On Multicast in Asynchronous Networks-on-Chip: Techniques, Architectures, and FPGA Implementation

Kshitij Bhardwaj

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2018

ABSTRACT

On Multicast in Asynchronous Networks-on-Chip: Techniques, Architectures, and FPGA Implementation

Kshitij Bhardwaj

In this era of exascale computing, conventional synchronous design techniques are facing unprecedented challenges. The consumer electronics market is replete with many-core systems in the range of 16 cores to thousands of cores on chip, integrating multi-billion transistors. However, with this ever increasing complexity, the traditional design approaches are facing key issues such as increasing chip power, process variability, aging, thermal problems, and scalability.

An alternative paradigm that has gained significant interest in the last decade is asynchronous design. Asynchronous designs have several potential advantages: they are naturally *energy proportional,* burning power only when active, do not require complex clock distribution, are robust to different forms of variability, and provide ease of composability for heterogeneous platforms.

Networks-on-chip (NoCs) is an interconnect paradigm that has been introduced to deal with the ever-increasing system complexity. NoCs provide a distributed, scalable, and efficient interconnect solution for today's many-core systems. Moreover, NoCs are a natural match with asynchronous design techniques, as they separate communication infrastructure and timing from the computational elements. To this end, globally-asynchronous locally-synchronous (GALS) systems that interconnect multiple processing cores, operating at different clock speeds, using an asynchronous NoC, have gained significant interest.

While asynchronous NoCs have several advantages, they also face a key challenge of supporting new types of traffic patterns. Once such pattern is *multicast communication,* where a source sends packets to arbitrary number of destinations. Multicast is not only common in parallel computing, such as for cache coherency, but also for emerging areas such as neuromorphic computing. This important capability has been largely missing from asynchronous NoCs.

This thesis introduces several efficient multicast solutions for these interconnects. In particular, techniques, and network architectures are introduced to support high-performance and low-power multicast. Two leading network topologies are the focus: a variant mesh-of-trees (MoT) and a 2D mesh. In addition, for a more realistic implementation and analysis, as well as significantly advancing the field of asynchronous NoCs, this thesis also targets synthesis of these NoCs on commercial FPGAs. While there has been significant advances in FPGA technologies, there has been only limited research on implementing asynchronous NoCs on FPGAs. To this end, a systematic computer-aided design (CAD) methodology has been introduced to efficiently and safely map asynchronous NoCs on FPGAs. Overall, this thesis makes the following three contributions.

The first contribution is a multicast solution for a variant MoT network topology. This topology consists of simple low-radix switches, and has been used in high-performance computing platforms. A novel *local speculation* technique is introduced, where a subset of the network's switches are speculative that *always broadcast* every packet. These switches are very simple and have high performance. Speculative switches are surrounded by non-speculative ones that route packets based on their destinations and also throttle any redundant copies created by the former. This hybrid network architecture achieved significant performance and power benefits over other multicast approaches.

The second contribution is a multicast solution for a 2D-mesh topology, which is more complex with higher-radix switches and also is more commonly used. A novel *continuous-time replication strategy* is introduced to optimize the critical multi-way forking operation of a multicast transmission. In this technique, a multicast packet is first stored in an input port of a switch, from where it is sent through distinct output ports towards different destinations concurrently, at each output's own rate and in continuous time. This strategy is shown to have significant latency and energy benefits over an approach that performs multicast using multiple distinct serial unicasts to each destination.

Finally, a systematic CAD methodology is introduced to synthesize asynchronous NoCs on commercial FPGAs. A two-fold goal is targeted: correctness and high performance. For ease of implementation, only existing FPGA synthesis tools are used. Moreover, since asynchronous NoCs involve special asynchronous components, a comprehensive guide is introduced to map these elements correctly and efficiently. Two asynchronous NoC switches are synthesized using the proposed approach on a leading *Xilinx FPGA in 28 nm:* one that only handles unicast, and the other that also supports multicast. Both showed significant energy benefits with some performance gains over a state-of-the-art synchronous switch.

Table of Contents

List of Figures **vi**

List of Tables **x**

1 Introduction **1**

1.1 Synchronous Design: Challenges 3

1.2 Asynchronous Design: An Alternative Paradigm 5

1.2.1 Advantages of Asynchronous Design 5

1.2.2 Challenges with Asynchronous Design 6

1.2.3 History and Overview of Recent Success 7

1.3 Networks-on-Chip: An Introduction 10

1.3.1 NoCs: Motivation, Basics, and Advantages 10

1.3.2 Advances in Synchronous NoCs 12

1.3.3 Advances in Asynchronous NoCs 14

1.4 Multicast Communication and its Applications 16

1.5 FPGAs: Architecture, and Applications 18

1.5.1 FPGA Architecture 18

1.5.2 FPGA Applications 20

1.6 Research Focus 20

1.6.1 Challenges with Supporting Multicast in Asynchronous NoCs 21

1.6.2 Challenges with Implementing Asynchronous NoCs on FPGAs 22

1.7 Contribution of Thesis 22

1.8 Organization of Thesis 24

2 Background: Asynchronous Design **26**
2.1 Handshaking Protocols 26
2.1.1 Four-Phase Protocol 27
2.1.2 Two-Phase Protocol 27
2.1.3 Trade-Offs 28
2.2 Data Encoding Schemes 28
2.2.1 Delay-Insensitive (DI) Codes 28
2.2.2 Single-Rail Bundled Data 29
2.2.3 Trade-Offs 29
2.3 Special Asynchronous Components 30
2.3.1 The C-Element 30
2.3.2 The Mutex 31
2.3.3 The N-Way Arbiters 32
2.4 Mousetrap Pipelines 34
2.4.1 Mousetrap Structure 34
2.4.2 Mousetrap Operation 34
2.4.3 Timing Constraints 35
2.5 Mixed-Timing Interfaces 35

3 Background: Networks-on-Chip **38**
3.1 Network Topologies 38
3.1.1 A Variant MoT Topology 39
3.1.2 A 2D-Mesh Topology 40
3.2 Routing Algorithms 41
3.2.1 Deterministic Routing 42
3.2.2 Oblivious Routing 42
3.2.3 Adaptive Routing 42
3.3 Packet Encoding Schemes 43
3.4 Synchronous Unicast Router: Micro-Architectures and Performance Optimizations 44
3.4.1 A Traditional 5-Cycle Router 44
3.4.2 Recent Single-Cycle Routers 45

3.4.3 Extreme Bypassing in Single Cycle Using SMART NoCs 46
3.5 Multicast Techniques and Related Work 47
3.5.1 Techniques 47
3.5.2 Related Work 48
3.6 Leading Synchronous Multicast NoCs 49
3.6.1 Multicast Using Single-Cycle Routers 49
3.6.2 Multicast Using SMART NoCs 49

4 A Local Speculation Approach for Multicast in Mesh-of-Trees NoCs 52

4.1 Introduction 52
4.2 Baseline Asynchronous NoC 53
4.2.1 Fanout Node 55
4.2.2 Fanin Node 56
4.2.3 Results 57
4.2.4 Baseline for the New Multicast Research 57
4.3 Proposed Multicast Approaches 58
4.3.1 Simple Tree-Based Multicast 58
4.3.2 Local Speculation-Based Multicast 59
4.3.3 Protocol Optimizations 61
4.3.4 Target Parallel Multicast Networks 62
4.4 Proposed Fanout Node Designs 62
4.4.1 Unoptimized Speculative Fanout Node 62
4.4.2 Unoptimized Non-Speculative Fanout Node 64
4.4.3 Optimized Speculative Fanout Node 68
4.4.4 Optimized Non-Speculative Fanout Node 71
4.5 Experimental Results 76
4.5.1 Experimental Framework 76
4.5.2 Node- and Network-Level Results 78
4.6 Conclusions 84

5 A Continuous-Time Replication Strategy for Multicast in 2D-Mesh NoCs 85

5.1 Introduction 85

5.2 Baseline Asynchronous NoC 86

5.2.1 The Baseline NoC Without VCs 87

5.2.2 Industrial Extension of the Baseline NoC to VCs 92

5.3 New Multicast Approach 96

5.3.1 Tree-Based Parallel Multicast 96

5.3.2 Continuous-Time Replication Strategy 97

5.3.3 Route Computation and Buffering Policy 98

5.3.4 Simulation of Multicast Routing 99

5.3.5 Resource-Dependent Deadlock Avoidance 101

5.4 Design Details: New Input Port Module (IPM) 104

5.4.1 IPM Structure and Operation 104

5.4.2 Route Computation Unit (RCU) 106

5.4.3 CMR Buffer 107

5.4.4 Address Modifier Unit (AMU) 111

5.5 Experimental Setup and Node-level Results 112

5.5.1 Experimental Framework 112

5.5.2 Node-Level Results 114

5.6 Network-level results 117

5.6.1 Multi-Flit Network-Level Results 118

5.6.2 Single-Flit Network-Level Results 127

5.6.3 Analytical comparison with state-of-the-art synchronous multicast NoCs . 128

5.7 Conclusions 131

6 Synthesizing Asynchronous NoCs on FPGAs: a Systematic Methodology 133

6.1 Introduction 133

6.2 Implementing Asynchronous Circuits on FPGAs: Related Work 135

6.3 Mousetrap Pipeline and Timing Requirements of Bundled-Data Circuits: A Brief Background 136

6.3.1 Mousetrap Pipeline 136

6.3.2 Timing Requirements of Bundled-Data Circuits 137

6.4 A CAD Methodology for Bundled-Data Asynchronous Circuits 137

6.4.1 Tool Flow 137

6.4.2 Validation Approach 142

6.4.3 Tool Flow Illustration: A Mousetrap Pipeline 142

6.5 Synthesis of Special Asynchronous Components on FPGAs 145

6.5.1 C-Element 145

6.5.2 Mutex 147

6.5.3 4-Input Arbiter 152

6.6 Case Study: Asynchronous NoC Routers 154

6.6.1 A Brief Recap: Unicast-Only and Multicast Asynchronous Routers 154

6.6.2 Implementing Asynchronous Routers on FPGAs 156

6.7 Experimental Results 158

6.7.1 Experimental Setup 159

6.7.2 Results 159

6.8 Conclusions 163

7 Conclusions and Future Work **165**

7.1 Conclusions 165

7.2 Future Work 168

Bibliography **168**

List of Figures

1.1 A typical synchronous system 3
1.2 An asynchronous system 5
1.3 A GALS system 9
1.4 Network-on-chip structure 12
1.5 Xilinx 7 Series FPGA block view [86] 19
1.6 FPGA programmable logic fabric 20

2.1 A typical asynchronous communication 27
2.2 Two common handshaking protocols 27
2.3 Two widely-used data encoding schemes 29
2.4 The C-element: (a) symbol, (b) a transistor-level design, and (c) a standard-cell based design 30
2.5 Mutex: (a) block-level view, (b) design details [147] 31
2.6 A 4-input arbiter [135] 33
2.7 A 3-stage Mousetrap pipeline [171] 34
2.8 Put/get interfaces for mixed-timing FIFOs: (a) synchronous interfaces, (b) asynchronous interfaces [29] 36
2.9 Basic *sync-sync* FIFO architecture [29] 37

3.1 Original 4×4 MoT topology [6] 39
3.2 A variant mesh-of-trees (MoT) topology, connecting processors to memory modules [7] 40
3.3 A 2D-mesh topology and the details of its nodes 41

3.4 A traditional 5-cycle router micro-architecture [103] 45
3.5 Pipeline stages in single-cycle routers 46
3.6 A SMART NoC structure and operation 47
3.7 Multicast techniques: (a) path-based, (b) tree-based 48
3.8 A single-cycle multicast router architecture [149] 50

4.1 Variant mesh-of-trees (MoT): connecting processors to memory modules . . 54
4.2 Baseline fanout node 55
4.3 Baseline fanin node 56
4.4 New fanout network architectures: (a)-(c) full range for 8x8 MoT, (d) One possible hybrid network for 16x16 MoT 59
4.5 Hybrid network: unicast/multicast simulations 60
4.6 Unoptimized speculative fanout node 63
4.7 Unoptimized non-speculative fanout node 65
4.8 Unoptimized non-speculative fanout node: Input Channel Monitor and Control Unit details 67
4.9 Unoptimized non-speculative fanout node: Ack Module details 67
4.10 Optimized speculative fanout node 70
4.11 Optimized speculative fanout node: control unit details 72
4.12 Optimized speculative fanout node: Ack Module details 72
4.13 Optimized speculative fanout node: Input Channel Monitor and Phase Corrector details 73
4.14 Optimized non-speculative fanout node 74
4.15 Optimized non-speculative fanout node: Control Unit details 75
4.16 Optimized non-speculative fanout node: Phase Corrector details 76
4.17 Contribution trajectory: network latency at 25% saturation load of respective networks 79
4.18 Contribution trajectory: saturated throughput 80
4.19 Contribution trajectory: total network power 82
4.20 Architectural design space exploration: network latency at 25% saturation load of respective networks 83

4.21 Architectural design space exploration: saturated throughput 83
4.22 Architectural design space exploration: total network power 83

5.1 Baseline IPM micro-architecture 88
5.2 Baseline circular FIFO 89
5.3 Baseline OPM micro-architecture 90
5.4 Node structure for double-plane baseline asynchronous router 93
5.5 VC control for an output channel interface 94
5.6 Baseline asynchronous vs. synchronous router: basic comparison for 2 VCs 96
5.7 Asynchronous vs. synchronous router: projected results 96
5.8 New router architecture and tree-based multicast operation 97
5.9 Continuous-time replication strategy operation for a 3-flit packet: highlighting interesting aspects of parallel route computation, independent read operations, and packet-based buffering 100
5.10 Cyclic dependency between multicast packets 102
5.11 Centralized arbitration: an alternative approach to avoid resource-dependent deadlocks for multicast 103
5.12 New IPM micro-architecture with CMR buffer 105
5.13 New route computation unit architecture 106
5.14 CMR buffer: write interface control 109
5.15 CMR buffer: read interface control 111
5.16 Address modifier unit (AMU) 112
5.17 Node-level energy results for different unicast and multicast transmission scenarios 116
5.18 Network latency for unicast and mixed unicast/multicast benchmarks at 25% saturation load of *baseline* 119
5.19 Unicast network latency at varying injection loads 120
5.20 Mixed unicast/multicast latency at varying injection loads 121
5.21 Output throughput at saturation load 124
5.22 Total network energy results measured at 25% saturation load of *baseline* . . 125
5.23 Isolated multicast case study: network latency for corner source 125

5.24 Isolated multicast case study: network latency for center source 126
5.25 Isolated multicast case study: network energy for corner source 126
5.26 Isolated multicast case study: network energy for center source 126
5.27 Network latency for single-flit traffic at 25% saturation load of *baseline* . . . 128
5.28 Saturation throughput for single-flit traffic 128
5.29 Total network energy for single-flit traffic at 25% saturation load of *baseline* 129

6.1 A 3-stage Mousetrap pipeline 137
6.2 Tool flow for implementing bundled-data asynchronous circuits on FPGAs . 141
6.3 Illustrating the proposed tool flow of performance-oriented stage on a Mousetrap pipeline example 143
6.4 Illustrating the proposed tool flow of robustness-oriented stage on a Mousetrap pipeline example 144
6.5 C-element: Standard-cell designs and FPGA mappings 146
6.6 The analog mutex 147
6.7 The digital standard-cell mutex 149
6.8 Validation setup for mapped mutex 152
6.9 A 4-input arbiter 153
6.10 Asynchronous unicast router input port module 155
6.11 Asynchronous unicast/multicast router output port module 156
6.12 Asynchronous multicast router input port module 157
6.13 Resource utilization for the three switch designs 160
6.14 Switch latency for the header and body flits 161
6.15 Switch energy per packet for different unicast and multicast transmissions . 162

List of Tables

4.1 Node-level area and latency comparisons 79

5.1 Node-level area and latency comparisons 115

5.2 Zero-load latency for *all-multicast* and *all-broadcast* 120

5.3 Minimum, average, and maximum multicast delivery time for different benchmarks, averaged over the total number of multicast packets injected for each benchmark. 122

Acknowledgments

I would like to thank many people who have helped me throughout my PhD at Columbia and have made this thesis possible.

First, I would like to thank my advisor and mentor, Steven Nowick, for his guidance and support. I admire Steve's way of conducting research, which he has taught me during my Ph.D.: understanding the fundamentals of a research problem, defining the problem in the simplest possible ways, performing a careful review of the relevant literature, and then coming up with systematic new solutions. Steve has also helped me improve my presentation skills. While new and interesting research ideas are important, presenting them clearly and simply, when writing a paper or giving a talk, is also critical. Steve is an excellent teacher and I have learned a lot from him, both by taking his classes and as his teaching assistant. Thanks Steve for patiently guiding me and making me a better researcher.

I would like to thank other members of my dissertation committee – Simha Sethumadhavan, Luca Carloni, Mingoo Seok and Gennette Gill – for their time, insightful feedback and valuable comments. During my Ph.D., I have approached Simha, Luca and Mingoo several times with my research-related questions, and they were always happy to answer them. I also enjoyed excellent classes from both Simha and Luca that helped me develop solid background in computer architecture and system design. I also had an excellent opportunity to collaborate with Luca on my final project of synthesizing asynchronous NoCs on FPGAs, and his guidance was very helpful.

I would like to thank Ram Krishnamurthy for giving me an opportunity to work at Intel Labs for a 3-month internship. During this time, I enjoyed working with other members of his team: Gregory Chen, Himanshu Kaul, Huseyin Sumbul, Phil Knag, and Raghavan Kumar. The internship was a success and we jointly filed for a US patent on my work.

I also had the pleasure of interning at Cadence Design Systems under the expert guidance of Ping-Sheng Tseng. At Cadence, I enjoyed working with Ping's team: Cheoljoo Jeong (a former student of Steve), Paraminder Sahai, and Rajiv Roy.

I would like to thank my colleagues in the Computer Systems Lab at Columbia. I am thankful for their support and for all the interesting discussions: Weiwei Jiang, Paolo Mantovani, Kanad Sinha, Emilio Cota, Andrea Lottarini, Davide Giri, Tom Repetti, Richard Townsend, Luca Piccolboni, and Yipeng Huang. I really enjoyed working with Paolo on my final FPGA project. I always enjoyed the delicious Indian lunches with Kanad at Doaba Deli!

This thesis would not have been possible without the support from my family. My brother, Kartikeya, a fellow Ph.D. student at CMU and an excellent researcher, was always there to support me and put a smile on my face. Thanks for introducing me to several awesome restaurants of Pittsburgh. My wife, Prachi, also a Ph.D. student at BU, who constantly motivated me. Her love and support have been very valuable and helped me through some difficult times. A special thanks to my parents-in-law and my brother-in-law, who are always there for me.

Finally, none of this would have been possible without my parents' unparalleled guidance, love and blessings. They are always a great source of inspiration and encouragement. My father's interesting stories from his own Ph.D. years motivated me, and his words energized me. My mother's care and affection have always been there, and her cooking was very helpful! Thanks mom and dad!

I would also like to acknowledge some miscellaneous things that played important roles. Since I commuted a lot during my Ph.D., a special thanks to Metro-North for making my train travel (between Poughkeepsie and NYC) a relaxing commute, albeit a slow one. Merritt parkway for some high-speed driving experiences between Boston and NYC. I am also thankful to the hilarious British comedy TV shows - I could rely on them after a long day. Last but not the least, several grants have made this research possible: NSF Grant No. CCF-1527796 and NSF Grant No. CCF-1219013.

This thesis is dedicated to:

My late grandfather, an eminent scholar.

Chapter 1

Introduction

We are in the age of exascale computing [131]. Shrinking transistor sizes have led to ultra-scale integration with the number of transistors on a single chip in the multi-billion range. Intel recently revealed its behemoth 72-core chip called *Knights Landing,* to be used in supercomputers [141]. The consumer market is already replete with many-core processors, ranging from 16 cores to almost thousand cores on chip. Examples include AMD 16-core Opteron 6000, Intel 24-core Xeon-E7, Intel 80-core Xeon Phi, and graphic processors such as AMD FirePro and Nvidia Titan X that have 2500-3500 GPU cores. However, given the ever-increasing scale of integration, conventional design approaches are facing unprecedented challenges: process variability, aging, chip power and thermal challenges, and scalability issues [147].

In the last two decades, asynchronous, or a hybrid mix of asynchronous and synchronous design approaches, has emerged as an alternative paradigm to address the challenges faced by the conventional synchronous or centralized clock-based approaches. Asynchronous designs are naturally *energy proportional,* where they burn dynamic power only when active. These systems also do not require complex clock distribution, are highly modular and support 'object-oriented' design style with ease of composability, and are very robust to process- and environment-induced variability [147]. Asynchronous design is gaining visibility in not just general-purpose processors and embedded systems, but are also suitable for the emerging areas of neuromorphic computing, quantum cellular automata, energy harvesting, and systems used in space missions [147], [148].

Another concept that has seen significant recent interest is networks-on-chip (NoCs), that have effectively replaced traditional bus-based interconnects in today's complex many-core systems.

NoCs have several advantages over the conventional interconnects such as buses or point-to-point networks [22]: (i) they provide a distributed architecture, which can be shared by multiple traffic flows in parallel leading to high performance; (ii) NoCs support modularity by separating communication from computation, thus allowing easy integration of processing cores, possibly operating at different rates; (iii) NoCs are more scalable, where their effective bandwidth scales with size, unlike buses; and (iv) they are also more cost-effective than point-to-point interconnects, which can have large area/power overheads.

In recent years, there has been significant research on combining asynchronous design with networks-on-chip [66], [83], [89], [96], [185]. Asynchronous NoCs can be used not only to interconnect components in a fully-asynchronous system but also for globally-synchronous locally-synchronous (GALS) systems, where synchronous components can be running at different speeds. Asynchronous NoCs are promising due to their several advantages in terms of no clock distribution, no clock switching power, and ease of integration of heterogeneous processing elements.

While asynchronous NoCs provide several advantages, they also face a key challenge of supporting new types of traffic patterns. These NoCs must be able to support communication requirements of advanced parallel computer architectures, new interconnect technologies, such as wireless and photonics, and traffic due to emerging computing paradigms, e.g. neuromorphic computing. One important class is multicast, i.e. 1-to-many communication, which has recently seen growing interest [15].

While there is significant research on supporting multicast in synchronous NoCs [81], [88], [149] [164], [193], this capability has been largely missing from asynchronous NoCs. The aim of this thesis is to significantly advance the field of asynchronous NoCs by designing efficient multicast solutions that (i) exploit potential advantages of asynchronous NoCs in terms of low area/energy footprint while maintaining high performance, and (ii) introduce novel multicast paradigms that are unique to asynchronous, which can lead to lightweight NoC designs. To this end, new multicast strategies are introduced for two leading NoC topologies: simpler variant mesh-of-trees, and more complex but common 2D mesh. *To the best of our knowledge, these are the first general-purpose asynchronous NoCs to support parallel multicast.*

Additionally, for a more realistic analysis and evaluation, and also to advance the field of asynchronous NoCs, this thesis targets synthesis of these NoCs on commercial field-programmable gate

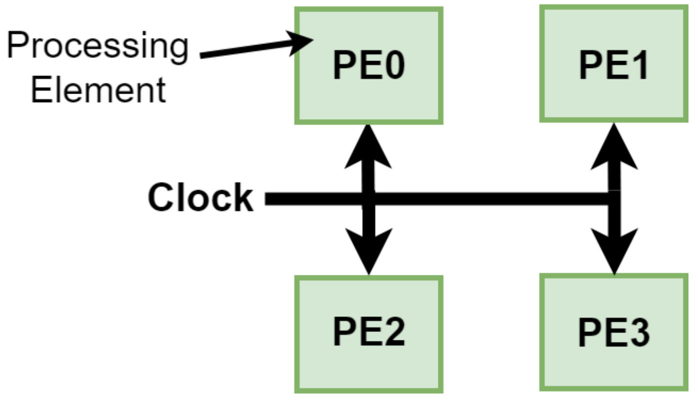

Figure 1.1: A typical synchronous system

arrays (FPGAs). Modern FPGAs not only include standard programmable logic but also consist of multiple cores, accelerators, GPUs and memories. These FPGAs have gained significant importance in hardware-assisted acceleration [151]. Even though there has been major advancement in FPGAs, there has been only limited research on mapping asynchronous NoCs to these devices. *To this end, this thesis proposes the first systematic CAD methodology to efficiently synthesize asynchronous NoCs on commercial FPGAs.* The target is a challenging two-fold goal of not only correctness but also high performance. Only the existing synchronous FPGA synthesis tools are used for ease of implementation. This methodology will enable efficient implementation of asynchronous NoCs on FPGAs to help with their deployment in accelerating real-world applications.

1.1 Synchronous Design: Challenges

Digital systems most commonly use the synchronous design style. In such systems, usually all computing elements communicate with each other using a global clock as shown in Figure 1.1. The clock is used to validate computations, which are performed during a clock cycle, and must be complete by the end of the current cycle. This discrete-time operation greatly simplifies the implementation and verification of digital systems.

The use of global clock in synchronous systems leads to a simple solution but also faces numerous challenges. Modern chips are highly-complex, can integrate thousands of computing cores, accelerators and memories, that perform different functionalities, and have distinct performance and power requirements. Designing such large-scale heterogeneous systems poses major challenges for the globally-synchronous design paradigm, as discussed below.

Clock distribution. Designing a robust clock distribution network for today's highly-complex systems is a major challenge [174]. A single clock distributed to a large chip can suffer from static and dynamic uncertainties, such as skew and jitter, respectively. The generated clock period must include appropriate margins to account for these uncertainties, which can increase as the chip size grows, leading to performance degradations. This problem is further exacerbated when high clock frequencies (upwards of 3 GHz) are used. Moreover, physical design of a global clock distribution network for such chips is also an arduous task, and can have its own challenges.

Clock power dissipation. Modern processor clock frequencies can no longer exceed 3-4 GHz as power consumption has hit a so-called *power wall* [129]. The switching energy of the clock network, even when the system is idle, can lead to significant power dissipation: almost 25-30% of a synchronous chip's power is due to the clock distribution network [173]. Clock power is a major issue for consumer electronics and embedded systems such as smart phones, where it directly impacts the battery life [73]. While techniques such as clock gating can be used to 'turn off' the clock for the inactive components [155], they can also incur extra overhead due to the complex clock-gating circuitry. In particular, clock gating can be applied at two levels: coarse-grain, where entire clock network can be shut down or fine-grain, where individual flip-flops can be selectively enabled or disabled. The former is partially effective when only a portion of the entire chip, connected to the same clock tree, is active [16]. The latter requires adding clock-gating logic at the leaves of a clock network, which has its own challenges pertaining to physical design [87].

Worst-case performance. The operating clock rate of a synchronous system is determined by its slowest component, and are therefore bound to worst-case performance. These systems cannot easily exploit the variation in data-dependent computation times to achieve higher performance.

Interfacing with heterogeneous components. There is a big push towards heterogeneous systems, for applications such as Internet-of-Things (IoT), consisting of different accelerators, cores, and memories. These components usually operate at different clock rates, and will therefore require complex synchronizers for interfacing between the different clock domains, which can lead to extra power, performance and area overheads.

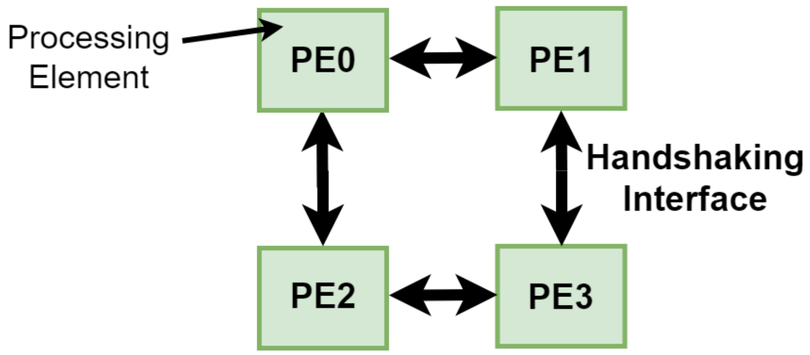

Figure 1.2: An asynchronous system

1.2 Asynchronous Design: An Alternative Paradigm

To address the challenges of the conventional synchronous designs, this thesis targets an alternative design style: asynchronous or clockless design. Figure 1.2 shows a high-level view of an asynchronous system, where the processing elements communicate with each other using local handshaking channels. In the absence of a global clock, these components are free to operate at different speeds.

Asynchronous designs exhibit several potential advantages over the conventional synchronous designs, which have led to a recent surge in the use of asynchronous circuits for various industrial applications. However, these designs also have their own unique challenges that must be addressed by the research community to make asynchronous more mainstream.

1.2.1 Advantages of Asynchronous Design

Asynchronous circuits can exhibit several potential advantages over synchronous in terms of lower power, higher performance, better scalability and design reuse.

Potential lower power. The absence of a clock and the associated clocking circuitry can lead to significant power savings for the asynchronous designs. Asynchronous circuits are also *energy proportional,* where they burn dynamic power only when active [147]. In contrast, as highlighted earlier, synchronous designs consume power even when inactive due to switching of the clock; clock gating can be used to inactivate modules that are not being used, but it can have its own limitations and overheads.

Potentially higher average-case performance. Unlike synchronous, performance of asynchronous systems is not limited by its slowest component, rather it is determined by an average of the operating speeds of different components, which can potentially lead to overall higher performance.

Better scalability and design reuse. Asynchronous systems exhibit high composability, where a larger complex system can be built using a simple aggregation of smaller components. In contrast, adding new modules to a synchronous system requires re-designing clocking networks. In addition some of these components might not be compatible with the target clock frequency, and therefore need to be re-designed as well. These issues do not exist for asynchronous systems. Recently, several large-scale industrial fully-asynchronous or GALS systems have been introduced that exhibit this ease of integration: Intel's asynchronous Ethernet switch [45], STMicroelectronics' P2012 [11], IBM's TrueNorth [2], and Intel's Loihi neuromorphic chip [117].

