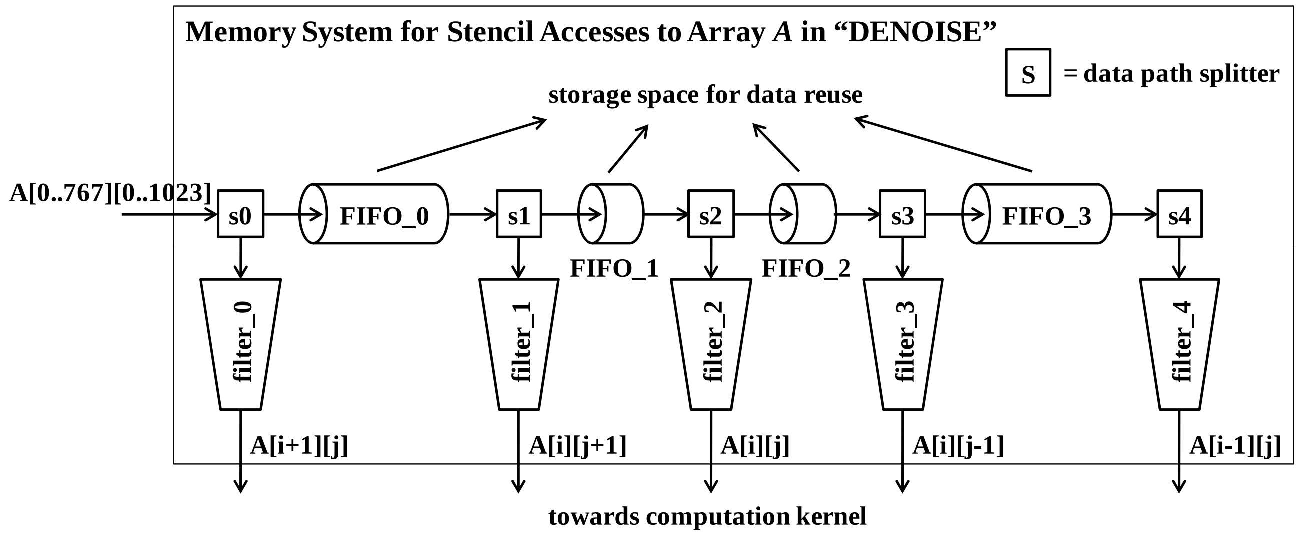


Figure 5.5: The example circuit structure of our memory system generated for array $A$ in the stencil computation of Listing 5.1.

The internal structure of a memory system in our microarchitecture is illustrated by the example in Fig. 5.5, which is generated for the stencil computation in Listing 5.1. Suppose the stencil window contains $n$ points ($n = 5$ in the example of Listing 5.1). Our memory system will contain $n - 1$ data reuse FIFOs as well as $n$ data path splitters and $n$ data filters connected together in the way shown in Fig. 5.5. The data reuse FIFOs provide the same storage as conventional data reuse buffers, and the data path splitters and filters work as memory controllers and data interconnects. In contrast to conventional cyclic memory partitioning [12, 13, 14] which uses uniform buffer sizes, the sizes of reuse buffers in our design are nonuniform. They are customized to the shape of the stencil window in the target application.

### 5.3.2 Denotations

To better explain the working principle of our memory system, we provide a table of denotations that will be used in the following sections in Table 5.1. The precise definitions of the denotations are given in Section 5.2.4.

| Denotations | Meanings | Example by Fig. 5.1 |
|---|---|---|
| $\vec{i}$ | loop iteration vector | $\vec{i} = (1, 2)$ |
| $A$ | data array | the array $A$ with the five references |
| $A_x$ | array reference | the five references such as $A[i+1][j]$ |
| $\vec{h}$ | data access index | $\vec{h}_x = (2, 2)$ being accessed by $A_x$ |
| $\vec{f}$ | data access offset | $\vec{f}_x = \vec{h}_x - \vec{i} = (1, 0)$ constant for $A_x$ |
| $\mathcal{D}_{A_x}$ | data domain | $\{(i, j) \,\lvert 2 \le i \le 767, 1 \le j \le 1022\}$ |
| $\mathcal{D}_A$ | input data domain | $\{(i, j) \,\lvert 0 \le i \le 767, 0 \le j \le 1023\}$ |
| $\vec{r}_{A_x \leftarrow A_y}$ | reuse distance vector | (1,-1) from $A[i+1][j]$ to $A[i][j+1]$ |
| $\lVert\vec{r}_{A_x \leftarrow A_z}\rVert$ | maximum reuse distance | 2048 from $A[i+1][j]$ to $A[i-1][j]$ |
| $\succ_l$ | lexicographic order | $(1, 0) \succ_l (0, 1) \succ_l (0, 0) \succ_l (-1, 0)$ |

Table 5.1: Denotations used in the working principle of our memory system.

### 5.3.3 Working Principle

Since our microarchitecture is based on nonuniform memory partitioning in contrast to uniform partitioning in prior work, the memory controlling mechanism cannot follow the modulo scheduling of data accesses among memory banks in prior work [12, 13, 14]. Instead, our microarchitecture is a novel design based on data streaming. Each module in our design is autonomous and can work in a full pipeline as long as its upstream module produces a data element and its downstream module consumes an element every clock cycle. As we shall discuss next, our design achieves function correctness, data streaming without stalling, the minimum reuse buffer size, and the minimum number of buffer banks.

#### 5.3.3.1 Function Correctness

Each data path splitter in Fig. 5.5 reads any existing data element from its precedent FIFO and sends the data element to the successive FIFO as well as to the data filter below. Each data filter customizes the data stream that flows into the computation kernel to fit the access patterns of the associated array reference. A data filter for an array reference $A_x$ receives the data stream which iterates in $\mathcal{D}_A$ and sends out the data which iterates in $\mathcal{D}_{A_x}$. For example, filter 0 in Fig. 5.5 sends the data element in set $\mathcal{D}_{A_0} = \{(i, j)|2 \leq i \leq 767, 1 \leq j \leq 1022\}$ out of the input data domain $\mathcal{D}_A$ in Table 5.1 and discards the first two rows in the 2D grid of Fig. 5.1. This guarantees the correctness of the data set sent to each data port of the computation kernel. Due to the property of stencil computation, the data elements accessed by an array reference are in the same lexicographic order as the loop iteration (see details in [134]). Our microarchitecture based on data streaming enforces this order, as long as the input data stream is also in the lexicographic order (i.e., data iterated from innermost loop to outermost loop). The lexicographic order of input data is usually realized without hardware overhead since it fits well with burst accesses to external memory or inter-accelerator communication patterns (see discussion in Section 5.3.6). By providing the correct data set and correct data order to the array references, our design can guarantee that as the stencil window slides, the data elements received by the computation kernel are always consistent with the array references.

#### 5.3.3.2 Data Streaming without Stalling

One key challenge is to ensure that the data streaming structure will not be stalled due to any FIFO emptiness or fullness. To prevent FIFO emptiness, we can sort the data access offsets $\vec{f}$ of the data array references in the descending lexicographic order when we map them to data filters from 0 to $n - 1$, e.g., $(1, 0) \rightarrow (0, 1) \rightarrow (0, 0) \rightarrow (0, -1) \rightarrow (-1, 0)$ in Fig. 5.5. To prevent FIFO fullness, we calculate the maximum reuse distances of all the pairs of adjacent array references and allocate reuse FIFO sizes accordingly, as shown in Table 5.2 for the example in Listing 5.1. These two

| FIFO ID | precedent/successive references | FIFO size | physical impl. |
|---|---|---|---|
| FIFO 0 | $A[i+1][j] \rightarrow A[i][j+1]$ | 1023 | BRAM |
| FIFO 1 | $A[i][j+1] \rightarrow A[i][j]$ | 1 | register |
| FIFO 2 | $A[i][j] \rightarrow A[i][j-1]$ | 1 | register |
| FIFO 3 | $A[i][j-1] \rightarrow A[i-1][j]$ | 1023 | BRAM |

Table 5.2: Reuse FIFOs with nonuniform sizes calculated from maximum reuse distances of adjacent array references and mapped to different physical implementations (block memory, distributed memory, or register) if targeted an FPGA platform.

conditions can be summarized as follows:

1. For data access offsets $f_x$ of $f_y$ of two array references,

$$x < y \Rightarrow \vec{f_x} \succ_l \vec{f_y} \tag{5.2}$$

2. For the reuse FIFO between adjacent $A_x$ and $A_y$,

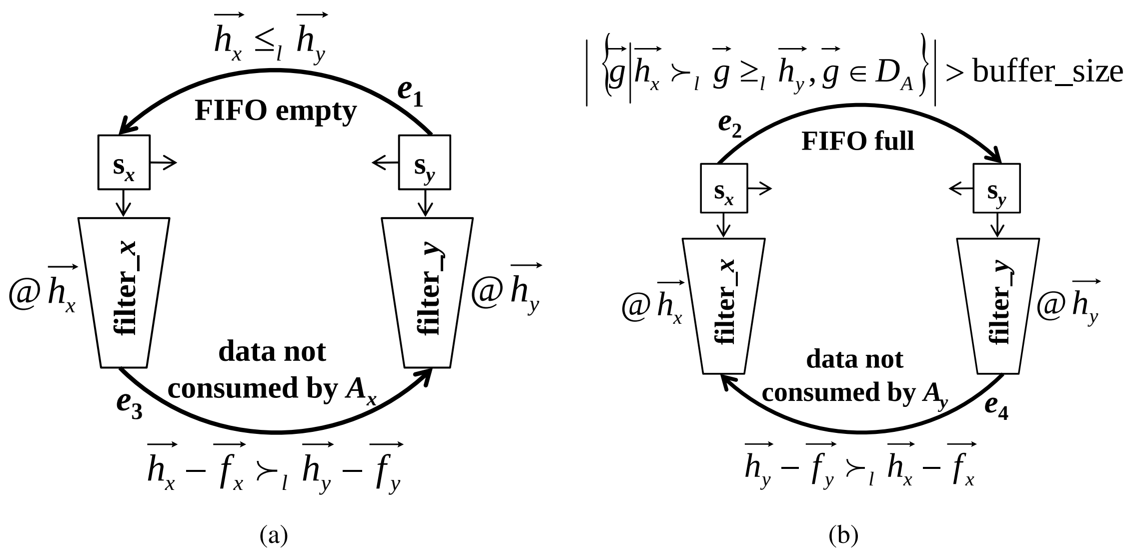
$$\text{buffer\_size} \geq \left\| \vec{f_x} - \vec{f_y} \right\| = \left\| \vec{r}_{A_x \rightarrow A_y} \right\| \tag{5.3}$$

Next we will show that our data streaming structure is deadlock-free. We draw the system dependency graph with the dependency edges $e1, e2, e3, e4$ in Fig. 5.6. Suppose at a certain moment, 'filter_$x$' is iterating over $\vec{h_x} \in \mathcal{D}_A$ as shown in Fig. 5.6(a), and 'filter_$y$' is iterating over $\vec{h_y} \in \mathcal{D}_A$, as shown in Fig. 5.6(b). As we know, if there are no cycles in the dependency graph of a system, this system is deadlock-free. Therefore we prove the following theorem.

**Theorem 5.** *The two conditions listed in Eq.* (5.2) *and Eq.* (5.3) *are the sufficient conditions to eliminate dependency cycles in Fig. 5.6.*

Before the formal proof, first let us explain the expressions of the four dependency relations.

A data filter will stall if there is data emptiness of a precedent reuse FIFO or data fullness of a successive reuse FIFO. When the condition of $e_1$ is true, there will be no data stored in the reuse



$\overrightarrow{h_x} \preceq_l \overrightarrow{h_y}$
$e_1$
FIFO empty
$s_x$
$s_y$
filter_x
filter_y
$@\overrightarrow{h_x}$
$@\overrightarrow{h_y}$
data not consumed by $A_x$
$e_3$
$\overrightarrow{h_x} - \overrightarrow{f_x} \succ_l \overrightarrow{h_y} - \overrightarrow{f_y}$
(a)
$\left|\left\{\vec{g} \,\middle|\, \overrightarrow{h_x} \succ_l \vec{g} \succeq_l \overrightarrow{h_y}, \vec{g} \in D_A\right\}\right| >$ buffer_size
$e_2$
FIFO full
$s_x$
$s_y$
filter_x
filter_y
$@\overrightarrow{h_x}$
$@\overrightarrow{h_y}$
data not consumed by $A_y$
$e_4$
$\overrightarrow{h_y} - \overrightarrow{f_y} \succ_l \overrightarrow{h_x} - \overrightarrow{f_x}$
(b)

Figure 5.6: The dependency graph of data filters 'filter_$x$' and 'filter_$y$' where $x < y$, with polyhedral expressions of dependency relations appended.

FIFO(s) between 'filter_$x$' and 'filter_$y$', and 'filter_$y$' will stall. It will wait for 'filter_$x$' to pull more data from the data path splitter 's_$x$' so that 's_$x$' can push more data to the downstream modules. When the condition of $e_2$ is true, the number of data elements stored in the reuse FIFO(s) between 'filter_$x$' and 'filter_$y$' (i.e., the number of data elements between $\vec{h_x}$ and $\vec{h_y}$) exceeds the sum of the sizes of reuse FIFOs between 'filter_$x$' and 'filter_$y$', and 'filter_$x$' will stall. It will wait for 'filter_$y$' to pull more data from 's_$y$' so that 's_$y$' can pull more data from the upstream.

A data filter will also stalled if the data element sent to the computation kernel is not consumed. This situation happens when the kernel is waiting for data elements from other data filters. When the condition of $e_3$ is true, 'filter_$x$' will stall. It has sent the data element needed by the computation kernel in iteration $\vec{i_x} = \vec{h_x} - \vec{f_x}$, but the computation kernel is waiting for 'filter_$y$' to send the data element needed in an iteration $\vec{i_y} = \vec{h_y} - \vec{f_y}$, which is earlier than $\vec{i_x}$. Similarly, when the condition of $e_4$ is true, 'filter_$y$' will wait for 'filter_$y$' to send data.

*Proof.* When the first condition in Eq. (5.2) is true, we have both

$$e_1 \text{ is true} \Rightarrow \vec{h_x} \preceq_l \vec{h_y} \prec_l \vec{h_y} - \vec{f_y} + \vec{f_x} \Rightarrow e_3 \text{ is false}$$

and

$$e_3 \text{ is true} \Rightarrow \vec{h_x} - \vec{f_x} \succ_l \vec{h_y} - \vec{f_y} \succ_l \vec{h_y} - \vec{f_x} \Rightarrow e_1 \text{ is false}$$

Therefore $e_1$ and $e_3$ in Fig. 5.6(a) are mutually exclusive.

When the condition in Eq. (5.3) is true, due to the linearity property of maximum reuse distances in stencil computation (Property 3 in Section 5.2.4), the condition applies any pair of array references. Then we can apply the similar scheme in the first condition to the second condition, and prove that $e_2$ and $e_4$ in Fig. 5.6(b) are mutually exclusive.

Then all the cycles in the dependency graph of data filters 'filter_$x$' and 'filter_$y$' will not be simultaneously satisfied, and our distributed system is deadlock-free. □

#### 5.3.3.3 Design Optimality

**Theorem 6.** *The design proposed in Fig. 5.5 uses the minimum reuse buffer size.*

*Proof.* Due to the linearity of maximum reuse distances, the sum of the sizes of all the reuse FIFOs is equal to the maximum reuse distance between array reference $A_0$ and $A_{n-1}$. Due to the sorting of array references by $\vec{f}$ in the descending lexicographic order, $A_0$ is the earliest reference and $A_{n-1}$ is the latest reference. Therefore, the total reuse buffer size is equal to the maximum reuse distance between the earliest and latest references, which is the theoretical minimum. □

As shown in Table 5.2 for the example in Listing 5.1, the total size is 2048, the same as the minimum value discussed in Section 5.2.3. If the maximum reuse distance is so large that the buffer sizes exceed the on-chip memory capacity, our microarchitecture allows trade-off of off-chip bandwidth occupation for smaller memory usage (see discussion in Section 5.3.7).

**Theorem 7.** *The design proposed in Fig. 5.5 uses the minimum number of buffer banks.*

*Proof.* The structure of our design guarantees that for $n$ array references, there are $n - 1$ buffer banks (reuse FIFOs) as described in Section 5.3.1. It is the theoretical minimum value as discussed in Section 5.2.3. □

Since our design achieves both the minimum reuse buffer size and the minimum number of buffer banks, our microarchitecture is optimal.

### 5.3.4 Insights Gained From RTL Simulation

Our microarchitecture is quite different from conventional designs with centralized controllers [12, 68, 13, 14]. The major tasks of a conventional controller includes two aspects:

1. *Filling up reuse buffers.* Before the computation starts, the controller will first fill reuse buffers with data elements needed by the computation kernel.

2. *Evict expired data from reuse buffers.* The challenging part in this function is when the reuse distance between array references changes as the execution advances. This often happens on a skewed data grid. In this case, the number of data elements stored in reuse buffers changes as time goes, and the symmetry between read and write is broken.

There is no specific module that takes charge of these key tasks in our microarchitecture. Instead, by observing the execution of our design in RTL simulation, we found that these key tasks are done automatically by the coordination of our distributed modules.

#### 5.3.4.1 Automatic Filling of Reuse Buffers

The filling process of reuse buffers in our microarchitecture is shown in Table 5.3. The data filter 4

| clock | data in | filter status (forwarding/bypassing/stalled) | | | | | FIFO status (# of data) | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| cycle | stream | filter 0 | filter 1 | filter 2 | filter 3 | filter 4 | FIFO 0 | FIFO 1 | FIFO 2 | FIFO 3 |
| 1 | $A[0][1]$ | bypass | bypass | bypass | bypass | forward→stall | 0 | 0 | 0 | 0 |
| 1024 | $A[1][0]$ | bypass | bypass | bypass | forward→stall | stall | 0 | 0 | 0 | 1023 |
| 1025 | $A[1][1]$ | bypass | bypass | forward→stall | stall | stall | 0 | 0 | 1 | 1023 |
| 1025 | $A[1][2]$ | bypass | forward→stall | stall | stall | stall | 0 | 1 | 1 | 1023 |
| 2028 | $A[2][1]$ | bypass→forward | stall→forward | stall→forward | stall→forward | stall→forward | 1023 | 1 | 1 | 1023 |
| 2049–... | $A[2][2]$–... | forward | forward | forward | forward | forward | 1023 | 1 | 1 | 1023 |

Table 5.3: The execution flow of our microarchitecture in the example of Listing 5.1. The latency among the data streams at different modules is ignored here for demonstration purpose only.

associated with $A[i-1][j]$ is first stalled at cycle 1. This is when the filter 4 tries to send data $A[0][1]$ to the computation kernel but all the other data filters are bypassing this data. As a result, the computation kernel will be waiting for data from the other data filters and will not consume data from the filter 4. This stalling will lead to filling up of data in the FIFO 3 between $A[i-1][j]$ and $A[i][j-1]$. The other four filters will keep the data stream advancing since all of them bypass the first row in the data domain. 1023 cycles later, the filter 3 will try to to send data $A[1][0]$ to the computation kernel but will be stalled as well. Then the FIFO 2 will start being filled up, and the data path splitter s3 will stop sending data to FIFO 3. The following process is similar until FIFO 1 and FIFO 0 are filled up consecutively, as shown in Table 5.3. Then at cycle 2049, the filter 0 receives $A[2][1]$ and will send the data to the computation kernel. All the data at the five inputs of the computation kernel become valid, and the kernel consumes all the five data to produce the first output. Then all the stalled filters can continue to send new data to the computation kernel every clock cycle until the end of the iteration domain.

#### 5.3.4.2 Automatic Adjustment of Reuse Data Amount

An application can have a skewed data grid as shown in Fig. 5.7. This kind of application is usually needed when a rectangular grid is iterated along the $45^\circ$ direction after certain loop transform [132]. The skewed grid leads to the challenge that the number of data stored in each reuse buffer will change as the iteration goes on, and often requires a complex memory controlling scheme in a centralized design [12, 13, 14]. However, this difficulty can be automatically handled by our distributed modules. Following the design schematic in Section 5.3.1, we will order the array references and map them to five filters 0–4. Among them, filter 2 is for reference $A[i][j]$ and filter 3 is for reference $A[i-1][j]$. Note that $\vec{h}_2$ of filter 2 advances $\vec{h}_3$ of filter 3 by one row when $\vec{h}_2$ and $\vec{h}_3$ are synchronized by the computation kernel, as shown in Fig. 5.7. At each turn around to the next row in the data domain, the filter 3 will fetch one more data from the FIFO splitter than the filter 2 since the filter 3 is iterating over a longer row. Then the number of data stored in FIFO 2 between filter 2 and filter 3 will be reduced by one. This achieves the dynamic adaption of

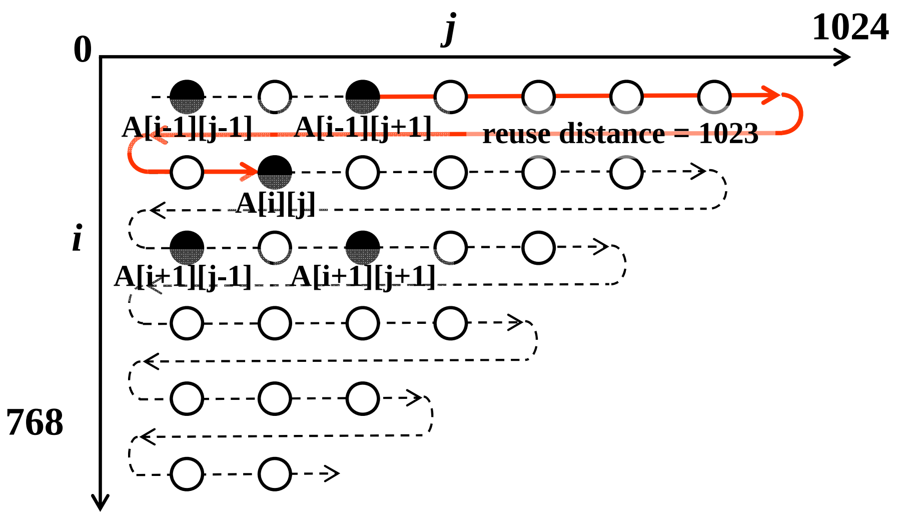


Figure 5.7: Non-rectangular iteration domain of an example application with dynamically changeable reuse distance.

the number of data stored in a reuse buffer to the change of reuse distance in the case of a skewed data grid.

### 5.3.5 Other Design Issues

#### 5.3.5.1 Heterogeneous Mapping of Reuse Buffers

Reuse buffers in our design have different sizes, as shown in Table 5.2, and may prefer different physical implementations. For example, if the target platform is FPGA, the physical implementation candidates include block memory, distributed memory and slice registers. They are efficient for a large buffer, a medium buffer and a small buffer respectively. Table 5.2 shows the heterogeneous mapping of reuse buffers to different physical implementation.