1.2.2 Challenges with Asynchronous Design

Even though the asynchronous approach has the above advantages, there are some major challenges associated with these designs that must be addressed: hazard-free operation, the lack of CAD tool support, and testing of asynchronous circuits.

Hazard-free designs. In synchronous designs, freedom from hazards is usually not a requirement as long as the result of a computation becomes stable before the start of the next clock cycle. However, in asynchronous designs, since there is no clock, freedom from hazards is therefore a requirement to achieve glitch-free operation. Freedom from hazards must be guaranteed at different levels of the synthesis flow: from two-level/multi-level logic minimization to technology mapping [148].

Lack of CAD support. While small asynchronous designs can be synthesized manually, larger designs will require automation for faster time-to-market, targeting both correctness and efficiency. There is a two-fold goal for developing automated tool flows for asynchronous circuits [148]: (i) these flows must be compatible with the existing synchronous languages and CAD tools, and (ii) new specification languages need to be developed, which can capture the asynchronous-specific aspects: absence of clock, fine-grain concurrency, and distributed synchronization.

Testability of asynchronous circuits. Testing of asynchronous circuits faces unique challenges compared to synchronous designs [148]. A typical testing procedure for synchronous designs, called *single-stepped approach,* involves pausing or slowing down the system, and checking the internal states by comparing with the 'golden' results. However, this testing approach is not possible for asynchronous designs due to the absence of a global clock. In addition, the testing tools for asynchronous designs should not only check for functional correctness but also for hazards, which adds further complications.

1.2.3 History and Overview of Recent Success

There has been a large amount of work in addressing various challenges of asynchronous design as well as exploiting its advantages to develop highly-efficient systems. This section presents a brief history of asynchronous designs, followed by an overview of the research on hazard-free logic synthesis, CAD flows, asynchronous processors, GALS systems, and various commercial applications of asynchronous designs.

Brief history. The early years between 1950s to 1970s mostly saw advent of the classical asynchronous theory by Unger [187], and Muller [143], as well as the use of asynchronous design in many commercial processors (Iliac, Iliac II, Atlas, MU-5) and in LDS-1 graphics system. The mid 1970s to early 1980s did not see much advancement in asynchronous designs and was a period of reduced activity. However, the mid 1980s to late 1990s was a "coming-of-age" era for asynchronous with significant new developments: correct and efficient methodologies for asynchronous controllers [9], [37], [60], [146] and pipeline designs [61], [114], [203], the first academic asynchronous processors from Caltech [124], and University of Manchester [65], and initial commercial uptake by Philips Semiconductor for low-power consumer products [63]. In the modern era, from early 2000s, there has been a surge of recent activity: new CAD tool development [8], [116]; asynchronous networks-on-chip [23], [66], [185]; asynchronous FPGAs [183]; industrial uptake such as from STMicroelectronics [11] and Intel [45]; and applications, for example, emerging areas of neuromorphic computing [2], [46], sensor networks [55], and energy harvesting [34].

Hazard-free logic synthesis. Two widely-used methods for the specification and synthesis of hazard-free asynchronous controllers have been proposed: burst-mode (BM) [145], and Petri-net based [38]. BM is effectively an asynchronous state machine, where state transitions are event-driven: once an "input burst" of one or more signal transitions arrives, the dependent output changes, and the machine advances to the next state. Although a BM specification can leverage existing synchronous approaches for synthesis, it requires a simple hold time requirement, also known as *generalized fundamental mode* timing requirement: no new input burst may arrive until the machine has stabilized from the previous burst. On the other hand, a Petri-net based method uses a directed bipartite graph, in which the nodes represent transitions (i.e. events that may occur) and places (i.e. conditions). Petri-net based controllers only use a quasi-delay insensitive (QDI) timing assumption, which requires that at each wire fanout point, the forked wires have roughly the same delay. Alternatively, *NULL Convention Logic (NCL)*, introduced by Karl Fant, targeted a unified synthesis of both control and datapath, where synchronous netlists are translated to asynchronous threshold gates, such as m-of-n gates [116].

Synthesis CAD flows. Several CAD flows have been developed for asynchronous circuits. The earliest is the *Caltech Synthesis Method* from Alain Martin's group [123], followed by Philips' *Tangram Compiler*, which was used for development of 80C51 microcontrollers, that had several applications with one of most prominent one being for smart card applications [14]. In addition, several automated tool flows have been developed for NCL-based systems, from as early as 2000 [116] to more recent ones [139]. Multiple pipeline optimization frameworks have also been proposed, which use techniques such as slack matching and loop unrolling. One of these pipeline optimization flows is *Proteus,* from 2011, which has been used in the development of Intel's Ethernet switch [8], [45]. Also, around the same period as Proteus, *Tiempo* was introduced, which uses high-level transaction-level modeling (TLM) of SystemVerilog, for specification entry, and then compiles it to a gate-level netlist, followed by synthesizing to layout using commercial synchronous tools [160].

Asynchronous processors. The first modern asynchronous 16-bit RISC processor was developed at Caltech by Martin's group in 1988 [124]. In 1993, another influential processor, Amulet 1, was designed at University of Manchester, which is an asynchronous version of the ARM processors [65]. Since then, there have been several processors that have advanced the field with new pipeline optimizations, cache and memory design, exception handling, speculative operation [148]: TITAC-1/2 [179], Amulet2e and Amulet3i [65], and MiniMIPS [125].

GALS systems. An alternative to fully-asynchronous systems are globally-asynchronous locally-synchronous (GALS) systems. An example GALS system is shown in Figure 1.3, where syn-

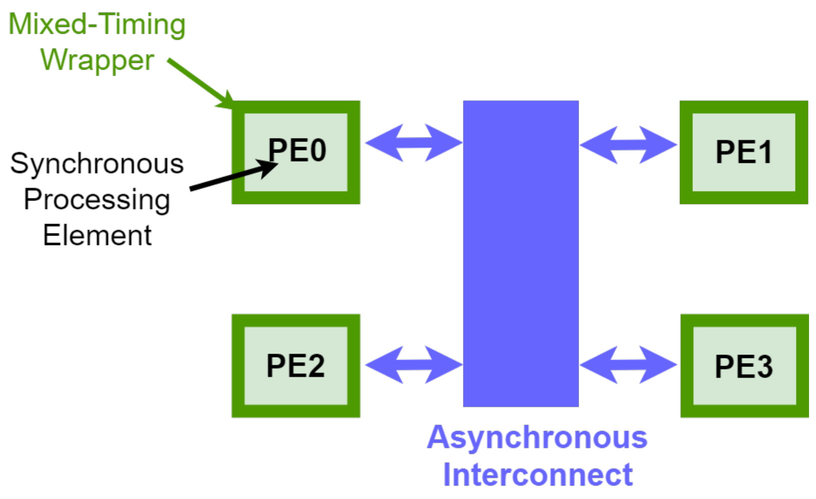

Figure 1.3: A GALS system

chronous computing elements, operating at different clock frequencies, are connected using an asynchronous interconnection network. For correct synchronization between synchronous and asynchronous components, mixed-timing wrappers are required at the sync-async boundaries. The GALS approach eliminates the use of global clock for the entire chip [107], [147], [182].

Commercial applications. Asynchronous designs have been used in several industrial applications.

Philips Semiconductors (now NXP) developed an asynchronous 80C51 microcontroller in the late 1990s to early 2000s [63]. This microcontroller showed 3-4$\times$ lower power as well as lower electromagnetic interference (EMI) noise than its commercial synchronous version. This combined advantage led to the use of these controllers in wide-range of consumer electronics, such as pagers, cell phones, smart cards and digital IDs.

Asynchronous design has also made in-roads in FPGA development. Achronix Semiconductor introduced the Speedster 22i family of FPGAs in the mid 2000s [183]. These high-performance FPGAs used fine-grain bit-level pipelining, and could operate at 1.5 GHz in 22 nm technology.

Intel acquired the asynchronous startup Fulcrum Microsystems in 2011 to develop its industry-leading FM5000/6000 series Ethernet switch chips [45]. This switch supports a 40 Gigabit Ethernet using a fully-asynchronous high-speed crossbar, providing a maximum of 640 Gbps bandwidth with high energy efficiency.

Asynchronous design has also seen an increasing interest in brain-inspired neuromorphic computers, from both IBM [2] and Intel [46], aiming to achieve the high efficiency of the biological brain. These computers follow a non-Von-Neumann architecture, where neurons are the main computing elements, with closely-coupled memories, and these neurons are connected to each other using synapses, modeled as the interconnection framework of the computer. IBM's *TrueNorth* is 5.4 billion transistor neuromorphic GALS chip, which connects 4096 synchronous cores, modeling 1 million neurons and 256 million synapses using a fully-asynchronous NoC. Intel's *Loihi* is a fully-asynchronous 2.1 billion transistors system, comprising 128 cores, each modeling 1024 neurons.

1.3 Networks-on-Chip: An Introduction

Networks-on-Chip (NoCs) are becoming the de facto standard of communication for many-core systems, and are the focus of this thesis. This section presents the motivation behind the rise of NoCs, its basics and advantages, and recent advances in synchronous as well as asynchronous NoCs.

1.3.1 NoCs: Motivation, Basics, and Advantages

System performance and power depend not only on computing efficiency but are also governed by the communication efficacy of the on-chip interconnects [122]. In particular, on-chip interconnects have become the limiting factor to achieve high performance and low power for today's many-core systems due to two reasons: (i) cores for these systems operate on different clock frequencies, and a reliable and efficient interconnect is required to manage interaction between these different timing domains; and (ii) with technology scaling, computational elements and memories have become faster and more energy-efficient but the performance and power of the interconnects has not scaled down. Given these reasons, it is important to consider designing efficient on-chip communication framework as a first-class research problem.

Traditional global buses are often not suitable for today's large-scale many-core processors [122]. In bus-based interconnects, a single bus is shared between multiple processors, graphics cards, memory modules, and accelerators, which is not scalable as the number of units increase, both in terms of performance and power. Due to the centralized architecture, buses have limited support for handling multiple communication flows in parallel. To start a transmission, a sender first requests access to the bus. Since, there can be multiple senders active at one time, arbitration is performed and the bus access is granted to the winner. The winner then broadcasts messages on the bus, which are received by all ‘slave’ units but are accepted only by the intended receiver, and ignored by the others. Such bus-wide broadcasts can have significant power overheads.

Another conventional interconnect is the point-to-point interconnect. These interconnects use dedicated wires connecting each source-destination pair, e.g. a crossbar. This network can lead to high performance but at a cost of significant power and area overheads, and may not scale well with large networks. These interconnects also suffer from significant physical design issues, where routing of these large number of dedicated wires is an arduous task.

As shown in Figure 1.4, NoC provides a distributed communication infrastructure, consisting of switches and channels. Each processing element (PE) is connected to the switch through a network interface (or NI), and the switches are in turn connected to each other using channels or links. The switches and channels are organized in a fixed structure called a topology, which can be of different types, e.g. mesh, torus, ring, etc. [122]. During a transmission, the PE sends a message to the NI, which performs packetization, and converts the message into multiple packets, which are sent to the attached switch. The switches use an underlying routing algorithm to determine the path for routing the packets to the destination switch, traversing intermediate channels. The destination switch sends the received packets to the NI, which converts them back to messages, compatible to the formats used by the PEs, and sends them to the destination PE. In cases where the PEs are operating at a different clock rate than the NoC, the NIs may also use mixed-timing interfaces to synchronize between the NoC and the PEs.

NoCs have several advantages over the traditional interconnects:

- **High performance and energy efficiency.** NoCs provide a shared communication infrastructure, which can be utilized by many traffic flows at the same time. This parallel and distributed operation leads to high performance, without utilizing extra dedicated wiring resources, hence with minimum area and power overheads.

- **Scalability and reliability.** The aggregated bandwidth of the NoCs scales with the network size. In contrast, bandwidth is limited in traditional global buses, which is shared by all the attached units and suffers when the number of units increase. Further, NoCs have regular ar-

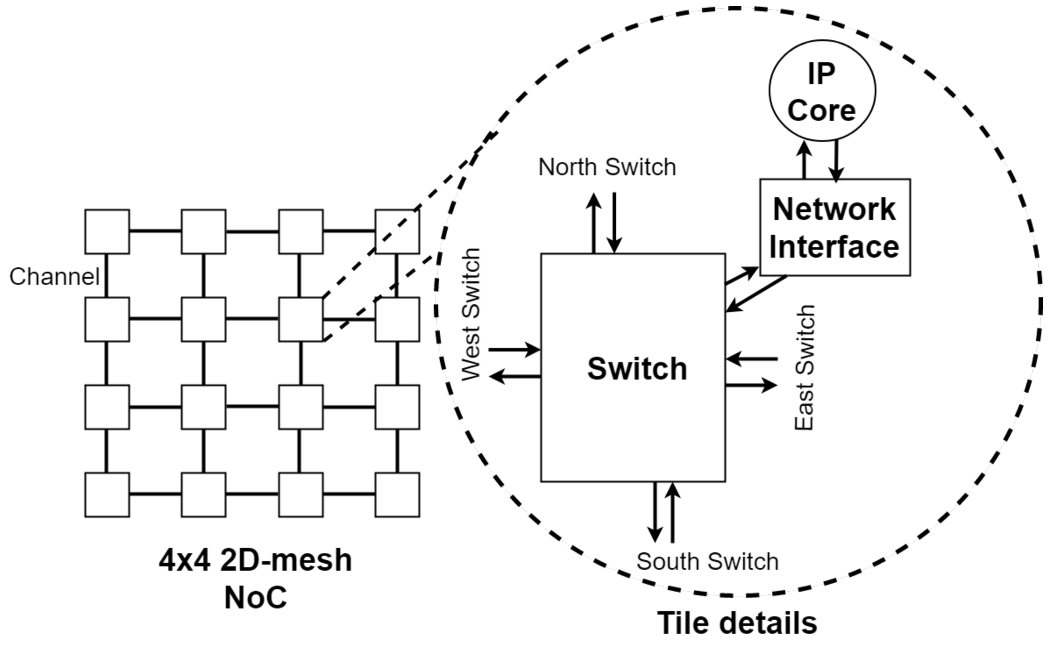

Figure 1.4: Network-on-chip structure

chitectures, with short wires that have controlled and predictable electrical properties, leading to a more reliable operation compared to global long wires.

- **Modularity and ease of integration.** NoCs support modularity by separating communication from computation. They also facilitate design reuse where optimized standard IPs can be simply plugged in, considerably decreasing the design efforts and allowing faster testing and validation, hence improving the overall design cycle.

1.3.2 Advances in Synchronous NoCs

The synchronous design style is the most common approach used for NoCs. The earliest synchronous NoCs were seen in the early 2000s [12], [40], [72], [76], [197]. Since then, there has been much advancement in this field.

Topologies. Many different network topologies have been used, as well as new ones proposed for NoCs. The most common topologies are: mesh [70], [132], torus [204], ring [93], and trees [72]. Some high-radix topologies are also proposed for high-performance computing, such as Dragonfly [99].

Routing Algorithms. There has been significant research on routing algorithms for NoCs. There are two main categories: *deterministic,* where the path taken by a packet is fixed statically [79], [93], [177], [204], and *adaptive routing,* where a packet can dynamically select the best path based on network state such as congestion [69], [72], [80], [132].

Guaranteed service. NoCs also play a critical role in systems with hard real time deadlines in order to deliver packets on time. Such systems require NoCs to support guaranteed service (GS) and multiple service levels [70], [93], [132].

Power and performance optimization. Several power and performance optimization techniques have been introduced for NoCs. To minimize power, novel router architectures have been proposed [192], and techniques such as dynamic voltage and frequency scaling (DVFS) have been used to select the best *V,F* settings depending on channel utilization [169]. To improve performance, optimization techniques such as speculation [150], prediction [127] and bypassing or lookahead [149] have been used within routers. Recently, SMART NoCs were introduced that use *extreme bypassing*, where multiple routers on the correct path are bypassed by a packet in a single clock cycle [104].

Support for new traffic patterns. There has also been significant recent research to support communication patterns common in parallel computing application, such as cache coherency, and emerging areas of deep neural network architectures. These patterns involve multicast (1-to-many) and aggregation (many-to-1) traffic. Several approaches have been proposed to support these patterns while simultaneously achieving high performance with low overheads [106], [81], [88], [106], [193].

Reliability. Fault tolerance and reliability are major concerns for NoCs as feature sizes keep scaling down [22]. Fault-tolerant routing algorithms that route around the faulty components have been introduced [59], [156], [209], some of which can tolerate any number of faults as long as good connections exist [156]. Defect-tolerant router architectures have also been proposed that use error detection codes such as CRC and redundant hardware to improve reliability [36].

Real chips and prototypes. Synchronous NoCs have been used in several academic and industrial chips, which are large-scale, involve many processors, and are used for different applications. Some of the academic prototypes include: TRIPS [71], where a NoC replaced the traditional bus for the applications of operand networks, and more recent 36-core SCORPIO [47] with a mesh-based

NoC to support a scalable snoopy cache coherence protocol. Some of the industrial systems using NoCs are from Tilera [10], Intel [189], and IBM [208].

Emerging technologies. Recently, several high-performance NoCs using emerging technologies such as 3D, wireless, and photonics have been proposed. A 3D chip stacks multiple device layers, connected using vertical interfaces such as Through-Silicon Vias (TSVs) [44]. New 3D NoCs have been proposed to connect processing elements in these chips. These NoCs have many advantages: lower network diameter leading to potentially high performance, reduction in total wiring cost, and higher packing density [43], [92], [186]. In addition, wireless NoCs have been proposed that add high-speed wireless links on top of a conventional wired NoC to create a *small world* effect of bringing far nodes closer, significantly improving performance [49]. Such hybrid wireless/wired NoCs use different wireless technologies, e.g., mm-wave or surface wave [49], [95]. Similarly, photonics NoCs have been introduced that exploit *wavelength-division multiplexing (WDM)* to send multiple packet streams in parallel, using different wavelengths, on a single channel, at speed of light, leading to very high performance [119], [168].

1.3.3 Advances in Asynchronous NoCs

Even though synchronous NoCs are mainstream, they can still incur significant power and performance overheads, making asynchronous NoCs a promising alternative.

Asynchronous NoCs can be used to connect synchronous components, forming globally-asynchronous locally-synchronous (GALS) systems, or can be used in fully-asynchronous systems. The use of asynchronous NoCs eliminates global clock management, and the associated overheads of clock skew, clock power or any clock-gating circuitry. Given the promise of asynchronous NoCs, there has been much research in this area in the last decade, including industrial advancements and use of these NoCs in emerging areas of neuromorphic computing, as described below.

General asynchronous NoC research. A number of research challenges for asynchronous NoCs have been targeted. To achieve quality of service (QoS), asynchronous NoCs have been proposed that provide guaranteed service and multiple levels of services, in addition to best effort traffic [23], [51], [162]. There has been important research on improving fault-tolerance and reliability of asynchronous NoCs [84], [176], [205] with some works focusing on developing efficient asynchronous NoCs that also mitigate effects of process variation [56], [142]. Automated tool flows have also been proposed for asynchronous NoCs that guarantee not only correctness but also lead to high-performance implementation [66], [134], [184]. Interestingly, a recent asynchronous NoC that supports time division multiplexing (TDM) was proposed [96]; TDM is usually performed in a synchronous setting, since the use of clock helps provide a time reference. In addition, space division multiplexing (SDM) has also been used for asynchronous NoCs [206]. Further, virtual channels (VCs) have been added to asynchronous NoCs without significant overheads in terms of area and power [51], [136]. Multiple recent works target latency optimization of asynchronous NoCs using a low-overhead bypassing technique, where the routers on the path of a packet are informed in advance of the arrival so they can pre-allocate the resources, and the packet is then simply fast forwarded through the routers after arrival [57], [90]. Finally, there also has been recent interest in using asynchronous NoCs for 3D technology [190], as well as for vision applications [163].

Industrial comparisons with synchronous NoCs. Recently, there have been asynchronous and GALS NoCs developed at STMicroelectronics, Intel and AMD, which have been shown to be more efficient than their synchronous counterparts.

A GALS system called P2012 from STMicroelectronics uses a fully-asynchronous NoC to connect highly-customizable accelerators [11]. This system comprises 4 clusters, each consisting of 16 synchronous processors. The system delivers a performance of 80 GOPS but consuming only 2W power. Compared to recent Quadro and Nvidia commercial GPUs, P2012 achieves significantly better performance per unit area and performance per unit power.

Intel Labs proposed a hybrid packet/circuit-switched NoC, fabricated in advanced 22 nm tri-gate CMOS technology [30]. This NoC supports two modes: normal synchronous packet-switched, and a *source-synchronous* circuit-switched mode. The latter mode is actually an asynchronous circuit-switched implementation, which was shown to achieve 2.7× better network throughput and a 93% reduction in latency, compared to the synchronous packet-switched mode.

A recent asynchronous NoC router, developed jointly by AMD research and Asynchronous Circuits Lab of Columbia University, showed 55% less area and 28% reduction in latency in a head-to-head comparison with a state-of-the-art synchronous router used in high-end AMD processors [89]. Both the routers were synthesized in advanced 14 nm FinFET technology, and used two virtual channels. Similar improvements were also estimated for this asynchronous router with 8 VCs.

Neuromorphic computing. Neuromorphic computing is an emerging area of brain-inspired computers that target brain-like high performance and power efficiency. These systems have billions of computing elements, each following a non-Von Neumann architecture with a closely-coupled local memory. The neurons, in these systems, are implemented by computing elements, either using digital design [2] or analog [13], while the communication between neurons (or synapses) are implemented using an interconnect. Interestingly, asynchronous NoCs are often used as these interconnection networks, connecting billions of synchronous neural cores, forming a substantial GALS system. Neuromorphic computing fits the asynchronous paradigm because of the event-driven communication between neurons, and can utilize the various benefits of asynchronous NoCs in terms of scalability, low power, and ease-of-integration.

On of the first examples of a neuromorphic computer is *SpiNNaker (Spiking Neural Network Architecture),* developed at University of Manchester [62]. Spinnaker is a massively-parallel system, where 1 million neurons are modeled in software using synchronous ARM9 cores, which are connected using an asynchronous communication infrastructure. This system achieves burns only 50 mJ of energy when processing a 1024 by 1024 image, compared to 1.5 J for a GPU-based system.

TrueNorth from IBM is a recent example, which is a 5.4-billion transistor neuromorphic chip [2]. TrueNorth is also a GALS system with 4096 synchronous neurosynaptic cores, modeling 1 million neurons and 256 million synapses, connected using a fully-asynchronous NoC. This chip only consumes 63 mW of power while processing a 400×240 video input. TrueNorth is also being used for other applications such as face recognition.

Very recently, Intel announced the *Loihi neuromorphic chip*, which is a fully-asynchronous 2.1-billion transistor system [46], [117]. This chip comprises of 128 neural cores, modeling 1024 neurons each, and consuming only 25 pJ energy per operation at 1 V supply. These cores are connected using a 8×4 asynchronous mesh NoC. Loihi is the first system to support on-chip learning.

1.4 Multicast Communication and its Applications

While NoCs have several benefits, they also face a key challenge of supporting new types of traffic patterns. Modern NoCs must be able to support communication requirements of advanced parallel architectures. One such important class of traffic is multicast, i.e. 1-to-many communication, which has seen a growing interest recently [15]. Even though there has been significant research on supporting multicast in synchronous NoCs, this capability has been largely missing from asynchronous NoCs, despite the numerous applications of multicast as described below. This thesis focuses on supporting high-performance multicast in asynchronous NoCs while maintaining low overheads.

Multicast communication is defined as sending the same packet from one source to an arbitrary number of destinations. There are three main domains where this traffic pattern is common: (i) parallel computing applications, (ii) new interconnect technologies, and (iii) emerging NoC applications.

Multicast has been widely used in parallel computing domain for three main applications: *cache coherence protocols, shared operand networks,* and *barrier synchronization* [88]. In cache coherence, multicast can be used to send write invalidates to multiple processors in directory-based protocols. There are also multicast [21] and broadcast-based [126] snoopy cache coherence protocols. In shared operand networks such as RAW [181], TRIPS [166], Wavescalar [178], multicast can be used to deliver operands to multiple instructions. Also, barrier synchronization can be performed using multicast, crucial for message passing based shared memory systems. This approach is useful in bringing a set of processors to a known global phase before proceeding to a new phase of computation. In this case, a barrier-sync packet indicating thread synchronization is multicast/broadcast to all the participating processors [175]. Therefore, given these important applications, the NoCs for these advanced parallel architectures must support efficient multicast.

Multicast is also gaining importance with new technologies replacing the traditional electrical wires in NoCs. Emerging interconnect paradigms such as nanophotonics, wireless, and 3D integration have seen recent interest [26], [49], [168], [202]. Multicast and broadcast are inherent forms of communication used in radio frequency (RF) [28], [27], [94] wireless [49], [50], [111], photonics [33], [133], [201], and CDMA technologies [112], [153], [195]. Supporting lightweight, and power-performance efficient multicast in these emerging NoCs is an active area of research [94]. There is also interesting research in handling multicast in 3D NoCs [54].

Finally, 1-to-many (multicast) and 1-to-all (broadcast) communication patterns are also common in emerging areas of neuromorphic computing [2], [46] [130], [138], [144], and computational genomics [24]. A significant portion (sometimes 100%) of spiking neural networks (SNNs) [188] or deep neural networks (DNNs) [32], [109], traffic is multicast/broadcast, where a neuron communicates with several other neurons. Similarly, broadcast patterns are common for genomic applications such as sequence analysis and parallel sequence alignment [24]. NoCs designed for these emerging applications must be able to support multicast and broadcast efficiently. Recently, asynchronous NoCs have also been used to handle communication requirements of hardware implementations of SNNs [130]. However, an efficient multicast capability has been entirely missing from general-purpose asynchronous NoCs.

1.5 FPGAs: Architecture, and Applications

In the last decade, research and development in the field of reconfigurable computing has led to FPGAs becoming more mainstream and being deployed as SoCs for variety of applications. Modern FPGAs not only consist of the standard programmable logic fabric but also a processing system comprising multiple cores, GPUs, accelerators, and memories [151], [161]. The combination of programmability with a high-performance processing unit enables efficient implementation of large-scale systems, targeting different applications, at low cost, low power, and achieving fast time-to-market.

FPGAs have come a long way from the production of the first Xilinx FPGA in 1985, which could support 1000 ASIC gates to Xilinx's latest Zynq Ultrascale+ FPGA, which supports up to 6.2 million ASIC gates. Since 1999, there has been a major push towards integrating processors and memories with the programmable logic: from a single core to multi-core homogeneous and heterogeneous systems with different processors targeting specialized tasks and supporting different operating systems [151]. However, given the advancement in this field, there has been only limited research on implementing asynchronous circuits on FPGAs. This thesis takes on this challenge of safely and optimally implementing asynchronous NoCs on the modern FPGAs.

Before discussing the implementation of asynchronous NoCs on FPGAs, it is important to first understand the basic architecture of an FPGA, and the recent trends of its different applications.

1.5.1 FPGA Architecture

Figure 1.5 shows the high-level architecture of a Xilinx 7 series FPGA, such as Zynq 7000, Kintex 7, and Virtex 7 [20]. There are two main components: a processing system, and a programmable logic fabric.

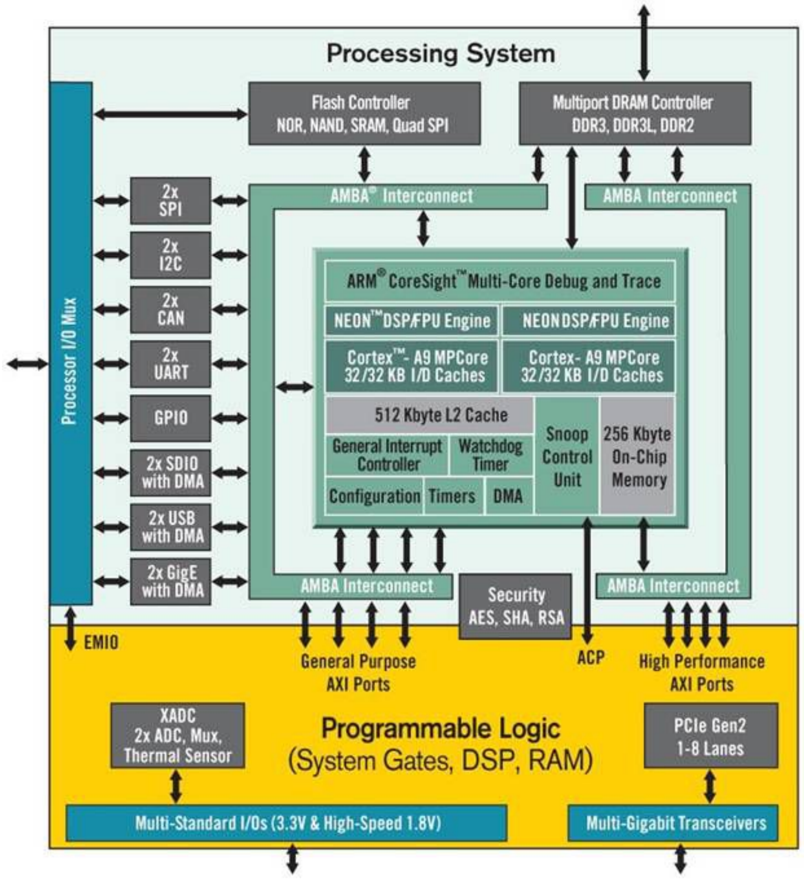
Processing System
Flash Controller NOR, NAND, SRAM, Quad SPI
Multiport DRAM Controller DDR3, DDR3L, DDR2
AMBA® Interconnect
AMBA Interconnect
Processor I/O Mux
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO with DMA
2x USB with DMA
2x GigE with DMA
ARM® CoreSight™ Multi-Core Debug and Trace
NEON™ DSP/FPU Engine
NEON DSP/FPU Engine
Cortex™- A9 MPCore 32/32 KB I/D Caches
Cortex- A9 MPCore 32/32 KB I/D Caches
512 Kbyte L2 Cache
General Interrupt Controller
Watchdog Timer
Configuration
Timers
DMA
Snoop Control Unit
256 Kbyte On-Chip Memory
AMBA Interconnect
Security AES, SHA, RSA
AMBA Interconnect
EMIO
General Purpose AXI Ports
ACP
High Performance AXI Ports
XADC 2x ADC, Mux, Thermal Sensor
Programmable Logic (System Gates, DSP, RAM)
PCIe Gen2 1-8 Lanes
Multi-Standard I/Os (3.3V & High-Speed 1.8V)
Multi-Gigabit Transceivers

Figure 1.5: Xilinx 7 Series FPGA block view [86]

The processing system consists of an advanced processor subsystem with dual-core ARM Cortex-A9 CPUs, L1/L2 caches, on-chip memory, and other useful units such as DMA and interrupt controller. Other important components of the processing system are the flash controller, multi-port DRAM controller that supports DDR2 and DDR3, and a rich set of standard I/O peripherals. This system connects to the programmable logic fabric using a high-performance low-latency AXI-based interconnect that can enable 16 parallel DMA channels and a functional bandwidth of over 300 MB/s.