#### 5.3.5.2 Data Filter in Polyhedral Domain

Note that though the data domains are rectangles in the example of Listing 5.1, they could be any polyhedrons on a multi-dimensional grid. Comparisons on loop bounds is not a universal solution to data filtering. To select $\mathcal{D}_{A_x}$ out of $\mathcal{D}_A$, the data filter in our microarchitecture is implemented with a data switch controlled by two counters, let's say input counter and output counter, as shown in Fig. 5.8. The input counter iterates over the input data stream $\mathcal{D}_A$ of array $A$,

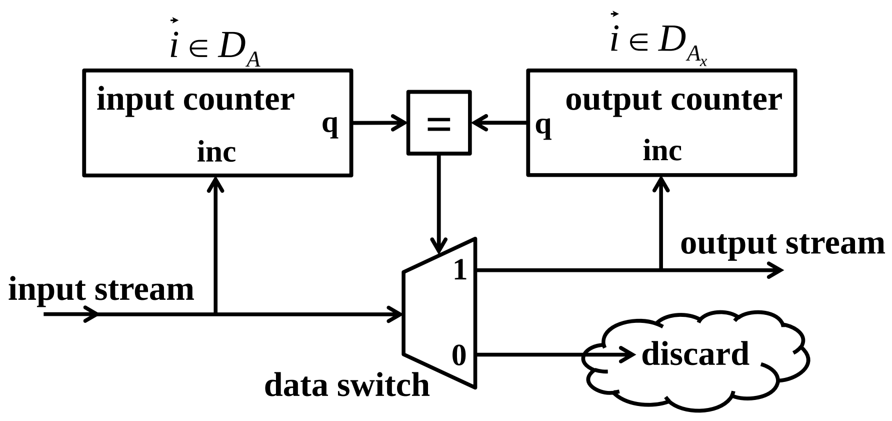


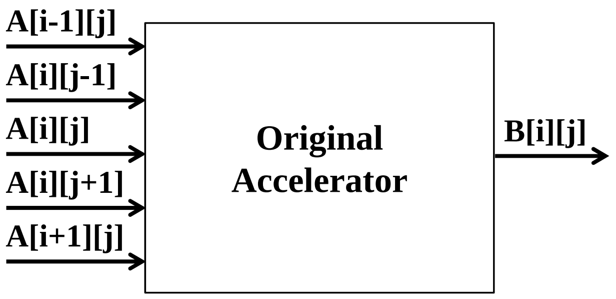
Figure 5.8: Structure of the data filter which can be applied to general polyhedral data domains.

e.g., $A[0..767][0..1023]$. The output counter iterates over the data domain $\mathcal{D}_{A_x}$ of array reference $A_x$, e.g., $A[2..767][1..1022]$. The input counter proceeds when the filter receives an input data. The output counter proceeds when its value is equal to the input counter. It is also the condition that the data switch forwards the input data to the output. In contrast, when the output counter is not equal to the input counter, the data switch bypasses the input data.

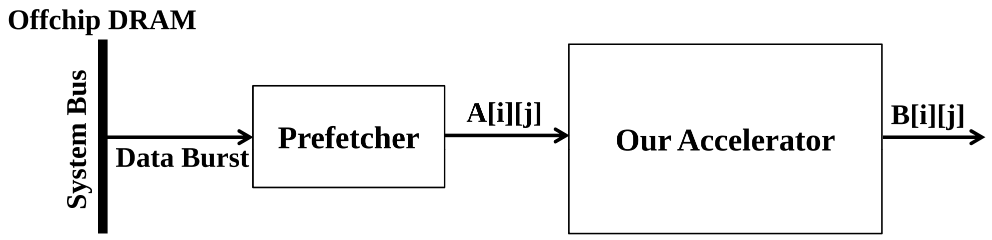
### 5.3.6 System Integration

Note that our methodology transforms the original accelerator with multiple data references, e.g., Fig. 5.9(a), to a new accelerator with a single reference, e.g., Fig. 5.9(b). An accelerator with our microarchitecture can be easily connected to an off-chip prefetch module with bus burst accesses. The prefetch module can directly forward the data stream from the bus pipeline to the accelerator and only needs a small buffer to hide the bus latency.

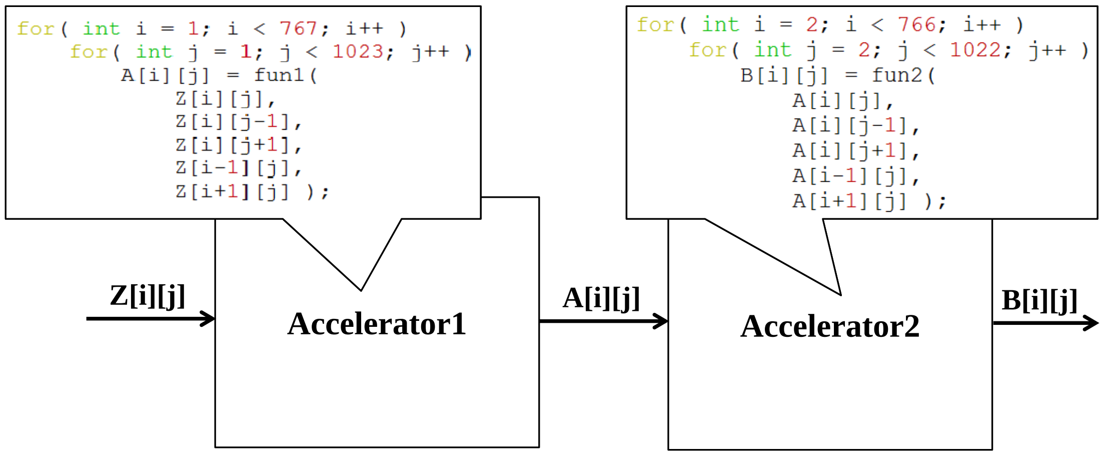
In addition, our microarchitecture can simplify the co-optimization of accelerators for inter-block communication. In the example of Fig. 5.9(c), accelerator 1 writes out $A[i][j]$. Accelerator 2 reads $A[i][j]$, $A[i][j-1]$, $A[i][j+1]$, $A[i-1][j]$ and $A[i+1][j]$. The load operations of accelerator 2



(a) Original accelerator with multiple data references.



(b) Our accelerator with a single data reference can be easily connected to an off-chip prefetch module with bus burst accesses.



```
for( int i = 1; i < 767; i++ )
    for( int j = 1; j < 1023; j++ )
        A[i][j] = fun1(
            Z[i][j],
            Z[i][j-1],
            Z[i][j+1],
            Z[i-1][j],
            Z[i+1][j] );
```

```
for( int i = 2; i < 766; i++ )
    for( int j = 2; j < 1022; j++ )
        B[i][j] = fun2(
            A[i][j],
            A[i][j-1],
            A[i][j+1],
            A[i-1][j],
            A[i+1][j] );
```

(c) Our accelerator with a single data reference can be easily co-optimized with another accelerator by loop transformation for data forwarding.

Figure 5.9: Integration of an accelerator with our microarchitecture in a system.

are very different from the store operation of accelerator 1. To implement the inter-block communication, accelerator 1 has to write the $1022 \times 766$ output data block to an on-chip block memory first. Then accelerator 2 reads data from the block memory. With our microarchitecture, however, the input data stream of accelerator 2 is transformed to $A[i][j]$, which has the same form as the output of accelerator 1. Then data can be directly forwarded from accelerator 1 to accelerator 2 without a large on-chip block memory.

In summary, our microarchitecture facilitates system-level synthesis in terms of optimization on module-level communication and off-chip communication.

### 5.3.7 Enabling Bandwidth/Memory Trade-off

There is trade-off between off-chip bandwidth and on-chip memory usage. When we have larger off-chip bandwidth, we can use smaller on-chip memory. The extreme case is that when we have sufficiently large off-chip bandwidth, we do not need any on-chip memory for data reuse. The different pairs of off-chip bandwidth occupation and on-chip memory usage form the points on a design curve. When we move along the curve, the array references to go through reuse buffers will change. The problem is that conventional memory partitioning does not guarantee an optimal solution for an arbitrary access pattern. Optimality of memory partitioning may get lost when we perform bandwidth/memory trade-off. One of the benefits of our microarchitecture is that it will automatically reduce the buffer size to the minimum when the off-chip bandwidth is increased, but keep the optimal design structure. As shown in the example of Fig. 5.10, when we allow one more off-chip access for each clock cycle, we will pick up the largest reuse buffer in our microarchitecture in Fig. 5.10(a), and replace it with the input data stream from off-chip in Fig. 5.10(b). By doing so, we achieve a graceful degradation of on-chip memory usage with the increase of off-chip bandwidth, as shown in Fig. 5.11 We use the example in Fig. 5.10 and the 19-point stencil in SEGMENTATION in Fig. 5.4(c), and sweep the number of off-chip accesses per cycle from 1 up to the number of stencil accesses in the two applications along the y-axis. The three phases in Fig. 5.11 show that we first give up data reuse across the image planes which need large buffers,

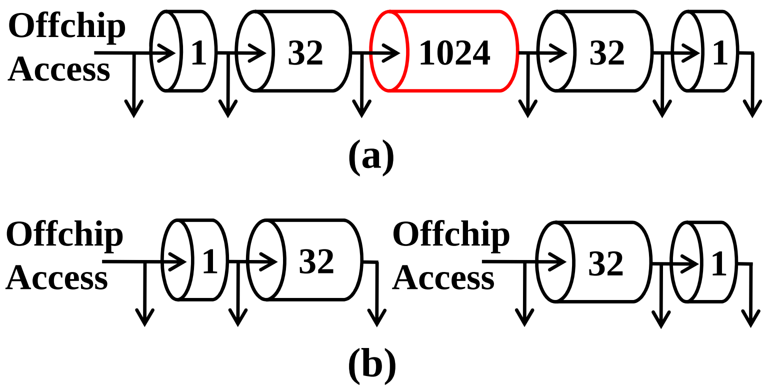


Figure 5.10: Breaking the chain at the largest reuse buffer when the number of off-chip accesses is increased. (a) An example of the original microarchitecture with one off-chip access per cycle. (b) The new microarchitecture with two off-chip accesses per cycle, but smaller total buffer size.

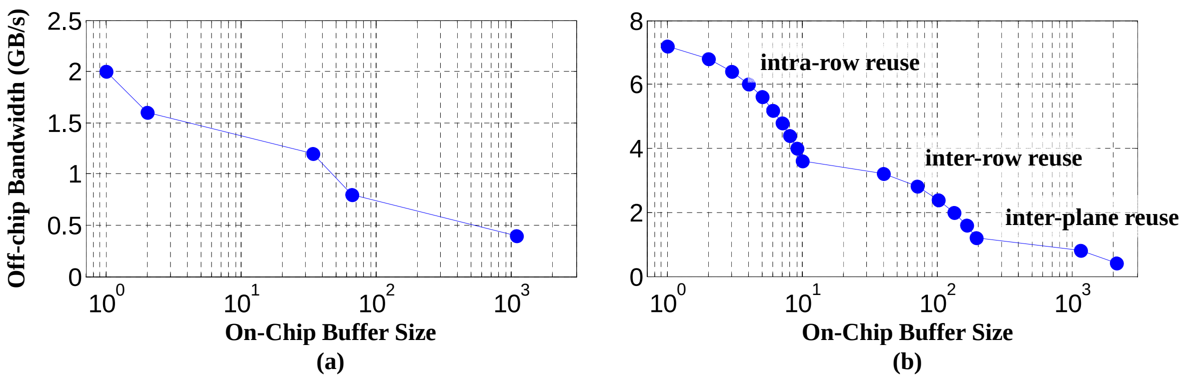
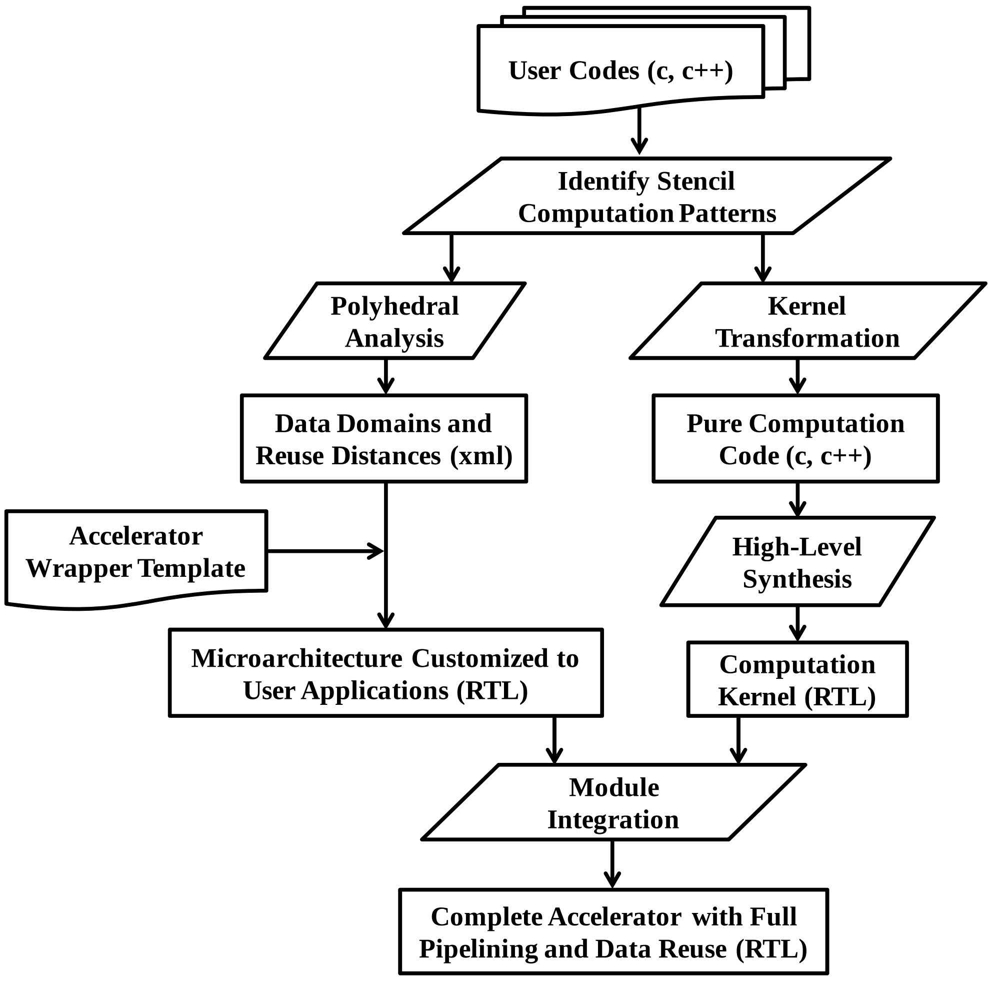


Figure 5.11: Graceful degradation of on-chip memory usage with the increase of off-chip bandwidth for (a) the example in Fig. 5.10, and (b) the 19-point stencil in SEGMENTATION in Fig. 5.4(c).

then give up data reuse across image rows which need medium buffers, and finally give up data reuse within an image row which needs small buffers.

## 5.4 Design Automation Flow

We develop a design automation flow to generate the complete accelerator for a given stencil application, as shown in Fig. 5.12. It starts from the original source codes of a user application, e.g.,



User Codes (c, c++)
Identify Stencil Computation Patterns
Polyhedral Analysis
Kernel Transformation
Data Domains and Reuse Distances (xml)
Pure Computation Code (c, c++)
Accelerator Wrapper Template
High-Level Synthesis
Microarchitecture Customized to User Applications (RTL)
Computation Kernel (RTL)
Module Integration
Complete Accelerator with Full Pipelining and Data Reuse (RTL)

Figure 5.12: Design automation flow of accelerator generation for stencil computation.

the code in Listing 5.1. We first analyze the data access patterns in the application and use Definition 6 to identify the stencil pattern. If there exists a part of computation that matches the stencil pattern, the flow will continue to optimize. In the left branch, we first apply polyhedral analysis to extract the polyhedrons of data arrays with stencil accesses. We calculate the data domain of each array reference and the reuse distance of each pair of adjacent array references. This information is used to instantiate the data filters and reuse FIFOs in our microarchitecture. Then the flow generates a microarchitecture instance, e.g., the design in Fig. 5.5, with the memory systems optimized for stencil accesses in user applications. In the right branch, we first apply source-to-source code transformation to extract the kernel code with pure computation, e.g., Listing 5.2. Then high-level synthesis is applied on the transformed code for a fully pipelined hardware implementation of the computation kernel in RTL. Finally, we integrate the microarchitecture with the computation kernel for a complete accelerator with full pipelining and data reuse, e.g., the design in Fig. 5.2. This flow is further integrated into CMOST, an automated compilation tool from C-code to executable FPGA accelerators [135].

## 5.5 Experiments

### 5.5.1 Experiment Setup

Our polyhedral analysis in Fig. 5.12 is implemented by the LLVM-Polly framework [136]. The kernel transformation is performed by the open source compiler infrastructure ROSE [137], and the high-level synthesis is performed by Xilinx Vivado HLS [72]. Although our methodology is applicable to both ASIC and FPGA designs, we choose FPGA as the target device in this work due to the availability of downstream behavioral synthesis and implementation tools. The Xilinx Virtex7 FPGA XC7VX485T and ISE 14.2 tool suite [3] are used in our experiments. The target clock frequency is set at 200MHz.

The benchmarks used in prior memory partitioning work [13, 14] make up a rich set of real-life stencil computation kernels. Among them, we select the more challenging benchmarks with non-rectangular stencil windows for our experiments. DENOISE (2D/3D), RICIAN (2D), and SEGMENTATION (3D) are from medical imaging [19]. BICUBIC (2D) is from bicubic interpolation process [15]. SOBEL (2D) is from Sobel edge detection algorithm [16]. We choose the more recent memory partitioning work [14] as our experiment baseline.

### 5.5.2 Results

| | Original | Target | # of Banks | | Total Size | |
|---|---|---|---|---|---|---|
| | *II* | *II* | [14] | Ours | [14] | Ours |
| DENOISE | 5 | 1 | 5 | 4 | 2050 | 2048 |
| RICIAN | 4 | 1 | 5 | 3 | 2050 | 2048 |
| SOBEL | 9 | 1 | 9 | 8 | 2054 | 2050 |
| BICUBIC | 4 | 1 | 5 | 3 | 2050 | 2048 |
| DENOISE_3D | 7 | 1 | 7 | 6 | 2240 | 2048 |
| SEGMENTATION | 19 | 1 | 20 | 18 | 2630 | 2112 |

Table 5.4: High-level partitioning results.

The comparison results of memory partitioning are shown in Table 5.4. We list the pipeline *II* of the original user codes which suffer memory port contentions before memory partitioning, which is equal to the number of memory load operations on the data array. We also list the *II* that the computation kernel targets to achieve via memory partitioning. The number and total size of reuse buffer banks are reported for both [14] and our method. The buffer size is in the unit of data element. As shown in Tabel 5.4, our method saves the partitioning bank number of all of the six benchmarks. In addition, our method does not need the padding technique in [14] which increases the grid size at certain dimensions to relax the partitioning complexity. Our methods saves the buffer size, especially when the padding introduces more overhead in a high-dimensional data grid, e.g., Fig. 5.4(c).

The post-synthesis results are listed in Table 5.5. Physical resource usage (block RAMs, logic slices, and DSPs) and timing information are extracted from Xilinx ISE report. As shown in Table 5.5, we use 66% fewer block RAMs than [14]. This stems from 1) the minimum number

| | | BRAM | Slice | DSP | CP (ns) |
|---|---|---|---|---|---|
| DENOISE | [14] | 5 | 703 | 5 | 4.502 |
| | ours | 2 | 636 | 0 | 4.519 |
| | **comp.(%)** | **-60** | **-9.5** | **-100** | 0.37 |
| RICIAN | [14] | 5 | 582 | 4 | 4.472 |
| | ours | 2 | 544 | 0 | 4.337 |
| | **comp.(%)** | **-60** | **-6.5** | **-100** | **-3.02** |
| SOBEL | [14] | 9 | 1937 | 9 | 4.416 |
| | ours | 2 | 1088 | 0 | 4.239 |
| | **comp.(%)** | **-78** | **-43.8** | **-100** | **-4.01** |
| BICUBIC | [14] | 5 | 535 | 4 | 4.309 |
| | ours | 2 | 493 | 0 | 4.196 |
| | **comp.(%)** | **-40** | **-7.8** | **-100** | **-2.62** |
| DENOISE_3D | [14] | 7 | 980 | 7 | 4.656 |
| | ours | 2 | 859 | 0 | 4.762 |
| | **comp.(%)** | **-71** | **-12.3** | **-100** | 2.27 |
| SEGMENTATION_3D | [14] | 20 | 7533 | 19 | 4.995 |
| | ours | 2 | 2251 | 0 | 4.985 |
| | **comp.(%)** | **-90** | **-70.1** | **-100** | **-0.2** |
| **Average(%)** | | **-66** | **-25** | **-100** | **-1.2** |

Table 5.5: Synthesis experimental results.

of buffer banks achieved, and 2) the heterogeneous mapping of buffer banks to variable resources in addition to block RAMs as demonstrated in Table 5.2. We also use 25% fewer logic slices than [14], even though we implement some of the small reuse buffers in registers. That is because we avoid the modulo scheduling in conventional uniform memory partitioning which generates a hardware transformer to map the original data address to the bank ID and local address via a complex calculation involving multiplication and division. Instead our memory system only needs counters iterating over the data domains in the lexicographic order. This advantage is also reflected by the complete elimination of DPSs in our method. The clock period does not show too much difference between [14] and our method since the back-end flow will stop optimization as long as it meets the 200MHz target. However, our method generally has larger slacks from the target 5.0ns as shown in Table 5.5. It is mainly due to the distributed structure in our method. We tried to use the Xilinx XPower Analyzer for power estimation, but found that the FPGA power is dominated by the static power, and is almost invariant with custom circuits. If power gating is available in FPGA, the FPGA power will be proportional to resource usage, which is covered by Table 5.5.

## 5.6 Conclusions and Future Work

In this chapter, we propose non-uniform partitioning which opens a new design space compared to conventional cyclic partitioning framework. As a starting point, we use stencil computation as the initial design target and show a novel memory system that works with non-uniform sizes of reuse buffer. The memory system in the extended design space can achieve the optimal solution with the minimum reuse buffer size and the minimum number of buffer banks. We develop a design automation flow that generates a microarchitecture with our memory systems and integrates it with the computation kernel for a complete design. Experimental results show that our method outperforms the recent memory partitioning work in terms of utilization of variable FPGA resources.

As this is the first work on non-uniform memory partitioning, our primary goal is to show the potential of this new approach. Though stencil computation is a popular application domain and attracts the attention of most memory partitioning work, it is still important to extend non-uniform memory partitioning to general cases. Our data streaming method may not be the only solution for utilizing the non-uniform reuse buffers. A modified modulo scheduling extended from conventional uniform memory partitioning is also a good candidate. We believe that there are many opportunities in future research.

# CHAPTER 6

# Computation and Communication Optimization in Convolutional Neural Networks for Big Data

## 6.1 Introduction

While the previous chapters have gone through the communication optimization within a chip, we focus on the new challenges in big data in this chapter. As the capability of data collection continuously increases, the data processing and analytics relies more on machine learning to automatically mine out more insights from the big data. Biologically inspired deep learning based on artificial neural networks (ANNs) has gained its popularity in recent years since the network complexity can be easily scaled up with the increasing data volume to capture a better data model. Among the different kinds of ANNs, convolutional neural networks (CNNs) have achieved good success, particularly in computer vision applications, e.g., the recognition of handwritten digits [74, 75], and the detection of faces [76, 77]. In the 2012 ImageNet contest [78], a CNN-based approach named SuperVision [17] outperformed all the other traditional image recognition algorithms. On one hand, CNNs keep the advantage of artificial neural networks which use a massive network of neurons and synapses to automatically extract features from data. On the other hand, CNNs further customize their synapse topologies for computer vision applications to exploit the feature locality in image data.

The success of CNNs promises wide use for many future platforms to recognize images, e.g., micro-robots, portable devices, and image search engines in data centers. It will be beneficial to improve the implementation of the CNN algorithm to reduce the computational cost. One direction is to improve the CNN algorithm using hardware accelerators, e.g., GPUs and field-programmable gate arrays (FPGAs) [79, 80, 81]. Another orthogonal direction is to reduce the theoretical number of basic operations needed in the CNN computation from the algorithmic aspect, as will be discussed in this thesis.

In this chapter we first reveal the linear algebraic properties in the CNN computation. Based on these properties, we invent an efficient algorithm that can be applied to generic CNN architectures to reduce the arithmetic operations without any penalty on the image recognition quality or hardware cost. Furthermore, we develop a communication-avoidance algorithm to minimize the disk access.

## 6.2 Background

### 6.2.1 Algorithm Review of CNNs

Convolutional neural networks (CNNs) were extended from artificial neural networks (ANNs) and customized for computer vision [74]. An example of a CNN is given in Fig. 6.1. As shown in

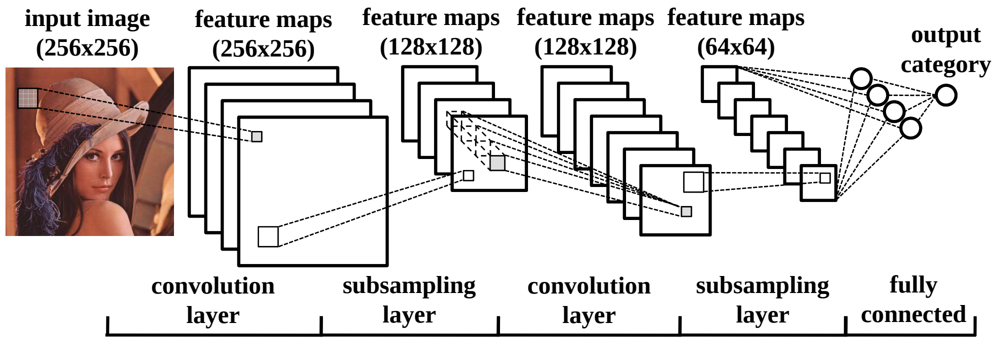


Figure 6.1: An example of a convolutional neural network.

this figure, the intermediate results in a CNN are different sets of feature maps. The main working principle of a CNN is to gradually extract local features from feature maps of higher resolutions, and then to combine these features into more abstract feature maps of lower resolutions. This is realized by the two alternating types of layers in a CNN: convolution layers and subsampling layers. The last few layers in the CNN still use fully connected ANN classifiers to produce the abstracted classification results. The detailed computation patterns of different layers in the CNN are described as below:

*Convolution Layer.* In this layer, features, such as edges, corners, and crossings, are extracted from the input feature maps via different convolution kernels, and are combined into more abstract output feature maps. Assume there are $Q$ input feature maps and $R$ output feature maps, and the feature map size is $M \times N$. Also assume the convolution kernel size is $K \times L$. Then the

```
for(r=0; r<R; r++)              //output feature map
  for(q=0; q<Q; q++)            //input feature map
    for(m=0; m<M; m++)          //row in feature map
      for(n=0; n<N; n++)        //column in feature map
        for(k=0; k<K; k++)      //row in convolution kernel
          for(l=0; l<L; l++)    //column in convolution kernel
            Y[r][m][n]+=W[r][q][k][l]*X[q][m+k][n+l];
```

Figure 6.2: Example loop-nest representing the computation in a convolution layer of a CNN.

computation in the convolution layer can be represented in a nested-loop description, as shown in Fig. 6.2. The array $X$ contains the input feature maps, and the array $Y$ contains the output feature maps which are initialized to zeros. The array $W$ contains the weights in the convolution kernels. To regularize the computation pattern, we do not explicitly add the network bias to the output feature maps. Instead, we put a dummy input feature map of all 1's in array $X$ and put the bias on the weights associated with this dummy input map in array $W$. The computational workload in the convolution layer is in the order of $O(R \cdot Q \cdot M \cdot N \cdot K \cdot L)$.

*Subsampling Layer.* The purpose of this layer is to achieve spatial invariance by reducing the resolution of feature maps. In the example of Fig. 6.1, each feature map is scaled down by a subsample factor $2 \times 2$. The computational workload in this layer is in the order of $O(Q \cdot M \cdot N)$, which is much smaller than that in the convolution layer.

At the output of each layer, an activation function is further applied to each pixel in the feature maps to mimic the neuron activation.

### 6.2.2 Architecture of Real-Life CNN

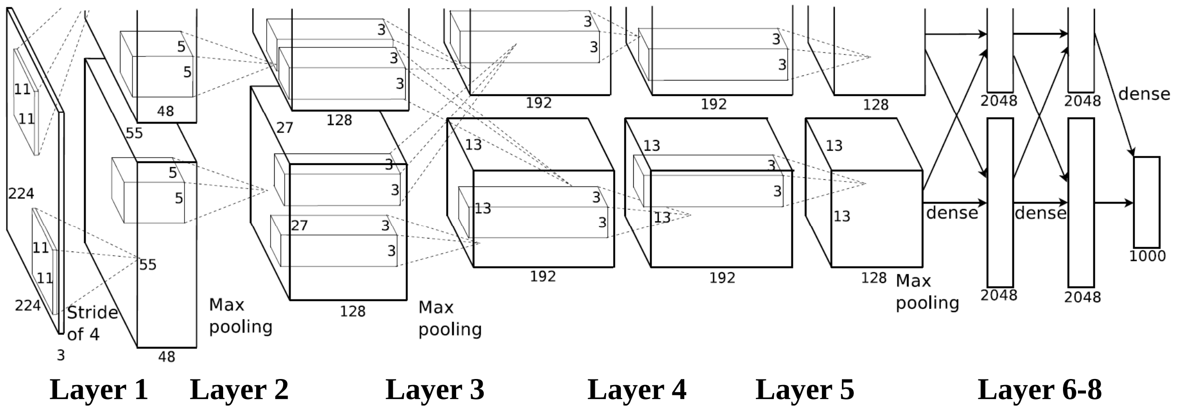


Figure 6.3: A real-life CNN that was used in the 2012 ImageNet contest [17].

The architecture of a real-life CNN that was used in the 2012 ImageNet contest [17] is shown in Fig. 6.3. It consists of eight layers. The first layer contains three $224 \times 224$ input images that are obtained from the original $256 \times 256$ image via data augmentation. The 1,000 neurons in the last layer report the likelihoods of the 1,000 categories that the input image might belong to. Layer 2 contains 96 feature maps, and each feature map is sized $55 \times 55$. They are partitioned into two sets, each containing 48 feature maps, so as to fit into two GPUs used in [17]. The other layers also follow notations similar to Fig. 6.3. Note that the convolution layer and the subsampling layer are merged together in Layers 1, 2, 3, and 6 of this architecture. There are no subsampling layers but only convolution layers in the other layers. The convolution kernel size is 11 in Layer 1, 5 in Layer 2, and 3 in the other layers. The default subsample factor is 2, except for a factor of 4 in Layer 1. The subsampling operations are not trainable in this architecture, but the function $\max$ is applied to the $2 \times 2$ or $4 \times 4$ pixel windows in each feature map, and is marked as max pooling in Fig. 6.3. The activation function is simplified to the Rectified Linear Unit (ReLU) function, $\max(0, x)$, as discussed in [17]. In Layer 2, 4, and 5, to avoid the inter-GPU communication, the features extracted from the two partitioned sets of input feature maps are not combined together at the output. The design choice of removing this combination is made by trial and error, and proves effective in [17].

### 6.2.3 Runtime Breakdown of Real-Life CNN

To better understand the time-consuming part of the process of image recognition via a CNN, we reimplement the CNN in Fig. 6.3 in a single-thread CPU so that the workload can be measured by runtime. A breakdown of runtime is given in Table 6.1. We see that the runtime is dominated by the convolution layers. The main focus of this work is to optimize the computation in the convolution layers.

| | Convolution Layer | Subsampling Layer | ReLU Activation | Fully Connected ANN |
|---|---|---|---|---|
| Layer 1 | 364s | 720s | 1.83s | - |
| Layer 2 | 1728s | 416s | 1.19s | - |
| Layer 3 | 5710s | 147s | 0.42s | - |
| Layer 4 | 2564s | - | 0.42s | - |
| Layer 5 | 2652s | - | 0.27s | - |
| Layer 6 | - | 29s | 0.12s | 8.60s |
| Layer 7 | - | - | 0.12s | 5.63s |
| Layer 8 | - | - | 0.03s | 1.78s |
| Total | 13018s | 1313s | 4.41s | 16.0s |
| Breakdown | 90.7% | 9.15% | 0.03% | 0.11% |

Table 6.1: Breakdown of CNN runtime in the recognition of 256 images.

## 6.3 Properties of CNN Computation

### 6.3.1 Another View of CNN Computation

In this section we offer another view of the CNN computation in Fig. 6.2 that enables optimization of the computational workload. First denote the $Q$ input feature maps as $x_1, x_2, ...x_Q$, and the $R$ output feature maps as $y_1, y_2, ..., y_R$. Also denote the $R \times Q$ convolution kernels (each sized $K \times L$) as $w_{rq}$ where $r = 1, 2, ..., R$ and $q = 1, 2, ..., Q$. We further denote the convolution operation between a kernel $w_{rq}$ and a feature map $x_q$ as

$$w_{rq} * x_q = z, \text{ where } z(m,n) = \sum_{k=0}^{K-1}\sum_{l=0}^{L-1} w_{rq}(k,l)x_q(m+k,n+l). \tag{6.1}$$

Here $z$ represents the convolution result in the form of an $M \times N$ image, and $z(m,n)$ represents an image pixel in $z$. Then the computation in Fig. 6.2 can be represented as

$$\begin{aligned}
y_1 &= w_{11} * x_1 + w_{12} * x_2 + \cdots + w_{1Q} * x_Q \\
y_2 &= w_{21} * x_1 + w_{22} * x_2 + \cdots + w_{2Q} * x_Q \\
y_3 &= w_{31} * x_1 + w_{32} * x_2 + \cdots + w_{3Q} * x_Q \quad . \\
&\cdots \\
y_R &= w_{R1} * x_1 + w_{R2} * x_2 + \cdots + w_{RQ} * x_Q
\end{aligned} \tag{6.2}$$

If we reorganize these $x_q$, $y_r$ and $w_{rq}$ in the form of column vectors and matrices

$$\overrightarrow{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_Q \end{pmatrix}, \ \overrightarrow{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_R \end{pmatrix}, \ W = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1Q} \\ w_{21} & w_{22} & \cdots & w_{2Q} \\ \vdots & \vdots & \ddots & \vdots \\ w_{R1} & w_{R2} & \cdots & w_{RQ} \end{pmatrix},$$

then the computation in Eq. (6.2) can be redefined as a special matrix/vector multiplication

$$\overrightarrow{y} = W \times \overrightarrow{x}.$$

Each element in the left operand $W$ is a convolution kernel. Each element in the right operand $\overrightarrow{x}$ is an input feature map. Each element in the result $\overrightarrow{y}$ is an output feature map. The element-wise multiplication is redefined as the convolution between a kernel $w_{rq}$ and a feature map $x_q$ in Eq. (6.1). If a web server provider receives a batch of $P$ images to recognize, we will have $\overrightarrow{x}_1, \overrightarrow{x}_2, \cdots, \overrightarrow{x}_P$ as parallel inputs, and $\overrightarrow{y}_1, \overrightarrow{y}_2, \cdots, \overrightarrow{y}_P$ as parallel outputs. Their computation can be merged as

$$(\overrightarrow{y}_1, \overrightarrow{y}_2, \cdots, \overrightarrow{y}_P) = (W \times \overrightarrow{x}_1, W \times \overrightarrow{x}_2, \cdots, W \times \overrightarrow{x}_P) = W \times (\overrightarrow{x}_1, \overrightarrow{x}_2, \cdots, \overrightarrow{x}_P).$$

This can be further simplified to a matrix multiplication

$$Y = W \times X,$$

where

$$X = (\overrightarrow{x}_1, \overrightarrow{x}_2, \cdots, \overrightarrow{x}_P),\ Y = (\overrightarrow{y}_1, \overrightarrow{y}_2, \cdots, \overrightarrow{y}_P).$$

Both the left operand $X$ and the result $Y$ are matrices of feature maps. This matrix multiplication representation provides a new view of the computation in convolution layers of CNNs. We name this representation convolutional matrix multiplication (convolutional MM).

### 6.3.2 Enabling New Optimization Opportunities

We can optimize the computation of convolutional MM by revisiting techniques that have been built for normal matrix multiplication (normal MM). For example, we can use the classical Strassen algorithm [138] to reduce the computational workload. In each recursion of matrix partitioning, the Strassen algorithm can reduce the number of multiplications by 1/8, but it incurs many extra additions. Note that in normal MM, the element-wise multiplication is a multiplication between two numbers, while in our convolutional MM, the element-wise multiplication is redefined as the convolution between a kernel and a feature map, which has a sufficiently high complexity to make the extra additions negligible. Our convolutional MM is expected to experience more benefits from the Strassen algorithm than the normal MM; this will be discussed in Section 6.4.

### 6.3.3 Properties of Convolutional MM

Before we go through any optimization, we first identify the properties of our convolutional MM. If the addition of two convolution kernel matrices $W_1$ and $W_2$ of the same size are intuitively defined as the additions of all the pairs of weights at the same positions, i.e.,

$$W_1 + W_2 = W_3,\ \text{where } w_{3_{rq}}(k,l) = w_{1_{rq}}(k,l) + w_{2_{rq}}(k,l),$$

combined with the linearity of the operation defined in Eq. (6.1), we have

$$(W_1 + W_2) \times X = W_1 \times X + W_2 \times X. \tag{6.3}$$

Similarly, if the addition of two feature map matrices $X1$ and $X2$ of the same size are intuitively defined as additions of all the pairs of pixels at the same positions, we have

$$W \times (X_1 + X_2) = W \times X_1 + W \times X_2. \tag{6.4}$$

## 6.4 Computation Optimization

### 6.4.1 Algorithm

In this section we show how to extend the Strassen algorithm [138] from normal MM to reduce the computational workload of our convolutional MM. We start from

$$Y = W \times X,$$

where the number of elements in both rows and columns of $Y, W, X$ is assumed to be even. We partition $W$, $X$ and $Y$ into equally sized block matrices

$$W = \begin{pmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{pmatrix}, \; X = \begin{pmatrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{pmatrix}, \; Y = \begin{pmatrix} Y_{1,1} & Y_{1,2} \\ Y_{2,1} & Y_{2,2} \end{pmatrix}.$$

Then we have

$$\begin{aligned} Y_{1,1} &= W_{1,1} \times X_{1,1} + W_{1,2} \times X_{2,1} \\ Y_{1,2} &= W_{1,1} \times X_{1,2} + W_{1,2} \times X_{2,2} \\ Y_{2,1} &= W_{2,1} \times X_{1,1} + W_{2,2} \times X_{2,1} \\ Y_{2,2} &= W_{2,1} \times X_{1,2} + W_{2,2} \times X_{2,2} \end{aligned}. \tag{6.5}$$

Here, we still need 8 multiplications, the same number that we need in matrix multiplication before partitioning. We define new matrices

$$\begin{aligned}
M_1 &:= (W_{1,1} + W_{2,2}) \times (X_{1,1} + X_{2,2}) \\
M_2 &:= (W_{2,1} + W_{2,2}) \times X_{1,1} \\
M_3 &:= W_{1,1} \times (X_{1,2} - X_{2,2}) \\
M_4 &:= W_{2,2} \times (X_{2,1} - X_{1,1}) \\
M_5 &:= (W_{1,1} + W_{1,2}) \times X_{2,2} \\
M_6 &:= (W_{2,1} - W_{1,1}) \times (X_{1,1} + X_{1,2}) \\
M_7 &:= (W_{1,2} - W_{2,2}) \times (X_{2,1} + X_{2,2})
\end{aligned} \quad . \tag{6.6}$$

Followed by the properties in Eq. (6.3) and Eq. (6.4), we can compute the result of the matrix multiplication from the 7 multiplications in Eq. (6.6) as follows:

$$\begin{aligned}
Y_{1,1} &= M_1 + M_4 - M_5 + M_7 \\
Y_{1,2} &= M_3 + M_5 \\
Y_{2,1} &= M_2 + M_4 \\
Y_{2,2} &= M_1 - M_2 + M_3 + M_6.
\end{aligned}$$

Here, we reduce the number of the redefined multiplications from 8 to 7 without changing the computation results. We can iterate this matrix partitioning process recursively until the submatrices degenerate into basic elements, i.e., separate convolution kernels and feature maps in our convolutional MM. We can see that each recursion will reduce the number of multiplications by 1/8, but will incur 18 additions on the submatrices. In the normal MM, all the elements are numbers, and either multiplications or additions are performed between these numbers. The overhead of 18 additions could completely eliminate the benefits brought by the multiplication savings in normal MMs. In our convolutional MM, however, the element-wise multiplication is redefined as the convolution between a kernel and a feature map in Eq. (6.1). Suppose the convolution kernel size is $K \times L$, and the feature map size is $M \times N$. As shown in Table 6.2, the number of FLOPS (floating-point operations) in an element-wise multiplication will be $2K \cdot L \cdot M \cdot N$, which is much

|  | element-wise addition | element-wise multiplication |
|---|---|---|
| the normal MM | 1 | 1 |
| our convolutional MM | $K \cdot L$ or $M \cdot N$ | $2K \cdot L \cdot M \cdot N$ |

Table 6.2: The FLOPS comparison of different operations in the normal MM and our convolutional MM.

larger than either $K \cdot L$ FLOPS in a kernel addition or $M \cdot N$ FLOPS in a feature map addition. This makes the reduction of the number of multiplications very meaningful to our convolutional MM.

### 6.4.2 Experimental Results

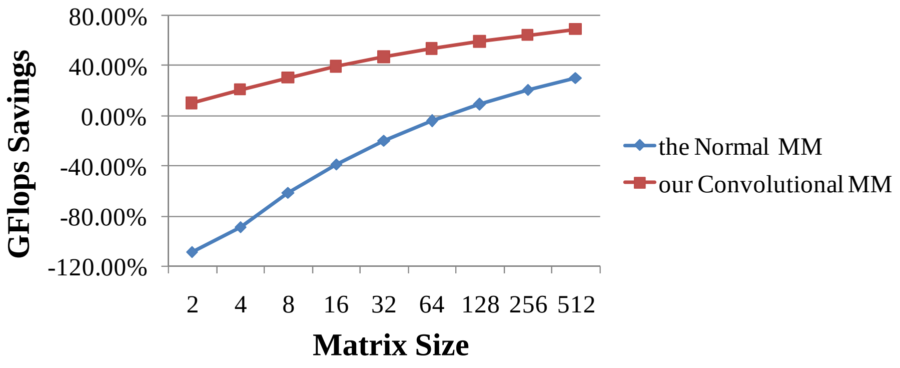


Figure 6.4: Comparison of our convolutional MM with the normal MM in terms of the savings of GFLOPS by the Strassen algorithm.

A comparison of our convolutional MM with the normal MM in terms of the savings of GFLOPS by the Strassen algorithm is shown in Fig. 6.4. We use the convolution kernel size $5 \times 5$ and the feature map size $55 \times 55$ in Layer 2 of Fig. 6.3, and sweep different square sizes for matrices $W, X, Y$ in this experiment. We see that for the normal MM, the Strassen algorithm may not bring benefits, but could lead to a $>100\%$ overhead, especially when the matrix size is small. In [139], even if the nested loops in Fig. 6.2 are unrolled and the computation is represented in a normal MM to make the Strassen algorithm applicable, the Strassen algorithm still brings relatively few benefits. But if the computation is represented in our convolutional MM, many more benefits can be achieved due to the redefined granularities of matrix elements and element-wise multiplications.

A sensitivity study on the convolution kernel size and the feature map size is provided in Fig. 6.5. Here, we fix the matrix size to 256. As shown in Fig. 6.5(a), the convolution kernel

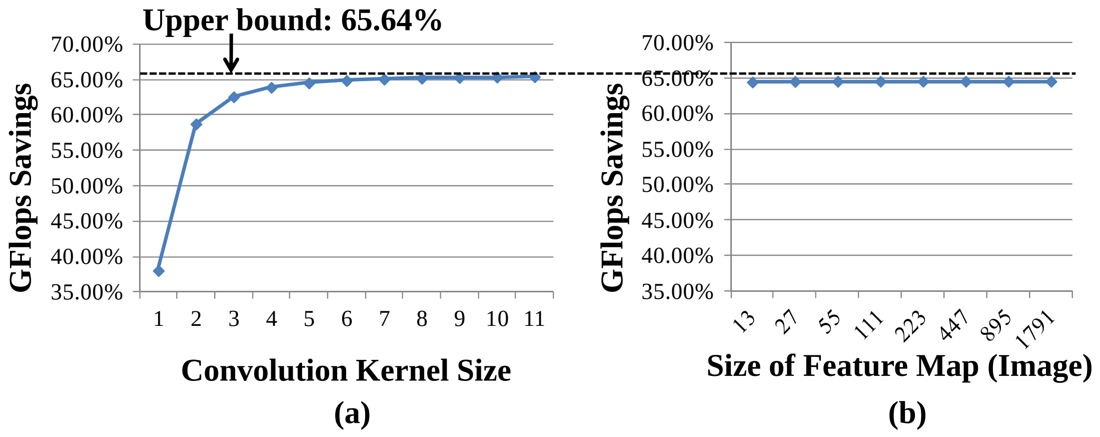


Figure 6.5: A sensitivity study on convolution kernel size and feature map size.

size has a high impact on GFLOPS savings. This matches the analysis that the FLOPS difference between the element-wise multiplication and the element-wise addition of our convolutional MM is proportional to the kernel size (assume the feature map size is much larger than the kernel size). Since the matrix size is limited to 256, there could be at most eight recursions of the Strassen algorithm, which imposes an upper bound of $1 - (7/8)^8 = 65.64\%$ on the GFLOPS savings. Fig. 6.5(a) shows that we approach this upper bound as the convolution kernel size increases. The GFLOPS savings are invariant with the increase of the feature map size, as shown in Fig. 6.5(b), since the computational workloads of both the element-wise multiplication and addition will increase.

We reimplement the real-life CNN in Fig. 6.3 and apply the Strassen algorithm to reduce the computational workload. Experimental results are listed in Table 6.3. As shown in this table, no matrices are square, and no matrices have sizes equal to the power of 2. To deal with the non-square matrices, we stop matrix partitioning once either the row size or the column size becomes small. To solve the not-the-power-of-2 problem, we pad a dummy row or column in the matrices once we

| | | Layer 1 | Layer 2 | Layer 3 | Layer 4 | Layer 5 |
|---|---|---|---|---|---|---|
| Matrix Parameters | $Q$ | 3 | 48 | 256 | 192 | 192 |
| | $R$ | 96 | 128 | 384 | 192 | 128 |
| | $K, L$ | 11 | 5 | 3 | 3 | 3 |
| | $M, N$ | 224 | 55 | 27 | 13 | 13 |
| Runtime | original | 364s | 865s | 5710s | 1282s | 1326s |
| | our optimization | 433s | 864s | 3863s | 683s | 998s |
| | savings | -18% | 27% | 32% | 47% | 24% |

Table 6.3: Workload reduction by extending the Strassen algorithm to the real-life CNN in Fig. 6.3 via our convolutional MM.

encounter an odd number of rows or columns during matrix partitioning. Note that the Strassen algorithm is based on recursive matrix partitioning, which is a cache-oblivious algorithm that can take advantage of a CPU cache without knowing the cache size. For the sake of fairness, we also implement the baseline matrix multiplication in a cache-oblivious algorithm in Eq. (6.5). The hardware platform is a Xeon server with CPUs running at 2GHz. We limit the number of threads initialized by our convolutional MM computation to 1 since we are measuring the reduction of total workloads by the runtime. Table 6.3 shows that we can get up to a $47\%$ savings in certain convolution layers. Note that this gain is achieved without any change in image recognition results or the addition of any hardware accelerators.