The structure of a programmable logic fabric is shown in Figure 1.6 [98]. It uses an island-style architecture with two main components: configurable logic blocks (CLBs) and an underlying programmable interconnect, connecting the CLBs. The CLBs consist of configurable one-to-five input look-up tables (LUTs) that can implement different logic functions, and flip-flops for storage

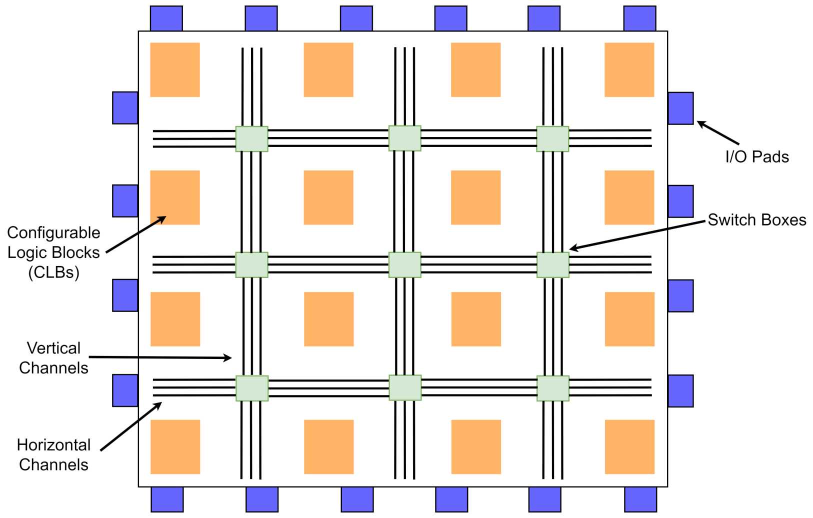

Figure 1.6: FPGA programmable logic fabric

purposes. The programmable interconnect consists of switch boxes, and horizontal and vertical channels. The switch boxes mainly consist of multiplexers that can be configured, as well as buffers to drive the wires of the channels.

1.5.2 FPGA Applications

Due to the advancement in FPGA technology, these devices are being deployed for several industrial applications [151], [161]. Some of the important ones are [161]: to develop efficient digital signal processing systems, implementing vision applications in FPGA for aerial vehicles such as UAVs, and in automative systems for vehicle velocity estimation, etc. With the ever-increasing interest in machine learning algorithms, FPGAs are also being used for hardware-assisted acceleration of these algorithms: prominent examples are at Microsoft, where FPGAs are being used to accelerate the Bing search engine [3], [25]. Additionally, other applications of FPGAs include: developing security circuits such as PUFs [91], as well as for big-data analytics acceleration [191].

1.6 Research Focus

This thesis aims to significantly advance the field of asynchronous NoCs by introducing the first systematic and efficient approach to support multicast in these NoCs. While there has been significant work on handling multicast in synchronous NoCs, this capability has been entirely missing from general-purpose asynchronous NoCs.

While the primary focus of this thesis is on supporting multicast in asynchronous NoCs, a secondary focus is on efficiently and safely synthesizing these NoCs on modern FPGAs. Synthesis and evaluation of these NoCs on commercial FPGAs is required to not only perform a realistic analysis of the actual physical design in terms of performance and energy, but also to advance the field of asynchronous NoCs by introducing a systematic and efficient methodology to map these NoCs on FPGAs, targeting both robustness and high performance. Such a methodology has been largely missing from asynchronous NoCs research, even though there has been significant advancement in FPGAs and their increased commercial uptake.

1.6.1 Challenges with Supporting Multicast in Asynchronous NoCs

To achieve high-performance and low-overhead multicast, there are multiple challenges that must be addressed: (i) selecting a multicast addressing that can efficiently encode multiple destinations; (ii) efficient multi-way forking of the flits of multicast packets; and (iii) avoiding any deadlocks that can occur due to multicast.

There are trade-offs involved in selecting the best multicast addressing scheme. There can be several possibilities: encoding addresses (x,y coordinates) of all destinations, using bit-string to encode destinations, or encoding the paths taken at various intermediate switches to the destinations. It is important that the selected addressing not only has a good coding efficiency but also leads to a simpler decoding logic in the switches for low latency.

Replication of flits of multicast packets at switches is the core operation of parallel multicast, and must be performed efficiently. In synchronous NoCs, this replication can either take multiple clock cycles [81], [88], or can be performed in parallel in a single clock cycle [149], [164], [193]. For high-performance multicast in asynchronous NoCs, switches must support forking of the flits of a multicast packet in parallel towards multiple output ports, and this replication must be performed without adding significant area/energy overheads. In addition, due to the asynchronous operation, this multi-way routing is performed in continuous time without waiting for discrete clock cycles, which can also potentially lead to high performance.

Multicast operation is potentially prone to deadlocks. Deadlocks can occur within a switch, due to cyclic dependency between multiple multicast packets, where one packet acquires some resources needed by the other packet, and vice versa, and both packets wait on each other to free up these resources [164]. Such deadlocks must be avoided but without incurring significant costs in terms of area, power, and performance.

1.6.2 Challenges with Implementing Asynchronous NoCs on FPGAs

There are three main challenges with implementing asynchronous NoCs on FPGAs: (i) mapping of special asynchronous elements that are not used in traditional synchronous designs; (ii) in the absence of clocks, asynchronous circuits rely on timing constraints that must be satisfied for correct operation; and (iii) developing an automated CAD tool flow for implementation on FPGAs.

There are elements, which are not used in synchronous designs but widely-used in asynchronous NoCs. These are: C-element, used for storage, and a mutual exclusion (mutex) element for arbitration. Mapping these elements on FPGAs, both safely and efficiently, is required.

The focus of this thesis is on single-rail bundled-data asynchronous NoCs, a design style which has gained significant interest recently to achieve high performance at low overheads, but also rely on moderate timing constraints for correctness [66], [83], [89], [163]. These NoCs use a single-rail bundled data encoding, where synchronous-style datapath is strobed by a *req* wire. While the use of single-rail data encoding leads to low overheads, two timing constraints are required for correctness: in the datapath, *req* must arrive only after data is stable (i.e. a bundling constraint), and in the control, relative timing constraints (RTCs) between any two paths must be satisfied. The methodology to map these NoCs on FPGAs must guarantee that these constraints are satisfied.

Finally, developing an automated tool flow to map asynchronous NoCs on FPGAs is challenging. These flows must not only satisfy all timing constraints for correctness but also lead to a high-performance mapping. In addition, these flows must use existing commercial tools for convenience of the designers.

1.7 Contribution of Thesis

In this thesis, new multicast strategies are introduced for two different network topologies, first starting with a simpler *variant mesh-of-trees* with low-radix routers, followed by the more common and complex *2D mesh* with higher radix routers. *To the best of our knowledge, these are the first general-purpose asynchronous NoCs to directly support parallel multicast.*

Additionally, for a more realistic analysis and evaluation, and also to advance the general field of asynchronous NoCs, a systematic CAD methodology is introduced to synthesize these NoCs on FPGAs. A challenging two-fold goal is targeted for the final implementation: it must be highly robust and also achieve high performance. *To the best of our knowledge, this is the first systematic methodology to efficiently map asynchronous NoCs on FPGAs.* To demonstrate the effectiveness of the proposed flow, two distinct 5-port asynchronous NoC switches are implemented using the proposed tool flow: one highly-efficient switch that only handles unicast, and the other that also supports multicast, proposed as part of this thesis. A head-to-head comparison was performed between these switches and a state-of-the-art synchronous switch: the asynchronous switches significantly outperform the synchronous one in terms of energy and idle power, as well as in some critical performance metrics.

Multicast in variant mesh-of-trees NoCs using local speculation. The first topology targeted in this thesis for supporting multicast is variant mesh-of-trees (MoT). This topology is simple, involves low-radix switches, and has been used for high-performance parallel computing to interconnect processing cores with memory modules.

For variant MoT, a novel strategy, *local speculation,* is introduced for high-performance and low-overhead multicast [18]. In this strategy, a packet (unicast or multicast) is always broadcast at a fixed subset of speculative routers in the network. To restrict the distance traveled by any redundant packets to small 'local' regions, these packets are throttled by neighboring non-speculative routers, hence, limiting the penalties of speculation to minimal impact on congestion and power. Speculative routers are very simple and fast as they do not perform route computation or channel allocation. Non-speculative routers perform throttling with almost no hardware overhead. This mix of speculative and non-speculative routers leads to a hybrid network architecture. For multicast traffic, significant performance benefits with small power savings are obtained by the new hybrid network over a fully non-speculative baseline. Interestingly, similar improvements are also shown for unicast traffic.

Multicast in 2D-mesh NoCs using a continuous-time replication strategy. The second topology used for handling multicast is 2D mesh, which is a more common topology. Multicast in 2D mesh is a more challenging problem, as it involves higher-radix routers, and exhibits significant amount of parallelism with multiple packets routed through different input ports of a router at the same time.

For 2D mesh, a new *replication strategy* is introduced to achieve high-performance multicast but still maintaining low overheads [17], [19]. In this approach, the flits of a multicast packet are routed through the distinct outputs of the router according to each output's own rate, concurrently and in continuous time. Unlike synchronous, this unique asynchronous approach, not discretized to clock cycles can provide considerable performance benefits by accommodating subtle variations in network congestion and exploiting 'sub-cycle' differences in interface operating rates. Exhaustive experiments on wide-ranging multicast benchmarks, created with traffic patterns common in cache coherence protocols and neural networks, show significant latency and throughput gains, with small-moderate energy improvements, over a baseline network that performs multicast using several unicasts, injected and routed serially. Interestingly, moderate latency improvements were also shown for unicast traffic.

A systematic methodology for synthesizing asynchronous NoCs on FPGAs. Finally, a comprehensive CAD tool flow is proposed for mapping asynchronous NoCs on modern commercial FPGAs. The target of this work is bundled-data asynchronous NoCs, which are shown to be highly-efficient but also involve moderate timing constraints for correct operation. The proposed methodology not only makes sure that all timing constraints are satisfied, but also achieves high performance for these NoCs. For ease of implementation and convenience for the designs, only existing FPGA synthesis tool is used. In addition, asynchronous NoCs also use some special asynchronous components such as a C-element and a mutex; a systematic guide on how to map these components both efficiently and safely is also proposed. Two distinct bundled-data 5-port switches are synthesized using the proposed tool flow on *Xilinx Virtex 7 in 28 nm:* one that only supports unicast [66], and the other that also handles multicast (Chapter 5). Compared to a high-performance synchronous switch, the unicast asynchronous switch achieved 30.7% lower latency and 44.4% lower energy. Interestingly, the multicast asynchronous switch also achieved 11.5% lower latency with similar energy as synchronous for a unicast transmission, despite the extra instrumentation for multicast support.

1.8 Organization of Thesis

The thesis is organized as follows.

Background of the thesis is provided in Chapter 2 and Chapter 3. Chapter 2 covers the basic concepts of asynchronous design: different handshaking protocols, data encoding schemes, special asynchronous components, overview of asynchronous pipelines with a particular focus on the *Mousetrap pipeline*, and mixed-timing interfaces, which are used to design GALS systems. Chapter 3 presents the background on networks-on-chip: different topologies that are used in this thesis, routing algorithms, packet encoding schemes, basic synchronous router architecture and optimization strategies, multicast techniques and related work on supporting multicast in synchronous NoCs, followed by examples of state-of-the-art synchronous NoCs that support multicast.

Chapters 4, 5 and 6 present the new research, and form the core of the thesis. Chapters 4 and 5 introduce the new multicast strategies, targeting simpler mesh-of-trees topology and more complex 2D mesh, respectively. Chapter 6 introduces a new CAD methodology to implement asynchronous NoCs on FPGAs, addressing various challenges, and using a unicast asynchronous NoC router, and the multicast-aware asynchronous router from Chapter 5 as case studies.

Finally, concluding remarks and future research directions are presented in Chapter 7.

Chapter 2

Background: Asynchronous Design

Since the focus of this thesis is on asynchronous NoCs, a brief background on asynchronous design style is presented in this chapter before getting to the new research. This background covers several fundamentals for designing asynchronous NoCs: the types of handshaking protocols that can be used and their trade-offs, different data encoding schemes, some special asynchronous components that will be used in these NoCs, an overview of Mousetrap pipelines that form the basis of these NoCs, and different types of mixed-timing interfaces that are required to build GALS systems.

2.1 Handshaking Protocols

Figure 2.1 shows a typical asynchronous communication between a sender and a receiver. The communication channel involves a request *(req)* wire and an acknowledge *(ack)* wire. The *req* wire indicates when the data sent by the sender is valid, and the *ack* wire indicates that the receiver successfully received the data.

There are two most common handshaking protocols used for the communication channel. The first is a *four-phase (return-to-zero (RZ) protocol),* and the second is a *two-phase (non-return-to-zero (NRZ) protocol)* [147]. There are other non-standard communication protocols as well, such as *pulse mode* [152], which combine the advantages of the RZ and NRZ protocols, but have complex timing requirements and implementation. Hence, the focus of this thesis is only on two-phase and four-phase protocols, and the trade-offs of selecting one over the other.

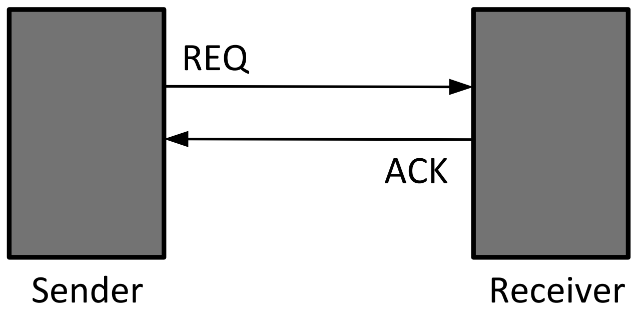

Figure 2.1: A typical asynchronous communication

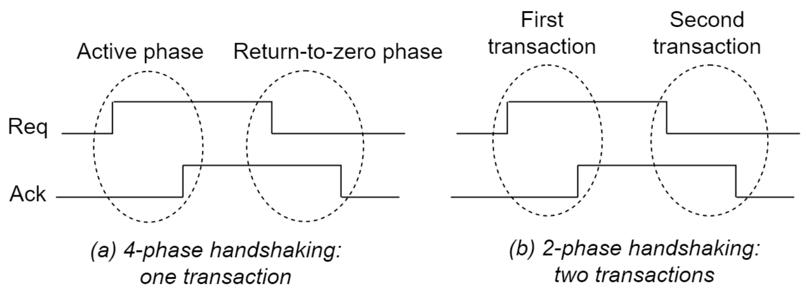
Active phase
Return-to-zero phase
First transaction
Second transaction
Req
Ack
(a) 4-phase handshaking: one transaction
(b) 2-phase handshaking: two transactions

Figure 2.2: Two common handshaking protocols

2.1.1 Four-Phase Protocol

Figure 2.2(a) shows a single transaction using a four-phase protocol [45], [172], [183], [185], [198]. In this protocol, *req* and *ack* are initially low. The sender initiates a transaction by asserting *req,* and the receiver responds by asserting *ack.* These events form the active phase. In the following reset phase, these two wires are in turn deasserted low. Therefore, a single transaction requires two roundtrip communications.

2.1.2 Two-Phase Protocol

Figure 2.2(b) shows two transactions using a two-phase protocol [85], [154], [171]. In this protocol, a single transition on the *req* wire starts the transaction, which is terminated by a single transition on the *ack* wire. Hence a transaction only involves one roundtrip communication.

2.1.3 Trade-Offs

There are interesting cost trade-offs involved in selecting the right handshaking protocol. A four-phase protocol may lead to simpler designs due to the return-to-zero phase. However, two roundtrip communications per transaction can lead to performance overheads. A two-phase protocol, on the other hand, may lead to complex designs but is a better choice for performance due to only a single roundtrip communication per transaction. Moreover, recently, the two-phase protocol has been used to design asynchronous NoCs which are not only high-performance but also very simple [66], [83], [89], [96]. Therefore, a two-phase protocol is mainly used in this thesis.

2.2 Data Encoding Schemes

After deciding the handshaking protocol, the next major decision is which encoding scheme to use for the data. There are two widely-used data encoding schemes: *delay-insensitive (DI) codes,* and *single-rail bundled-data* [147].

2.2.1 Delay-Insensitive (DI) Codes

In DI encoding, the *req* wire is typically replaced by the data bits, where a code is used that identifies both the data value and its validity [45], [183], [185], [198]. The data bits in this encoding can be sent to the receiver in any arbitrary order, and the arrival times of the bits can also be arbitrarily skewed, but the receiver is still able to successfully identify when data is valid and to extract the data. A completion detector is required at the receiver to detect the termination of a data transmission.

A common and simple instance of the DI codes is *dual-rail encoding* [45], [183], [198]. As shown in the Figure 2.3, the dual-rail code encodes a data bit using two wires. The bit combination '01' is used to encode 0 and '10' represents 1. '00' is called a spacer, used between two valid transmissions, and '11' is not used and is invalid in this scheme. Other popular approaches include 1-of-4, more generalized m-of-n codes, level-encoded dual-rail (LEDR) [48], and level-encoded transition-signaling (LETS) codes [128].

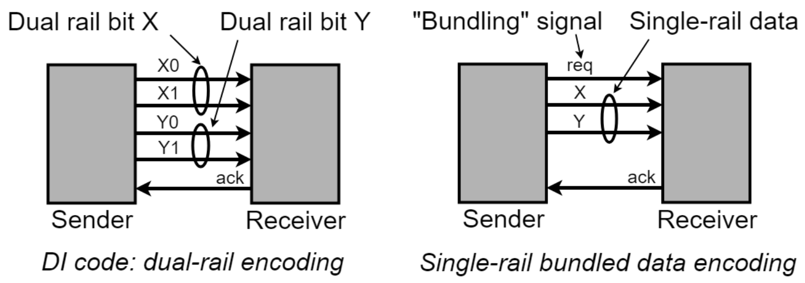

Figure 2.3: Two widely-used data encoding schemes

2.2.2 Single-Rail Bundled Data

An alternative is a single-rail bundled data encoding. As shown in the Figure 2.3, this scheme uses a synchronous-style single-rail data channel with an extra *req* wire, called bundling signal that is used as a local strobe [74], [89], [100], [171]. A simple single-sided timing constraint is required for correctness: the *req* arrive after the data is stable, and therefore models a worst-case delay. This matched bundling delay can be implemented as an inverter chain or a carefully replicated critical path taken from the datapath [147]. Recently, a set of heuristics have been proposed to select the best mix of gates, taken from the same standard-cell libraries used to synthesize the datapaths and wire lengths to generate high-performance and dynamically reconfigurable bundling delay lines [140]. Moreover, these delays can be made fairly tight.

2.2.3 Trade-Offs

Like the handshaking protocols, there are interesting trade-offs involved with the encoding schemes. DI encoding may have lower coding efficiency ($2n$ wires for n bits in dual-rail), which can lead to high dynamic power and area, but is also more timing-robust to static process variations or dynamic timing deviations. Single-rail bundled data, on the other hand, can be less timing-robust but only involves simple timing constraints, which, in practice, are easily satisfied, while also providing higher coding efficiency with lower energy/area footprint. Given these advantages, the single-rail bundled data encoding has been widely-used in recent asynchronous NoCs [66], [83], [89], [96], and is therefore, also used in this thesis.

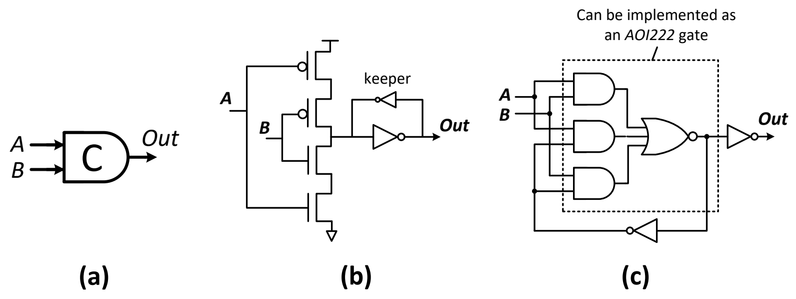

Figure 2.4: The C-element: (a) symbol, (b) a transistor-level design, and (c) a standard-cell based design

2.3 Special Asynchronous Components

There are some special asynchronous elements that are not typically used in synchronous designs. These components are: the C-element and the mutual exclusion (mutex) element. Another critical component is an n-way asynchronous arbiter, which mediates between multiple requests for a shared resource. These arbiters, and in particular, the 4-way arbiters, are important parts of the proposed asynchronous NoC routers.

2.3.1 The C-Element

The C-element is an asynchronous state-holding component [147]. As shown in Figure 2.4(a), it has two inputs: A and B, and one output. The output is deasserted when both inputs are low, and asserted when both inputs are high. For all other cases, the element holds its output.

Figure 2.4 shows two possible designs of the C-element. One possibility is at the transistor-level, as shown in Figure 2.4(b), which has advantages of higher performance and lower power. The other option is a standard-cell design, as shown in Figure 2.4(c), which can be easily integrated into an automated design flow. The latter was also used in *Tangram*, an early design flow, developed at Philips Semiconductors [14], and used in a number of products. Moreover, both the designs can be simply extended to support higher number of inputs.

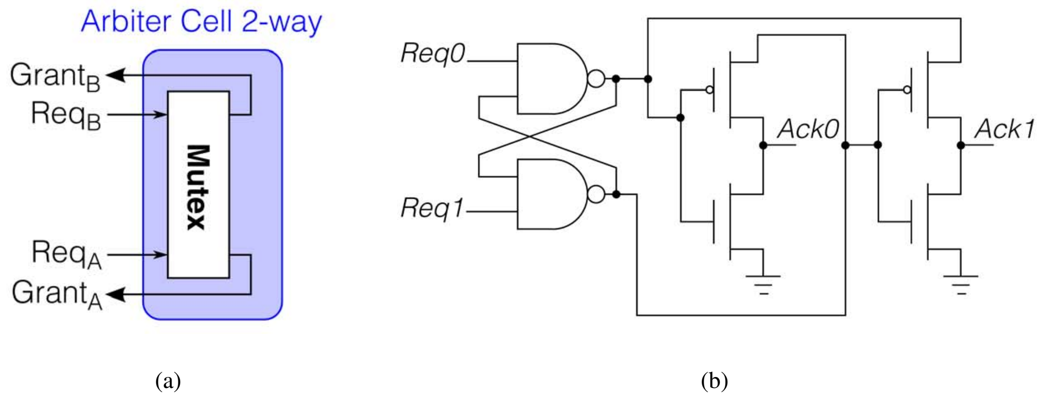

Figure 2.5: Mutex: (a) block-level view, (b) design details [147]

2.3.2 The Mutex

The mutex is used for basic arbitration in asynchronous systems. Arbitration in an asynchronous design is more challenging than in the synchronous designs. In synchronous, the result of the arbitration between requests is decided based on the order of arrival of the requests during discrete clock cycles. While in asynchronous, these requests arrive in continuous time, and can be only picoseconds apart. The mutex element is specially designed to handle this continuous-time arbitration.

There are two requests input to the mutex, with one grant output corresponding to each request, as shown by a block-level view in Figure 2.5(a). The design of the mutex is shown in the Figure 2.5(b), introduced by Seitz [147]. This mutex contains two stages: a digital arbiter stage and an analog filter stage. The arbiter stage performs arbitration between requests using a cross-coupled NAND structure, similar to an SR latch. If the two requests arrive close to each other, the arbiter can become metastable. The filter stage keeps the grant outputs of the mutex low while the arbitration is being resolved, and once the arbitration is complete, it cleanly asserts exactly one grant high.

The mutex operates using a four-phase handshaking protocol. In a basic scenario with no contention, an assertion on one request leads to an assertion on the corresponding grant, followed by deassertion of both. In a scenario with contention, with one request arriving slightly before the other, the arbiter stage performs arbitration, during which the filter stage keeps the outputs low. Once the arbitration is resolved, the grant corresponding to the winning request is asserted. This set phase is then followed by a reset phase, where both the request and the grant go back to low, in turn. This deassertion is then followed by the other pending request being granted.

The resolution time for arbitration during contention depends on the difference in arrival times of the requests. When the requests are far apart, the resolution is quite fast, with the resolution time degrading as the difference in arrival times gets smaller. When the requests arrive extremely close to each other (nearly 1 ps apart) then the time to resolve arbitration can be relatively long. However, in practice, ambient physical conditions, such as noise, can lead to fast resolution of arbitration.

2.3.3 The N-Way Arbiters

An n-way arbiter is an important component that mediates between n requests, which are trying to gain access to a shared resource. This resource can be a shared bus in a multi-processor system or an output channel of a NoC router node requested by multiple input ports. Designing these arbiters efficiently and robustly is critical, especially for NoCs, where the current trend is towards the use of topologies with high-radix (5-7) routers that will require n-way arbiters with n greater than 3.

Three cost metrics are considered when designing n-way arbiters: high performance, impartiality, and scalability [135]. High performance not only includes the latency of the arbiter to access the shared resource but also its throughput in processing the requests in various competing scenarios. Impartiality involves two facets: (i) *latency equalization,* i.e., latency from each input request to the corresponding grant output must be similar, and (ii) *arbitration fairness,* where every input request should have similar probability of winning the arbitration. In addition, the arbiters must also be scalable to larger sizes, and its performance and impartiality should not degrade as n increases.

A recent work introduced a novel tree-based architecture for n-way arbiters that mitigates various forms of impartiality and also achieves high performance [135]. This tree architecture is highly balanced and therefore has equalized latency response, as well as a higher degree of arbitration fairness. This paper introduced a scalable family of n-way arbiters, with n ranging from 3 to 9, that also achieved lower latency with higher throughput, compared to other arbiters.

A 4-way arbiter is an important case of n-way arbiters. An efficient and robust design of this arbiter is required, as it not only forms a building block of bigger arbiters but is also crucial for 5-port asynchronous NoC routers, which are the focus of this thesis.

The design of a 4-way arbiter is shown in Figure 2.6 [135]. The arbiter takes four input requests and has grant output corresponding to each request. This design achieves high performance using three mutexes in parallel: left and right mutexes arbitrate between the requests *req0/req1* and *req2/req3,* respectively, while the middle mutex arbitrates between the two pairs. The final decision

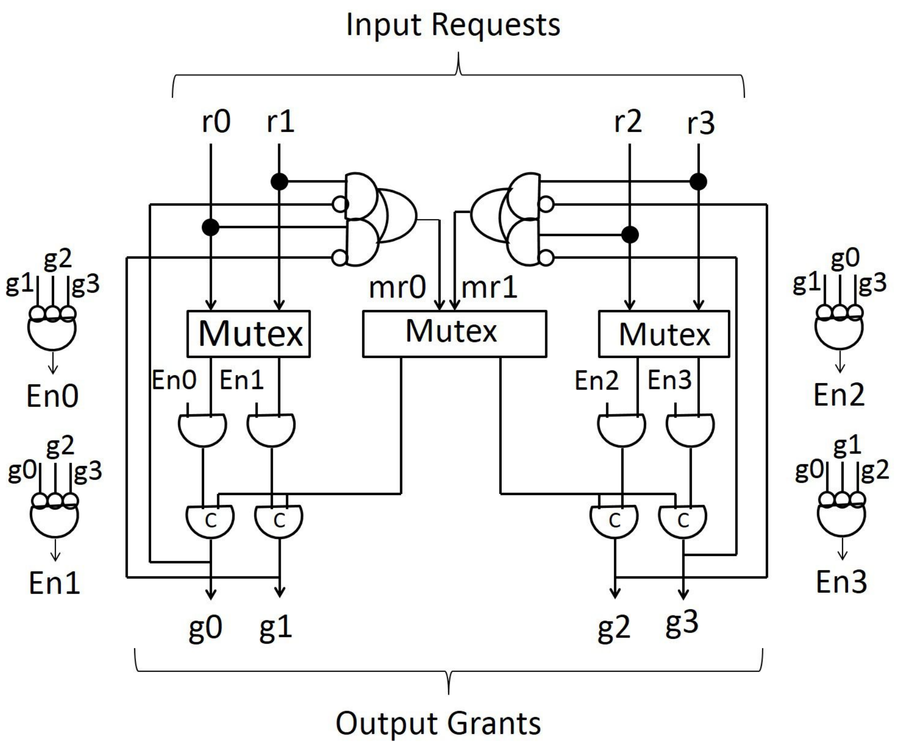

Figure 2.6: A 4-input arbiter [135]

on which request to grant access is made using a merge of the individual arbitration results of the mutexes, performed using C-elements, which synchronize the result of the middle mutex with the side ones.