## 6.5 Communication Optimization

### 6.5.1 Algorithm

In this section we show how to minimize the disk access. Again, we start from

$$Y = W \times X.$$

When a large CNN is applied to big data, the main memory cannot hold all the data in the matrices $Y$, $W$ and $X$. We can load only a part of it from the disk to the main memory each time, as shown in Fig. 6.6. And we want to maximize the computational workload that can be completed by this

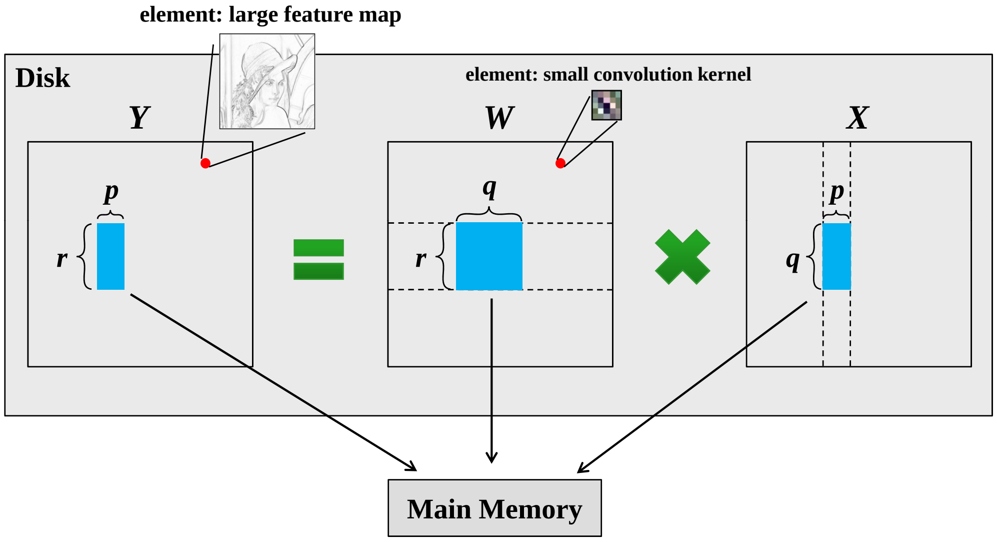


Figure 6.6: Partitioning of large matrices into smaller data blocks that can be loaded in the main memory for computation.

part of data. We use the number of element-wise multiplications $|V|$ to measure the computation workload. Given the matrix sizes $R$, $Q$ and $P$ of the convolutional MM for a convolution layer, the total workload is $|V| = R \cdot Q \cdot P$. Suppose the main memory to be a size of $C$ bytes which hold $|V_W|$ elements in $W$, $|V_X|$ elements in $X$, and $|V_Y|$ elements in $Y$. According to the discrete Loomis-Whitney inequality [140], we know that

$$|V|^2 \leq |V_W| \cdot |V_X| \cdot |V_Y|$$

This upper bound is attained when the computation covered by the partial data is a block matrix multiplication, as indicated by the blue data blocks in Fig. 6.6. Let's say we load an $r \times q$ submatrix from $W$, a $q \times p$ submatrix from $X$, and an $r \times p$ submatrix from $Y$. Then we have,

$$|V| = rqp = \sqrt{rq \cdot qp \cdot rp} = \sqrt{|V_W| \cdot |V_X| \cdot |V_Y|}$$

The problem that remains is to choose the optimal settings of $(r, q, p)$ to maximize $|V|$ for a given $C$. This problem can be formulated as:

$$\begin{aligned}
&\max_{r,q,p} r \cdot q \cdot p \\
&\text{s.t. } rqKL + qpMN + rpMN \leq C
\end{aligned}$$

where $KL$ is the data size of an element in the convolution kernel matrix $W$, and $MN$ is the data size of an element in the feature map matrix $X$ and $Y$. In the normal MM, the data sizes of all the elements are equal, which leads to the optimal solution $m = n = r$. So in the past, when we divided the computation into the block matrix multiplication to fit data in the main memory, we usually used uniform submatrices. However in our convolutional MM, the sizes of elements from $W$, $X$ and $Y$ are asymmetric, which breaks the balance among $r$, $q$ and $p$ in the original optimal solution. Instead, we know that

$$\begin{aligned}
& r \cdot q \cdot p \\
= \; & \frac{1}{MN\sqrt{KL}} \sqrt{rqKL \cdot qpMN \cdot rpMN} \\
\leq \; & \frac{1}{MN\sqrt{KL}} \sqrt{\left(\frac{rqKL + qpMN + rpMN}{3}\right)^3} \\
\leq \; & \frac{1}{MN\sqrt{KL}} \sqrt{(\frac{C}{3})^3}
\end{aligned}$$

This upper bound is attained when

$$r = q = \sqrt{\frac{C}{3KL}}, p = \frac{1}{MN}\sqrt{\frac{KLC}{3}}$$

In most cases, $MN$ is larger than $KL$, which means that $r$ and $q$ are larger than $p$, as shown in Fig. 6.6. The underlying insight is that a communication-avoid algorithm for our convolutional MM prefers large submatrices from $W$ and small submatrices from $X$ and $Y$ since the element size in $W$ is smaller. By using our optimal nonuniform matrix partitioning, we can finish more computation given the limited data volume in the main memory than by using the conventional uniform matrix partitioning. Therefore, the total disk access for CNNs can be reduced.

### 6.5.2 Experimental Results

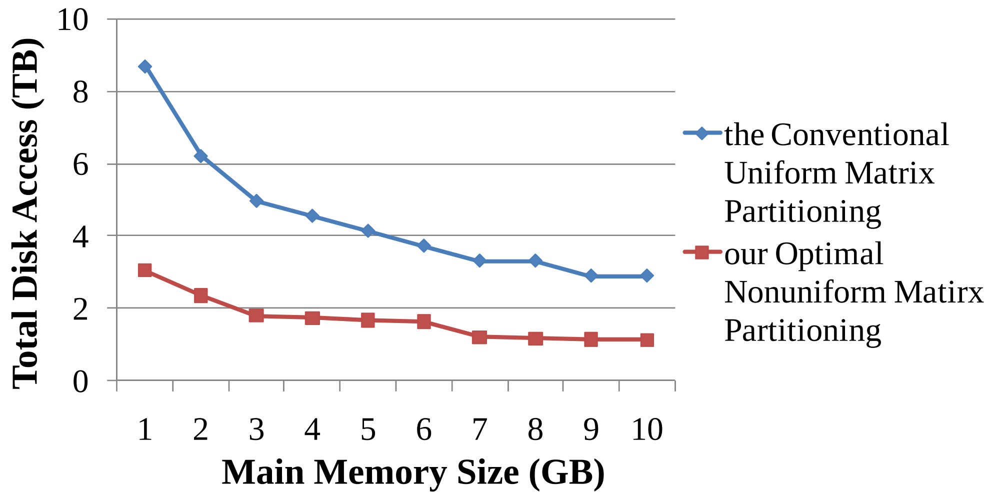


Figure 6.7: A comparison of disk accesses using the conventional uniform matrix partitioning and our optimal nonuniform matrix partitioning for CNNs.

Fig. 6.7 provides a comparison of disk accesses using the conventional uniform matrix partitioning and our optimal nonuniform matrix partitioning based on our convolutional MM. Since the optimization targets a large CNN and big data, we use the settings where $R = Q = P = 4096$. We sweep the main memory size that is allocated for one process of the convolutional MM program from 1GB to 10GB. We measure the total disk access by our convolutional MM, as shown in Fig. 6.7. We can see that our optimal nonuniform matrix partitioning outperforms the conventional method over the whole range of memory sizes. Note that the curve is not always smooth due to the imperfect partitioning when the matrix size cannot be divided by the optimal settings of $(r, q, p)$.

### 6.5.3 Combining with Strassen Algorithm

It is beneficial to combine our communication optimization with the Strassen algorithm that reduces the computational workload. As the recursion which is described in Section 6.4 goes on, the matrix size becomes smaller and smaller. When it is small enough to fit into the main memory, the disk access can be saved in further recursion steps. The challenge is that in the Strassen algorithm, the matrix size is always divided by two in each recursion. If we directly use the Strassen algorithm, no matter which recursion level we reach, the sizes of submatrices from $Y$, $W$ and $X$ are always the same as each other, as shown in Fig. 6.8. This limitation prevents us from optimizing

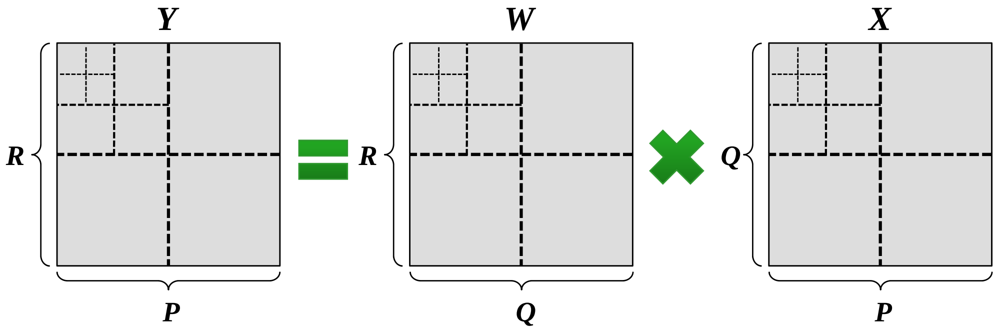


Figure 6.8: The Strassen algorithm before communication optimization. The different dashed lines indicate the matrix partitioning af the different levels of the recursions.

the matrix partitioning $(r, q, p)$ as discussed in Section 6.5.1. Since the ratio among submatrices sizes is kept constant during the process of the Strassen algorithm, we need to optimize the ratio before starting the Strassen algorithm, as shown in Fig. 6.9. As concluded in Section 6.5.1, the

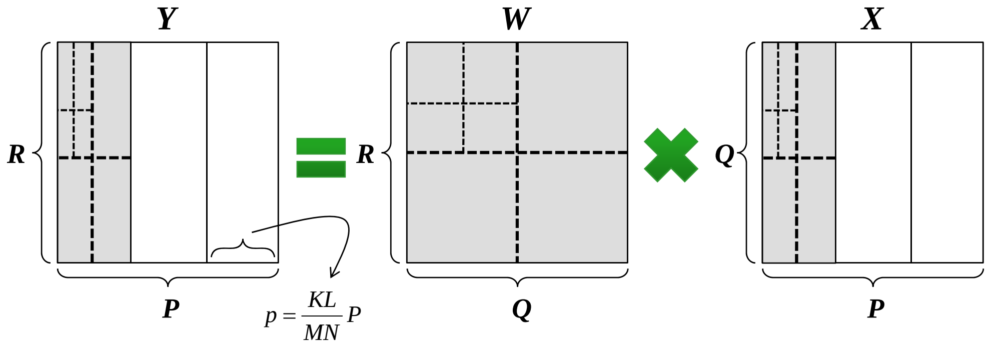


Figure 6.9: The Strassen algorithm after our communication optimization. The solid lines indicate the matrix partitioning to optimize the ratio of matrix sizes.

optimal ratio of $(r, q, p)$ is $r : q : p = 1 : 1 : \frac{KL}{MN}$ so that the submatrices from $Y$ and $X$ are smaller than the one from $W$. Before going into the Strassen algorithm, we can partition the matrices $Y$ and $X$ along the dimension $P$ as shown in Fig. 6.9. Then for each submatrix $Y_i$ and $X_i$ in $Y$ and

$X$, we perform the Strassen algorithm to compute $Y_i = WX_i$ and combine $Y_i$ together for the final result. In the computation of $Y_i = WX_i$ for each $i$, the ratio of $(r, q, p)$ is kept optimal.

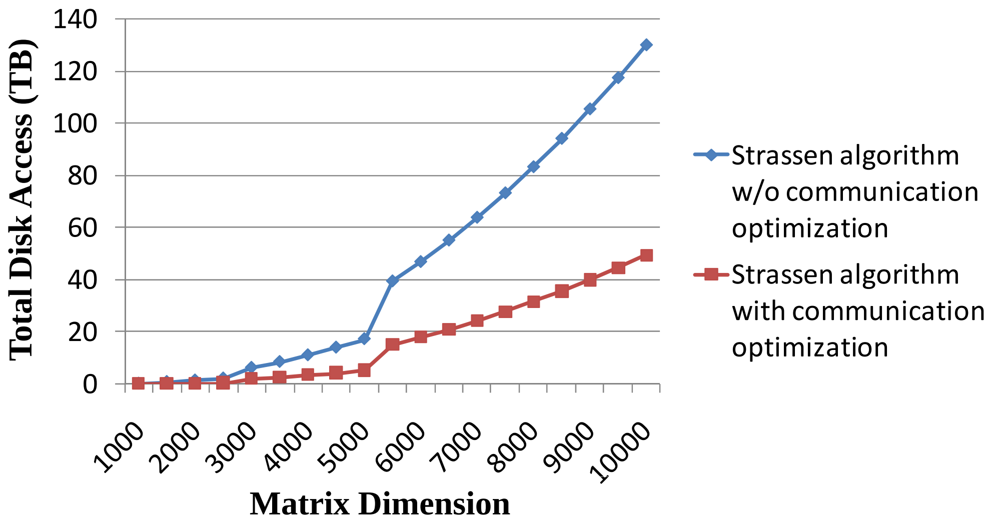


Figure 6.10: A comparison of disk accesses using the Strassen algorithm with and without communication optimization.

Fig. 6.10 provides a comparison of disk accesses using the Strassen algorithm with and without our communication optimization. We set the main memory size to 10GB and sweep the matrix dimension from 1,000 to 10,000. We can see that after we combine our communication optimization with the Strassen algorithm, the total disk access can be reduced over the whole range of matrix sizes. Note that the jumps in the curves are due to the changes of the recursion depth in the Strassen algorithm.

## 6.6 Conclusion

In this chapter the computation in the convolution layers of a CNN is expressed in a new representation—convolutional matrix multiplication (convolutional MM). This representation helps identify the linear algebraic properties of the CNN computation, and enables extension of state-of-art algorithms that have been built for normal matrix multiplication (normal MM) to CNNs to reduce the computational workload and the disk access. This kind of reduction does not change any image recognition results, and does not require any extra hardware accelerators. Our methodology is verified on a real-life CNN. Experimental results show that we can reduce the computation by up to $47\%$ and the communication by up to $64\%$. More well-studied algorithms on linear algebra can be further extended to our convolutional MM to optimize the CNN computation.

# CHAPTER 7

# Accelerator-Centric Cluster Computing with Communication Optimization

## 7.1 Introduction

While the previous chapter discusses optimizing the communication from the application's point of view, this chapter focuses on the optimization in a cluster computing system. As we enter the era of big data, the data volume to be timely processed becomes increasingly larger. Meanwhile, the performance improvement of a single CPU server has slowed down in recent years. To meet the increasing computation demand from big data, there are two major directions in computing solutions. One is horizontal scaling, i.e., add more server nodes to a system. The other is vertical scaling, i.e., improve the performance of a single node using hardware accelerators. There are two popular candidates for hardware accelerators: GPGPUs and FPGAs. In this thesis, we mainly consider FPGAs due to the reasons discussed in Section 7.2.1.

The use of FPGAs in literature [141, 142, 143] is often limited to single-node computation acceleration. In the era of big data, user data are usually partitioned into many data blocks and stored on many servers in a cluster. It is important to exploit parallelism among data blocks by distributing computing tasks to these servers. This capability is missing in single-node solutions of FPGAs. To exploit both cluster parallelism and microarchitecture customization, FPGA-centric cluster computing solutions are a promising trend. Microsoft recently developed such a cluster, where all the servers in its cluster are equipped with an FPGA and can work together to accelerate a single application [83]. However, this development at Microsoft is limited to the single application of Bing search. All the proposed methodologies are coupled with the properties of this search application.

On the other hand, the existing methodologies for the pure-CPU clusters are general enough to cover most big data applications. One popular category among them is the MapReduce framework, e.g., Google MapReduce [84], Apache Hadoop [85] and Spark [86]. These systems accept the parallel programming model MapReduce as user inputs. The MapReduce model allows users to express their computation in terms of a *Map* function that processes all the data blocks in parallel, and a *Reduce* function that merges all the mapped outputs with the same keys to produce a final result. This parallel programming model has even been used to synthesize FPGA accelerator designs by the compilation flows in [87, 88]. These flows map the program parallelism which is explicitly expressed in the MapReduce programming model to the hardware duplication of the customizable pipelines in FPGAs. However, their scope is still limited to single-node FPGA solutions. An enhanced compilation flow in [89] further supports partitioning of jobs in users' MapReduce programs among the FPGAs, GPUs and CPUs in a cluster. After the job partitioning, the compiler generates an optimized MPI host program for the collaboration of the FPGAs, GPUs and CPUs in a cluster on the target application. However, while this flow may provide a good static solution for a standalone application, it cannot handle the situation when multiple applications are launched in the cluster.

Instead of using the MapReduce model for design optimization, we want to keep the capability of the existing MapReduce systems that automatically distribute map and reduce tasks from user applications to the servers in a cluster. If we enhance these MapReduce systems so that the map and reduce tasks can further get acceleration from the FPGA devices installed on the local servers of these tasks, we will be able to achieve both cluster parallelism and microarchitecture customization. There are some research projects along this direction. In [90], Hadoop is deployed on a server with NetFPGAs connected via an Ethernet switch. However, in this system MapReduce programmers need to perform low-level interactions with the FPGA device in their designs of *Map* and *Reduce* functions. In addition, no results are provided to show whether this system can be extended to a cluster of multiple servers. In [91], Hadoop is deployed on a cluster of Xilinx Zynq SoC devices [92] where CPU cores and programmable logics are fabricated on the same chip. However, this methodology is tightly coupled with the underlying Zynq platform and is hard to port to clusters with commodity CPU servers.

We observe that there are several challenges in building a general MapReduce system with FPGA accelerators:

1. The methodology to support FPGA accelerators in the MapReduce system should not be specific to a single application, but should allow multiple different jobs to share the system during the runtime.

2. The methodology should not be platform-specific, but should be feasible for clusters of commodity servers.

3. The methodology should not require MapReduce programmers to acquire knowledge about FPGAs, but should abstract away FPGA hardware details from them.

In this work we build a MapReduce system extended from Spark [86] to support FPGA accelerators and solve the challenges listed above. We prototype our system in a Xeon cluster equipped with FPGAs, and run multiple machine learning applications on big data. Note that with hardware accelerators, the communication often becomes the system bottleneck. Our cluster computing solution also optimizes the data transfer at different levels as follows:

1. We redesign algorithms to minimize the inter-node communication within a training iteration.

2. We exploit the in-memory processing of Spark to minimize the inter-node communication across training iterations.

3. We group multiple tasks in a batch to offload to an FPGA device to minimize the setup overhead of the data transfer between FPGAs and host servers.

4. We analyze the properties of iterative machine learning applications and cache the training data in FPGA device memory to minimize the data transfer between FPGAs and host servers.

Experimental results validate the effectiveness and scalability of our approach.

## 7.2 MapReduce System with FPGAs

### 7.2.1 Choice of Accelerators

First, we need to decide which type of accelerator to use in our accelerator-centric cluster computing. Different accelerators have their own advantages/shortcomings when used for computation acceleration. There are two popular candidates: GPUs and FPGAs.

GPUs usually have a fixed hardware architecture for parallel computing. For example, the NVIDIA GeForce 8800 GTX GPU is comprised of 16 streaming multiprocessors (SMs). Each SM has 8 streaming processors (SPs), with each group of 8 SPs sharing 16 KB of per-block shared memory. Each SP is deeply multithreaded, supporting 96 co-resident thread contexts which are free of the thread scheduling overhead. GPUs also have a fixed programming model of a single program, multiple data (SPMD) fashion. This model prefers applications to be free of interdependencies in the data flow so that the computation can be decomposed into a large number of threads. In addition, when the data-level parallelism in user applications does not fully match the architectural parameters of the target GPU device, it becomes hard to achieve the peak GPU performance.

FPGAs are an array of uncommitted programmable logic elements which are connected by programmable interconnects. This hardware fabric is most often used to approximate an ASIC (application-specific integrated circuit). This kind of usage eliminates the inefficiencies caused by the conventional von Neumann execution model and the pipelined implementations of GPUs and CPUs. Due to its ability for total customization of the hardware architecture for a user application's need, FPGAs can achieve a much higher energy efficiency.

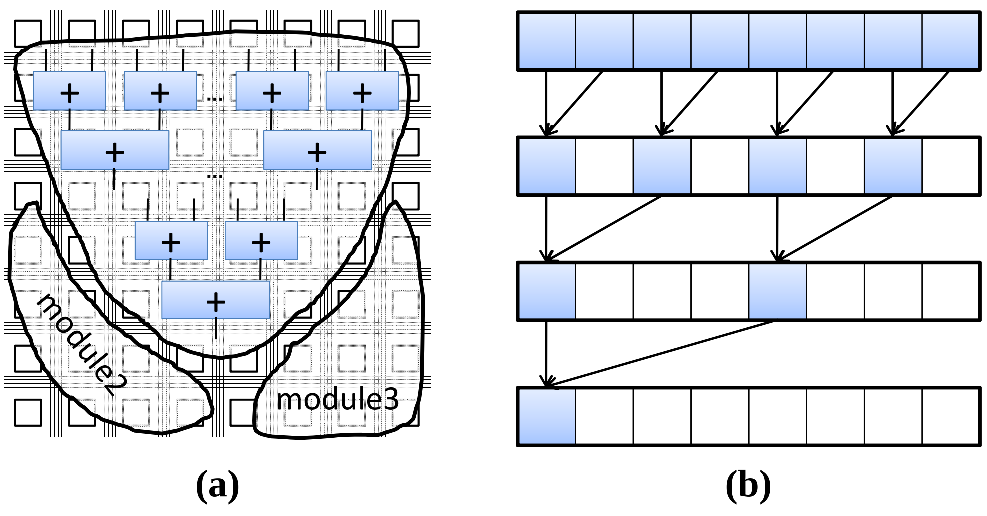


Figure 7.1: Implementations of the sum of vector elements on (a) an FPGA, and (b) a GPU.

Fig. 7.1 shows an example of the comparison between FPGAs and GPUs. Suppose we want to calculate the sum of the elements in a vector in an application. On one hand, we can instantiate an adder tree using some logic elements in an FPGA, as shown in Fig. 7.1(a). We can design the tree width to be the same as the vector size in our application. The unused logic elements left in the FPGA can further realize some other modules in our application. On the other hand, when this sum operation is executed on the fixed GPU architecture, it is converted to a number of parallel threads, as shown in Fig. 7.1(b). The number of working threads is reduced by one-half in each iteration, since the number of parallel threads is a constant value coupled with the architecture of the target GPU device. To make things worse, when the vector size cannot be divided by the number of parallel threads, there will be more idle threads. Because of these reasons, while GPUs have been employed as powerful coprocessors for some ranges of applications, FPGAs can achieve 10-1000x acceleration over CPU solutions in a wide range of applications, e.g., [141, 142, 143, 144]. In the machine learning applications that we are interested in, FPGAs can outperform GPUs by 4.4-6.5x [80].

However, we should admit the fact that today, FPGAs are still not so popular as GPUs for computation acceleration. The popularity of FPGA-based computation has been limited by its low programming productivity compared to CPUs and GPGPUs in the past. The price paid by FPGA customization for near-ASIC performance is that an FPGA implementation usually needs to be described in a hardware description language at register transfer level (RTL). This low level of programming abstraction leads to the long design cycle of FPGA solutions. The development of high-level synthesis (HLS) in recent years alleviates this problem [65]. Users are allowed to describe the FPGA implementation in a high-level programming language, e.g., C, C++ or systemC, and generate optimized RTL codes. The commercial HLS products such as AutoESL/VivadoHLS [72] and CatapultC [145] significantly reduce the programming burden of FPGA users. They also deliver even better designs with higher performance per area than manual designs in the past. The source to source capability as demonstrated in [135] further improves the program ability of FPGAs. Due to this recent progress, we believe that FPGAs are promising accelerators, and we choose FPGAs for our accelerator-centric cluster computing.

### 7.2.2 Target Cluster Architecture

Our target cluster architecture is shown in Fig. 7.2. In this architecture all the CPU servers are

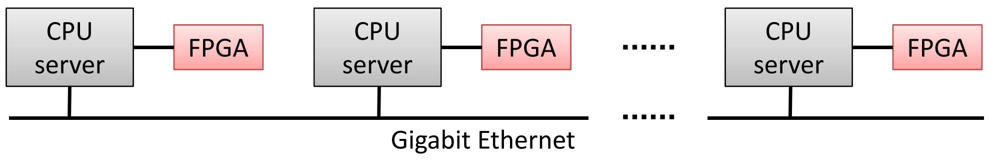


Figure 7.2: Our target cluster architecture.

connected by gigabit Ethernet via one or more Ethernet switches. Each CPU server can install one or more FPGA devices. To preserve the generality of our method, we do not put any assumption on the types of CPU servers and FPGA devices. A real prototyping of such a cluster is discussed in Section 7.4.1.

### 7.2.3 Execution Flow Overview

Fig. 7.3 shows the overall flow of our MapReduce system with FPGA accelerators. In this system

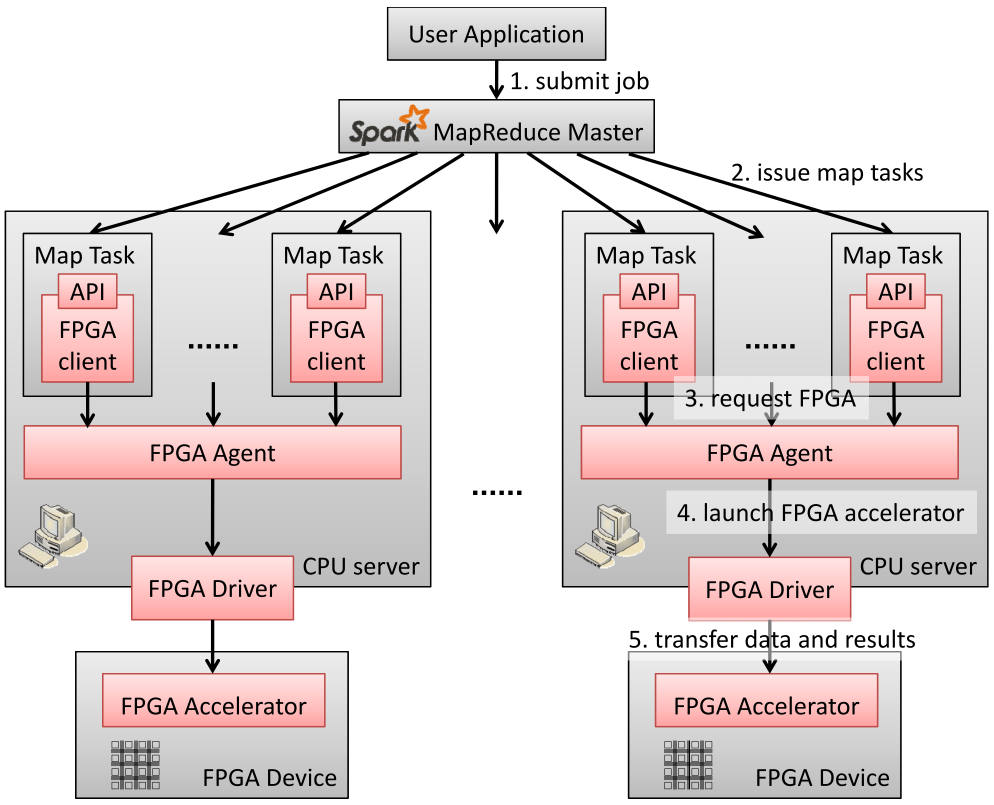


Figure 7.3: The overall flow of our MapReduce system with FPGA accelerators.

we develop the FPGA client and agent to help abstract away the FPGA hardware details from users. When a user writes a MapReduce program and calls our APIs for FPGA acceleration, the following sequence of actions occurs during the program execution:

1. When the user application enters the code segment of a MapReduce operation, it submits a MapReduce job to the MapReduce Master in the system.

2. The MapReduce Master partitions the data into data blocks (mainly according to the storage organization in the distributed file system). Then it issues a map task for each data block and distributes all the tasks to the servers in the cluster.

3. When the execution of a map task encounters our API for FPGA acceleration, an FPGA client is instantiated by the API call and requests FPGA resources from the FPGA agent in our system.

4. If the request is approved, the FPGA agent will retrieve the predesigned FPGA accelerator from the FPGA library and will launch it on the FPGA device via the driver.

5. Data are sent from the map task to the FPGA accelerator, and then results are retrieved after the completion of computation on the FPGA.

6. A similar process goes on for reduce tasks.

### 7.2.4 FPGA Management

Unlike CPUs, FPGAs are not time-divisible processors since they cannot be preempted and their internal states cannot be preserved during context switch. When multiple map/reduce tasks at the same node simultaneously request acceleration from the FPGA, the resource contention cannot be solved by existing OS implementation or MapReduce framework. In most existing systems with hardware accelerators, users have to handle this resource contention on their own by explicitly requesting resources, checking resource availability, and releasing resources [146]. In our system, we provide FPGA management to save users the burden of interfacing with hardware resources. Users only need to call our API for FPGA acceleration in their MapReduce programs, and then wait for the computation results. The features of our FPGA management include:

1. Resource scheduling. When multiple tasks request FPGA acceleration at the same time, our FPGA management schedules their executions over the timeline to ensure that an FPGA serves only one task at any point in time.

2. Resource virtualization. When the FPGA resources are fully occupied, our FPGA management will automatically switch the FPGA acceleration back to the CPU execution if this switch leads to faster job completion.

3. Resource allocation. When different jobs have different priorities and acceleration demands, our FPGA management will offer the high-priority jobs more FPGA resources and time periods in a quantitative way.

The FPGA management is mainly done in our FPGA agent which is launched on every server node equipped with FPGAs. Our FPGA client is instantiated upon each API call for FPGA acceleration in user tasks. It requests FPGA resources from our FPGA agent on behalf of users and then executes the responses. The details of these two modules are described below.

#### 7.2.4.1 FPGA Agent

Our FPGA agent maintains a task queue for each job to record all of its tasks waiting for FPGA acceleration, as shown in Fig. 7.4. Whenever an FPGA finishes its last task and becomes idle,

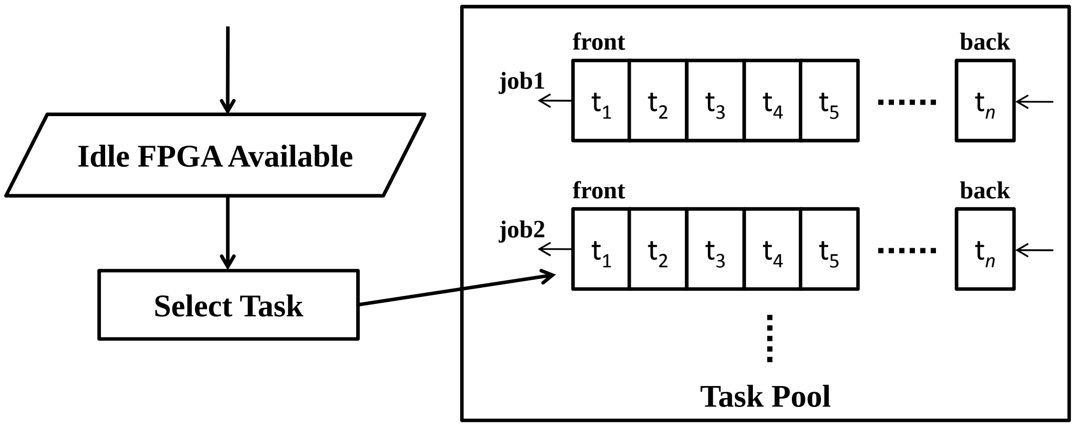


Figure 7.4: The task queues maintained in our FPGA agent for resource scheduling.

the scheduler in our FPGA agent will select a task from the pool of task queues. It will first select a job according to both priority and fairness. Each job is allocated with a certain FPGA capacity based on its priority. Our FPGA agent tracks the FPGA resource utilization of each job starting from its instantiation and compares the utilization to the allocated capacity. The scheduler always selects the job which has the largest gap between utilization and allocation. If all the jobs have the same priority, this selection will converge to the round-robin scheduling that guarantees fairness.

After the job selection, the task at the front of the queue for that job will be popped out for FPGA acceleration.

In addition to task scheduling, the FPGA agent also needs to respond to the acceleration requests from FPGA clients. Fig. 7.5 shows the request processing flow of our FPGA agent. After

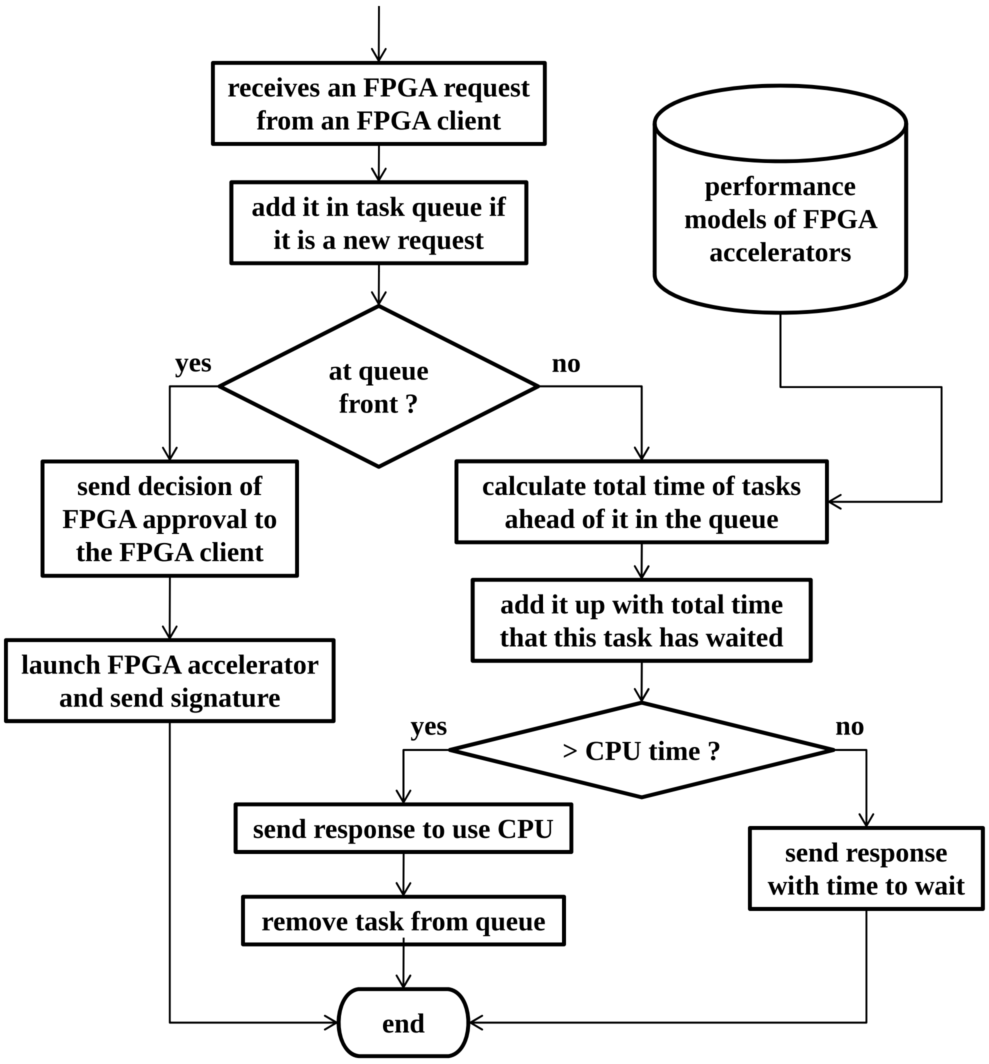

receives an FPGA request from an FPGA client
add it in task queue if it is a new request
performance models of FPGA accelerators
at queue front ?
yes
no
send decision of FPGA approval to the FPGA client
calculate total time of tasks ahead of it in the queue
add it up with total time that this task has waited
launch FPGA accelerator and send signature
> CPU time ?
yes
no
send response to use CPU
send response with time to wait
remove task from queue
end

Figure 7.5: The flowchart of our FPGA agent to process acceleration requests from FPGA clients.

receiving a request, our FPGA agent first checks whether this request already exists in a task queue. If it is a new request, the agent adds it to the task queue that belongs to the request's job. Then the agent checks whether this task is at the queue front. If yes, it means either it is this task's turn to get FPGA acceleration, or the FPGA is idle now. Then the agent responds to the FPGA client with the approval of FPGA acceleration and launches the FPGA accelerator for the task. If this task is not at the queue front, then it needs to wait for the other tasks ahead of it to finish. The agent retrieves the performance models of FPGA accelerators from our FPGA library to estimate the total wait time. This estimation is more accurate in the FPGA case than the CPU case since an FPGA accelerator design usually uses the static operation scheduling and scratchpad memory access optimization instead of the dynamic instruction execution and cache speculation in CPUs. If the wait time exceeds the CPU execution time, it will no longer be beneficial to get FPGA acceleration. In this case, the agent responds to the FPGA client with the CPU execution decision and removes the task from the queue. Otherwise, the agent responds to the client with the time to wait. After the client waits for this agreed-upon time period and resends the FPGA request of this task, the task probably just reaches the queue front and can get an immediate FPGA acceleration.

#### 7.2.4.2 FPGA Client

The job of our FPGA client is to get the computation done as required by the kernel acceleration API called by users. Fig. 7.6 shows the flowchart of this process. The client first sends an FPGA request to our FPGA agent on behalf of users. As mentioned before, our FPGA agent may respond with three kinds of decisions, depending on the FPGA status. In the first branch, the FPGA request is approved by the agent, and the client receives an FPGA accelerator signature from the agent. Then the client sends function parameters and data arrays to the granted FPGA accelerator for computation, and retrieves results. In the second branch, the agent suggests switching the computation back to the CPU execution. Then the client retrieves the binary code of the computation requested by the called API and executes it. In the third branch, the agent suggests waiting for a while. Then the client receives a wait time from the agent, and puts the thread of the map/reduce task to sleep for the agreed-upon time. After this time period, the client sends the FPGA request again to the agent.

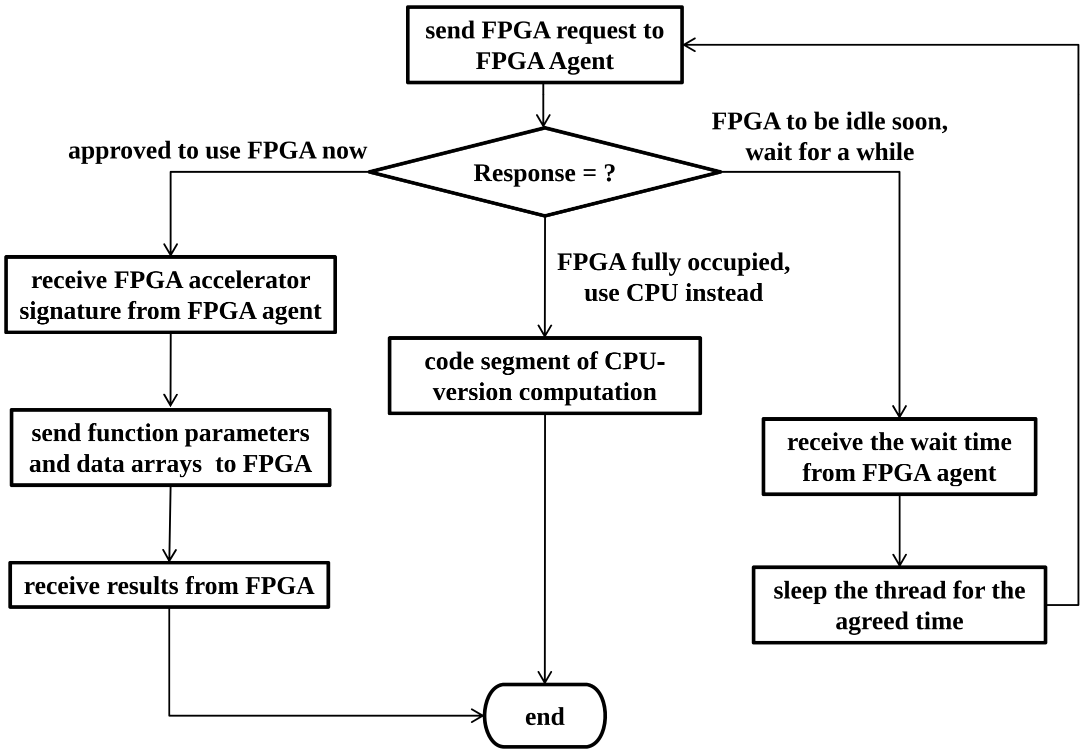

send FPGA request to FPGA Agent
Response = ?
approved to use FPGA now
FPGA to be idle soon, wait for a while
FPGA fully occupied, use CPU instead
receive FPGA accelerator signature from FPGA agent
send function parameters and data arrays to FPGA
receive results from FPGA
code segment of CPU-version computation
receive the wait time from FPGA agent
sleep the thread for the agreed time
end

Figure 7.6: The flowchart of our FPGA agent to get the computation done as required by the kernel acceleration API called by users.

## 7.3 Communication Optimization in Cluster Computing

### 7.3.1 Application Analysis

Many machine learning applications, e.g., logistic regression and neural network training, are basically solving optimization problems to maximize the target likelihoods. The most intensively used computation is stochastic gradient descent (SGD). It is an iterative training process, as shown in Algorithm 7. Here $\Delta w$ is the gradient being calculated. $\nabla Q_s(w)$ is the gradient of the objective

**Algorithm 7:** Pseudo code of stochastic gradient descent in machine learning applications.

```
Input  : Training data $D$, and objective function $Q(w)$
Output: Model weights $w$

1 for $i = 1, 2, 3, \cdots$, MAX ITER do
2     foreach mini-batch $b$ in $D$ do
3         $\Delta w = 0$ ;
4         foreach data sample $s$ in $b$ do
5             $\Delta w += \nabla Q_s(w)$ ;
6         end
7         $w -= \alpha \Delta w$ ;
8     end
9 end
```

function at the data sample $s$. $\alpha$ is the step size (or sometimes called the learning rate). The parallelism exists at the loop in line 4 and accordingly, the computation can be distributed to multiple server nodes in a cluster. After this computation finishes, each server node has to add its own result to the node that holds the model weights for aggregation in line 7. This step is performed for each mini-batch. Since the number of mini-batches increases linearly as the data size goes up, the inter-node communication per server node becomes nontrivial for the cases of big data.

### 7.3.2 Reducing Inter-Node Communication

To reduce the inter-node communication, we redesign Algorithm 7 to create Algorithm 8. Here

**Algorithm 8:** Pseudo code of distributed stochastic gradient descent for inter-node communication reduction.

```
Input  : Training data $D$, and objective function $Q(w)$
Output: Model weights $w$
1  for $i = 1, 2, 3, \cdots$ ,MAX ITER do
2      foreach data partition $p$ in $D$ do
3          $\Delta w_p = 0, w_p = w$ ;
4          foreach mini-batch $b$ in $p$ do
5              foreach data sample $s$ in $b$ do
6                  $\Delta w_p + = \nabla Q_s(w_p)$ ;
7              end
8              $w_p - = \alpha \Delta w_p$ ;
9          end
10     end
11     $w = \frac{1}{P} \sum_{p=1}^{P} w_p$ ;
12 end
```

$P$ is the number of data partitions. We first partition the training data among the server nodes as shown in line 2 of Algorithm 8. Then we apply the stochastic gradient descent in Algorithm 7 to the local data partition on each node. In the end, we aggregate the model weights from all the servers for the average $w$ as the final result, as shown in line 11 of Algorithm 8. In each iteration of this algorithm, the inter-node communication is only once per training iteration, and the solution is scalable even when the data size is large.

### 7.3.3 Reducing File Accesses

The training data are usually large and are stored in a distributed file system. At the beginning of each training iteration in Algorithm 8, each server node needs to load the corresponding data partition of its assigned task from the file system. This data loading may cause either disk accesses or network traffic. We want to further reduce the file accesses for training data. By analyzing Algorithm 8, we determine that we can fix the data partitioning across the training iterations so that the data partition used by each server node is kept constant. Then we can let each server node cache the data partition in its main memory to eliminate the file accesses. To make it happen, we use the Hadoop Distributed File System to store the training data and turn on the data caching option in the Spark MapReduce framework. When the Spark master distributes tasks among server nodes, it will check the data content cached on each node and will place tasks to favor the data locality.

### 7.3.4 Reducing Data Transfers between CPUs and FPGAs

When we deploy the machine learning applications in our MapReduce framework with FPGA accelerators, we offload the kernel computation in line 6 of Algorithm 8 to FPGAs. If we initialize the FPGA execution for every data sample, we need to set up the data transfer channel between the CPU and the FPGA and bear the setup cost each time. Since the number of data samples increases linearly as the data size goes up, the total data transfer latency can be large for the cases of big data. To solve this problem, we group all the data samples in the partition stored at a server node together and send to the FPGA in a batch. After this improvement, we only need to bear a single setup cost of the data transfer between the FPGA and the host server in each training iteration.

We further observe that since we offload the major computation to the FPGA, the training data partition used by an FPGA is also kept constant, the same as the CPU case discussed in Section 7.4.3. Therefore, we implement the data caching in the FPGA device memory to save the unnecessary data transfer to the FPGA. Currently, we identify whether the training data already exists in the FPGA by simply checking whether the current iteration is the first training iteration or not. If yes, we bypass the data transfer between the FPGA and the host server.