The arbiter effectively implements a round robin arbitration policy, both between the left and right side, and between the requests on the same side, in active scenarios. For example, in a contention scenario, with three or four requests, whenever one of these requests is granted, any other request from the the same side of the winner will be masked for the time being. The idea behind this masking is to make sure that there is no other active request on the same side when the winning request is deasserted so that the middle mutex can be released. This operation results in an advantage for the requests from the other side to now acquire the middle mutex, hence leading to a fair arbitration.

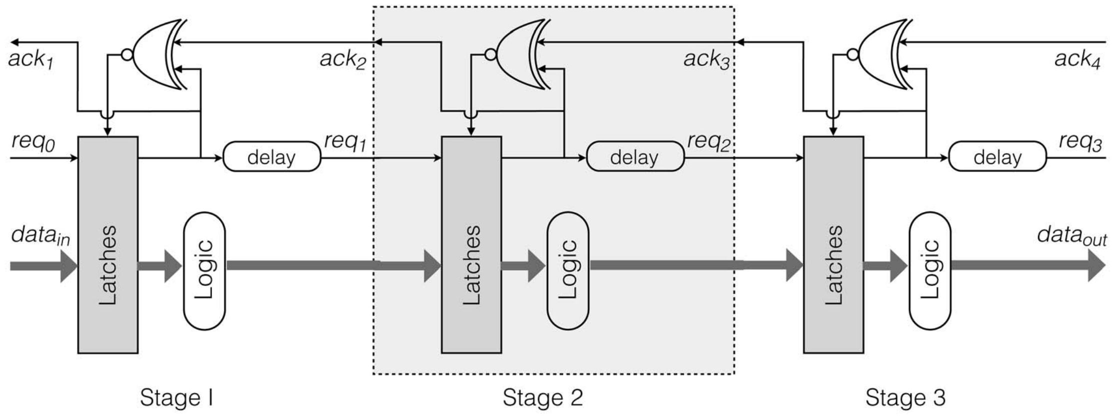

Figure 2.7: A 3-stage Mousetrap pipeline [171]

2.4 Mousetrap Pipelines

Mousetrap is a high-performance pipeline that uses a two-phase handshaking protocol and single-rail bundled data encoding [171]. This pipeline forms the basis of the proposed asynchronous NoCs, and it is therefore important to understand its structure and operation. Also, since Mousetrap is a bundled-data design, it relies on one-sided timing constraints for correct operation. These constraints are also reviewed below.

2.4.1 Mousetrap Structure

Figure 2.7 shows a 3-stage Mousetrap pipeline. Each stage consists of a single bank of normally-transparent D-latches with a simple local control: an XNOR gate. The interface between adjacent stages has single-rail data and a bundling *req* going forward and an *ack* going backward. A delay element is added on the *req* wire to match the worst-case logic block delay.

2.4.2 Mousetrap Operation

Mousetrap operation is based on a *capture-pass protocol,* where the latches are initially transparent with all the *req/ack* wires at 0. At a stage i, as new data arrives with its bundling *req,* it is passed through the latch and the corresponding req_i is toggled, causing the XNOR of that stage to close the latch, storing the data. In parallel, ack_i is sent upstream to request new data. Finally, when an ack_{i+1} is received from right, the $XNOR_i$ makes the register transparent, again completing the entire cycle.

2.4.3 Timing Constraints

To ensure correct operation of the bundled-data circuits, two types of timing constraints must be satisfied. First, a bundling constraint in datapath: *req* must transition after data is stable. Second, relative timing constraints (RTCs) in control: the delay across one path should be less or greater than the other.

Mousetrap exhibits examples of these timing constraints. *Bundling constraint:* a delay element must be added on the *req* wire to match the worst-case delay across the logic block between two stages, as shown in Figure 2.7. *Data protection RTC:* once data enters a stage (e.g. stage 2), it must be securely stored in the latch register before new data is produced by the previous stage. This constraint involves two paths starting after req_2 toggles: one path to close the $latch_2$ through the $XNOR_2$, and the other path comprising of re-enabling $latch_1$ through $XNOR_1$, followed by the delay of $latch_1$ and $logic_1$. Clearly the delay of first path must be less than the second.

2.5 Mixed-Timing Interfaces

Mixed-timing interfaces are crucial in building GALS systems [147]. In these systems, the mixed-timing interfaces are required to connect processing cores, accelerators, memories, operating at different clock frequencies with an asynchronous NoC. Although this thesis does not immediately focus on GALS systems, an important future work is to implement such systems on FPGAs.

There are three kinds of mixed-timing interfaces. These are: (i) *sync-sync* that connects a synchronous sender with another synchronous receiver that is operating at a different clock frequency, (ii) *sync-async* that connect a synchronous sender to an asynchronous receiver, and (iii) *async-sync,* which is a dual of *sync-async.*

Robust and high-performance FIFO-based mixed-timing interfaces have been introduced for the above three cases [29]. Each FIFO has a put interface for the sender, and a get interface for the receiver. These interfaces can be either synchronous or asynchronous, depending on the type of FIFO: *sync-sync*, *async-sync* or *sync-async.* For example, for an *async-sync* FIFO, an asynchronous put interface is combined with a synchronous get interface.

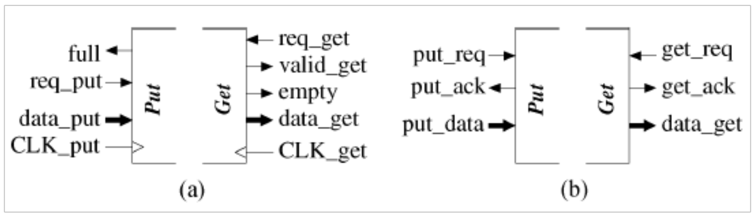

Figure 2.8: Put/get interfaces for mixed-timing FIFOs: (a) synchronous interfaces, (b) asynchronous interfaces [29]

The block-view of the synchronous put and get interfaces is shown in Figure 2.8(a). The synchronous put interface has three inputs: a clock (CLK_{put}), a control request to initiate put operation (req_{put}), and a data bus ($data_{put}$). It has one status output, $full$, which indicates if the FIFO is full or not. The synchronous get interface has two inputs: CLK_{get}, and req_{get}. This interface has two outputs: a data bus $data_{get}$, where the data is placed, and an $empty$ status signal, which indicates if the FIFO is empty or not.

The block view of the asynchronous put and get interfaces is shown in Figure 2.8(b). These interfaces are more simplified compared to the synchronous ones. Each interface only contains a data bus as input (for put) or output (for get), and handshaking controls req and ack. A *req* is used to request a put or get operation, while an *ack* indicates completion of these operations. Note that there are no full or empty status signals. In case of FIFO being full, the put interface will simply hold the put_{ack}, and when empty, the get interface will hold the get_{ack}.

As an example, the basic architecture of a *sync-sync* FIFO is shown in Figure 2.9. Similar architectures can be defined for other FIFOs: different synchronous and asynchronous put and get interfaces can be freely combined to construct these FIFOs.

Each FIFO is a circular array of identical cells, comprising put and get interfaces, that communicate with the sender and receiver data buses. The input and output behaviors of the FIFO are controlled by two tokens: a put token to insert data items, and a get token to remove data items. The cell with the put token is the tail of the queue, while the cell with the get token is the head. After a token has been used by the current cell for data operation, it is passed to the next cell. An interesting feature of these circular FIFOs is that data remains immobile and is not moved between cells: data is enqueued and dequeued in place.

There are several advantages of the above architecture, which are common to the all the FIFOs.

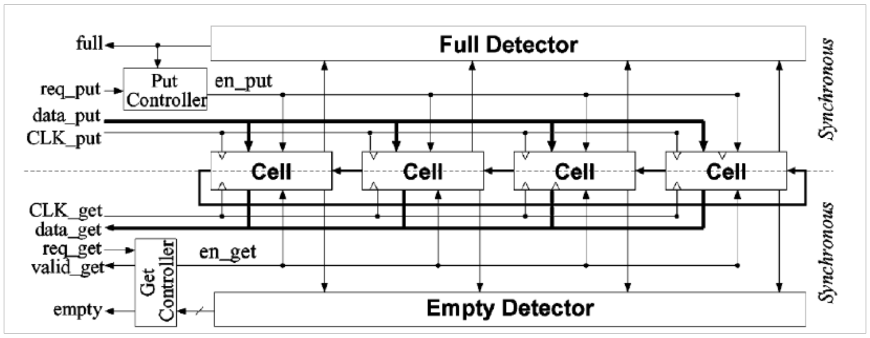

Figure 2.9: Basic *sync-sync* FIFO architecture [29]

Since data is immobile, these FIFOs have a potential for low latency: as soon as a data item is enqueued, it is also available for dequeuing. This immobility can also lead to potentially lower power. Finally, the FIFOs are very scalable as the FIFO capacity can be changed with only small design modifications.

Now that we have covered the relevant background on asynchronous design, in the next chapter we go over the basics on networks-on-chip, which form the second thread of this thesis.

Chapter 3

Background: Networks-on-Chip

Since Networks-on-Chip (NoCs) are at the center of this thesis, it is important to understand their basics before presenting the new research. This chapter covers several NoC fundamentals: network topologies, routing algorithms, and packet encoding schemes. The different synchronous NoC switch micro-architectures and optimizations to improve their performance are also presented.

In addition, the focus of the thesis is to support multicast in NoCs. Therefore, the main techniques to perform multicast in synchronous NoCs and related work are also covered in this chapter. As case studies, two leading synchronous NoCs with multicast capability are discussed in terms of their structure and protocols.

3.1 Network Topologies

A network topology defines the structure in which the switches and the channels are arranged [41], [22]. Selecting an appropriate topology is usually the first step in designing a NoC, and depends on the applications and power/performance requirements of the NoC. This thesis focuses on two topologies: a variant mesh-of-trees (MoT) and a 2D-mesh topology.

These topologies belong to two different classes: an indirect topology and a direct topology. The variant MoT is an indirect topology, where the network consists of two different types of router nodes: *terminal* and *switching*. The terminal node is a source or a sink for data transmission, sending packets from or to an attached processing element. The switching node is not connected to any processing elements and only forwards data to another node. The 2D mesh, on the other hand,

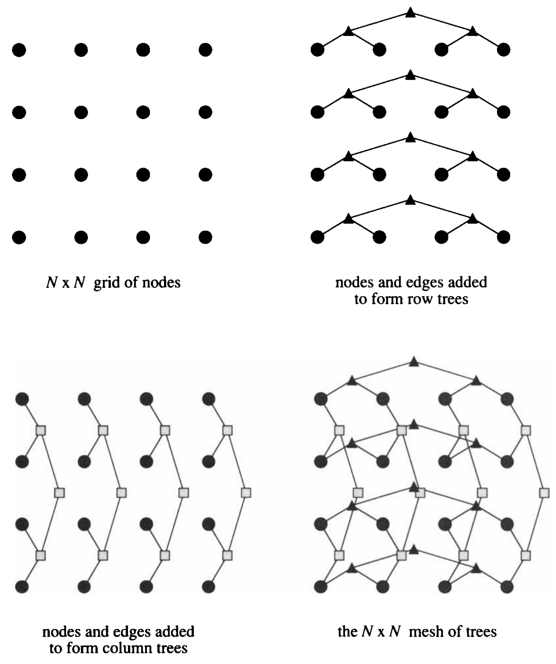
N x N grid of nodes
nodes and edges added to form row trees
nodes and edges added to form column trees
the N x N mesh of trees

Figure 3.1: Original 4×4 MoT topology [6]

is a direct topology, where the router nodes act as both terminal and switching nodes; each router is connected to a processing or storage element and also forwards data to other nodes.

3.1.1 A Variant MoT Topology

Before discussing the variant MoT, it is important to first go over the original MoT. The structure of the original 4×4 MoT is shown in the Figure 3.1 [6]. A grid of terminal nodes, each attached to a processing element, are connected using horizontal tree connections (row of trees) and vertical tree connections (column of trees) to form this topology. The triangles and squares in the figure show intermediate nodes, while the circles are the terminal nodes. The routing path between a source and a destination, involves sending packets up or down the trees. These paths can also be shared by different packets and therefore can cause congestion.

Figure 3.2 shows the structure of a 4×4 variant MoT [7]. It contains two binary trees: a fanout tree composed of routing (i.e. fanout) nodes, and a fanin tree composed of arbitration (i.e. fanin) nodes. The trees are mirror copies: the fanout network from a source root to leaves, and the fanin network from leaves to a destination root. The fanout nodes route packets from a single input

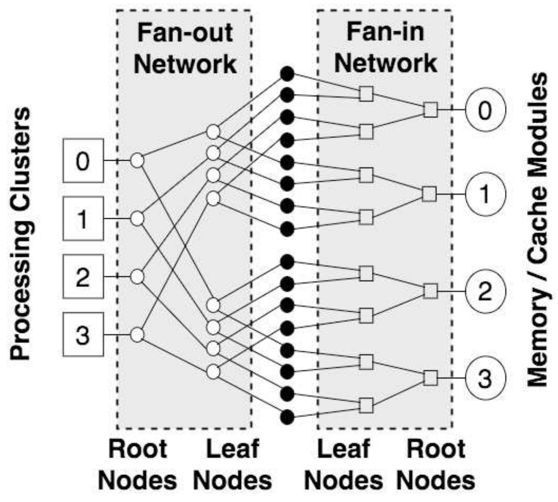

Figure 3.2: A variant mesh-of-trees (MoT) topology, connecting processors to memory modules [7]

channel to the correct one of the two output channels. The fanin nodes arbitrate between two input streams and routes the winner on the output channel.

Variant MoT is a high-performance topology [7]. It provides two key advantages: (a) the hop count from source to destination is always a small constant, log(n), leading to low latency; and (b) unlike the original MoT, each distinct source-destination pair has a unique path through the network, which can minimize network contention significantly. However, the lack of path diversity can be a potential performance bottleneck for some adversarial cases, where traffic is extremely unbalanced. Overall, though, recent results demonstrate significant benefits for saturation throughput in high-performance systems, where variant MoT has been used to connect processing cores with memory [5], [78], [158].

3.1.2 A 2D-Mesh Topology

Figure 3.3 shows the architecture of a 4×4 2D-mesh topology connecting 16 tile nodes. The structure of each tile is also shown in this figure, which consists of a router, connected to an IP core through a network interface (NI). Unlike the variant MoT with only one-input fanout nodes and two-input fanin nodes, 2D-mesh exhibits more parallel operations with generally 5-port routers that handle both routing and arbitration functions. Therefore there are five input port modules (IPMs),

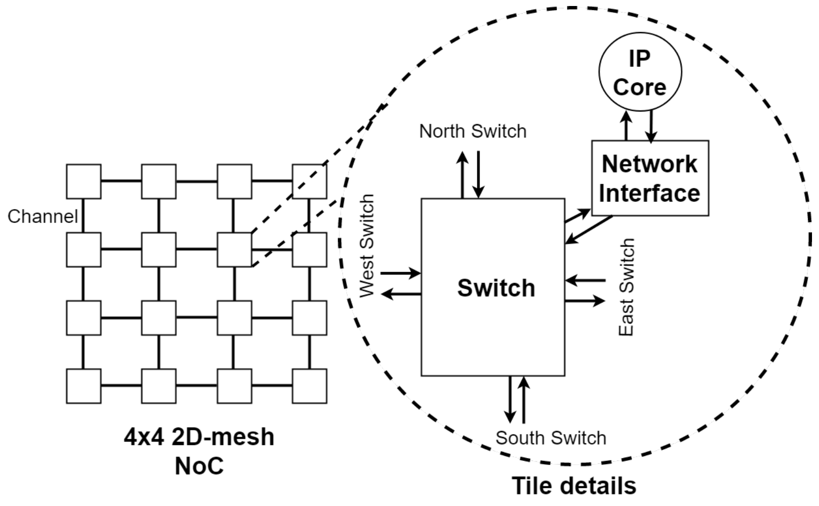

Figure 3.3: A 2D-mesh topology and the details of its nodes

which perform the input buffering and route computation to select one of the other four outputs, and five output port modules (OPMs), each of which arbitrates between four other inputs. These IPMs and OPMs are connected using a shared crossbar. Compared to a variant MoT, another major difference is the existence of multiple routing paths between a source and destination in 2D mesh. This path diversity can be used for load balancing to minimize congestion, hence improving overall network performance.

3.2 Routing Algorithms

After deciding the network topology, the next step is to select an appropriate routing algorithm that determines a path between a source and a destination. These algorithms are important and have impact on network performance and power consumption. While the usual approach is to select a shortest routing path, it is sometimes beneficial to pick a longer path in order to avoid an already congested minimal path.

The routing algorithms can be broadly divided into three categories: deterministic, oblivious, and adaptive [22].

3.2.1 Deterministic Routing

In deterministic routing, the route between a source and a destination is statically pre-determined and fixed. This approach is the simplest way of routing. *Dimension-ordered routing,* such as *XY* is an example of deterministic routing, where the packet first travels in the X direction until it reaches the destination's X dimension, after which it travels in the Y direction and reaches the destination. XY routing, and its dual YX are very commonly used [47], [166]. Recently, other deterministic routing approaches have also been introduced [58], [159].

Although deterministic routing is simple and easy to implement, it also has some limitations. This routing approach does not provide any path diversity – always selecting the same path for routing between a source and a destination, which can lead to congestion that can incur performance overheads. Deterministic routing is also not fault-tolerant: if a links on the fixed path break due to issues such as aging, the packets are unable to route around these faulty links and the network has to stop operation. However, for uniformly distributed traffic scenarios, and non-faulty conditions, this routing approach is often an excellent solution.

3.2.2 Oblivious Routing

Oblivious routing is more flexible than deterministic routing, where the routing paths are dynamically determined and are not fixed [170], [101], [167]. Therefore, multiple routes exist between the same source/destination pair in oblivious routing. An example is *O1TURN model,* where the first dimension for routing a packet is selected randomly, and the packet is only allowed to turn once [167]. Although more flexible and simple, this routing approach still does not consider network state, such as congestion, when deciding routes, and therefore can incur performance bottlenecks for non-uniform traffic.

3.2.3 Adaptive Routing

In adaptive routing, paths between sources and destinations are dynamically selected based on the network state, such as congestion [53], [115], [120]. This approach is more advanced, and often more effective for non-uniform traffic. Adaptive routing can also be used to route around the faults in the network. However, this scheme usually results in more complex switch design that can have impact on its latency, area and power. Moreover, since the routes are selected on the fly, there is no guarantee that these paths will not lead to deadlocks. Therefore, additional deadlock-avoidance techniques are used with adaptive routing. A prominent work introducing adaptive deadlock-free routing algorithms is from Glass and Ni [69]. Their proposal was to prohibit a small number of turns such that packets do not form a cyclic dependency on each other, hence avoiding deadlocks.

3.3 Packet Encoding Schemes

The basic unit for data transmission between IP cores is a *packet.* The packet is further divided into *flits* or *flow control units*. A packet typically contains a *header flit,* which carries the destination address information and may contain some data, and *body* and *tail flits*, which usually contain only data. There are two approaches to encode addressing in the header flits: *source routing,* and *destination-based routing* [22].

In source routing, the address field of the header directly contains routes to take at each router node on the path to destination. This encoding is performed at the source network interface during the injection of the packet. Source routing can lead to very simple router node design, efficient in terms of latency, area, and power. However, this approach can have lower coding efficiency with long address fields. It also can not be used with adaptive routing, where the paths are determined dynamically.

In destination-based routing, the address field of the header only contains the destination address. This address can be in the form of *X, Y* coordinates of the destination, or a bit string that contains a bit for each node of the network, which is 1 if the node is a destination, and 0 otherwise. The destination-based routing generally have small address field but can lead to more complex switch design as each switch must decode and select the correct path based on the destination address.

In this thesis, for multicast transmissions, both source routing and destination-based routing have been used. Packet encoding for multicast is a challenging problem, where multiple destinations must be encoded in the header but still maintaining a good trade-off between coding efficiency and switch design complexity. For multicast in a variant MoT, source routing was used, which led to very simple switch designs and had only small overheads in terms of coding efficiency. This is because each switch has only two output ports so there are very small number of possible routes to select. On the other hand, for 2D mesh, source routing was infeasible for multicast as each switch has five ports and therefore there are a large number of possible routes to encode, leading to a long addressing field. Hence, destination-based routing with a simple bit string was used. Interestingly, bit-string addressing still yielded simple switch designs.

3.4 Synchronous Unicast Router: Micro-Architectures and Performance Optimizations

In the early 2000s, a synchronous NoC router typically consisted of 4-5 pipeline stages, each of which performed a different functionality in series [41]. These functionalities include buffering packets in one clock cycle, followed by route computation to select the correct path in the next cycle, then arbitration to gain access to the shared resources such as an output channel, etc. Later, several optimizations had been proposed to perform some or all of these operations in parallel, resulting in a single-cycle router operation [102], [108], [149]. Recently, more extreme optimizations have been proposed that can route a packet through multiple routers in a single cycle, which has resulted in very high-performance SMART NoC designs [104].

3.4.1 A Traditional 5-Cycle Router

The early classic synchronous routers involved five stages as shown in Figure 3.4 [41]. These stages perform the following functionalities serially, each taking one clock cycle: (i) *buffer write:* a flit is written to a free input buffer space corresponding to a pre-determined virtual channel (VC) (ii) *route computation (RC):* decode the addressing in the header and select the correct output port for routing, (iii) *virtual channel allocation (VA):* assign a free VC of the downstream router to the flit buffered at the current router, (iv) *switch allocation (SA):* for each output channel of the current router, multiple input ports arbitrate to gain access, and (v) *switch traversal (ST):* route the flit from the input to the output of the router through a crossbar. Finally, the flit then traverses the link in the next clock cycle to reach the neighboring router, which is called *link traversal (LT)*.

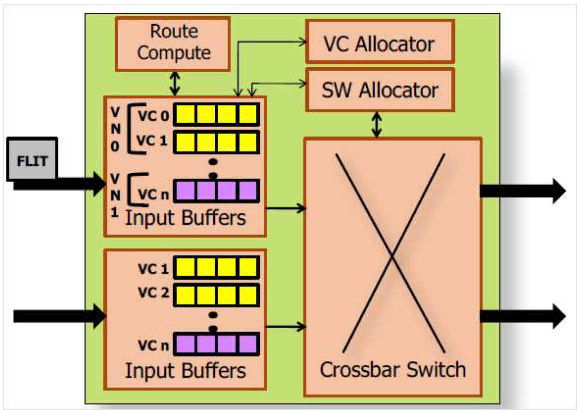

Figure 3.4: A traditional 5-cycle router micro-architecture [103]

3.4.2 Recent Single-Cycle Routers

The above 5-cycle router can be radically transformed to perform all its operations in parallel in a single cycle [102], [108], [149]. Such a transformation is possible using two pipeline optimization techniques: *lookahead* and *speculation.* In lookahead, the results of a pipeline stage are fast forwarded to a later stage, which is waiting on these results, so it can start early computation. In speculation, two pipeline stages, which are normally performed in series, can be performed in parallel, where the latter stage is completed assuming certain outcomes of the previous stage. Speculation may require rollback of the latter stage, if it is found out that previous stage resulted in a different outcome than the assumed one.

Figure 3.5 shows the pipeline operations of a single-cycle router. This router receives a *route information* from an upstream router, along with the flit. This route information includes which output port to select at the current router and the destination of the packet. The destination is used to compute the output port to select, but for the downstream router (i.e. RC). In parallel, based on the correct output port information for the current router, switch allocation and VC allocation are performed speculatively, assuming that there is a free VC at the downstream router. The buffer write is also bypassed speculatively, assuming that the switch allocation will be completed successfully for the flit and there is a free VC at the next router. Under these assumptions, the flit directly goes into switch traversal. By the end of the ST, SA and VA results are available, and it is checked if all assumptions were correct or any rollbacks are required. Usually, no rollbacks are needed for uniform uncongested traffic scenarios and the flit traverses the link (LT). However, under congestion, these

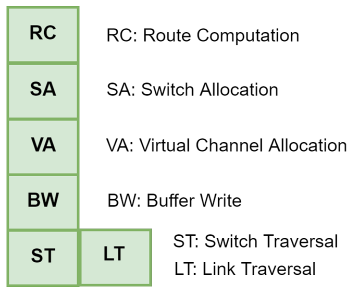

Figure 3.5: Pipeline stages in single-cycle routers

rollbacks may be required, which can take additional cycles to re-do SA and/or VA.

3.4.3 Extreme Bypassing in Single Cycle Using SMART NoCs

While in the previous router, a flit traverses only a single switch in one clock cycle, a high-performance SMART NoC has been introduced, where a flit can route through multiple routers and links in a single cycle [104]. This NoC uses a high-speed monitoring network, that shadows the regular datapath network, to dynamically pre-allocate the routers on the path to destination in advance of the arrival of a flit. Before injecting the data flit at the source, a small 'monitoring' packet is injected, which rapidly traverses through the monitoring network to the destination, pre-allocating the routers on the path, which will allow them to operate in a simple very low-latency "bypass" mode. The data flit then simply follows the pre-allocated path to reach the destination in a single cycle. For a SMART NoC, operating at 1 GHz in 45 nm, it is shown that 9-11 hops can be bypassed in a single clock cycle.

Figure 3.6 briefly shows the structure and operation of a SMART NoC. For simplicity, only four routers are assumed with the leftmost one as the source and the rightmost one as the destination. These routers are connected using both the regular datapath network and a monitoring network with dedicated links between the source router and each of the other routers. Assume that the injected flit at the source gets stored in buffer 1 of the router. In the first cycle, local arbitration is performed between the stored flits to gain access to the east output channel. Assuming that the flit in buffer 1 wins this arbitration, in the second cycle, a request is forwarded on the monitoring network to all the routers on the path to destination. At each of these routers, pre-allocation is performed using the monitoring request, i.e., the routers are configured into a bypass mode. In the third cycle, the data

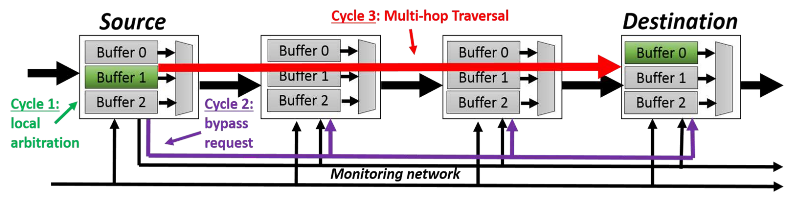

Figure 3.6: A SMART NoC structure and operation

flit simply traverses through all these routers in a single cycle to reach the destination.

Although SMART NoCs can achieve very high performance, they can also incur significant power and area overheads. These overheads are due to the expensive monitoring network with dedicated links (nearly 24) emanating from each router to all the other routers. Moreover, each such link can be 2-4 bits wide. As a result, a number of these switches have several dozen extra wires (e.g. 48-96) solely devoted to monitoring.

3.5 Multicast Techniques and Related Work

There are two main techniques to perform multicast in NoCs: one is serial and the other parallel. There has been significant research on both of these approaches.

3.5.1 Techniques

Figure 3.7 illustrates the two main multicast techniques: *serial path-based multicast* and *parallel tree-based multicast.* In path-based, a multicast packet is serially routed from the source to its first destination, from there to the next, and so on. For example, in Figure 3.7(a), a packet is first routed from source A to destination D, from there to E and C and finally to B. This technique is simple but can incur significant latency overheads for large number of destinations. Tree-based multicast is more widely-used, where a multicast packet is first routed on a common path from the source towards all destinations. When this common path ends, the packet is replicated and the new copies also follow a recursive tree approach, replicating multiple times to reach the destinations. In Figure 3.7(b), the packet first gets replicated at the source and one copy is sent to D, E and the other copy is sent to B, C. Each of these copies is replicated further at intermediate nodes to finally reach the destinations.

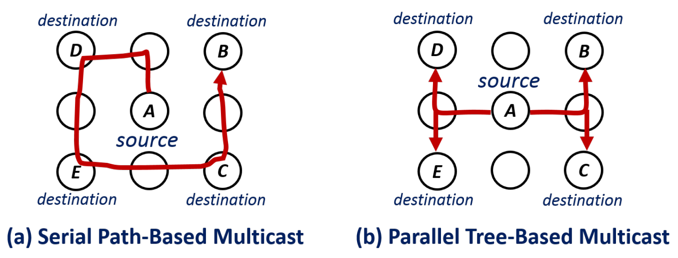

Figure 3.7: Multicast techniques: (a) path-based, (b) tree-based

3.5.2 Related Work

The path-based technique is far less attractive than the tree-based due to its serial operation, where packet cycles through all destinations one by one, which can have serious latency overheads, specially for large number of destinations. However, there has been previous work that tries to improve upon this issue by dividing the paths into smaller sub-paths using partitioning schemes and selecting the best ordering of routing to the destinations such that the overall path is optimized [4], [42]. A similar approach was also proposed for 3D NoCs recently [54].

Several recent synchronous NoCs use the tree approach for high-performance multicast, but these can still incur significant cost overheads. Early tree approaches used multiple unicast packets, sent to each destination, to set up paths for a multicast packet. The output port taken by each individual unicast packet at any router, is stored in a routing table inside that router, which is later accessed by the multicast packet. This preconfiguration phase can be expensive in terms of network latency, extra congestion and power [81], [88]. Recent approaches avoid setup entirely and dynamically compute the multicast tree paths based on the destinations and an underlying routing algorithm. However, these approaches lead to complex router designs due to highly-customized route computation [193], [200], multiple virtual channels per port [149], and turn prohibitions and other expensive methods to avoid deadlocks [106], [165], [193].

Finally, a recent NoC supports high-performance multicast [105], extending the unicast-only state-of-the-art SMART NoC [104], to perform multicast. This NoC is discussed in more detail later. The multicast SMART NoC achieves full broadcast in just 2 cycles for an 8x8 2D mesh using an early arbitration and channel pre-allocation approach, facilitated by a high-speed monitoring network that shadows the datapath. However, this NoC supports multicast either using a full-chip broadcast and then dropping packets at non-destinations or by routing several unicasts serially to each destination. Both of these approaches can have serious energy overheads. Moreover, the SMART monitoring network, as highlighted earlier, also adds significant area and power costs.

3.6 Leading Synchronous Multicast NoCs

Two state-of-the-art synchronous multicast NoCs are discussed: one that uses single-cycle routers to support tree-based multicast [149], and the other is a multicast SMART NoC that uses the SMART framework to optimize the performance of multicast packets [104].

3.6.1 Multicast Using Single-Cycle Routers

The architecture of a single-cycle multicast router is shown in Figure 3.8 [149]. This router performs lookahead route computation (RC), switch allocation (SA), virtual channel allocation (VA), switch traversal (ST), as well as link traversal (LT) in a single clock cycle. There are five basic differences from a single-cycle unicast router in order to support multicast: (i) multicast addressing is used in the packet header that encodes multiple destinations; (ii) a new lookahead route computation is able to decode the new multicast addressing and select multiple output ports, instead of just one, in the downstream router; (iii) switch allocation and virtual channel allocation are now performed for multiple output ports based on the paths taken by a multicast packet; (iv) a new crossbar design is used to support multi-way forking of a multicast flit in parallel; and (v) a low-swing crossbar and links are used, that have low latency, allowing the switch and link traversals to be performed within a single clock cycle.

In Chapter 5, this leading synchronous multicast NoC is analytically compared with the proposed 2D-mesh asynchronous multicast NoC. The proposed asynchronous NoC is shown to achieve better network latency for a multicast scenario as well as lower switch-level area than this synchronous NoC.

3.6.2 Multicast Using SMART NoCs

The basic idea is to use the original unicast-only SMART framework [104] for multicast also without major modifications [105]. To this end, the multicast SMART NoC uses almost the same monitoring

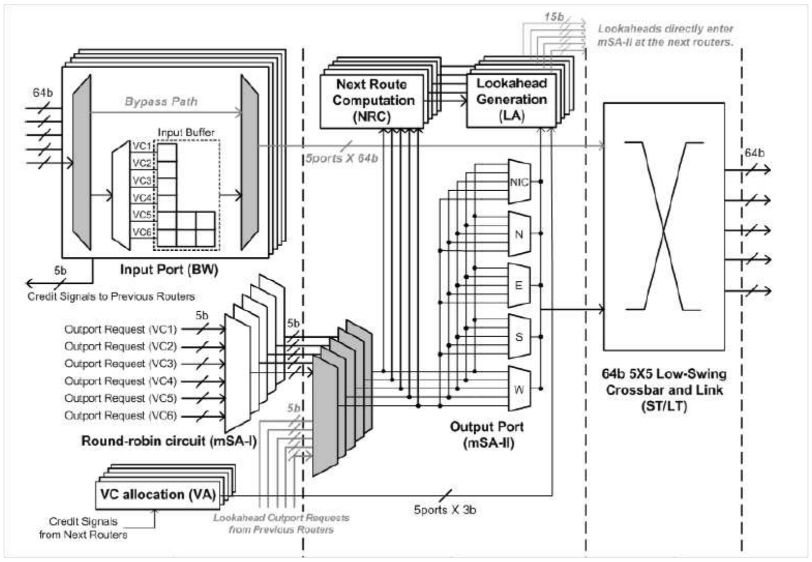
Input Port (BW)
Bypass Path
Input Buffer
VC1
VC2
VC3
VC4
VC5
VC6
64b
5b
Credit Signals to Previous Routers
Outport Request (VC1)
Outport Request (VC2)
Outport Request (VC3)
Outport Request (VC4)
Outport Request (VC5)
Outport Request (VC6)
Round-robin circuit (mSA-I)
VC allocation (VA)
Credit Signals from Next Routers
Lookahead Outport Requests from Previous Routers
5ports X 64b
5ports X 3b
Next Route Computation (NRC)
Lookahead Generation (LA)
15b
Lookaheads directly enter mSA-II at the next routers.
NIC
N
E
S
W
Output Port (mSA-II)
64b 5X5 Low-Swing Crossbar and Link (ST/LT)

Figure 3.8: A single-cycle multicast router architecture [149]

network as the unicast SMART NoC with small changes in the switch design. The monitoring network is now used to pre-allocate routers on different paths to all destinations of a multicast packet, instead of just a single path as in the original SMART NoC. The multicast packet then simply follows these pre-allocated paths, bypassing multiple routers in a single clock cycle to reach its destinations. The new router designs are now able to support multicast addressing and efficient multi-way forking of multicast flits.

The multicast SMART NoC, however, is optimized only for broadcast, and uses inefficient ways to handle multicasts. To support multicast, there are two techniques: 1) if a multicast is to large number of destinations (dense multicast), then it is performed by broadcasting the packet to all nodes, using the SMART framework, followed by dropping copies at the non-destinations, and 2) if multicast is to only small number of destinations (sparse multicast), then it is performed by breaking the multicast packet into several unicasts and sending each unicast serially to each destination using the SMART technique. Both of these methods can have severe power/energy overheads due to significantly extra network utilization compared to a standard tree-based parallel multicast.

This multicast SMART NoC is also later analytically compared with the proposed 2D-mesh asynchronous multicast NoC. Although the SMART NoC achieves better latency than the proposed asynchronous NoC, it can incur significant power and area overheads. Higher power dissipation can result for multicast transmissions in the SMART NoC due to the use of an expensive protocol as highlighted above. In contrast, the proposed asynchronous NoC only uses parallel tree-based transmission to handle multicast and does not involve any wasted energy. Moreover, the multicast SMART NoC can still incur both area and power overheads due to the extra substantial monitoring network, while no such network is used in the proposed asynchronous NoC.

Now that we have covered the background on the three main threads of this thesis: asynchronous design, networks-on-chip, and multicast, the remaining chapters will cover the new research contributions.

Chapter 4

A Local Speculation Approach for Multicast in Mesh-of-Trees NoCs

4.1 Introduction

The first research contribution of this thesis is an efficient asynchronous multicast solution for a variant mesh-of-trees (MoT) topology. Variant MoT is an indirect topology, combining a binary fan-out network (i.e. routing network), and a binary fan-in network (i.e. arbitration network) consisting of low-radix fan-out and fan-in nodes, respectively, as discussed in Section 3.1.1. This simple topology is targeted first, before moving on to the more complex 2D-mesh topology.

In this chapter, multiple routing strategies and network architectures are proposed and evaluated to support efficient parallel multicast in asynchronous NoCs [18], which are now presented briefly.

As a first solution, the standard tree-based multicast approach is performed in an asynchronous NoC with variant MoT topology. In this approach, a multicast packet follows a tree path from source, replicating at intermediate routing nodes if needed, to reach all its destinations. This solution, although simple, is not very efficient in terms of performance. Therefore, further enhancements are proposed.

A novel strategy, *local speculation,* is next introduced for high-performance parallel multicast. Local speculation is a new paradigm, where some routing nodes of a network always locally broadcast every packet (unicast or multicast), i.e. send a packet received on its input through all its output ports. Since some redundant copies might be created due to this 'always broadcast' approach, these copies are throttled by the neighboring routing nodes, which always route based on the packet's actual destination. Therefore, the redundant copies are restricted to small 'local' regions, not allowing them to travel long distances. The former nodes are called *speculative nodes,* which have highly simplified designs as they do not perform any route computation or channel allocation, thereby improving network performance. The latter nodes are called *non-speculative nodes,* which perform the usual route computation and channel allocation as well as have a new capability of throttling redundant packets. Local speculation, therefore, leads to a hybrid network architecture that provides interesting opportunities to mix these two types of nodes.

Local speculation is a unique asynchronous paradigm. This strategy leads to very simple speculative routers, operating much faster than a clock cycle, surrounded by slower non-speculative routers with 'sub-cycle' low-overhead throttling capability, while still maintaining a pipelined operation. This level of simplicity and lightweight designs may not be possible with synchronous techniques.

For an exhaustive design space exploration, two more architectures are introduced besides hybrid, with extreme degrees of speculation. The first does not use any speculation, which is the same as our tree-based solution, while the other is almost fully speculative.

Although the new speculative and non-speculative fanout nodes are very efficient, further power and performance improvements are achieved using novel protocol optimizations. These optimizations are performed for multi-flit packets and are triggered by the header. Speculative routers are very fast but can dissipate extra power due to redundant packets. These extra copies can be minimized by switching to non-speculative mode for body/tail flits after the header is broadcast. In contrast, in case of non-speculative, routing of the header is used to pre-allocate correct outputs for the trailing flits, which are then fast forwarded, optimizing performance.

4.2 Baseline Asynchronous NoC

The new multicast research builds on a previous recent asynchronous NoC, targeting a variant MoT topology, but which only supports unicast communication [78]. This NoC was developed as part of the *C-MAIN* (Columbia University/University of Maryland Asynchronous Interconnection Network) Project: an effort to develop low-cost and flexible NoCs for high-performance shared memory

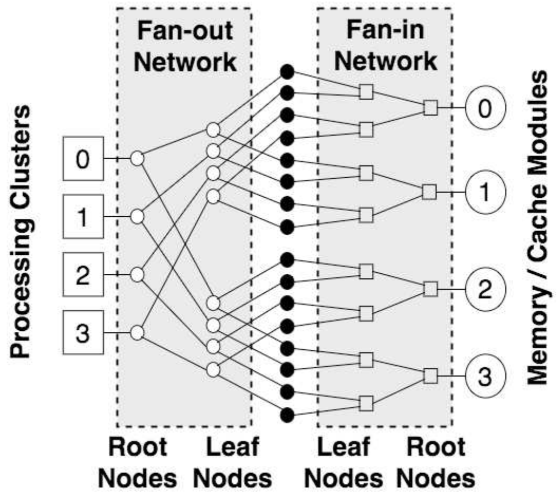

Figure 4.1: Variant mesh-of-trees (MoT): connecting processors to memory modules

architectures. This earlier NoC will be significantly enhanced for multicast and will also form the baseline for the new work.

This baseline NoC uses the variant MoT topology, as shown in Figure 4.1, which consists of very simple routing and arbitration nodes. These nodes are designed using a 2-phase communication protocol with only one roundtrip communication per transaction, resulting in higher performance, compared to 4-phase protocol with two roundtrip communications. Single-rail bundled data encoding is used, which exhibits higher coding efficiency and lower area and energy overheads than the delay-insensitive encoding [147].

The fanout network of the variant MoT, consisting of the fanout nodes, route packets from a source towards the fanin network. A fanout node receives packets on one input channel and forwards them to one of the two output channels, based on the *source routing addressing*. Source routing leads to simpler fanout node designs, where each packet header only contains the address fields of every fanout node on its path, which is only 1-bit wide, identifying the output port for routing: top or bottom.

The fanin network, on the other hand, consists of the fanin nodes that arbitrate between multiple packets and send the winner to the attached destination. A fanin node receives packets on two input channels, performs arbitration in continuous time, and forwards the winning packet on the single output channel.

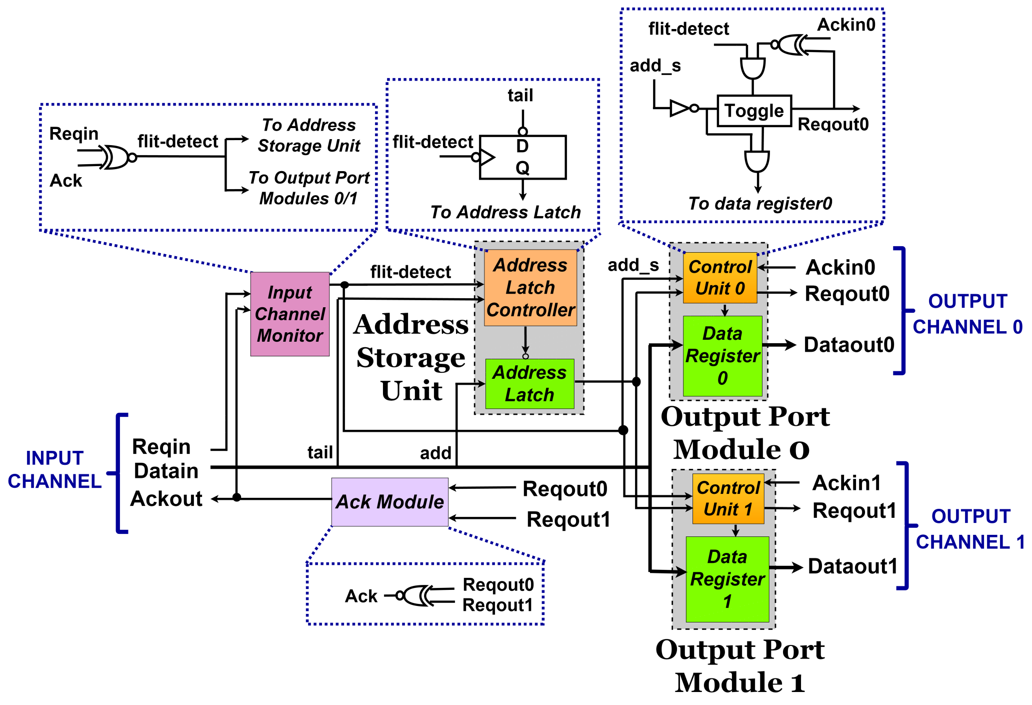

Figure 4.2: Baseline fanout node

4.2.1 Fanout Node

The micro-architecture of a baseline fanout node is shown in Figure 4.2. There are 5 main components: *Input Channel Monitor, Address Storage Unit,* two *Output Port Modules*, and *Ack Module.* The Input Channel Monitor detects the arrival of each flit of a packet. The Address Storage Unit stores the address of the header flit, which it holds until after the arrival of the tail flit. The two Output Port modules, which are *normally opaque,* manage routing and flow control of each output channel. Finally, the Ack Module observes when either output channel transmits a flit, in unicast traffic, then completes handshaking on the input channel.

The fanout node has relatively simple operation [[78]]. When a packet arrives on the input channel, the header is directed to inputs of both Output Port Modules. The Input Channel Monitor detects the arrival of the flit, enabling storing of its address in the Address Storage Unit. The monitoring *flit-detect* also partially-enables both Output Modules; the one receiving the correct address is then activated, and the flit is sent out on that output channel along with a toggled *Reqout*. This signal enables two concurrent operations: closing of the Output Port Module (for data protec-

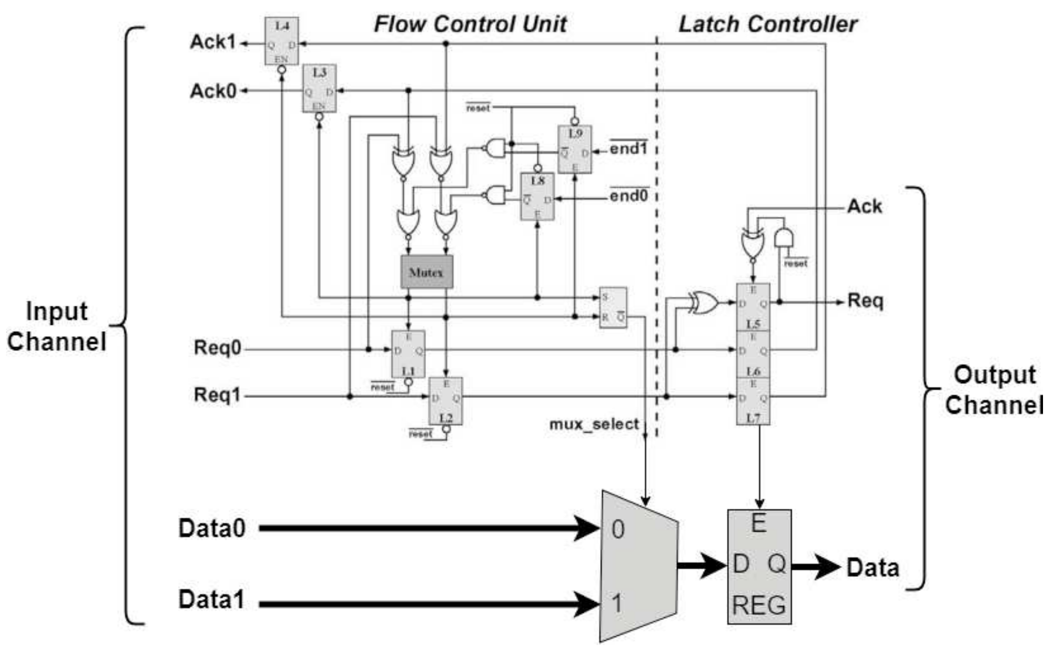

Figure 4.3: Baseline fanin node

tion) and enabling of the Ack Module to generate the acknowledge on the input channel. Finally, the downstream node acknowledges (toggles the *Ackin* signal), completing the handshaking on the output channel. Similar operation occurs for the remaining body and tail flits.

4.2.2 Fanin Node

The micro-architecture of a baseline fanin node is shown in Figure 4.3. There are six main components: a 2-way arbiter, two on-demand (i.e. normally-closed) input registers corresponding to each input stream, a 2:1 data mux, a capture-pass (i.e. normally-transparent) output register, and an ack generator consisting of two on-demand latches.

The functionality of these components is first presented before describing the operation of the fanin node. The arbiter must operate in continuous time (in contrast to synchronous arbiters), where inputs arrive at arbitrary temporal intervals not discretized to clock cycle. Therefore, the arbiter is implemented as an analog mutual-exclusion (mutex) element, and is the only except to use of standard-cell components in the fanin node (see [147] and Section 2.3.2 for more details). Based on the arbitration, the correct input channel of the fanin node is allocated: enable the selected input register and select the winning packet's data stream of the data mux. The output register performs flow control on the output channel to the downstream fanin node. The ack generator is used to complete handshaking on the winning input channel after routing each flit of the packet.

In a basic scenario with no contention, a header flit of a packet arrives on one of the input channels, accompanied with its bundling request. This request is used by the mutex to perform arbitration for every flit. In the absence of competition, the mutex grants access to this header. This acquisition is followed by three operations in parallel: (i) the correct L1-L2 input request register is allocated i.e., made transparent, (ii) the appropriate data stream in the data mux is selected, (iii) the corresponding latch of the ack generator is also enabled, and (iv) disable any competing request (pending or future) to the arbiter to enforce wormhole routing: this 'kill your rival' approach biases the mutex towards the winning packet so that the next flits of this packet are prioritized for routing through the fanin node instead of any competing packet. Next the header is sent out on the output channel through the capture-pass register; the toggling of the output request closes the normally-transparent output register transiently for protection and flow control. In parallel, the mutex is released, and the ack is sent on the input channel through the already enabled latch of the ack generator.

Similar operations happen for body and tail flits. In addition, once the tail flit arrives and is being sent out, the blocking of any competing request to the arbiter is also released so as to enable routing of the other packet through the fanin node.

4.2.3 Results

This baseline NoC has been shown to handle unicast traffic very efficiently. At the node-level, the asynchronous fanout and fanin nodes showed 5.6-10.7x lower energy per packet and 2.8-6.4x lower area than similar synchronous nodes in post-layout comparisons. At the network-level, an 8x8 MoT asynchronous NoC, built using the above nodes, showed 1.7x lower network latency with comparable throughput than a 800 MHz synchronous NoC.

4.2.4 Baseline for the New Multicast Research

This baseline NoC cannot directly handle parallel multicast, and only supports unicast. The only way to route multicast packets in this NoC is using a serial unicast-based approach, where for each multicast packet, the source NI creates multiple unicast copies destined to different destinations, which are then injected and routed serially. However, as shown in Section 4.5, performing multicast using the serial unicast approach can have severe performance and power overheads.

In this chapter, the above network will be significantly enhanced to support multicast operation. Only fanout nodes will be modified; existing fanin nodes are directly reused. In particular, fanout nodes are responsible for all routing, so must support distribution of multiple packet copies; hence they must be instrumented with new addressing and replication capability. In contrast, even with multicast, the existing fanin network will still arbitrate between multiple packets, and direct all packets to their destinations without making any routing decisions.

4.3 Proposed Multicast Approaches

Complete new asynchronous solutions to support multicast in a variant MoT topology are presented in this section. Building loosely on the previous unicast solution, the new work starts with a basic tree-based parallel multicast approach, and then enhances it with novel "local speculation" technique. Moreover, additional optimizations are also applied to the new routers to further improve their power and performance.

4.3.1 Simple Tree-Based Multicast

The first contribution is a simple tree-based parallel multicast, applied for the first time to a general-purpose asynchronous NoC. A multicast packet follows a common path towards its destinations and replicates when this common path ends, then copies are routed in parallel towards the destinations. Routing of a unicast packet is the same as in the baseline network. The fanout network architecture, in Figure 4.4(a), has all non-speculative nodes. New fanout nodes are designed to handle parallel replication, as described in Section 4.4(b).

Source routing is used to encode the address for every fanout node on each path to the destination(s). The address at each fanout node must encode 3 symbols: top route, bottom route or both. Therefore, 2-bit encoding is used for the address field of each fanout node.

This basic parallel tree-based multicast is simple, but not efficient in terms of latency and throughput. The new fanout nodes are relatively slow due to expensive route computation and channel allocation protocols, required to handle a more complex set of transmission modes. An-

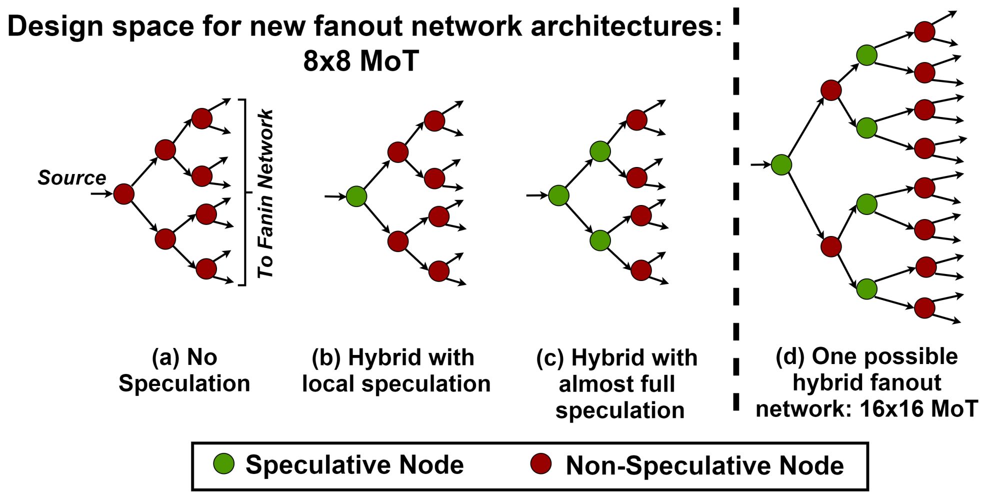
Design space for new fanout network architectures: 8x8 MoT
Source
To Fanin Network
(a) No Speculation
(b) Hybrid with local speculation
(c) Hybrid with almost full speculation
(d) One possible hybrid fanout network: 16x16 MoT
Speculative Node
Non-Speculative Node

Figure 4.4: New fanout network architectures: (a)-(c) full range for 8x8 MoT, (d) One possible hybrid network for 16x16 MoT

other limitation is that the source routing, as described above, leads to low packet coding efficiency, which does not scale with larger network sizes.

4.3.2 Local Speculation-Based Multicast

A new strategy, *local speculation,* is introduced for high-performance parallel multicast. In local speculation, a subset of fanout nodes is speculative and always locally broadcasts a multicast (or unicast) packet. These nodes are surrounded by non-speculative nodes that always send packets on the right path(s) and throttle any received redundant packets from the speculative nodes, restricting these packets to small "local" regions. A hybrid network architecture is introduced to mix these two types of fanout nodes. Figure 4.4(b) shows one possible hybrid fanout network for an 8x8 MoT. The detailed design of a speculative node is presented in Section 4.4(a).

Interestingly, the use of this new hybrid network also improves packet coding efficiency using a *simplified source routing*, which is not required to encode the addressing for the speculative nodes on the path to the destination(s). This simplification is a direct consequence of the simplicity of speculative nodes that always broadcast and therefore do not require any addressing. As a result, only a *subset* of fanout nodes (i.e. non-speculative ones) requires address fields in the packet header.

The operation of the hybrid fanout network is illustrated using two simulations: for unicast and

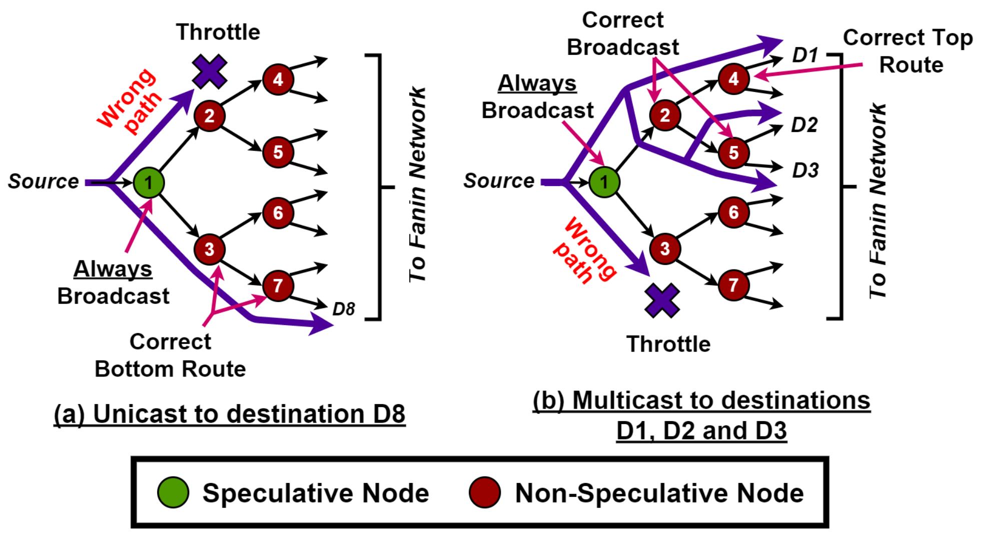

Figure 4.5: Hybrid network: unicast/multicast simulations

multicast.

Figure 4.5(a) shows the routing of a unicast packet in an 8x8 MoT network. The packet is first broadcast by the speculative root node, sending one copy on the right path and the other on the wrong path. The copy on the wrong path is throttled in the top sub-tree by non-speculative node 2. The copy on the right path is forwarded by non-speculative nodes 3 and 7 through their bottom output ports, based on actual addressing, towards D8.

Figure 4.5(b) shows the routing of a multicast packet. Similar to unicast, the speculative root node broadcasts, sending copies on the right and wrong paths. The latter is throttled in the bottom sub-tree by non-speculative node 3. As non-speculative nodes can *also* broadcast, the copy on the right path is broadcast by node 2 on both output channels. One copy is correctly routed to D1 by node 4 through its top output port, while the other is correctly routed to D2 and D3 by another broadcast at node 5.

So far, two architectural design points have been covered, non-speculative and hybrid, as shown in Figures 4.4(a) and (b). To complete the design space, a third extreme point is introduced, an almost fully-speculative architecture. As shown in Figure 4.4(c), only the last level must be non-speculative, since the fanin network cannot throttle any misrouted packets. Such global speculation can achieve high performance, but suffers from major power overheads due to the large distances traveled by misrouted packets. Overall, this hybrid architecture approach provides an interesting design space, allowing different mixes of speculative and non-speculative nodes, resulting in various cost trade-offs.

While the focus of the discussion of the above contributions and the evaluations in this chapter is on 8x8 MoT, it is interesting to consider future directions of larger-sized networks. The hybrid architecture for the larger networks has more degrees of freedom to mix the speculative and non-speculative nodes and therefore a wider design space. Figure 4.4(d) shows one possible hybrid fanout network for a 16x16 MoT, out of a family of many possibilities.

4.3.3 Protocol Optimizations

The above speculative and non-speculative nodes are efficient, but may incur some overheads, which can be minimized for multi-flit packets using protocol optimizations triggered by the header. Speculative nodes can create redundant copies and therefore lead to extra switching power. Non-speculative nodes have complex route computation and channel allocation steps, which are performed for every flit, and therefore, can lead to performance bottlenecks.

Extra power due to speculative nodes is minimized by reverting immediately to the non-speculative mode for body flits of packets. Therefore, no redundant copies are created for the body flits. Routing of the header is used by the node to identify and close the output port on the incorrect path before the trailing body flits arrive. The body flits are then routed through only the correct output port(s), effectively reverting to non-speculative mode and thus saving power. Arrival of tail flit is used by the node to return to always broadcast state. These optimized nodes are called as *power-optimized speculative nodes,* and are described in Section 4.4(c).