## 7.4 Prototyping and Experiments

### 7.4.1 Prototyping Setup

We deploy our accelerator-centric MapReduce system on a cluster of servers equipped with FPGA devices. Fig. 7.7 is a photograph of our cluster. The cluster contains five Xeon servers running at

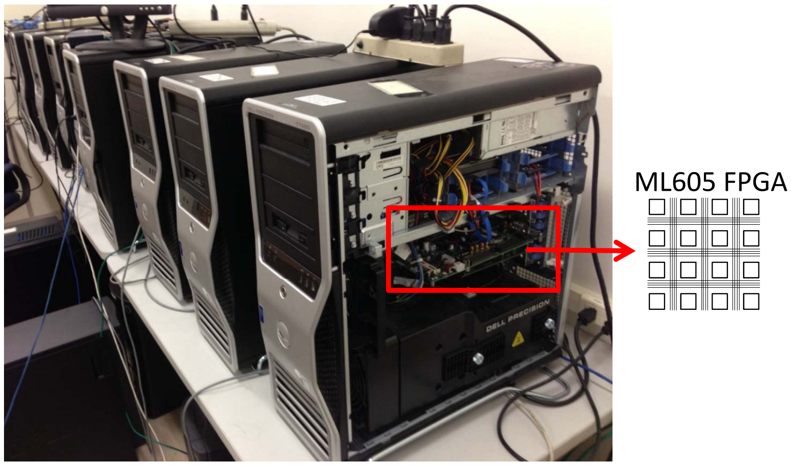


Figure 7.7: A photograph of our cluster with FPGA devices.

2GHz. Each server has an ML605 FPGA board connected via the PCIe bus. We use the host-FPGA PCIe communication library in [147] as the underlying link layer of data transfer to the FPGAs.

We deploy the Hadoop distributed file system (HDFS) [148] in the cluster for data storage. We launch our Spark-based MapReduce framework along with our FPGA agents in the whole cluster. Our FPGA clients will be launched during the executions of MapReduce applications in our system.

We test two MapReduce applications: logistic regression and neural network training from [149]. They are used to classify large amounts of data into different categories. As the process of data training and model updating is iterated over time, higher classification accuracy will be achieved. The major computation lies in the calculation of weight gradients, and we offload this part to FPGA accelerators. Some training parameters, such as the learning rate and the regularization coefficient, are adopted from [149]. We use the training data from the MNIST database of handwritten digits [74]. This database contains a training set of 60,000 examples and a test set of 10,000 examples. After training with this database, the classifier can categorize previously unseen handwritten digit images into the ten digit numbers.

### 7.4.2 System Validation

We first validate that our system can produce the correct results for the logistic regression and neural network training. For machine learning applications, an important criterion for justifying the correctness of results is to see whether the classification accuracy increases as the training iteration goes on. We record this accuracy trend during the training and plot it in Fig. 7.8. At the

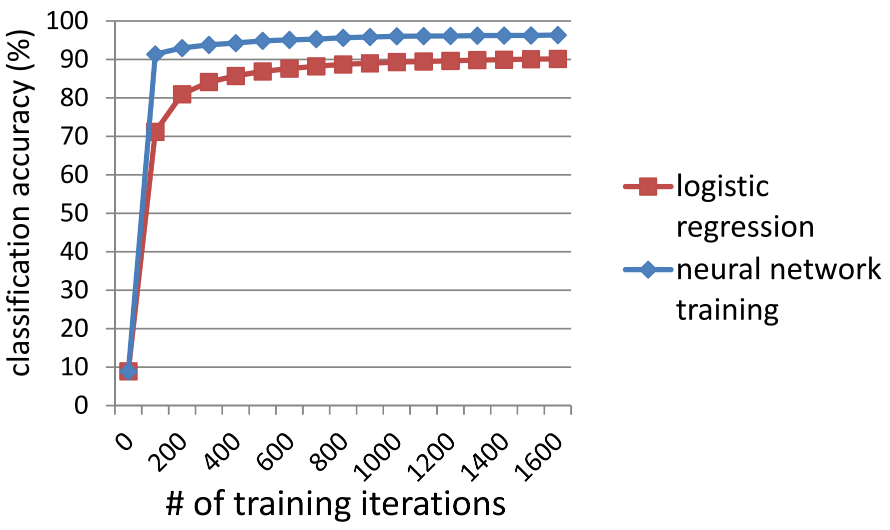


Figure 7.8: The classification accuracy during the logistic regression and neural network training running on our MapReduce system with FPGA accelerators.

beginning of training, the classifiers in the two applications behave like a random guess among the ten possible digits, and the accuracy is close to $10\%$. With the increase in the number of training iterations, the accuracy gradually approaches $100\%$. The accuracy of the neural network training grows faster than the logistic regression since the neural network training uses a complex model with more powerful logic abstraction. These results mean that our system successfully enables FPGA accelerators in MapReduce applications.

### 7.4.3 Communication Optimization Breakdown

Step by step, we study the effect of our communication optimization technologies discussed in Section 7.3. Fig. 7.9 shows the system performance after each optimization step. First, after

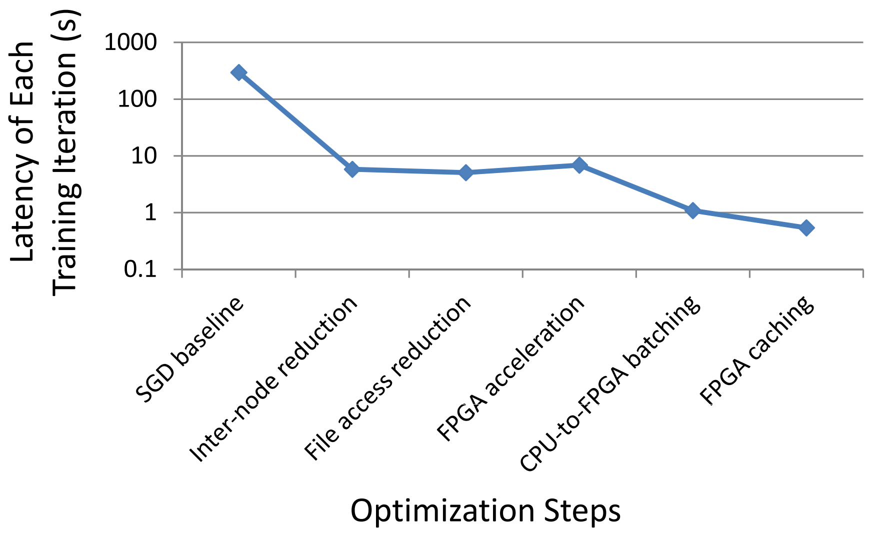


Figure 7.9: Communication optimization breakdown.

the inter-node communication reduction, the latency of each training iteration is reduced significantly compared to the baseline. In the original algorithm, the inter-node communication has to be performed for each mini-batch, and there are 3,000 mini-batches in our benchmark. After optimization, we only need to perform the inter-node communication once per iteration. Second, after the file access reduction, the latency drops further due to the savings of loading data from the disk or the network at the beginning of each training iteration. Third, after FPGA acceleration, the latency goes up a bit. This is because without optimization, we transfer data to FPGAs for acceleration at a fine granularity, and the total setup cost of the CPU-to-FPGA data transfer undermines the computation acceleration benefit. After we further group data to send to FPGAs in a batch, the latency goes down. Finally, we cache data in the FPGA device memory to eliminate unnecessary CPU-to-FPGA data transfers, and the latency further deceases.

### 7.4.4 Runtime Breakdown

After system optimization, we perform a full system analysis for a runtime breakdown of applications running on our MapReduce system with FPGA accelerators. To avoid the complexity caused by the thread-level parallelism and to better understand the runtime breakdown, we set up only one CPU executor per server node. Fig. 7.10 shows the results. The total number of training iterations

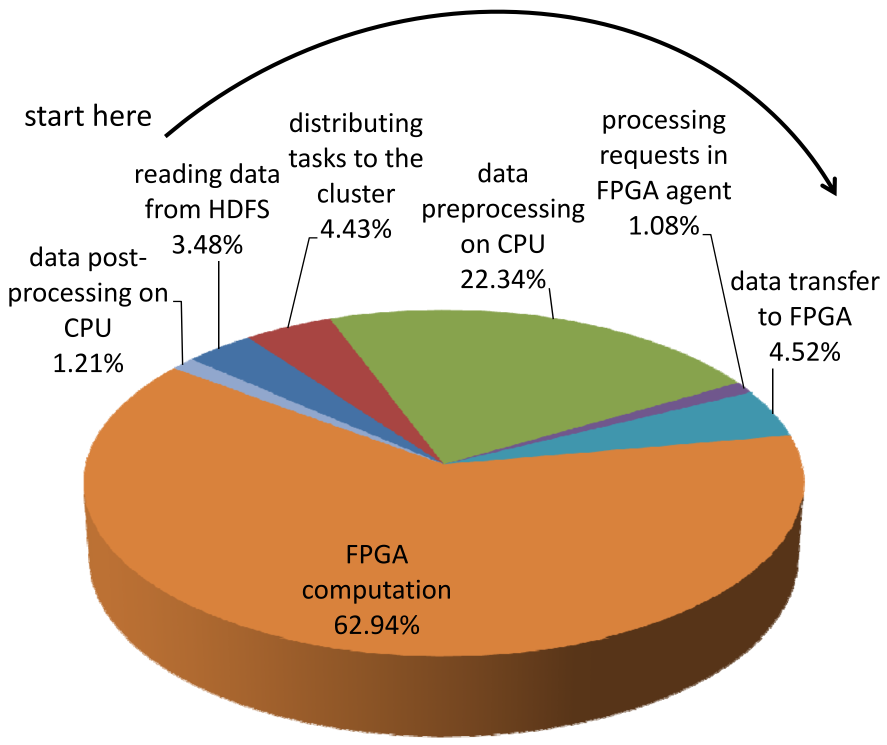


Figure 7.10: Runtime breakdown of an application running on our MapReduce system with FPGA accelerators.

is set to 1,000. The breakdown is listed according to the execution order, starting from reading data from the HDFS. There are several observations:

- Most of the execution time is spent on the FPGA computation after we try to minimize the communication cost.
- Around $23\%$ of runtime is still the pure-CPU execution that cannot be accelerated by FPGAs, which includes the data preprocessing and post-processing. This gives us opportunities to overlap the CPU execution with the FPGA execution, which is discussed in Section 7.4.5.
- The overhead of our FPGA agent for resource management is as small as $1.08\%$.

### 7.4.5 Resource Management for Multiple Jobs

We test the FPGA resource management in our system by simultaneously running multiple jobs of neural network training. We first set all the jobs to be the same priority and sweep the number of jobs running in parallel from one to eight. Fig. 7.11 shows the results. As the number of parallel

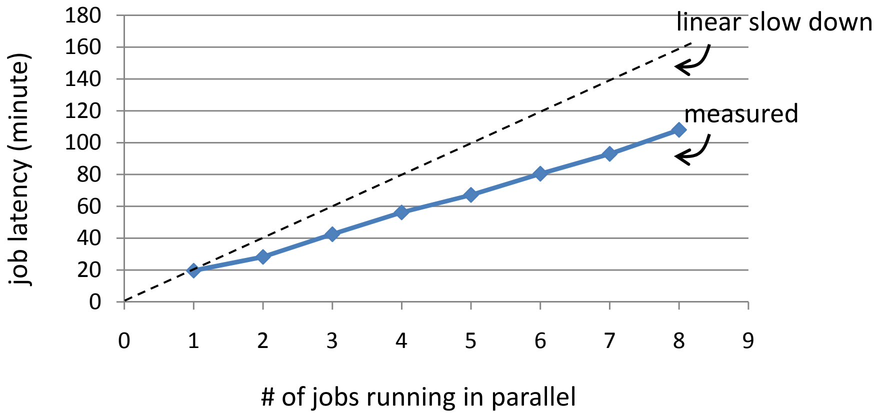


Figure 7.11: Latency of each job when we simultaneously run multiple jobs in our MapReduce system with FPGA accelerators.

jobs increases, the latency of each job also increases since these jobs have to share the same FPGA pool. An interesting observation is that the latency increase is slower than the linear slow down. We may intuitively think that the latency should be doubled against a single job when we run two jobs together. But this is not the case in our system. This is because a job contains some segments that belong to the pure-CPU execution (as discussed in Section 7.4.4). When a job is executing the CPU part, the other job can use the FPGAs exclusively. When there are more jobs running in parallel, there are longer periods of CPU execution that overlap with FPGA execution.

We further test our system feature of FPGA resource management in terms of job priority. We still run multiple jobs, but set one job to be of a higher priority. The result is shown in Fig. 7.12. As the number of parallel jobs increases, the latency of the high-priority job keeps nearly constant

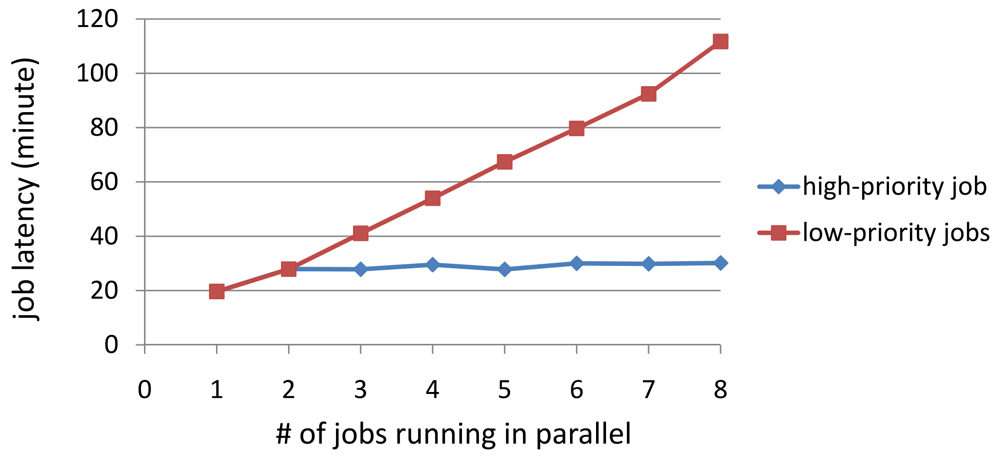


Figure 7.12: Latency of each job when we simultaneously run multiple jobs with different priorities in our MapReduce system with FPGA accelerators.

while the latency of the remaining low-priority jobs increases. It means that we successfully offer the FPGA resources first to the high-priority job. Only when this job is executing the CPU part, do the other low-priority jobs have a chance of FPGA acceleration. Note that there is still a small increase in the latency of the high-priority job compared to the latency of running a standalone job. That's because when the high-priority job returns from the CPU part to the FPGA-accelerable part, the FPGA may be executing a task from some other jobs. The high-priority job has to wait a little bit for the FPGA to finish its current task.

We also investigate the FPGA utilization in our system. In previous experiments, for simplicity, we set the number of CPU executors per server node to one. When there is only one job running in our system, this setting will prevent the FPGA from being fully utilized due to the CPU portion in the job as analyzed in Section 7.4.4. So, we set the number of CPU executors per server node to the number of CPU cores in this experiment. Fig. 7.13 shows the result. Here we run two jobs

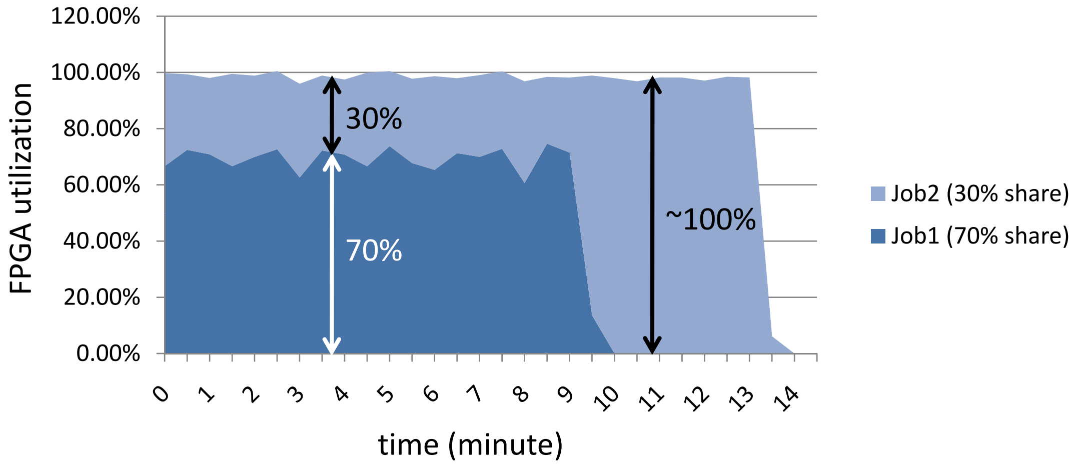


Figure 7.13: FPGA utilization of each job when we run parallel jobs in our system.

simultaneously and monitor the FPGA utilization of each job separately. Job1 has a high priority and is allocated 70% of the FPGA shares when submitted to our system. Meanwhile, job2 has a low priority and is allocated 30% of FPGA shares. All the tasks in these jobs are evenly distributed to all the server nodes, and we pick up a random node to monitor the FPGA utilization. There are several observations in Fig. 7.13:

1. The sum of the FPGA utilization by these two jobs is close to 100% over the time period. It means that our resource management can achieve full utilization of FPGAs when the workloads are bounded by FPGA resources.

2. When both jobs are being executed in the left part of the time line in Figure 7.13, job1 occupies around 70% of the FPGA working time while job2 occupies around 30%. This utilization matches the resource allocation decision made during the submission of the two jobs. It means that our resource management correctly manages the FPGA resources among parallel jobs with different priorities in a quantitative way.

3. After job1 finishes in the right part of the time line in Figure 7.13, our resource management sees no other jobs that demand the FPGA resources, except for job2. The FPGA shares of

job2 are then automatically increased to boost the job performance.

### 7.4.6 System Scalability

Finally we test the scalability of our system by sweeping the number of nodes included in our cluster from one to five. We run either logistic regression or neural network training on these different settings and measure the system performance by counting how many training iterations are completed per minute. Fig. 7.14 shows the results. As the number of nodes increases, the

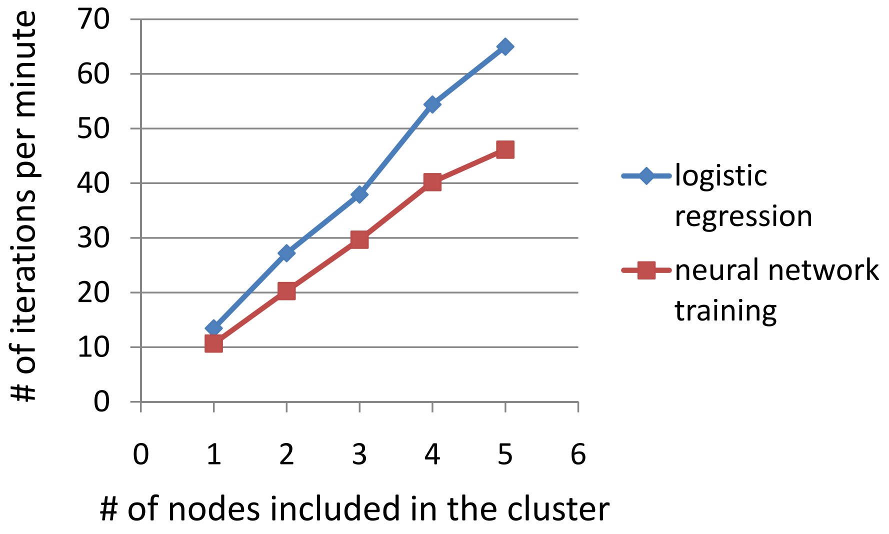


Figure 7.14: The performance of our MapReduce system with FPGA accelerators scales with the number of nodes included in the cluster.

system performance grows linearly. This result indicates that our MapReduce system with FPGA accelerators preserves the scalability.

## 7.5 Conclusions

In this chapter we developed a MapReduce system with FPGA accelerators. It supports general MapReduce programs and provides FPGA resource management to allow multiple jobs running in parallel. We prototype our system in a Xeon cluster equipped with FPGAs, and test multiple machine learning applications on big data. While FPGAs accelerate the computation, we minimize the communication at different levels to make the acceleration more beneficial. We redesign algorithms to minimize the inter-node communication within a training iteration. We exploit the in-memory processing of Spark to minimize the inter-node communication across training iterations. We group multiple tasks in a batch to offload to a target FPGA to minimize the setup overhead of the data transfer between FPGAs and host servers. We analyze the properties of iterative machine learning applications and cache the training data in FPGA device memory to minimize the CPU-to-FPGA data transfer. The experimental results validate the effectiveness and scalability of our approach.

# CHAPTER 8

# Summary and Concluding Remarks

## 8.1 Technology Summary

This dissertation explores the communication optimization for customizable domain-specific computing. Table 8.1 summaries our exploration efforts. To save users the programming effort, these

| Optimization Level | Building Components | Communication Patterns to Customize | Driving Force |
|---|---|---|---|
| fabric-level (Chapter 2 and Chapter 3) | Look-Up-Tables (LUTs) | Boolean network in netlist of application-specific accelerators | emerging devices |
| chip-level (Chapter 4 and Chapter 5) | On-chip memories and accelerators | data access patterns and accelerator execution patterns | accelerator-centric architecture |
| server/cluster-level (Chapter 6 and Chapter 7) | CPU and FPGA boards | task patterns | big data |

Table 8.1: Summary of proposed communication optimizations for customizable domain-specific computing.

technologies are also automated under a general design automation framework as discussed in Fig. 8.4 of Chapter 1.