Latency and throughput of the non-speculative nodes is optimized by performing route computation and channel allocation only for the header and not for the other flits. Routing of the header is used to reserve the correct output channel(s) for the remainder of the packet (body/tail flits), i.e. the correct path through the node is pre-allocated for these flits. The body/tail flits are then fast forwarded through the allocated output channel(s) after their arrival, optimizing latency of these flits and improving overall network latency and throughput. These optimized nodes are called as *performance-optimized non-speculative nodes,* and are described in Section 4.4(d).

4.3.4 Target Parallel Multicast Networks

The above parallel multicast approaches are incorporated into five network configurations, representing three distinct points in the design space of speculative architectures: (i) *non-speculative,* (ii) *hybrid,* and (iii) *almost fully speculative.* The goal is to explore the tradeoffs associated with varying degrees of speculation and protocol optimizations.

In particular, the chapter targets two non-speculative networks (**BasicNonSpeculative, OptNonSpeculative**), two hybrid networks (**BasicHybridSpeculative, OptHybridSpeculative**), and one extreme case of a nearly fully speculative network, with non-speculative nodes only at its leaves (**OptAllSpeculative**). To support the design of these networks, four distinct fanout nodes are introduced.

4.4 Proposed Fanout Node Designs

This section presents the design and operation of the new fanout nodes, which are the main building blocks of the new parallel multicast networks. The basic new networks are composed of the unoptimized fanout nodes, and the more advanced new networks are composed of the power- and performance-optimized fanout nodes, discussed in turn.

4.4.1 Unoptimized Speculative Fanout Node

The structure and operation of the simple unoptimized speculative fanout node are first presented.

4.4.1.1 Structure

The micro-architecture of the simple speculative node is shown in Figure 4.6. There are three main differences from the baseline fanout:

(i) *Drastically simplified design:* due to elimination of the Input Channel Monitor and Address Storage Unit, which are not needed as the new node always broadcasts every packet and does not perform any route computation.

(ii) *New Output Port Modules with normally-transparent registers:* these registers can now be used in the new node as it always sends both ways, unlike the baseline node that used opaque

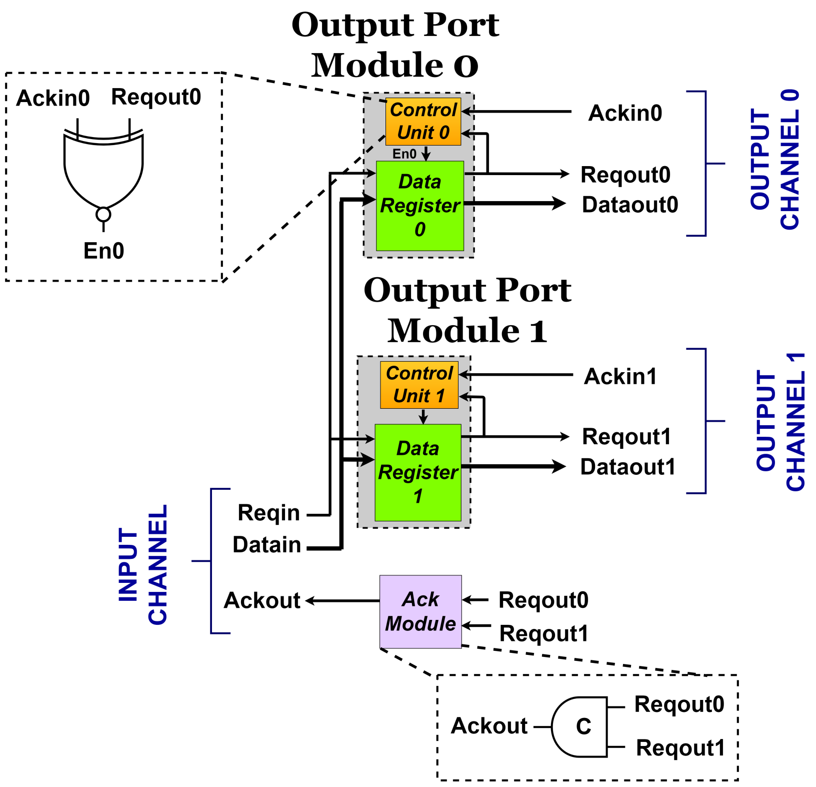

Figure 4.6: Unoptimized speculative fanout node

registers since it needs to select the single correct path before sending the packet forward. Normally-transparent registers in the node have a very simple flow control, implemented by a single XNOR gate, and also provide very low forward latency.

(iii) *a new Ack Module:* this Ack Module now synchronizes both output channels before completing the handshaking on the input channel, i.e. ack to left is sent only after a flit is sent on both the output channels. In contrast, no such synchronization is required for the ack module of the baseline as it sends ack after a flit is sent out on one of the two output channels. Hence a *C-element* is used to implement the new ack module, as opposed to an *XOR* gate in the baseline.

4.4.1.2 Operation

This node has a very simple operation, which is the same for any type of packet and its flits: *always broadcast.* A packet on the input channel can be a correctly routed unicast, or multicast going to either or both outputs, or any misrouted packet from previous node. When a flit arrives on the input channel, it is directed to both output channels, along with the generation of *Reqouts*. These *Reqouts* perform two concurrent operations: close the Output Port Modules for data protection, and enable Ack Module to generate *Ack*. Finally, when the downstream nodes toggle *Ackin(s)*, handshaking is complete on the output channel(s).

4.4.2 Unoptimized Non-Speculative Fanout Node

The structure, operation and design details of the non-speculative fanout node are now presented, but without considering any protocol optimizations.

4.4.2.1 Structure

The micro-architecture of the new node is shown in Figure 4.7. Overall, the structure is similar to the baseline fanout with identical key components: Address Storage Unit, Input Channel Monitor, two Output Port Modules with their Control Units, and an Ack Module. However, all these units are now more complex, to support parallel replication for multicast and throttling of any misrouted received packets.

The Address Storage Unit now stores a 2-bit source routing multicast addressing than 1-bit as in the baseline.

Similar to the baseline fanout node, the Input Channel Monitor detects arrival of the flits. Additionally, this unit now decodes the address bits and generates the correct routing information, and also checks if the packet is misrouted from the upstream node.

Each Output Port Module still consists of a normally-opaque register with a Control Unit. However, both Control Units can now enable their corresponding registers for a packet going both ways, or keep the registers disabled for a misrouted packet.

Finally, the Ack Module now completes handshaking on the input channel for three cases: if a flit is sent out on exactly one of the output channels, or both output channels, or if it is a misrouted flit. The Ack Module also notifies the Output Port Module(s) control unit, when a flit has been sent on the corresponding output channel(s).

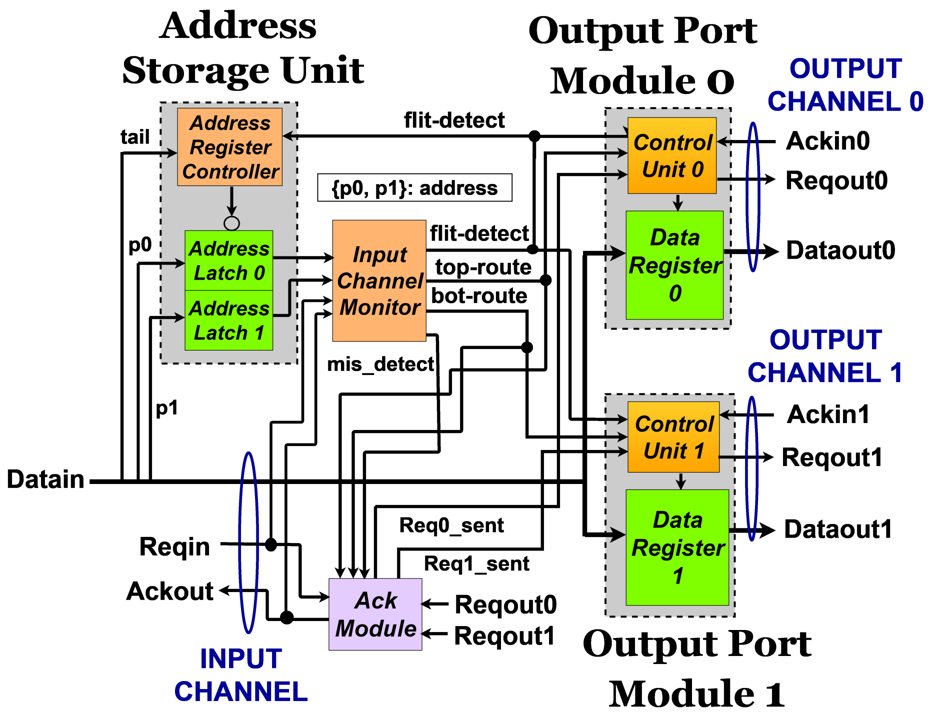

Figure 4.7: Unoptimized non-speculative fanout node

4.4.2.2 Operation

Three packet types can arrive on a node's input channel: unicast, multicast (going to one or two output ports), and a misrouted packet from the previous node.

In case of a correct unicast packet, the header is first directed to both the Output Port Modules. The Input Channel Monitor detects the flit arrival, enables storing of the address and also partly enables both Output Port Modules. Depending on the address, the monitor generates the *top-route/bot-route* routing signals to enable the correct Output Port Module, which then sends out the flit along with its bundling request if there is no congestion on the output channel. The generation of *Reqout* leads to two concurrent operations: closing the Output Port Module transiently for data protection, and enabling the Ack Module to generate the correct *Req0/1_sent* status signal. *Req0/1_sent* signals perform two operations: (i) identify to the Output Port Modules if the flit on the input channel is new or stale; they disable the Output Port Module, right after a flit is sent out on the output channel, hence avoiding any potential resampling, and (ii) these signals, together with the *top_route* and *bot_route*, are used in the Ack Module to generate ack on the input channel. Similar operations occur for body and tail flits.

In the case of a correct multicast packet, if intended for exactly one direction, a similar protocol to the unicast packet is followed. For multicast packets going to both outputs, the Input Channel Monitor enables both output ports for routing. After *Reqouts* are generated on both output channels, the Ack Module completes handshaking on the input channel. All internal operations (data protection, no resampling) are similar to unicast, but now done for both Output Port Modules.

Finally, for a misrouted packet on the input channel, the Input Channel Monitor detects this packet, and keeps the Output Port Modules closed while enabling the Ack Module to complete handshaking on the input channel so that the next packet can arrive and throttle the earlier misrouted one.

4.4.2.3 Design details of sub-modules

The design details of three important components of the unoptimized non-speculative node are described below: Input Channel Monitor, Control Units, and Ack Module.

As shown in the Figure 4.8(a), in addition to generating the *Flit_detect,* the new Input Channel Monitor decodes a 2-bit source routing address to determine whether the packet is intended for one of the four options: top output port, bottom output port, both output ports or is misrouted from the upstream node. In particular, the Input Channel Monitor uses the stored address bits from the Address Register to compute *top_route*, *bot_route*, and *Mis_detect* status signals. If both address bits are '0' then it is misrouted packet, otherwise appropriate routing signals are generated.

Figure 4.8(b) shows the details of the Control Unit of an Output Port Module. This unit produces the enable for normally-opaque data register of the Output Port Module, and also generates the output request. To generate both the enable signal for the data register as well as the output request, all of the following four conditions, corresponding to the four different inputs to this unit, must be satisfied: (i) a flit has arrived on the input channel (*flit_detect*), (ii) the flit is intended for this output port (*top_route* or *bot_route*), (iii) the flit is not stale, i.e. the output request for this flit has not been generated yet (*Req_sent* from Ack Module), and (iv) the output channel is not congested, i.e. *Ackin* has been received from the downstream node for the previous flit.

Figure 4.9 shows the design details of the Ack Module. The Ack Module generates the ack on the input channel, and also produces two status signals (*Req0_sent* or *Req1_sent*), showing which

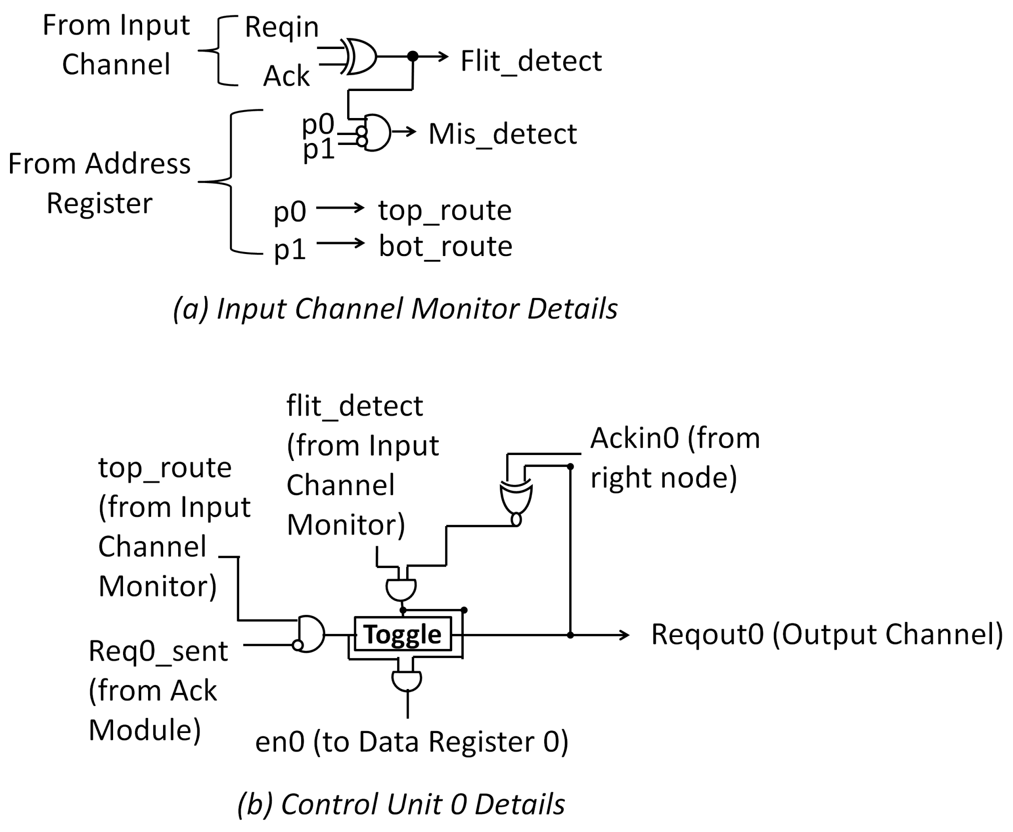

Figure 4.8: Unoptimized non-speculative fanout node: Input Channel Monitor and Control Unit details

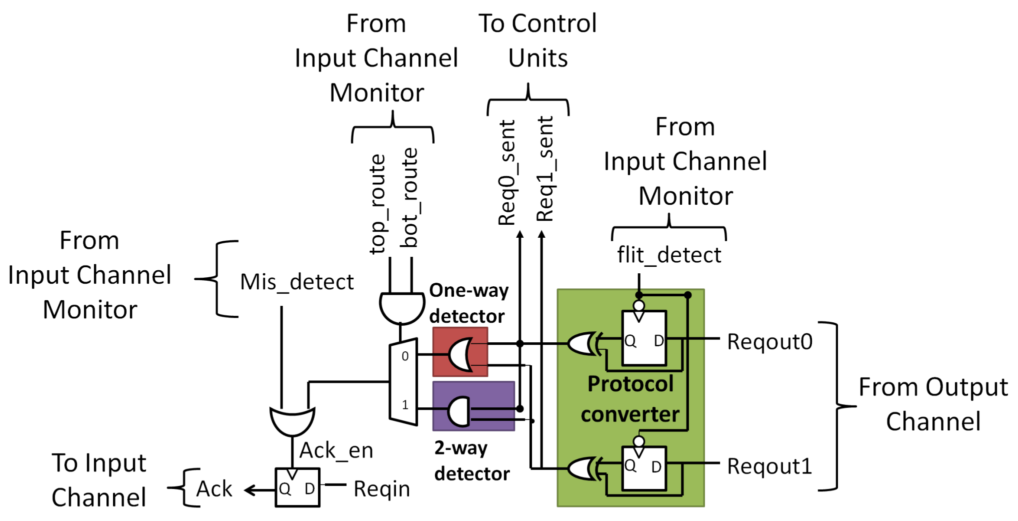

Figure 4.9: Unoptimized non-speculative fanout node: Ack Module details

output channels have generated the output requests. The Ack Module takes the two output requests as inputs, and the routing information from the Input Channel Monitor (*top_route*, *bot_route*, *Mis_detect*). The toggling of each two-phase *Reqout* leads to the generation of the corresponding level signal (*Req_sent*), through a protocol converter. If a flit was intended for only one output channel, then the one-way detector detects this case using the appropriate *Req_sent*, otherwise if the flit has been routed through both output channels then this case is detected by the two-way detector. Based on the routing information, the output of the correct detector is selected, which is then used to generate the ack on the input channel. If the flit was misrouted from the upstream node, then no output requests will be generated, in which case, *Mis_detect* will be simply used to generate the ack. After generation of the ack, the Input Channel Monitor resets the *Flit_detect* and *Mis_detect*; the former resets the Ack Module and the *Req_sents* in the case of the correctly routed flit, and the latter resets the Ack Module for the misrouted flit.

4.4.3 Optimized Speculative Fanout Node

The idea of this optimization is to speculatively route *only the header flit* through both output ports, while in parallel storing its address information, which can then be used for body flits to switch to non-speculative mode. Therefore, the body flits are not misrouted, and are only routed through the correct output ports, thereby potentially saving significant power. Finally, the arrival of the tail flit is used to return the node to speculative mode, and the tail is also routed through both the output ports.

4.4.3.1 Structure

Figure 4.10 shows the micro-architecture of the optimized speculative nodes. There are four main differences from the basic speculative nodes: (i) the new Input Channel Monitor is instrumented to detect the arrival of flits on the input channel, and in particular, to detect the tail flit; (ii) more complex Output Port Modules, which can now store address information while routing the header flit, and can switch between different operating modes: *speculative to non-speculative* after routing of the header flit, and *non-speculative to speculative* after routing of the tail flit; (iii) a new Ack Module that generates *Ack* for two different cases: those body flits that are routed correctly but only through one output channel in non-speculative, and all other flits in the speculative mode. For the former, *Ack* is sent after the flit is routed on exactly one output channel; for the latter, the protocol is same as the basic speculative nodes; and (iv) a new Phase Corrector Unit is added, which is used to generate the output request with the correct phase, corresponding to each output port.

4.4.3.2 Operation

The operation of the optimized speculative fanout node is the same for all types of packet: unicast or multicast or a misrouted packet from the previous node.

To demonstrate how the optimization works, a multiflit unicast packet intended for only one output port is assumed. Once the header of this packet arrives, it is speculatively routed in parallel through both the output channels. Its address is then used by the Output Port Modules to identify the correct route (top or bottom), and *to block the incorrect route for all the subsequent body flits:* the control unit of the incorrect Output Port Module will close the output register for the trailing body flits. This control unit also generates a *power-save* status signal to inform the Ack Module about which output port will be disabled for the body flits. After the ack is sent on the input channel, corresponding to the header, the Ack Module switches its mode to one of the power-save modes, so it now only needs to check for the routing of the body flits through one of the output ports before generating an ack.

After a body flit arrives, it is simply routed through the correct output channel, while the other channel remains blocked. The Ack Module completes handshaking on the input channel after the body flit is routed through a single output port.

The arrival of the tail flit leads to two operations in parallel: (i) the tail is detected by the Input Channel Monitor, which generates a *tail-detect* status signal, and (ii) the Ack Module switches back to the normally-transparent mode, where it now checks for routing of a flit through both outputs before generating an ack on the input channel. The *tail-detect* status signal is further used by the control unit of the blocked Output Port Module to re-enable its data register, switching it back to normally-transparent. Hence, *the node returns to speculative mode* and the tail flit is routed through both the output ports. After the tail's routing, the Ack Module completes handshaking on the input channel.

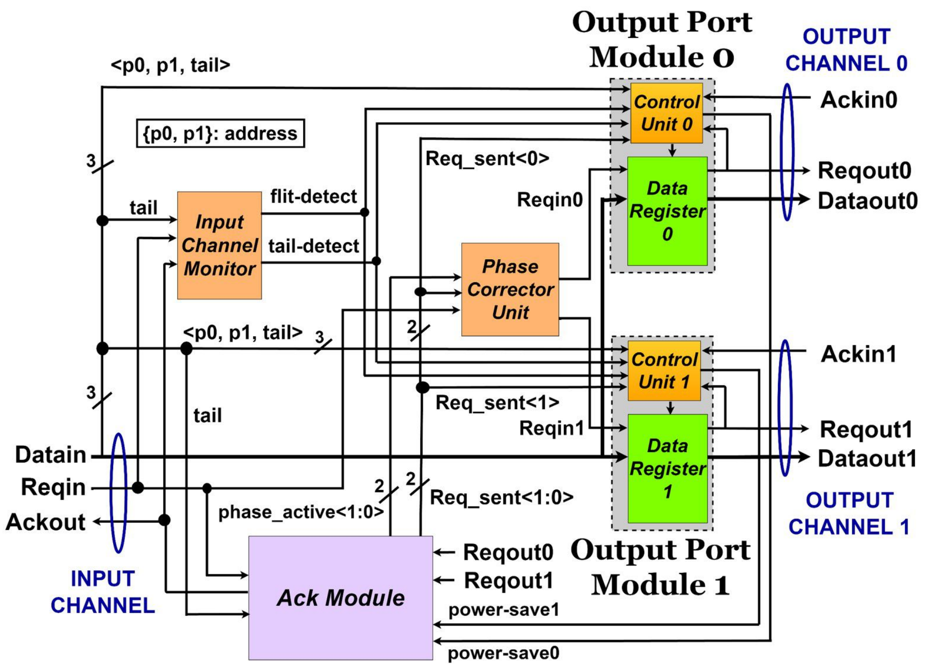

Figure 4.10: Optimized speculative fanout node

4.4.3.3 Design details of sub-modules

The designs details of three main components of the optimized speculative node are presented next: the control unit, the ack module, and the phase corrector.

Figure 4.11 shows the design details of the control unit for an Output Port Module. The request latch and the data register of the module are initially transparent, i.e. the output of the control unit is initially asserted. There are two main sub-units of this control unit: power_save/normal mode selector and the flow control unit. In the beginning, the mode selector is in the normal speculative mode with *power_save* de-asserted low. After the header flit is sent out (identified by *tail* and *Req_sent*, and this is not the correct path (determined using *p0, p1*), then the mode selector switches to non-speculative power-save mode, asserting *power_save* high, which disables the register in the Output Port Module for the body flits. After the tail flit arrives, the mode selector switches back to the speculative normal mode, and the register is back to normally transparent. The flow control unit simply checks if the output channel is available or congested.

As shown in the Figure 4.12, the Ack Module has a similar structure as the Ack Module of the unoptimized non-speculative node but now operates in three different modes: (i) normal speculative mode, where the ack is generated after a flit is routed through both output ports, (ii) power-save0 mode, where Output Port Module 0 is operating in power-save mode, and ack is generated after a body flit is routed through port 1, and (iii) power-save1 mode, where the Output Port Module 1 is operating in power-save mode, and the ack is generated after a body flit is sent out through port 0. A mode selector unit determines the mode of operation of the Ack Module using the *power_save* status signals from the control units: initially for the header, the Ack Module is in the normal speculative mode, after the header is routed, the module switches to one of the two power-save modes for the body flits, and stays in that mode until the tail flit arrives when it returns to the normal mode. In addition, these three modes are also used by the Phase Corrector Unit to perform the appropriate phase correction (using *Phase_active* status signals). For the normal speculative mode, the ack is generated using both the *Req0_sent* and *Req1_sent*, which are asserted high after requests are sent out on both output channels. For the other power-save modes, the ack is generated using the *Req_sent*, corresponding to the Output Port Module, which is not closed. Generation of the ack resets the *Flit_detect* in the Input Channel Monitor, which then resets the Ack Module.

The Figure 4.13 shows the design details of the Phase Corrector Unit, corresponding to Output Port Module 1. Phase correction may be required for an input request to an Output Port Module, if the flit is not intended for this output port. In particular, phase correction will be needed if the Output Port Module is in power-save mode and closed for the body flits. For each flit, initially, *Reqin* is toggled, which leads to a transition on *Reqin1*, which is input to the Output Port Module. But, if the output port is closed, then *Reqin1* is blocked from going out and its phase must be corrected. After the Phase Selector Unit has determined that the corresponding output port is in power-save mode using *Phase_active1* then *phase* is toggled so that *Reqin1* returns to its initial state.

4.4.4 Optimized Non-Speculative Fanout Node

The basic idea of optimization is to use the address of the header to pre-allocate the correct output channel(s) for remaining body and tail flits, thereby avoiding the overhead of flit-level allocation. These flits are then fast forwarded on their arrival, without route computation and output channel allocation, thereby improving network throughput as well as overall latency.

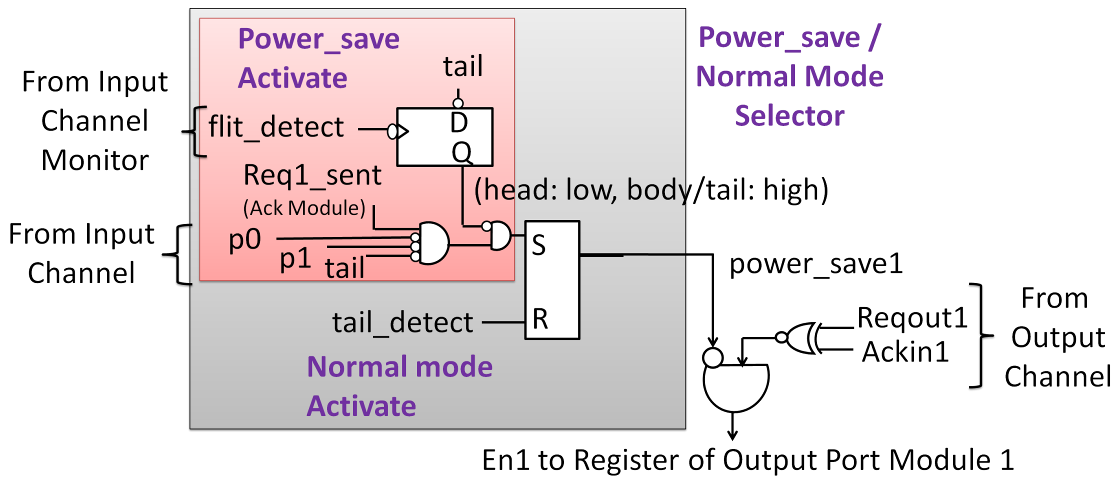

Figure 4.11: Optimized speculative fanout node: control unit details

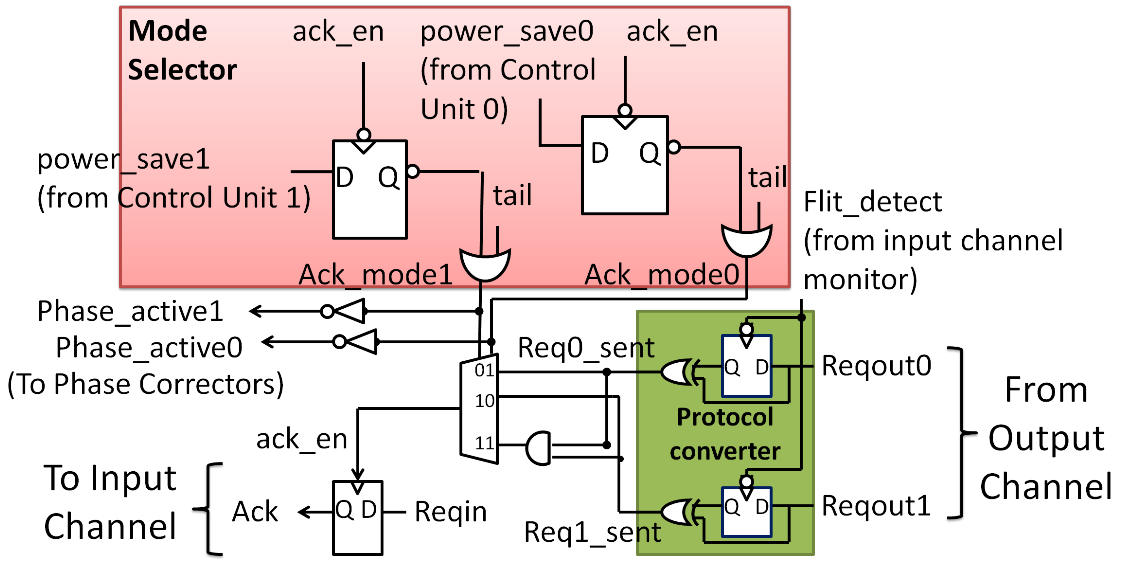

Figure 4.12: Optimized speculative fanout node: Ack Module details

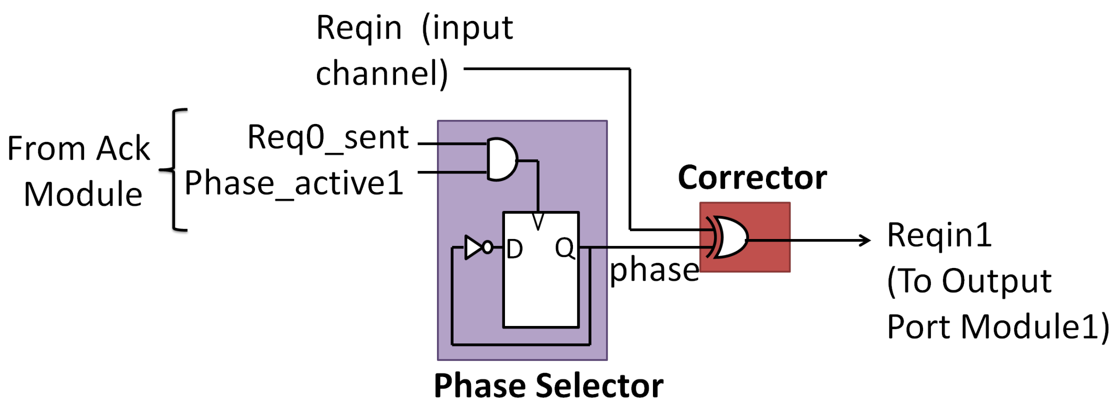

Figure 4.13: Optimized speculative fanout node: Input Channel Monitor and Phase Corrector details

4.4.4.1 Structure

As shown in the Figure 4.14, the high-level structure of the optimized non-speculative fanout node is almost similar to the unoptimized version, with two key differences: (i) new control units of Output Port Modules, which are able to pre-allocate the correct output channel(s), i.e. keep the correct output register(s) open after routing the header flit, and to also release the pre-allocated channel after the tail flit is routed. These control units are now also simplified as they do not generate the output requests, which are now generated as part of the datapath, similar to Mousetrap; and (ii) a new Phase Corrector Unit is added, which is used to generate the output request with the correct phase, corresponding to each output port. All the other units are the same as in the unoptimized node.

4.4.4.2 Operation

The optimization only affects the unicast and multicast packets that are correctly routed from the upstream fanout node. The operation for the incorrectly routed packets is the same as for the un-optimized nodes: these packets are simply throttled. Only correct packets are considered in the discussion below.

After the header flit of a correct packet arrives on the input channel, its address is stored in the Address Storage Unit, which is then used by the Input Channel Monitor to generate the *top-*

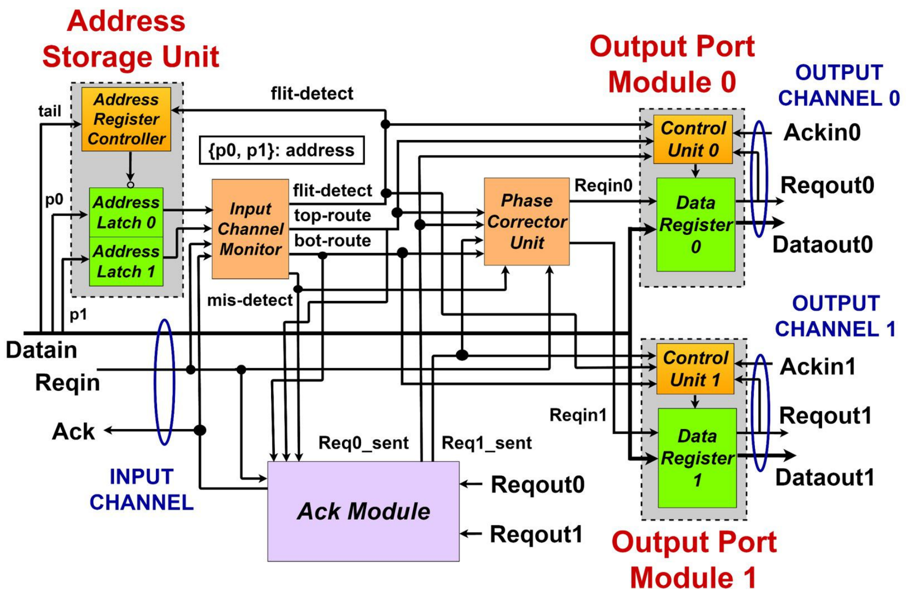

Figure 4.14: Optimized non-speculative fanout node

route and/or *bot-route* status signals. These signals are used by the control units of the Output Port Modules to enable the correct output register(s). Next, the header flit is sent out along with its bundling request. Sending of the output request(s) is used by the Ack Module to generate the *Req0/1_sent* status signals, which are then used by the control units of the correct Output Port Module(s) to trigger channel pre-allocation for the trailing body and tail flits: the output register(s) now switch to transparent mode for the body/tail flits. The Ack Module then completes handshaking on the input channel so the next body flits can arrive.

For each body flit, it is simply fast forwarded through the already pre-allocated output channel(s), followed by sending ack to the upstream node.

Finally, after the tail flit is fast forwarded through the pre-allocated output port(s), the corresponding control units release the output channel(s) by disabling the register(s), which return to their default opaque state.

4.4.4.3 Design details of sub-modules

The design details of the new control unit and the phase corrector are presented.

Figure 4.15 shows the design details of the new control unit of the Output Port Modules. The

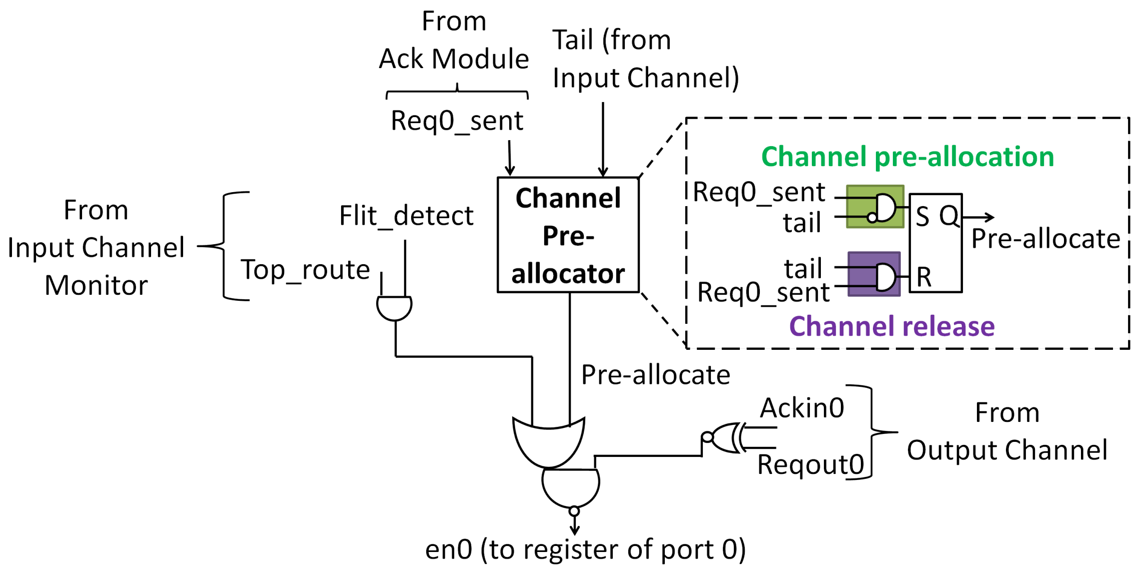

Figure 4.15: Optimized non-speculative fanout node: Control Unit details

data register and the request latch of the Output Port Module are initially normally-opaque as the outputs of the control units are initially de-asserted. The control unit consists of three sub-units: path selector, channel pre-allocator, and flow control unit. The path selector partially enables the Output Port Module based on the address decoding from Input Channel Monitor. The channel pre-allocator asserts its output (*pre-allocate*), if the header has been sent out of this output port, i.e. pre-allocates this output port for the trailing body/tail flits, and releases the output port after the tail has been routed. The flow control unit simply checks if there is congestion on the output channel or if the channel is available for routing. The output of the control unit is asserted for two cases: (i) for the header if the header is on the correct path (determined by the path selector) and the flow control unit sees the output channel as available, and (ii) for the body/tail flits, if the output channel was pre-allocated by the header and the channel is not congested as determined by the flow control unit.

Figure 4.16 shows the details of the new Phase Corrector Unit corresponding to the output port 0. Phase correction may be required for an input request to an Output Port Module if the flit is not intended for this output port. In particular, this correction is performed for two cases: (i) if the flit is sent out only through the other output port, or (ii) if the flit is misrouted from the upstream router. For each flit, initially, *Reqin* is toggled, which leads to a transition on *Reqin0*, which is input to the

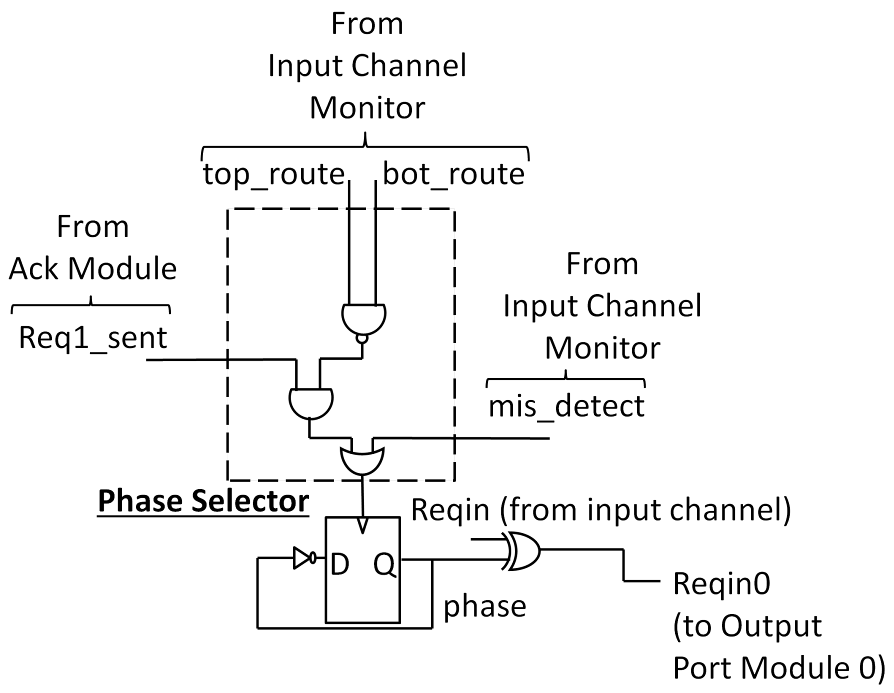

Figure 4.16: Optimized non-speculative fanout node: Phase Corrector details

Output Port Module. However, if the output port is not on the correct path, *Reqin0* is blocked from going out and its phase should be corrected. After the phase selector unit has determined if either of the above two cases applies, then *phase* is toggled so that *Reqin0* returns to its initial state.

4.5 Experimental Results

The experimental framework for evaluation of the new parallel multicast solutions, along with node and network-level results on area, performance, and power, are now presented.

4.5.1 Experimental Framework

Some interesting experimental case studies are presented to evaluate the effectiveness of the proposed new multicast approaches. In particular, the new tree-based multicast capability, local speculation approach, different protocol optimizations, and the novel network architectures. The setup and the benchmarks used are also presented before presenting the node- and network-level results in the next section.

Experimental case studies. Two distinct case studies are used to evaluate the proposed parallel multicast solutions: (a) *contribution trajectory*, and (b) *architectural design space exploration.* The contribution trajectory incrementally evaluates the effectiveness of each contribution, in order, against a serial baseline: use of parallel multicast; local speculation and a hybrid network; and protocol optimizations. Architectural design space exploration, on the other hand, only evaluates the effects of varying the degrees of speculation on the new parallel multicast networks. To isolate the focus, only optimized networks are targeted, thereby eliminating any interference from the optimization strategies.

In particular, the *contribution trajectory* compares 4 networks: (i) *Baseline* [78], only supporting serial multicast; (ii) *BasicNonSpeculative*, using simple tree-based parallel multicast; (iii) *BasicHybridSpeculative*, using local speculation in a hybrid network; and (iv) *OptHybridSpeculative*, similar to the previous one, but including protocol optimizations.

The *architecture design space exploration* compares 3 optimized new networks, each with varying degrees of speculation: (i) *OptNonSpeculative*, with no speculation; (ii) *OptHybridSpeculative*, with local speculation; and (iii) *OptAllSpeculative*, with almost full speculation.

Experimental setup. Six different 8x8 MoT networks are implemented using FreePDK Nangate 45 nm technology. Designs are technology-mapped and pre-layout. Six types of nodes are implemented, as building blocks: five fanout and one fanin. Nodes are mapped to the Nangate standard cell library in the Cadence Virtuoso tool. Accurate gate-level models are extracted using the Spectre simulator (typical process corner), to determine rise/fall times for every I/O path of each gate. Channel lengths and delays are borrowed from a synchronous MoT chip [78] and scaled to 45 nm technology. These extracted models of nodes and channels are used to implement the networks in structural Verilog.

An asynchronous NoC simulator is used for both unicast and multicast traffic. It includes a Programming Language Interface (PLI) to connect a C-based traffic generator and test environment to the technology-mapped network. This simulator extends a previous one that only supports unicast [67]. For multicast, the latter's traffic generator is modified to inject multicast packets with new source routing addressing, and also to include a new validation module that can now verify the routing of multicast packets to multiple destinations. A fixed packet size of 5 flits is used. Injection of headers of different packets follows an exponential distribution. A procedure similar to [41] is followed to ensure long warmup and measurement phases. Two steps are used to measure power: (i) record and annotate precise switching activity of every wire in the network over a benchmark run, and (ii) compute total power using the Synopsys PrimeTime tool.

Benchmarks. Experiments are conducted on six synthetic benchmarks. There are 3 unicast benchmarks [41]: 1) *Uniform random,* 2) *Bit permutation:shuffle,* and 3) *Hotspot.* There are 3 multicast benchmarks: 4) *Multicast5* and 5) *Multicast10,* where all sources inject multicast traffic at rates of 5% and 10%, respectively, to random subsets of destinations, and otherwise do uniform random unicast, and 6) *Multicast_static,* where 3 sources perform only random multicast, and the others do only uniform random unicast.

4.5.2 Node- and Network-Level Results

Node-level results. Area and latency of the four new fanout nodes (Section 4.4) and *Baseline* fanout are shown in Table 4.1.

In terms of area, the unoptimized speculative fanout node has the smallest area due to its very simple design. This node achieves 27.7% lower area than the *Baseline* node. Due to the added multicast capability in the unoptimized non-speculative fanout node, it incurs a moderate overhead of 18.7% over *Baseline.* The optimized speculative node includes extra logic to perform power-aware optimization, and therefore has significant (51%) overhead over the unoptimized version, but it still has almost the same area as the *Baseline.* Interestingly, the optimized non-speculative nodes not only improve performance but also have slightly lower area than the unoptimized ones.

In terms of latency, the unoptimized speculative fanout node has the lowest latency, which is only 52 ps, due to just the normally-transparent latch register on the forward critical path. Interestingly, even after adding the new replication capability and ability to decode multicast addressing in the unoptimized non-speculative node, it still has almost the same latency as unicast-only *Baseline* node. The optimized speculative node achieves 2$\times$ lower latency than the *Baseline* although incurs significant overhead over the unoptimized speculative node due to the added power-aware optimization. However, the optimization in the non-speculative node, interestingly, led to slightly better latency than both the *Baseline* node and the unoptimized non-speculative node.

Network-level results: contribution trajectory. This first experimental case study explores the incremental impact of each key contribution: parallel multicast, local speculation, and optimiza-

Metrics	***Baseline***	***Unoptimized Speculative***	***Unoptimized Non-speculative***	***Optimized Speculative***	***Optimized Non-speculative***
Node area	342 µm^2	247 µm^2	406 µm^2	373 µm^2	366 µm^2
Node latency	263 ps	52 ps	299 ps	120 ps	279 ps

Table 4.1: Node-level area and latency comparisons

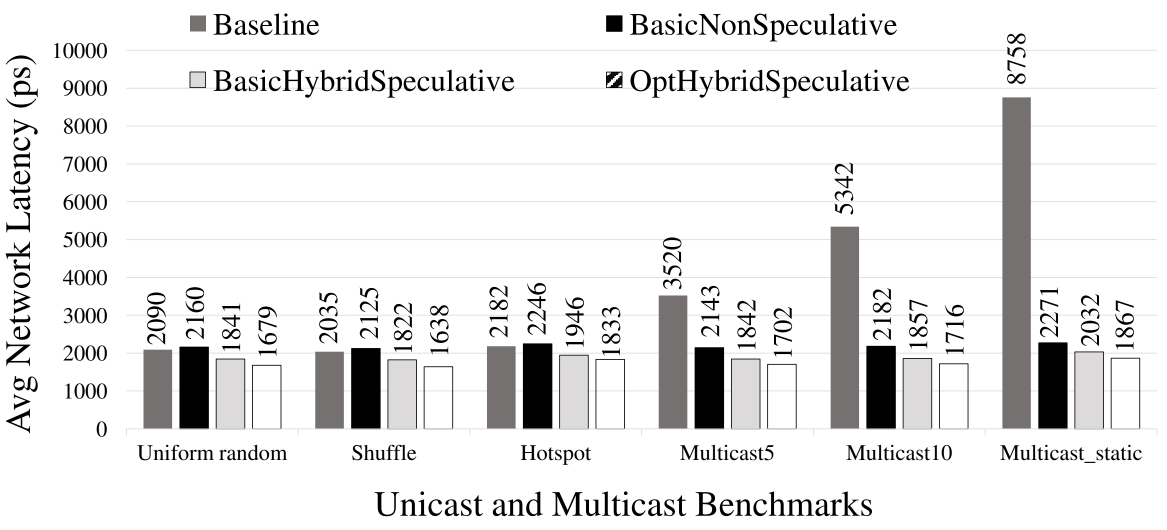

Figure 4.17: Contribution trajectory: network latency at 25% saturation load of respective networks

tions.

Network latency. Figure 4.17 shows the average network latency results. We measure latency of each network at 25% of the saturation throughput of that network, up to the arrival of all headers at destinations. This load is high enough to show the impact of different benchmarks, while keeping the network largely uncongested. Moreover, long warmup and measurement times are used, for example, for Uniform Random/Multicast_static benchmarks, warmup is 320 ns/640 ns, and measurement is 3200 ns/6400 ns with injection of 2100/4000 flits at each active source.

For multicast benchmarks, the simple tree-based parallel multicast network, *BasicNonSpeculative*, obtained significant benefits over the serial *Baseline*, from 39.1% (Multicast5) to 74.1% (Multicast_static), highlighting the severe overheads of the serial multicast approach. The *BasicHybrid-*

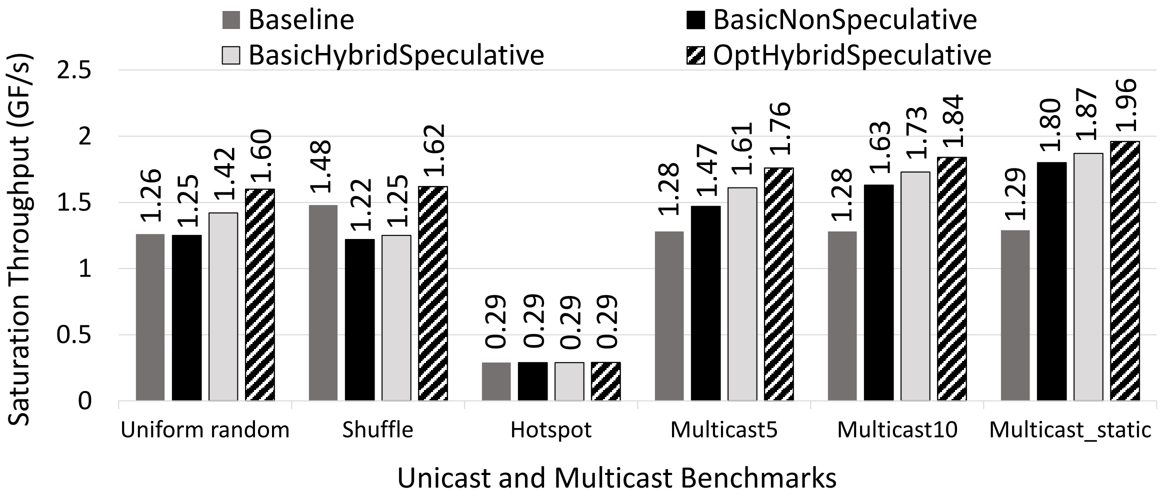

Figure 4.18: Contribution trajectory: saturated throughput

Speculative and *OptHybridSpeculative* show further improvements of 10.5-14.9% and 17.8-21.4%, respectively, over the *BasicNonSpeculative,* illustrating the individual benefits of hybrid design and optimizations.

For unicast benchmarks, *BasicNonSpeculative* incurs a small latency overhead over *Baseline:* since unicast is serial, the added node complexity to support parallel multicast becomes an overhead. However, the two hybrid networks provide noticeable benefits over *BasicNonSpeculative,* following similar trends as observed with multicast benchmarks. Interestingly, these latter results show that local speculation can significantly accelerate unicast traffic due to very fast speculative nodes.

Saturation throughput. Figure 4.18 shows saturation throughput results. For multicast benchmarks, the new simple parallel network, *BasicNonSpeculative*, shows considerable benefits over the serial *Baseline*, ranging from 14.8% (Multicast5) to 39.5% (Multicast_static). The two hybrid networks exhibit additional improvements up to 9.5% and 19.7%, respectively, over *BasicNon-Speculative,* demonstrating that local speculation, with accelerated packet transmission, provides a higher threshold for saturation.

For unicast benchmarks, results are more complex. Hotspot is highly-adversarial, with identical throughput for every network. For Uniform random, the *OptHybridSpeculative* network showed substantial improvements (28.0%) over *BasicNonSpeculative* due to the use of both fast and simple speculative nodes as well as the use of optimized non-speculative nodes with throughput-oriented protocol optimizations. For Shuffle, two new networks show moderate throughput degradation (*Ba-*

sicNonSpeculative, BasicHybridSpeculative) over the *Baseline*, while *OptHybridSpeculative* obtains 32.8% higher throughput than *BasicNonSpeculative* and 9.5% higher than *Baseline*.

The creation of redundant copies by the speculative nodes, in case of local speculation, may result in saturation throughput overheads, as shown for the Shuffle benchmark. The speculation nodes, although very simple, employ a conservative protocol to send ack upstream: this ack is only sent after a flit has been routed on both the output channels. While in the *Baseline,* the ack is sent after routing the flit through either of the outputs. The speculative node can become a performance bottleneck if the downstream path taken by the redundant copy gets blocked due to congestion. In this case, the new flit on the input channel is only routed through the 'correct' free output port, but not routed through the blocked channel, and therefore no ack can be sent upstream for this flit, and no new flits can arrive on the input channel. Hence, the redundant copy ripples the congestion upstream affecting the overall network throughput. However, as shown in Figure 4.18, such overheads can only occur for non-uniform traffic and at injection rates close to saturation.

Total network power. Figure 4.19 shows power results for 4 benchmarks. An injection rate that is 25% saturation load measured in *Baseline,* for a normalized comparison of energy per packet. Overall, as expected, *Baseline* has the lowest power due to its low complexity and serial multicast approach. *BasicNonSpeculative* has moderate overhead over *Baseline* (5.8-11.9%), due to more complex nodes. The overhead increases significantly for *BasicHybridSpeculative* (13.4-23.8% over *Baseline*), due to creation of redundant speculative copies. However, using *OptHybridSpeculative*, most of this overhead is removed (only 2.9-10.3% over *Baseline*): due to elimination of all redundant body flits (speculative nodes), and reduced switching activity because of channel pre-allocation (non-speculative nodes).

Network-level results: architectural design space exploration. This second experimental case study only includes evaluations of the optimized designs, while varying the degree of speculation.

Network latency. As shown in Figure 4.20, the hybrid network with local speculation (*OptHybridSpeculative*) achieves 9.7-11.9% latency improvements (unicast and multicast) over *OptNonSpeculative*, showing the effectiveness of the proposed techniques. The extreme case, *OptAllSpeculative*, exhibits 8.7-12.0% additional latency improvements over *OptHybridSpeculative* (18.5-21.7% over *OptNonSpeculative*), due to its almost fully speculative architecture (but will have significant

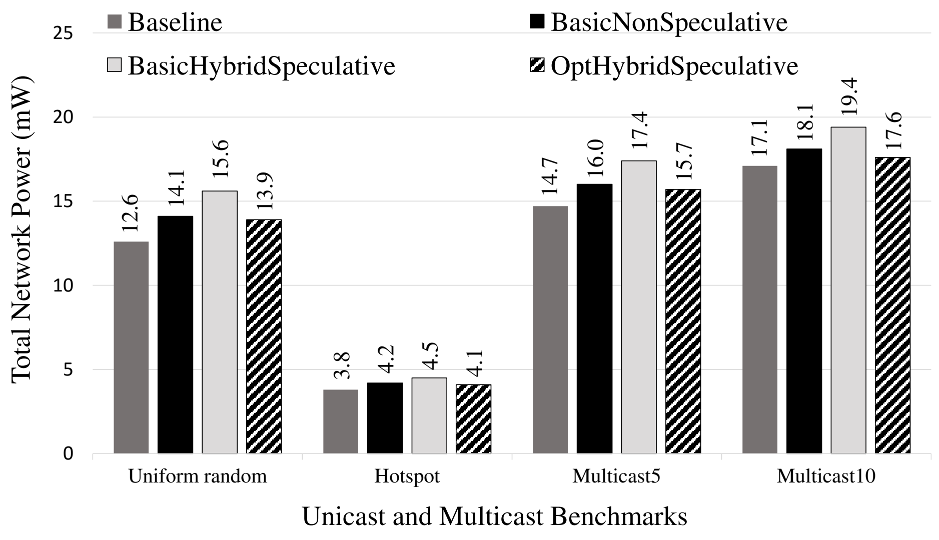

Figure 4.19: Contribution trajectory: total network power

power overheads).

Saturation throughput. For all benchmarks, in Figure 4.21, the hybrid approach (*OptHybrid-Speculative*) and extreme speculation (*OptAllSpeculative*) have nearly identical throughput to the non-speculative (*OptNonSpeculative*).

Total network power. Figure 4.22 shows the power results for the three optimized networks. Interestingly, even with its significant performance benefits, the optimized hybrid approach incurs only minor power overheads of 3.5-6.1% over the non-speculative approach, since redundant copies are restricted to small local regions, and a power-oriented optimization is applied to disable speculation for body flits. In contrast, the fully-speculative approach (*OptAllSpeculative*) incurs considerable power overheads (10.8-15.8% over *OptHybridSpeculative*, 14.7-22.9% over *OptNonSpeculative*) due to larger regions of speculation in *OptAllSpeculative*. It is expected that with larger MoT networks, these overheads will only increase, due to wider speculative regions.

Network-level results: addressing scheme comparisons. As highlighted earlier, an additional benefit of local speculation is to reduce the address field size. The serial Baseline has the shortest address field, using source routing, with a 1-bit address per fanout node on a unicast path: 3 bits for 8x8 MoT, and 4 bits for 16x16 MoT (not evaluated in this chapter). However, the large performance overheads of these designs make them impractical for multicast.

Of the three proposed parallel architectures, each using source routing, address field sizes for

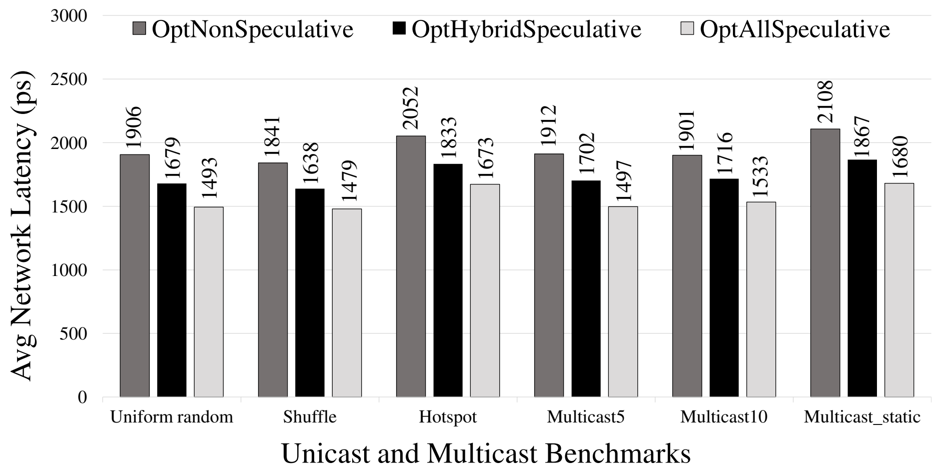

Figure 4.20: Architectural design space exploration: network latency at 25% saturation load of respective networks

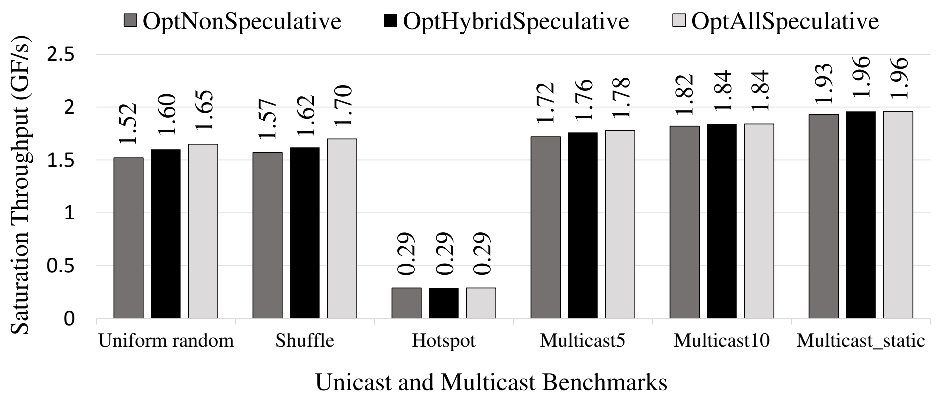

Figure 4.21: Architectural design space exploration: saturated throughput

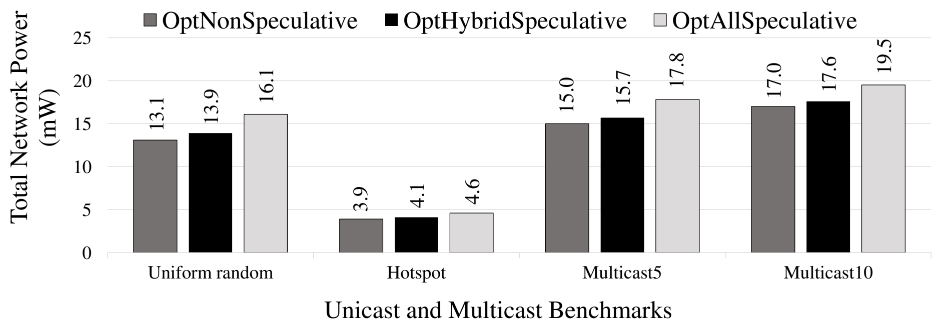

Figure 4.22: Architectural design space exploration: total network power

an 8x8 MoT are: 14 bits in *non-speculative*, 12 bits in *hybrid*, and 8 bits in *almost fully-speculative*. For a 16x16 MoT, the benefits of speculation are even greater: 30, 20 and 16 bits, respectively. Effectively, the speculative architectures reduce the total number of address fields, by only addressing non-speculative nodes.