## 8.2 Accumulative Impact

We create a generalized case study that can benefit from all the communication optimization technologies proposed in this thesis, as shown in Fig. 8.1. We use the convolutional neural network

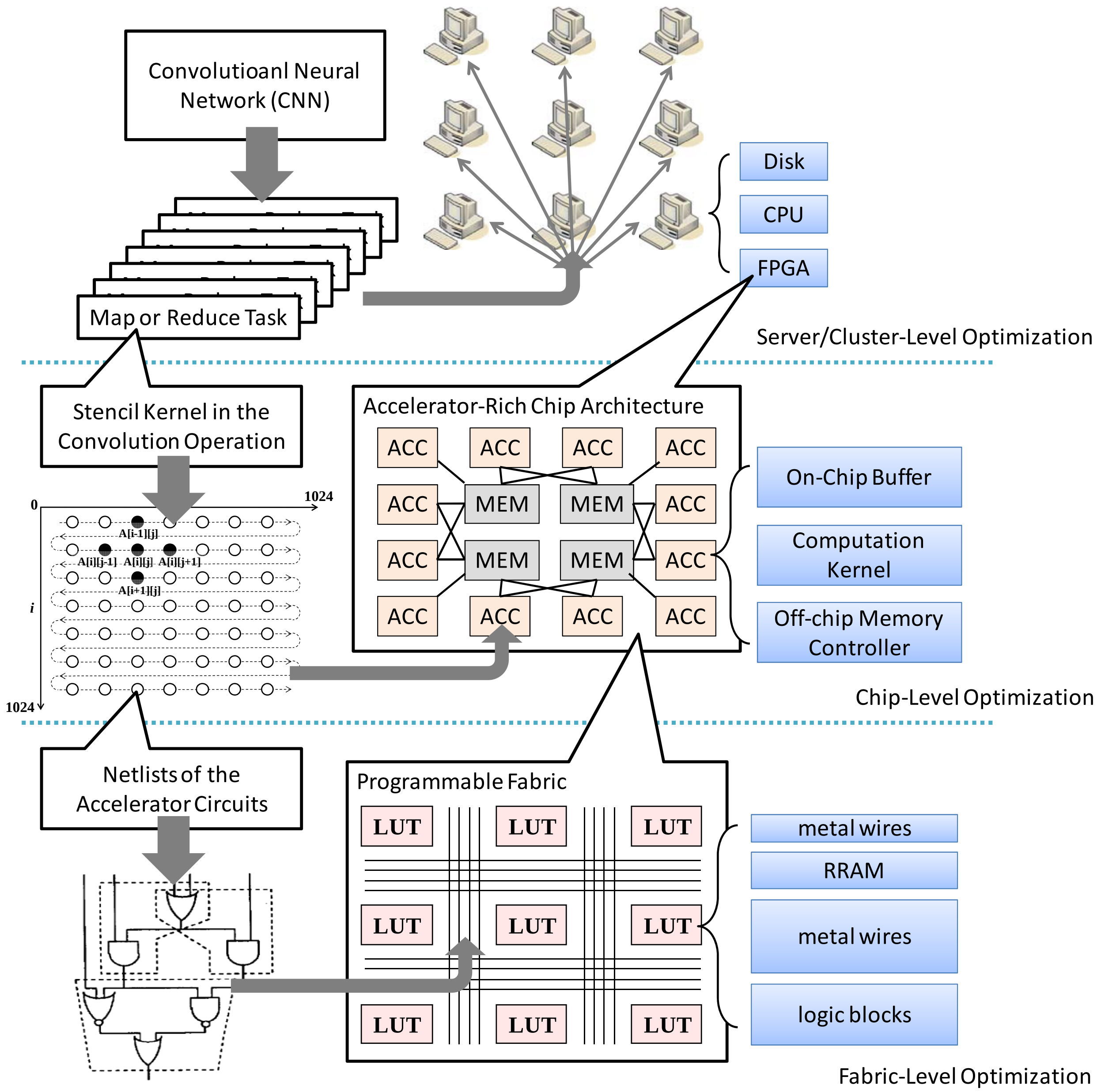

Convolutioanl Neural Network (CNN)
Map or Reduce Task
Disk
CPU
FPGA
Server/Cluster-Level Optimization
Stencil Kernel in the Convolution Operation
0
1024
i
1024
A[i-1][j]
A[i][j-1] A[i][j] A[i][j+1]
A[i+1][j]
Accelerator-Rich Chip Architecture
ACC
MEM
On-Chip Buffer
Computation Kernel
Off-chip Memory Controller
Chip-Level Optimization
Netlists of the Accelerator Circuits
Programmable Fabric
LUT
metal wires
RRAM
metal wires
logic blocks
Fabric-Level Optimization

Figure 8.1: A generalized case study that can benefit from all the communication optimization technologies proposed in this thesis.

(CNN) as the example case. First we decompose the computation into many tasks and distribute them among the server nodes in the cluster. We can optimize the inter-server communication as proposed in Chapter 7. At each server node, we need to move data among disks, CPUs and FPGAs. We can optimize the data transfer among these devices as proposed in Chapter 6. Each FPGA device is reconfigured to an accelerator-rich chip with many accelerators of frequently used computation kernels in the CNN. We can optimize the interconnects between accelerators and shared memories as proposed in Chapter 4. In each accelerator, we need to move data among the on-chip buffers, the computation kernel, and the off-chip memory controller. We can optimize the data transfer for better data reuse as proposed in Chapter 5. After physical synthesis, the netlists of accelerator circuits are mapped to the underlying programmable fabric of an FPGA device. We can optimize the programmable interconnects by using emerging devices RRAMs as proposed in Chapter 2. The emerging devices can have a high defect rate, and we can optimize the signal routing to tolerate the defects as proposed in Chapter 3.

Based on this generalized case, we collect the benefits brought by the different technologies proposed in this thesis together and project them onto a unified roadmap of energy-efficiency improvement. We insert the Intertek®power meter P4460.01 at the power supplies in our system to

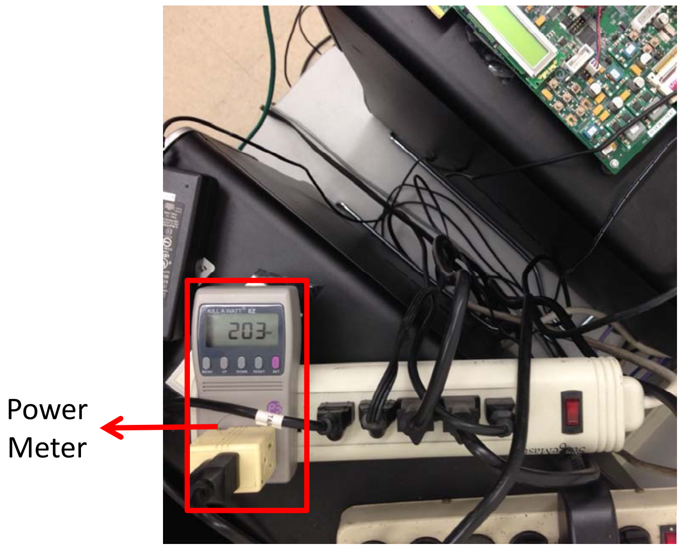


Figure 8.2: Power measurement of our system.

measure the power, as shown in Fig. 8.2. Because of the difficulty in recording the power of different components in real time, we measure their power numbers at different working states instead. Then we calculate the sum of the products between these power numbers and the time durations of the different states to estimate the total energy consumption. Table 8.2 lists the power numbers that we measured. Based on these power numbers, the energy consumption reductions brought

| Component | Working State | Power (W) |
|---|---|---|
| Xeon Server | Idle | 203 |
| | Network Communication | 214 |
| | Disk Access | 248 |
| | Computation @ $\sim 100\%$ Utilization | 296 |
| ML605 FPGA | Idle | 19.6 |
| | Active @ $\sim 100\%$ Utilization | 24.9 |

Table 8.2: Power numbers of different components at different working states measured by a power meter for the estimation of energy consumption.

by our methodologies are projected in Fig. 8.3. Here we use an CNN with 192 $13 \times 13$ feature maps in the neuron layer and the $3 \times 3$ stencil window in the synapse layer, and five Xeon servers for cluster computing. First, we apply inter-server communication minimization including the distributed stochastic gradient descent and Spark data caching proposed in Chapter 7. The energy consumption is reduced significantly compared to the baseline. Second, we perform the optimization of the tasks executed on each node based on our model of the convolutional MM proposed in Chapter 6 to reduce both the communication and computation. The energy consumption goes down further. Third, we use FPGAs to accelerate the most frequently used computation kernels in the CNN—the convolution operations. At this first step of FPGA acceleration, we temporarily turn off the communication optimization and focus on the computation resources. To fully utilize the DSP resources in an ML605 FPGA device in our platform, we can duplicate the accelerator of the convolution kernel into 17 copies for parallel execution. Then the energy consumption goes down by a bit, but does not reduce as much as usually reported in literature. This is because in spite of

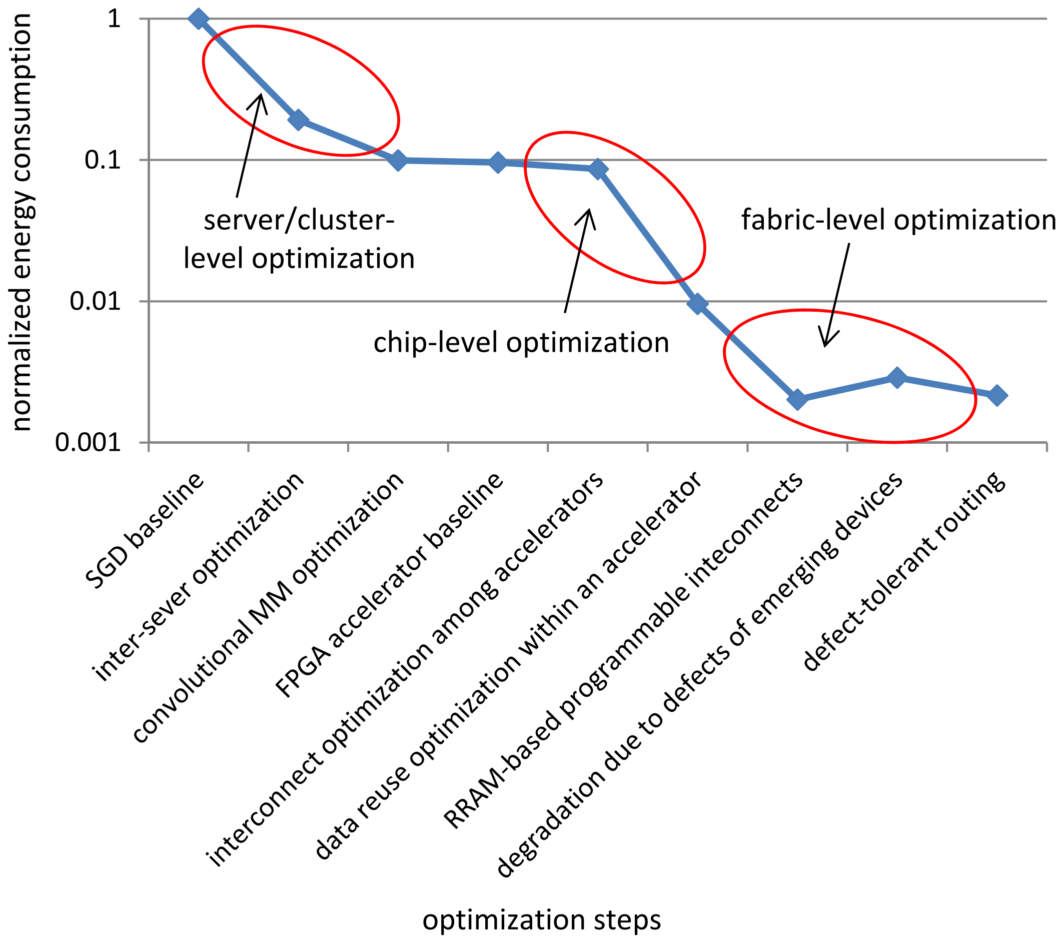


Figure 8.3: Step-by-step optimization breakdown of the example in Fig. 8.1 by the different technologies proposed in this thesis.

the accelerated computation, there are still two bottlenecks that limit the FPGA performance: the inefficient global interconnects, and the excessive off-chip accesses performed by an accelerator. After we replace the conventional global interconnects with the novel partial crossbar that was customized to the accelerator-rich architecture in Chapter 4, the accelerator performance and energy efficiency is improved a little, but is still limited by the off-chip data accesses within an accelerator. Then we reduce the nine off-chip accesses of the $3 \times 3$ stencil in the CNN to one via the data reuse technology proposed in Chapter 5, and the energy consumption goes down substantially. Afterwards, we can improve the underlying FPGA fabric by using the RRAM-based programmable interconnects proposed in Chapter 2, and the energy consumption further decreases. But if we count the degradation caused by the defects of the emerging devices, the energy consumption should go up. Finally, we use defect-tolerant routing to alleviate this degradation as proposed in Chapter 3, and the energy consumption decreases again.

## 8.3 Design Automation Tools

In addition to the optimization methodologies proposed in this thesis, we also realize these methodologies in design automation tools to save users the programming effort. Fig. 8.4 shows the general framework of how these tools fit in. It starts from the original source codes of a user application. We first analyze the communication patterns in the application. Then, given the specifications of the target platform, we perform the proposed optimizations on these communication patterns. After that, the optimization results will be combined with some pre-designed code templates to generate the new program source codes. The optimized program will be forwarded to the back-end compilation flow for the final mapping. The work in this thesis produces several design automation tools that fits into the following frameworks [44, 64, 135]:

1. Our *mrVPR* tool is a placement and routing tool that maps the netlists of accelerator circuits to the RRAM-based FPGAs (discussed in Chapter 2 and Chapter 3).
2. Our *ARAcompiler* tool is a software tool to generate an accelerator-rich architecture for pro-

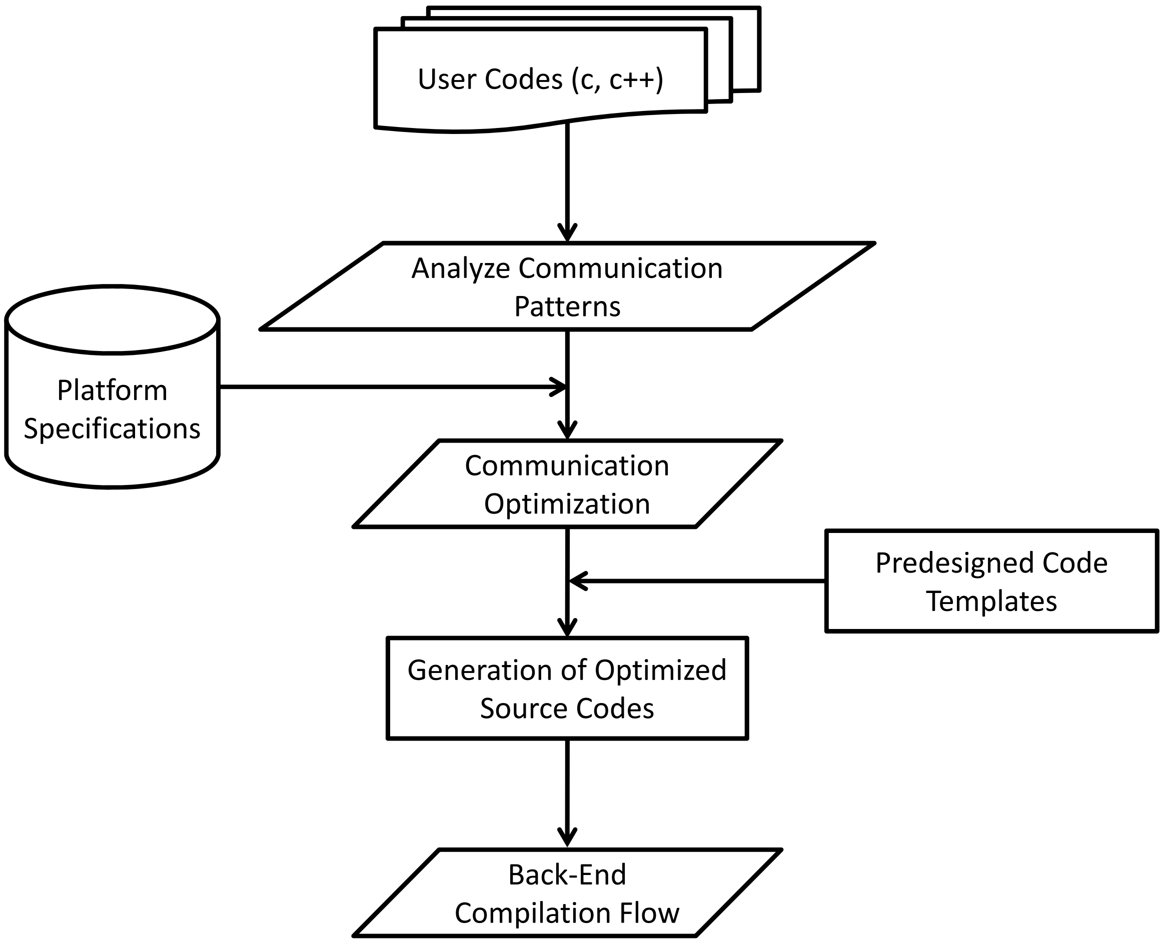

User Codes (c, c++)
Analyze Communication Patterns
Platform Specifications
Communication Optimization
Predesigned Code Templates
Generation of Optimized Source Codes
Back-End Compilation Flow

Figure 8.4: The general framework of our design automation tools with the integration of communication optimization.

totyping on FPGAs where the accelerators are connected by the interconnects discussed in (Chapter 4).

3. Our *CMOST* tool is an automated compilation flow from C-code to executable FPGA accelerators that integrates the stencil computation optimization proposed in (Chapter 5).

## 8.4 Open Issues

During the course of this study, we also realized the following limitations.

1. In the fabric-level communication optimization, we only considered the fabrication defects in emerging devices, but did not propose technologies to tolerate defects that appear during lifetime. We used a conservative scheme to deal with defects with the sacrifice of interconnect performance at an early age with fewer defective devices.

2. In the chip-level communication optimization, we assumed that we know a batch of accelerator executions in advance. This needed an interval-based resource scheduler to globally allocate data channels to the accelerator workloads that were received within a time period.

3. In the server/cluster-level communication optimization, we focused on one of the popular big data application domains, machine learning, and built up the corresponding methodologies of communication minimization. We hope that these methodologies will be applied to other big data application domains.

The journey just begins. Today, domain-specific computing has become increasingly important for achieving energy-efficient green computing. However, research that focuses on communication optimization for domain-specific computing is still far from a mature stage. Continued research will require a comprehensive effort from the domain experts, computer architects, and circuit designers. It is our hope that the proposals and methodologies discussed in this dissertation—including their limitations, of course—will inspire more innovations in the future for the communication optimization of customizable domain-specific computing.

# References

[1] M. Horowitz, “1.1 Computing’s energy problem (and what we can do about it),” in *International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb. 2014, pp. 10–14.

[2] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Norwell: MA:Kluwer, 1999.

[3] Xilinx, “Virtex-7 FPGA data sheets.” [Online]. Available: http://www.xilinx.com/support/documentation/7_series.htm

[4] K. Tsunoda, K. Kinoshita, H. Noshiro, Y. Yamazaki, T. Iizuka, Y. Ito, A. Takahashi, A. Okano, Y. Sato, T. Fukano, M. Aoki, and Y. Sugiyama, “Low Power and High Speed Switching of Ti-doped NiO ReRAM under the Unipolar Voltage Source of less than 3V,” in *International Electron Devices Meeting (IEDM)*, Dec. 2007, pp. 767–770.

[5] S.-S. Sheu, P.-C. Chiang, W.-P. Lin, H.-Y. Lee, P.-S. Chen, T.-Y. Wu, F. T. Chen, K.-L. Su, M.-J. Kao, and K.-H. Cheng, “A 5ns Fast Write Multi-Level Non-Volatile 1 K bits RRAM Memory with Advance Write Scheme,” in *VLSI Circuits, Symposium on*, 2009, pp. 82–83.

[6] W. Guan, S. Long, Q. Liu, M. Liu, and W. Wang, “Nonpolar Nonvolatile Resistive Switching in Cu Doped $ZrO_2$,” *IEEE Electron Device Letters*, vol. 29, no. 5, pp. 434–437, May 2008.

[7] C.-H. Wang, Y.-H. Tsai, K.-C. Lin, M.-F. Chang, Y.-C. King, C.-J. Lin, S.-S. Sheu, Y.-S. Chen, H.-Y. Lee, F. T. Chen, and M.-J. Tsai, “Three-Dimensional $4F^2$ ReRAM Cell with CMOS Logic Compatible Process,” in *IEDM Technical Digest*, 2010, pp. 664–667.

[8] S. Tanachutiwat, M. Liu, and W. Wang, “FPGA Based on Integration of CMOS and RRAM,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 11, pp. 2023–2032, Nov. 2011.

[9] I. Kuon and J. Rose, “iFAR – intelligent FPGA Architecture Repository.” [Online]. Available: http://www.eecg.utoronto.ca/vpr/architectures/

[10] “Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 70731.” [Online]. Available: http://www.eecs.berkeley.edu/$\sim$alanmi/abc/

[11] G. Lemieux, P. Leventis, and D. Lewis, “Generating highly-routable sparse crossbars for PLDs,” in *International Symposium on Field Programmable Gate Arrays*, 2000, pp. 155–164.

[12] J. Cong, W. Jiang, B. Liu, and Y. Zou, “Automatic memory partitioning and scheduling for throughput and power optimization,” in *International Conference on Computer-Aided Design*, 2009, p. 697.

[13] P. Li, Y. Wang, P. Zhang, G. Luo, T. Wang, and J. Cong, "Memory partitioning and scheduling co-optimization in behavioral synthesis," in *International Conference on Computer-Aided Design*, 2012, pp. 488–495.

[14] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong, "Memory partitioning for multidimensional arrays in high-level synthesis," in *Design Automation Conference*, 2013, p. 1.

[15] "Bicubic interpolation." [Online]. Available: http://www.mpi-hd.mpg.de/astrophysik/HEA/internal/Numerical_Recipes/f3-6.pdf

[16] S. Verdoolaege, H. Nikolov, and T. Stefanov, "pn: A Tool for Improved Derivation of Process Networks," *EURASIP Journal on Embedded Systems*, vol. 2007, pp. 1–13, 2007.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proceedings of Neural Information and Processing Systems*, 2012, pp. 1–9.

[18] M. J. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks, "The Accelerator Store: A Shared Memory Framework For Accelerator-Based Systems," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 4, pp. 1–22, Jan. 2012.

[19] J. Cong, V. Sarkar, G. Reinman, and A. Bui, "Customizable Domain-Specific Computing," *IEEE Design and Test of Computers*, vol. 28, no. 2, pp. 6–15, Mar. 2011.

[20] P. Schaumont and I. Verbauwhede, "Domain-specific codesign for embedded security," *Computer*, vol. 36, no. 4, pp. 68–74, Apr. 2003.

[21] "ITRS 2007 System Driver," Tech. Rep. [Online]. Available: http://www.itrs.net

[22] M. Lin, A. El Gamal, Y.-C. Lu, and S. Wong, "Performance Benefits of Monolithically Stacked 3-D FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 681–229, Feb. 2007.

[23] C. Chen, H.-S. P. Wong, S. Mitra, R. Parsa, N. Patil, S. Chong, K. Akarvardar, J. Provine, D. Lewis, J. Watt, and R. T. Howe, "Efficient FPGAs using Nanoelectromechanical Relays," in *International Symposium on FPGAs*, 2010, pp. 273–282.

[24] E. Ahmed and J. Rose, "The Effect of LUT and ClusterSize on Deep-Submicron FPGA Performance and Density," *IEEE Transactions on VLSI Systems*, vol. 12, no. 3, pp. 288–298, Mar. 2004.

[25] C. Dong, D. Chen, S. Haruehanroengra, and W. Wang, "3-D nFPGA: A Reconfigurable Architecture for 3-D CMOS/Nanomaterial Hybrid Digital Circuits," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 11, pp. 2489–2501, Nov. 2007.

[26] F. Li, Y. Lin, L. He, D. Chen, and J. Cong, "Power modeling and characteristics of field programmable gate arrays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 11, pp. 1712–1724, Nov. 2005.

[27] P. Khalili Amiri, Z. M. Zeng, P. Upadhyaya, G. Rowlands, H. Zhao, I. N. Krivorotov, J.-P. Wang, H. W. Jiang, J. A. Katine, J. Langer, K. Galatsis, and K. L. Wang, "Low Write-Energy Magnetic Tunnel Junctions for High-Speed Spin-Transfer-Torque MRAM," *IEEE Electron Device Letters*, vol. 32, no. 1, pp. 57–59, Jan. 2011.

[28] A. L. Lacaita and D. J. Wouters, "Phase-change memories," *Physica Status Solidi (a)*, vol. 205, no. 10, pp. 2281–2297, Oct. 2008.

[29] W. W. Jang, J. O. Lee, J.-B. Yoon, M.-S. Kim, J.-M. Lee, S.-M. Kim, K.-H. Cho, D.-W. Kim, D. Park, and W.-S. Lee, "Fabrication and characterization of a nanoelectromechanical switch with 15-nm-thick suspension air gap," *Applied Physics Letters*, vol. 92, no. 10, p. 103110, 2008.

[30] R. Huang, L. Zhang, D. Gao, Y. Pan, S. Qin, P. Tang, Y. Cai, and Y. Wang, "Resistive switching of silicon-rich-oxide featuring high compatibility with CMOS technology for 3D stackable and embedded applications," *Applied Physics A*, pp. 927–931, Mar. 2011.

[31] S. Paul, S. Mukhopadhyay, and S. Bhunia, "Hybrid CMOS-STTRAM non-volatile FPGA: Design challenges and optimization approaches," in *International Conference on Computer-Aided Design (ICCAD)*, Nov. 2008, pp. 589–592.

[32] N. Bruchon, L. Torres, G. Sassatelli, and G. Cambon, "New non-volatile FPGA concept using Magnetic Tunneling Junction," in *ISVLSI*, 2006, pp. 269–276.

[33] P.-E. Gaillardon, M. H. Ben-Jamaa, M. Reyboz, G. B. Beneventi, F. Clermidy, L. Perniola, and I. O'Connor, "Phase-change-memory-based storage elements for configurable logic," in *International Conference on Field-Programmable Technology (FPT)*, Dec. 2010, pp. 17–20.

[34] Y. Chen, J. Zhao, and Y. Xie, "3D-nonFAR: Three-Dimensional Non-Volatile FPGA ARchitecture Using Phase Change Memory," in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2010, p. 55.

[35] R. Chakraborty, S. Paul, Y. Zhou, and S. Bhunia, "Low-power hybrid complementary metal-oxide-semiconductor-nano-electro-mechanical systems field programmable gate array: circuit level analysis and defect-aware mapping," *IET Computers and Digital Techniques*, vol. 3, no. 6, pp. 609–624, 2009.

[36] Y. Y. Liauw, Z. Zhang, W. Kim, A. E. Gamal, and S. S. Wong, "Nonvolatile 3D-FPGA with monolithically stacked RRAM-based configuration memory," in *International Solid-State Circuits Conference (ISSCC)*, Feb. 2012, pp. 406–408.

[37] O. Turkyilmaz, S. Onkaraiah, M. Reyboz, F. Clermidy, C. Hraziia, J. Portal, and M. Bocquet, "RRAM-based FPGA for Normally Off, Instantly On Applications," in *International Symposium on Nanoscale Architectures (NANOARCH)*, 2012, pp. 101–108.

[38] P.-E. Gaillardon, M. Haykel Ben-Jamaa, G. Betti Beneventi, F. Clermidy, and L. Perniola, "Emerging memory technologies for reconfigurable routing in FPGA architecture," in *International Conference on Electronics, Circuits and Systems (ICECS)*, Dec. 2010, pp. 62–65.

[39] S. Onkaraiah, P.-e. Gaillardon, M. Reyboz, F. Clermidy, J.-m. Portal, M. Bocquet, and C. Muller, "Using OxRRAM memories for improving communications of reconfigurable FPGA architectures," in *International Symposium on Nanoscale Architectures (NANOARCH)*, Jun. 2011, pp. 65–69.

[40] P.-e. Gaillardon, D. Sacchetto, G. B. Beneventi, M. H. Ben Jamaa, L. Perniola, F. Clermidy, I. OConnor, and G. De Micheli, "Design and Architectural Assessment of 3-D Resistive Memory Technologies in FPGAs," *IEEE Transactions on Nanotechnology*, vol. 12, no. 1, pp. 40–50, Jan. 2013.

[41] G. S. Snider and R. S. Williams, "Nano/CMOS architectures using a field-programmable nanowire interconnect," *Nanotechnology*, vol. 18, no. 3, p. 035204, Jan. 2007.

[42] Q. Xia, W. Robinett, M. W. Cumbie, N. Banerjee, T. J. Cardinali, J. J. Yang, W. Wu, X. Li, W. M. Tong, D. B. Strukov, G. S. Snider, G. Medeiros-Ribeiro, and R. S. Williams, "Memristor-CMOS hybrid integrated circuits for reconfigurable logic." *Nano letters*, vol. 9, no. 10, pp. 3640–5, Oct. 2009. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/19722537

[43] B. Xiao, "FPGA-RR: A Novel FPGA Architecture with RRAM-Based Reconfigurable Interconnects," Master Thesis, University of California, Los Angeles, 2012.

[44] J. Cong and B. Xiao, "FPGA-RPI: A Novel FPGA Architecture With RRAM-Based Programmable Interconnects," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 4, pp. 864–877, Apr. 2014.

[45] J. Cong and B. Xiao, "mrFPGA: A Novel FPGA Architecture with Memristor-Based Reconfiguration," in *International Symposium on Nanoscale Architectures (NANOARCH)*, Jun. 2011, pp. 1–8.

[46] W. N. N. Hung, C. Gao, X. Song, and D. Hammerstrom, "Defect-Tolerant CMOL Cell Assignment via Satisfiability," *IEEE Sensors Journal*, vol. 8, no. 6, pp. 823–830, Jun. 2008.

[47] Y. Su and W. Rao, "Defect-Tolerant Logic Implementation onto Nanocrossbars by Exploiting Mapping and Morphing Simultaneously," in *International Conference on Computer-Aided Design (ICCAD)*, 2011, pp. 456–462.

[48] W.-J. Huang and E. J. Mccluskey, “Column-Based Precompiled Configuration Techniques for FPGA Fault Tolerance,” in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2001, pp. 137–146.

[49] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, “Efficiently Supporting Fault-Tolerance in FPGAs,” in *Itnernational Symposium on FPGAs*, 1998, pp. 105–115.

[50] A. Agarwal, J. Cong, and B. Tagiku, “Fault Tolerant Placement and Defect Reconfiguration for nano-FPGAs,” in *International Conference on Computer-Aided Design (ICCAD)*, Nov. 2008, pp. 714–721.

[51] R. Rubin and A. Dehon, “Choose-your-own-adventure routing,” *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, no. 4, pp. 1–24, Dec. 2011.

[52] L. Mcmurchie and C. Ebeling, “PathFinder : A Negotiation-Based Performance-Driven Router for FPGAs,” in *International Symposium on FPGAs*, 1995, pp. 111–117.

[53] J. Cong and B. Xiao, “Defect Tolerance in Nanodevice-Based Programmable Interconnects: Utilization Beyond Avoidance,” in *Design Automation Conference (DAC)*, 2013.

[54] H. Franke, J. Xenidis, C. Basso, B. M. Bass, S. S. Woodward, J. D. Brown, and C. L. Johnson, “Introduction to the wire-speed processor and architecture,” *IBM Journal of Research and Development*, vol. 54, no. 1, pp. 3:1–3:11, Jan. 2010.

[55] L. Seiler, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, and J. Sugerman, “Larrabee: A Many-Core x86 Architecture for Visual Computing,” *ACM Transactions on Graphics*, vol. 27, no. 3, p. 1, Aug. 2008.

[56] C. Huang and F. Vahid, “Dynamic coprocessor management for FPGA-enhanced compute platforms,” in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2008, p. 71.

[57] L. Bauer, M. Shafique, and J. Henkel, “Run-Time Instruction Set Selection in a Transmutable Embedded Processor,” in *Design Automation Conference (DAC)*, 2008, pp. 56–61.

[58] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, “Architecture Support for Accelerator-Rich CMPs,” in *Design Automation Conference*, 2012, pp. 843–849.

[59] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, “Understanding sources of inefficiency in general-purpose chips,” *International Symposium on Computer Architecture*, p. 37, 2010.

[60] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, “Conservation cores: reducing the energy of mature computations,” in *Architectural Support for Programming Languages and Operating Systems*, Mar. 2010, pp. 205–218.

[61] J. Cong, M. A. Ghodrat, M. Gill, C. Liu, and G. Reinman, “BiN: A Buffer-in-NUCA Scheme for Accelerator-Rich CMPs,” in *International Symposium on Low Power Electronics and Design*, 2012, pp. 225–230.

[62] L. Bauer, M. Shafique, and J. Henkel, “A computation- and communication- infrastructure for modular special instructions in a dynamically reconfigurable processor,” in *International Conference on Field Programmable Logic and Applications (FPL)*, 2008, pp. 203–208.

[63] J. Cong and B. Xiao, “Optimization of Interconnects Between Accelerators and Shared Memories in Dark Silicon,” in *International Conference on Computer-Aided Design (ICCAD)*, 2013.

[64] Y.-t. Chen, J. Cong, M. A. Ghodrat, M. Huang, C. Liu, B. Xiao, and Y. Zou, “Accelerator-Rich CMPs: From Concept to Real Hardware,” in *International Conference on Computer Design*, 2013.

[65] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, “High-Level Synthesis for FPGAs: From Prototyping to Deployment,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[66] J. Cong, P. Zhang, and Y. Zou, “Optimizing memory hierarchy allocation with loop transformations for high-level synthesis,” in *Design Automation Conference*, 2012, pp. 1229–1234.

[67] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, “Polyhedral-based data reuse optimization for configurable computing,” *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays - FPGA ’13*, p. 29, 2013.

[68] Y. Wang, P. Zhang, C. Xu, and J. Cong, “An integrated and automated memory optimization flow for FPGA behavioral synthesis,” in *Asia and South Pacific Design Automation Conference*, Jan. 2012, pp. 257–262.

[69] Y. Ben-Asher and N. Rotem, “Automatic memory partitioning,” in *International Conference on Hardware/Software Codesign and System Synthesis*, 2010, p. 155.

[70] S. H. Fuller and L. I. Millett, *The Future of Computing Performance: Game Over of Next Level*. Washington, D.C.: The National Academies Press, 2011.

[71] S. L. Graham, M. Snir, and P. Cynthia A., *Getting Up to Speed: The Future of Supercomputing*. Washington, D.C.: The National Academies Press, 2004.

[72] Xilinx, “Vivado High-Level Synthesis.” [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/index.htm

[73] J. Cong, P. Li, B. Xiao, and P. Zhang, “An Optimal Microarchitecture for Stencil Computation Acceleration Based on Non-Uniform Partitioning of Data Reuse Buffers,” in *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference - DAC ’14*, 2014, pp. 1–6.

[74] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, vol. 86, no. 11, 1998, pp. 2278–2324.

[75] P. Simard, D. Steinkraus, and J. Platt, "Best practices for convolutional neural networks applied to visual document analysis," in *International Conference on Document Analysis and Recognition (ICDAR)*, vol. 1, 2003, pp. 958–963.

[76] S. Behnke, "Hierarchical Neural Networks for Image Interpretation," in *volume 2776 of Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[77] M. Osadchy, L. C. Yann, and L. M. Matthew, "Synergistic Face Detection and Pose Estimation with Energy-Based Models," *Journal of Machine Learning Research*, vol. 8, pp. 1197–1215, 2007.

[78] "ImageNet Contest 2012." [Online]. Available: http://www.image-net.org/challenges/LSVRC/2012/index

[79] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "CNP: An FPGA-based processor for Convolutional Networks," in *International Conference on Field Programmable Logic and Applications*, vol. 1, no. 1, Aug. 2009, pp. 32–37.

[80] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *International symposium on Computer architecture*, 2010, p. 247.

[81] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for Convolutional Neural Networks," in *International Conference on Computer Design*, Oct. 2013, pp. 13–19.

[82] J. Cong and B. Xiao, "Minimizing Computation in Convolutional Neural Networks," in *Artificial Neural Networks and Machine Learning ICANN 2014*, ser. Lecture Notes in Computer Science, S. Wermter, C. Weber, W. Duch, T. Honkela, P. Koprinkova-Hristova, S. Magg, G. Palm, and A. E. P. Villa, Eds., vol. 8681. Springer International Publishing, 2014, pp. 281–290.

[83] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *International Symposium on Computer Architecture*. Ieee, Jun. 2014, pp. 13–24.

[84] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, p. 107, Jan. 2008.

[85] "Apache Hadoop." [Online]. Available: hadoop.apache.org

[86] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark : Cluster Computing with Working Sets,” in *USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[87] J. H. Yeung, C. Tsang, K. Tsoi, B. S. Kwan, C. C. Cheung, A. P. Chan, and P. H. Leong, “Map-reduce as a Programming Model for Custom Computing Machines,” in *International Symposium on Field-Programmable Custom Computing Machines*, Apr. 2008, pp. 149–159.

[88] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, “FPMR: MapReduce framework on FPGA,” in *International Symposium on Field Programmable Gate Arrays*, 2010, pp. 93–102.

[89] K. H. Tsoi and W. Luk, “Axel: a heterogeneous cluster with FPGAs and GPUs,” in *International Symposium on Field Programmable Gate Arrays*, 2010, pp. 115–124.

[90] D. Yin, G. Li, and K.-d. Huang, “Scalable MapReduce Framework on FPGA,” in *Lecture Notes in Computer Science*, S. Andreev, S. Balandin, and Y. Koucheryavy, Eds. Springer Berlin Heidelberg, 2012, pp. 280–294.

[91] Z. Lin and P. Chow, “ZCluster: A Zynq-based Hadoop cluster,” in *International Conference on Field-Programmable Technology*, Dec. 2013, pp. 450–453.

[92] Xilinx, “Zynq-7000 All Programmable So.” [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/

[93] G. Lemieux and D. Lewis, “Circuit design of routing switches,” in *International Symposium on FPGAs*, 2002, pp. 19–28.

[94] I. Kuon and J. Rose, “Measuring the Gap Between FPGAs and ASICs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, Feb. 2007.

[95] I. Kuon, R. Tessier, and J. Rose, “FPGA Architecture: Survey and Challenges,” *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 2, pp. 135–253, 2007.

[96] R. Nathanael, V. Pott, H. Kam, J. Jeon, and T.-J. K. Liu, “4-Terminal Relay Technology for Complementary Logic,” in *International Electron Devices Meeting (IEDM)*. Ieee, Dec. 2009, pp. 1–4.

[97] M.-J. Lee, C. B. Lee, D. Lee, S. R. Lee, M. Chang, J. H. Hur, Y.-B. Kim, C.-J. Kim, D. H. Seo, S. Seo, U.-I. Chung, I.-K. Yoo, and K. Kim, “A fast, high-endurance and scalable non-volatile memory device made from asymmetric Ta(2)O(5-x)/TaO(2-x) bilayer structures.” *Nature materials*, vol. 10, no. 8, pp. 625–30, Jan. 2011.

[98] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, “Phase Change Memory,” *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec. 2010.

[99] Y.-W. Chang, D. F. Wong, and C. K. Wong, "Universal switch modules for FPGA design," *ACM Transactions on Design Automation of Electronic Systems*, vol. 1, no. 1, pp. 80–101, Jan. 1996.

[100] Q. Shen, Y. Chen, L. Wang, and J. Tong, "FPGA Routing Architecture Optimization," *International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, pp. 1934–1936, 2006.

[101] Xilinx, "Virtex-6 FPGA Configurable Logic Block User Guide." [Online]. Available: http://www.xilinx.com/support/documentation/virtex-6.htm

[102] G. Lemieux, E. Lee, M. Tom, and A. Yu, "Directional and single-driver wires in FPGA interconnect," *International Conference on Field- Programmable Technology*, pp. 41–48, 2004.

[103] Xilinx, "Virtex-6 Family Overview." [Online]. Available: www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[104] I. Dobbelaere, M. Horowitz, and A. El Gamal, "Regenerative feedback repeaters for programmable interconnections," *IEEE Journal of Solid-State Circuits*, vol. 30, no. 11, pp. 1246–1253, 1995.

[105] "International Technology Roadmap for Semiconductors," 2011. [Online]. Available: http://www.itrs.net/Links/2011ITRS/Home2011.htm

[106] L. van Ginneken, "Buffer placement in distributed RC-tree networks for minimal Elmore delay," in *International Symposium on Circuits and Systems (ISCAS)*, 1990, pp. 865–868.

[107] J. Cong and D. Pan, "Buffer block planning for interconnect-driven floorplanning," in *International Conference on Computer-Aided Design (ICCAD)*, 1999, pp. 358–363.

[108] C. Alpert, S. Sapatnekar, and P. Villarrubia, "A practical methodology for early buffer and wire resource allocation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 5, pp. 573–583, May 2003.

[109] "VPR 5.0." [Online]. Available: http://www.eecg.utoronto.ca/vpr/

[110] W. Zhao and Y. Cao, "Predictive Technology Model (PTM) website." [Online]. Available: http://ptm.asu.edu/

[111] W. Zhao and Y. Cao, "New Generation of Predictive Technology Model for Sub-45nm Design Exploration," in *International Symposium on Quality Electronic Design (ISQED)*, 2006, pp. 585–590.

[112] S. Yang, "Logic synthesis and optimization benchmarks, version 3.0," MCNC, Tech. Rep., 1991.

[113] R. Y. Rubin and A. M. DeHon, "Timing-Driven Pathfinder Pathology and Remediation: Quantifying and Reducing Delay Noise in VPR-Pathfinder," in *International Symposium on FPGAs*, 2011, pp. 173–176.

[114] Xilinx, "Virtex-4 FPGA data sheets," Tech. Rep. [Online]. Available: http://www.xilinx.com/support/documentation/virtex-4.htm

[115] Xilinx, "Xilinx: EasyPath series overview." [Online]. Available: http://www.xilinx.com/products/silicon-devices/fpga/easypath-7/index.htm

[116] Z. Hyder and J. Wawrzynek, "Defect tolerance in multiple-FPGA systems," in *International Conference on Field Programmable Logic and Applications*, vol. 153, no. 3, 2006, pp. 247–254.

[117] N. Campregher, P. Y. K. Cheung, G. a. Constantinides, and M. Vasilko, "Yield enhancements of design-specific FPGAs," *International Symposium on FPGAs*, pp. 93–100, 2006.

[118] F. Hatori, T. Sakurai, K. Nogami, K. Sawada, M. Takahashi, M. Ichida, M. Uchida, I. Yoshii, Y. Kawahara, T. Hibi, Y. Saeki, H. Muroga, A. Tanaka, and K. Kanzaki, "Introducing Redundancy in Field Programmable Gate Arrays," in *Custom Integrated Circuits Conference (CICC)*, 1993, pp. 7.1.1–7.1.4.

[119] A. Yu and G. Lemieux, "FPGA Defect Tolerance: Impact of Granularity," in *International Conference on Field-Programmable Technology (FPT)*, 2005, pp. 189–196.

[120] N. Mehta, R. Rubin, and A. DeHon, "Limit Study of Energy and Delay Benefits of Component-Specific Routing," in *International Symposium on FPGAs*, 2012, pp. 97–106.

[121] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983.

[122] Xilinx, "Vivado Analytical Place and Route." [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/implementation/place-and-route/index.htm

[123] Altera, "Quartus II University Interface Program." [Online]. Available: http://www.altera.com/education/univ/research/quip/unv-quip.html

[124] J. Pistorius, M. Hutton, A. Mishchenko, and R. Brayton, "Benchmarking Method and Designs Targeting Logic Synthesis for FPGAs," in *International Workshop on Logic and Synthesis (IWLS)*, 2007.

[125] A. Carpenter, J. Hu, O. Kocabas, M. Huang, and H. Wu, "Enhancing effective throughput for transmission line-based bus," in *International Symposium on Computer Architecture*, vol. 40, no. 3, Sep. 2012, pp. 165–176.

[126] J. Balfour and W. J. Dally, "Design tradeoffs for tiled CMP on-chip networks," in *International Conference on Supercomputing*, 2006, pp. 187 – 198.

[127] J. Cong, Y. Huang, and B. Yuan, "ATree-based topology synthesis for on-chip network," in *International Conference on Computer-Aided Design*, Nov. 2011, pp. 651–658.

[128] R. Das, S. Eachempati, A. K. Mishra, V. Narayanan, and C. R. Das, "Design and evaluation of a hierarchical on-chip interconnect for next-generation CMPs," in *International Symposium on High Performance Computer Architecture*, Feb. 2009, pp. 175–186.

[129] A. Bui, J. Cong, L. Vese, and Y. Zou, "Platform characterization for Domain-Specific Computing," *Asia and South Pacific Design Automation Conference*, pp. 94–99, Jan. 2012.

[130] T. Henretty, J. Holewinski, N. Sedaghati, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Stencil Domain Specific Language (SDSL) User Guide 0.2.1 draft," OSU TR OSU-CISRC-4/13-TR09, Tech. Rep., 2013.

[131] A. A. Nacci, V. Rana, F. Bruschi, D. Sciuto, I. Beretta, and D. Atienza, "A high-level synthesis flow for the implementation of iterative stencil loop algorithms on FPGA devices," in *Design Automation Conference*, 2013, p. 1.

[132] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *International Symposium on FPGAs*, 2013, p. 9.

[133] P. Feautrier, "Some efficient solutions to the affine scheduling problem. I. One-dimensional time," *International Journal of Parallel Programming*, vol. 21, no. 5, pp. 313–347, Oct. 1992.

[134] J. Cong, P. Li, B. Xiao, and P. Zhang, "An Optimal Microarchitecture for Stencil Computation Acceleration Based on Non-Uniform Partitioning of Data Reuse Buffers," Computer Science Department, UCLA, TR140009, Tech. Rep., 2014. [Online]. Available: http://fmdb.cs.ucla.edu/Treports/140009.pdf

[135] J. Cong, H. Huang, M. Huang, B. Xiao, and P. Zhang, "CMOST: A System-Level FPGA Compilation Framework," in *Design Automation Conference*, 2015.

[136] "LLVM-polly." [Online]. Available: http://llvm.org/svn/llvm-project/polly/

[137] "ROSE compiler infrastucture." [Online]. Available: http://rosecompiler.org/

[138] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, no. 4, pp. 354–356, Aug. 1969.

[139] K. Chellapilla, S. Puri, and P. Simard, "High Performance Convolutional Neural Networks for Document Processing," in *International Workshop on Frontiers in Handwriting Recognition*, 2006.

[140] D. Irony, S. Toledo, and A. Tiskin, "Communication lower bounds for distributed-memory matrix multiplication," *Journal of Parallel and Distributed Computing*, vol. 64, no. 9, pp. 1017–1026, Sep. 2004.

[141] J. Cong and Y. Zou, "Lithographic aerial image simulation with FPGA-based hardware acceleration," in *International Symposium on Field Programmable Gate Arrays*, vol. 2, no. 3, 2008, p. 67.

[142] E. Sotiriades, C. Kozanitis, and A. Dollas, "FPGA based architecture for DNA sequence comparison and database search," in *International Parallel and Distributed Processing Symposium*, 2006.

[143] J. Cassidy, L. Lilge, and V. Betz, "Fast, Power-Efficient Biophotonic Simulations for Cancer Treatment Using FPGAs," in *International Symposium on Field-Programmable Custom Computing Machines*, May 2014, pp. 133–140.

[144] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks: An Analytical Approach based on Roofline Model," in *International Symposium on FPGAs*, 2015.

[145] T. Bollaert, "Catapult synthesis: A practical introduction to interactive C synthesis," in *High-Level Synthesis: From Algorithm to Digital Circuit*, P. Coussy and A. Morawiec, Eds. Dordrecht: Springer Netherlands, 2008, pp. 29–52.

[146] "OpenCL Document." [Online]. Available: https://www.khronos.org/opencl/

[147] J. Gong, T. Wang, J. Chen, H. Wu, F. Ye, S. Lu, and J. Cong, "An Efficient and Flexible Host-FPGA PCIe Communication Library," in *International Conference on Field Programmable Logic and Applications*, 2014.

[148] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Symposium on Mass Storage Systems and Technologies*, May 2010, pp. 1–10.

[149] "DeepLearning 0.1 Documentation." [Online]. Available: http://deeplearning.net/tutorial/contents.html