Overall, the optimized network with local speculation is the best design point considering multiple cost objectives. This network shows significant latency and throughput improvements with almost the same power as the non-speculative baseline and tree-based solutions, not just for the multicast traffic, but also for unicast traffic. The fully-speculative network can achieve better performance than local speculation, but it is not a practical solution given its major power overheads.

4.6 Conclusions

The chapter presents a new lightweight multicast using Mesh-of-Trees based asynchronous NoCs. A new strategy, local speculation, is introduced, where fixed speculative switches always broadcast, but redundant packets are restricted to small regions. A hybrid network architecture is proposed, mixing speculative and non-speculative switches. The approach is the first general-purpose multicast for asynchronous NoCs. For multicast, the network achieves 17.8-21.4% improvements in network latency with small power reductions over a tree-based non-speculative approach. Interestingly, similar performance benefits are also observed for unicast traffic. Therefore, local speculation can also be used for applications involving only unicast, with fast and simple speculative switches, surrounded by non-speculative switches that will only support one-way transmission, stripping off any logic for multi-way routing, which can result in even higher performance. For future work, we plan to extend the approach to larger MoT networks, alternative topologies (e.g. 2D-mesh), as well as synchronous NoCs.

Chapter 5

A Continuous-Time Replication Strategy for Multicast in 2D-Mesh NoCs

5.1 Introduction

In contrast to the previous chapter, where the target topology was a variant mesh-of-trees (MoT), this chapter focuses on the more challenging problem of handling multicast in a 2D-mesh topology [17], [19]. 2D mesh is more commonly used, contains higher radix routers with greater parallelism, and employs more complex routing than the variant MoT. Therefore, it is critical to find a simple yet high-performance multicast solution for this topology.

Achieving high-performance multicast in 2D-mesh topology is a difficult problem. Switches in 2D mesh exhibit high levels of concurrency, where multiple input ports of a switch can receive packets in parallel, which need to be routed through different output ports. In addition, some of these packets are usually intended for the same output ports, leading to contention at these ports. Multicast transmissions can manifest more contention than unicast, which can impact system performance: a multicast packet at an input port needs to be routed through multiple output ports, and there can be multiple such packets routed in parallel. For high performance, there is a need for a routing strategy, that can route copies of a multicast packet in parallel through different output ports, and *eagerly,* where as soon as an output port becomes available, a packet copy is sent through this output even if other outputs are still blocked. Of course, such a strategy must also be implemented with low area and energy overheads.

To this end, a novel *continuous-time replication strategy* is introduced in this chapter to achieve high-performance parallel multicast in an asynchronous NoC with low overheads. In particular, the flits of a multicast packet are first stored in a single buffer at an input port, from where these flits are then routed through multiple distinct output ports of the router according to each output's own rate, concurrently, and in continuous time. Unlike synchronous, this unique asynchronous approach, not discretized to clock cycles, can provide significant performance benefits by handling subtle variations in network congestion and operating speeds.

To enable the above replication strategy with low area and energy costs, a new *continuous-time multi-way read (CMR) buffer* is also introduced. The CMR buffer is a low-latency asynchronous FIFO with a single write port and multiple decoupled read controls. The write port is used to store the flits of each individual multicast packet, which can then be accessed by multiple output ports of the router in parallel using the independent read pointers of this buffer. Only one CMR buffer is used per input port, leading to minimum power/area overheads, compared to multiple buffers in recent synchronous multicast routers [193], [81]. In addition, this buffer is a standalone unit, which can be useful for a wide variety of asynchronous applications.

The new continuous-time replication strategy along with the CMR buffer form a unique asynchronous paradigm that leads to low-overhead 'sub-cycle' forking of flits, which is critical for high-performance multicast. This continuous-time yet simple multi-way routing may not be possible using synchronous techniques, which rely on discretization.

Finally, for further latency optimization in the new switch, the expensive input buffering operation is eliminated from the forward critical path by performing it in parallel with route computation for the packet header. In contrast, asynchronous routers generally perform buffering of the header and route computation serially [185], [66], [89], resulting in a latency bottleneck. To ensure energy efficiency, this route computation unit is deactivated for the remaining flits of the packet.

5.2 Baseline Asynchronous NoC

The new multicast NoC builds on a previous highly-efficient 2D-mesh asynchronous NoC, which only supports unicast [66]. This NoC was developed as part of a joint collaboration between Asynchronous Circuits and Systems Lab at Columbia University and the University of Ferrara, Italy. This early work is significantly enhanced to support parallel multicast, and will also be used as a baseline for the new work.

In addition, this baseline NoC was also extended to include VCs in a collaborative work between our group at Columbia University and AMD Research [89]. This section first briefly presents the design of the original NoC without VCs, followed by discussion on the extended design with VCs.

5.2.1 The Baseline NoC Without VCs

This baseline NoC uses 5-port routers, which are designed to be simple, high-performance with low energy and area costs. Most of the existing asynchronous routers use 4-phase protocol with delay-insensitive encoding on channels, which can have significant overheads [52], [23], [185]: 4-phase can be expensive for throughput due to two roundtrip communications per transaction compared to one roundtrip in 2-phase. DI encoding, on the other hand, leads to significant increase in area and energy-per-flit. Therefore, the less common but efficient 2-phase communication protocol with single-rail bundled data is utilized to achieve a cost-effective router design.

Each router of this baseline NoC has two main components: input port modules (IPMs) and output port modules (OPMs), connected using a shared crossbar. Each IPM has one input channel and four output channels connected through a crossbar to the OPMs. The IPM performs route computation on an incoming packet and selects the correct OPM for routing. Each OPM has four input channels and a single output channel. The OPM arbitrates between packets from the four IPMs and the winner is forwarded on the output channel. Both the IPMs and the OPMs are based on Mousetrap pipelines: for high performance, *capture-pass registers*, i.e. normally-transparent single-latch registers, are used throughout the router, which provide very low latency as no synchronization to a latch enable is required to open these registers [147].

5.2.1.1 Input port module

The micro-architecture of the baseline IPM is shown in Figure 5.1. There are four components: *Circular FIFO Buffer*, *Route Computation Unit*, four *Request Generators*, and *Internal Ack Generator*. The Circular Buffer stores the incoming flits. This buffer was proposed as an auxiliary unit in [66] but was not directly used in its IPM. However, the baseline in this chapter uses this buffer. The Route Computation Unit uses XY routing to select the correct Request Generator based on the

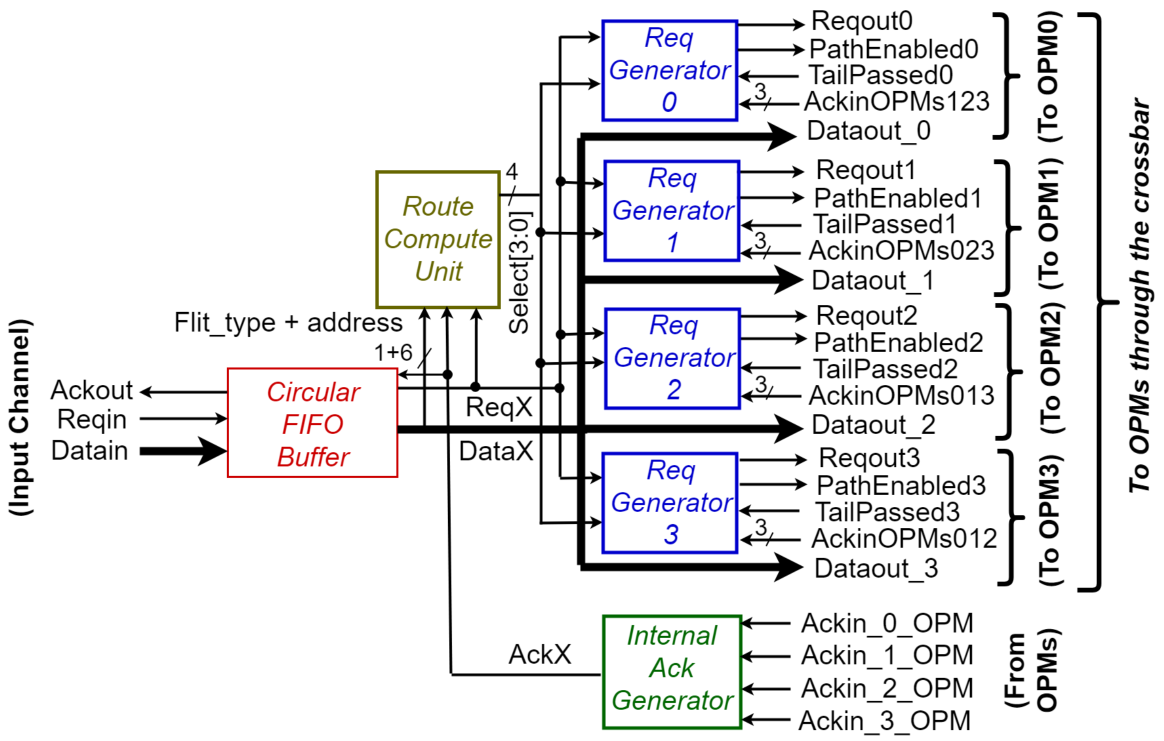

Figure 5.1: Baseline IPM micro-architecture

destination address in the packet header, which activates the correct OPM for the packet and generates an output request for each flit. The Internal Ack Generator observes completion of handshaking between the IPM and the OPM for each flit, and advances the read pointer in the Circular Buffer.

The baseline IPM has a simple operation. A header flit first arrives on the input channel and is immediately stored in the Circular Buffer. Next, three operations are performed in parallel: (a) the flit from the buffer is speculatively broadcast to all OPMs, (b) the bundled request is also broadcast to all the Request Generators, which then toggles the corresponding output requests, and (c) the Route Computation Unit receives the address field (X/Y coordinates of destination) and selects the one correct Request Generator. This Request Generator activates the correct OPM by asserting *PathEnabled*, which persists for the packet lifetime. Once the flit has been routed and an ack received from the OPM, the next flit is read from the buffer. Similar operations are performed for body and tail flits. Finally, the routing of the tail by the OPM deasserts *PathEnabled*, releasing the OPM's arbiter, and completing the packet's routing.

Circular FIFO structure and operation. The Circular FIFO is a low-latency, area- and energy-

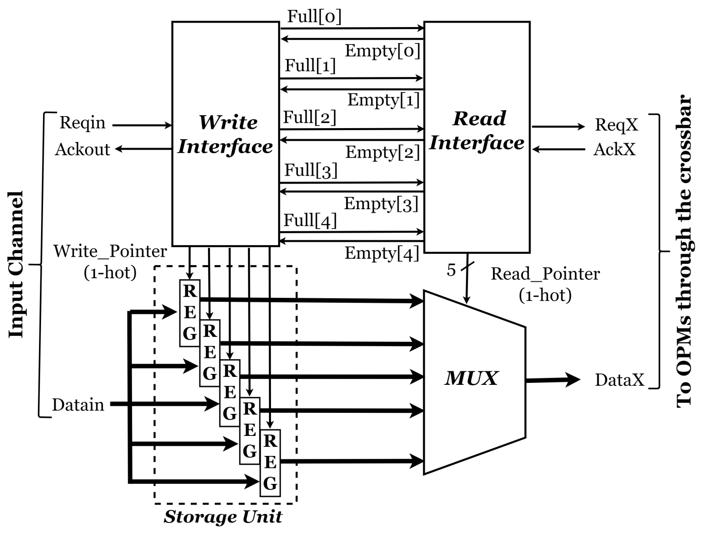

Figure 5.2: Baseline circular FIFO

efficient buffer. This buffer uses single latch-based registers, where each latch register can hold a distinct item. In contrast, synchronous designs typically use D flipflop-based registers with significant overheads, or pulse-mode single-latch designs with challenging two-sided timing constraints.[1] Interestingly, the use of capture-pass registers in this FIFO also improves forward latency. As shown in Figure 5.2 the buffer has one *write interface* and one *read interface*. After a flit is written in the tail cell, the *write interface* for the cell generates a *Full* request to the the *read interface*, followed by completing the handshaking on the input channel, and incrementing the write pointer. The *read interface*, after receiving the *Full* request, then reads the selected flit, which is forwarded to all the OPMs along with its req. After *AckX* is received from the OPM, the *read interface* advances its read pointer.

[1]A synchronous 2D circular FIFO has been proposed for low-voltage operation which, similar to us, uses single-latch registers; however, it can incur considerable overheads [39]. This FIFO comprises many buffer lanes, where each lane is a latch-based shift register. Due to the two dimensions, this FIFO can have significant area and power overheads. It is also expensive in terms of latency due to multiple latches in the shift registers. A 1D version of this FIFO, which is not shown in this chapter, will be a circular FIFO with each lane consisting of only one latch register. However, the operation of this 1D FIFO is still aligned to clock cycles and requires complex two-sided timing constraints for pulse-mode operation, unlike the proposed approach.

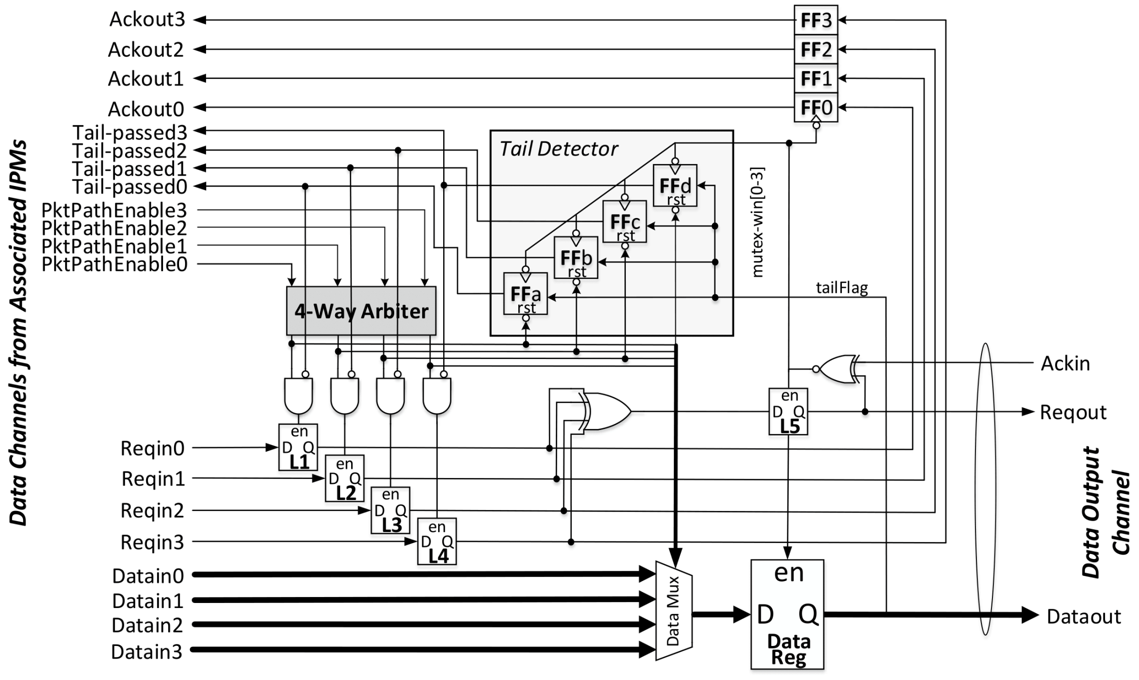

Figure 5.3: Baseline OPM micro-architecture

5.2.1.2 Output port module

The baseline OPM micro-architecture is shown in Figure 5.3. There are six main components: a 4-Way Arbiter, four on-demand (i.e. normally-closed) Input Registers corresponding to each IPM, a 4:1 Data Mux, a capture-pass (i.e. normally-transparent) Output Register, an Ack Generator, and a Tail Detector.

The low-latency arbiter mediates between packets from four distinct IPMs, and selects a winner. Since it must operate in continuous time (in contrast to synchronous arbiters), where inputs arrive at arbitrary temporal intervals not discretized to a clock cycle, it is built using an analog mutual-exclusion element, and is the only except to use of standard-cell components in the router (see [147], [66] and Section 2.3.2 for more details). Based on the arbitration, the correct input channel of the OPM is allocated i.e. enabling the selected input register. The data mux selects the data stream corresponding to the winning packet. The output register is used to perform flow control on the output channel to the downstream router. Two types of acks are sent to the winning IPM: one sent at a flit granularity by the ack generator after routing each flit on the output channel, and the other sent for the entire packet by the tail detector after routing the tail flit on the output channel.

In a basic scenario with no contention, a header flit of a packet, intended for the OPM, arrives from one of the IPMs through the crossbar. The flit is accompanied with its bundling request and address information in the form of *PktPathEnable,* which is input to the arbiter. In the absence of competition, the arbiter rapidly grants access to this header for the packet's lifetime. This acquisition is followed by two operations in parallel: (i) the correct L1-L4 input request register is allocated (i.e. held transparent) for entire processing of the packet, and (ii) the appropriate stream in the data mux is selected. Next, the header is sent out on the output channel through the capture-pass register. After the toggling of the output request, two operations occur in parallel: the normally-transparent output register is closed transiently for protection and flow control, and the ack generator sends the corresponding ack to the correct IPM.

Similar operations happen for body and tail flits. Since the correct path through the OPM is already allocated by the header, the new body/tail flits are simply fast forwarded through the OPM after their arrival. Once the tail flit arrives and is sent out, the tail detector asserts the appropriate *Tail-passed*, which then shuts the winning input register for safety. Finally, following a 4-phase handshaking with the IPM, assertion on the *Tail-passed* de-asserts *PktPathEnable* (in IPM), which then releases the arbiter in the OPM, leading to a complete resetting of the OPM's input registers, data mux, and the tail detector in parallel.

In summary, the entire path through the OPM remains flow-through and transparent, after the header wins the arbitration for all the body and tail flits. The only exception, is transient closing and opening of the output data register for flow control. If traffic is light or moderate, each body/tail flit effectively passes directly through the OPM.

5.2.1.3 Results

The baseline NoC router has been shown to handle unicast very efficiently. In a head-to-head post-layout comparison with a state-of-the-art synchronous router, this baseline asynchronous router significantly outperformed the latter, achieving 71% lower area, 44% lower energy-per-flit with 39% reduction in latency and comparable throughput [66]. Moreover, an exhaustive analysis was performed to measure the impact of increasing link lengths on latency: the asynchronous design showed significantly better latency than the synchronous one as link lengths were increased. In terms of power analysis, the asynchronous router achieved 90% lower idle power, and 73% and

45% dynamic power savings in case of hotspot and uniform traffic patterns, respectively.

5.2.1.4 Baseline for the new multicast research

The baseline network does not directly support parallel multicast, but can support serial multicast using a unicast-based approach. In this approach, for each multicast packet, the source NI creates multiple unicast copies destined for different destinations, which are then injected and routed serially. However, as shown in Section 5.5, performing multicast using the serial unicast approach can have severe performance and power overheads.

In this chapter, the baseline network will be significantly enhanced to support tree-based parallel multicast. Only the IPM module of the baseline is modified: since it performs route computation and selects the required output ports, it must be extended to support replication capability, which is essential for parallel multicast. In contrast, the OPM module remains the same as in the baseline as it still arbitrates between multiple packets and does not make any routing decisions.

5.2.2 Industrial Extension of the Baseline NoC to VCs

Recently, the above router was fully migrated to a leading commercial technology in an industrial setting [89]. This work was performed in collaboration between our group and AMD research. A head-to-head comparison was performed between the asynchronous router and AMD's leading synchronous router, where the asynchronous one outperformed the synchronous design. The asynchronous router also supports VCs in a high-performance and low-overhead manner.

This work made important significant advances to the field of asynchronous NoCs. First, it was the first published comparison between a high-performance asynchronous NoC router vs. an industrial synchronous NoC router using an advanced 14 nm technology. This synchronous router is used in high-end AMD processors and graphic products to handle a variety of configuration and control traffic. The comparison results are therefore more close to reality than the most other research works in this area. Second, a new end-to-end credit-based VC flow control was proposed, which leads to a lower number of backward credit synchronizations needed to the upstream router. Finally, the asynchronous router with 2 VCs significantly outperformed the synchronous AMD router, showing 55% lower area and 28% reduction in latency in a head-to-head comparison.

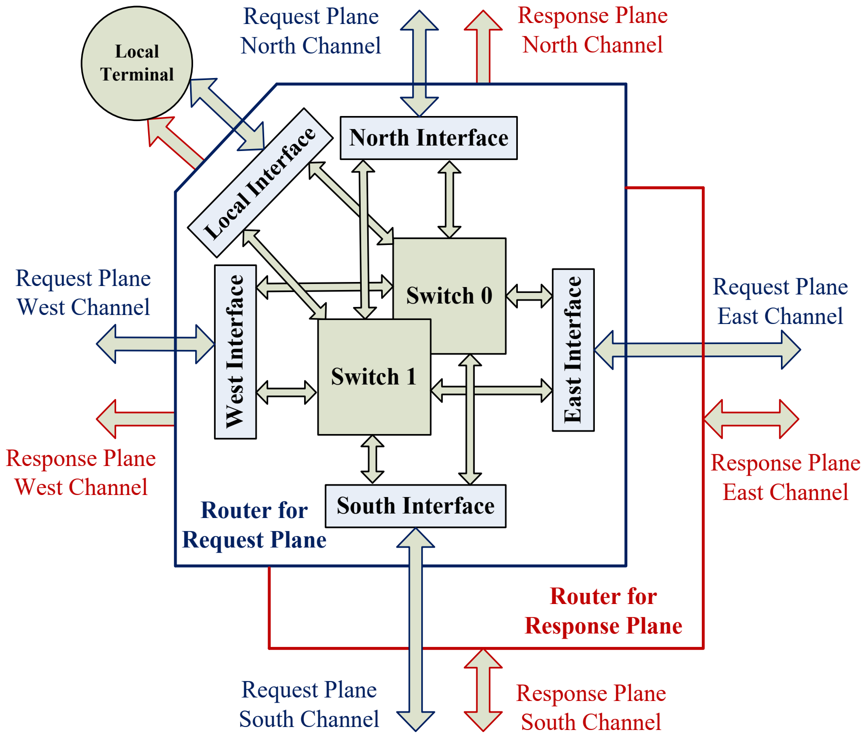
Local Terminal
Request Plane North Channel
Response Plane North Channel
North Interface
Local Interface
Switch 0
Switch 1
Request Plane West Channel
West Interface
East Interface
Request Plane East Channel
Response Plane West Channel
Response Plane East Channel
South Interface
Router for Request Plane
Router for Response Plane
Request Plane South Channel
Response Plane South Channel

Figure 5.4: Node structure for double-plane baseline asynchronous router

5.2.2.1 Overall router structure

Similar to the synchronous AMD router, the baseline router with VCs was designed for a double-plane NoC. The router for each plane contains uncorrelated and identical sub-routers, as shown in Figure 5.4. The two planes are: a *request plane* that routes read and write request packets, and a *response plane* that routes read and write response packets. Such request/response traffic is common for cache coherence protocols.

The router uses a multi-switch architecture to support VCs, which has been shown to be more effective than a shared-crossbar architecture for the asynchronous routers [136], although the latter is more common for synchronous routers. As shown in the Figure 5.4, in a multi-switch architecture, the number of switches are replicated as many times as the number of VCs, for each plane.

For each plane, in addition to the replicated switches, which are the same as in Section 5.2, interfaces are added for each of the five directions. These interfaces, on the input side, demux the arriving packet and route them to the switch according to the pre-assigned VC, and on the

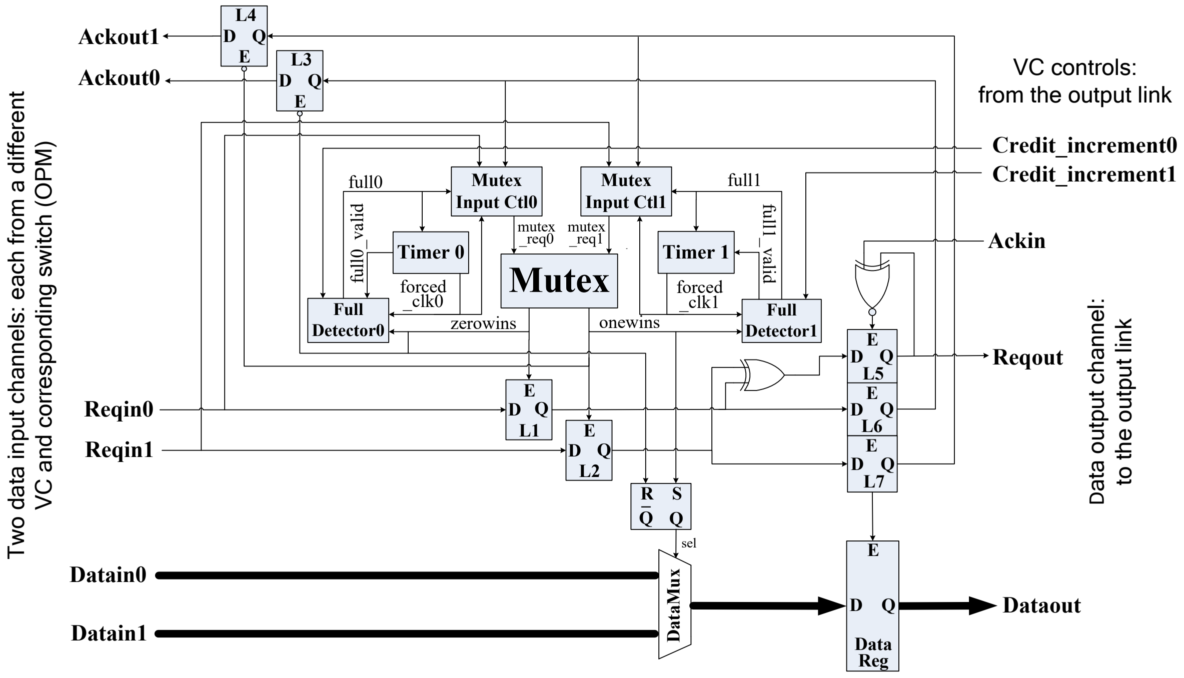

Figure 5.5: VC control for an output channel interface

output side, perform a merge operation of the traffic from all VCs on to the shared output channel. Therefore, the VCs separate the different traffic classes inside the router, but these classes are mixed on the inter-router channel.

5.2.2.2 VC flow control

A credit-based VC flow control is used, which forms the output interface for each direction. In this approach, the credit count is decreased when the flit is sent out, and increased when a free slot becomes available in the upstream router's input buffer. A 'lazy' approach is used for high-performance operation, where the non-critical credit increment requests are queued, and are updated only when performing the next and more critical credit decrement. This approach can lead to potentially better throughput than the other approaches, where the two credit updates are considered of the same priority, and the non-critical credit increment can block the critical credit decrement operation and sending out of a flit [136].

Figure 5.5 shows the VC flow control unit of the output interface. This module performs flit-level arbitration between the two data streams from the two VCs, and merges them on to the single output channel. There are two main components of this unit: the *full detector* and the *timer*. The full detector updates the credits every time a flit is sent out, considering both the queued credit-increment requests and the current credit decrement. This component also blocks further arbitration requests for a VC in the absence of credit availability. The timer is activated when credit is not available, and it periodically checks if any credits are released so that the blocking of arbitration can be removed.

5.2.2.3 Results

Experimental setup and some important results are summarized next.

The asynchronous router and the synchronous AMD router were compared at the pre-layout level. The synchronous router is a 3-cycle router with fine-grain clock gating, running at 1 GHz clock rate, based on the performance requirements of several high-end AMD products. The synchronous router was synthesized using the standard automated flow, while the asynchronous router was synthesized manually to achieve both high performance as well as robust operation. This router was also laid out using the standard P&R tools, however, the results presented here are only at pre-layout level, but similar trends are also expected at the post-layout level. Both the routers are configured to have 2 VCs, each with a 7-slot buffer. The technology used is low-power industrial 14 nm.

The results in terms of the area, latency, idle and active power are shown in Figure 5.6. The asynchronous router significantly outperforms the synchronous one: 55% lower area, 28% better latency, with 88% and 58% lower idle and active power.

Finally, Figure 5.7 shows the projected results for two alternative designs: (a) a 7-port router with 2VCs, which is important for 3D stacking and (b) a 5-port router with 8 VCs, which is a more realistic VC configuration. For both these cases as well, the asynchronous router dominates the synchronous router considerably. For the 7-port routers with 2 VCs, the asynchronous design showed 75% lower area and 25% lower latency. In terms of power, the synchronous design had 85% higher idle power and 50% higher active power than the asynchronous design. Similarly, for the 5-port routers with 8 VCs, the asynchronous one showed 38.4% lower area with almost the same latency. In terms of power, the synchronous design had 84.6% higher idle power and 46.1% higher active power than the asynchronous design.

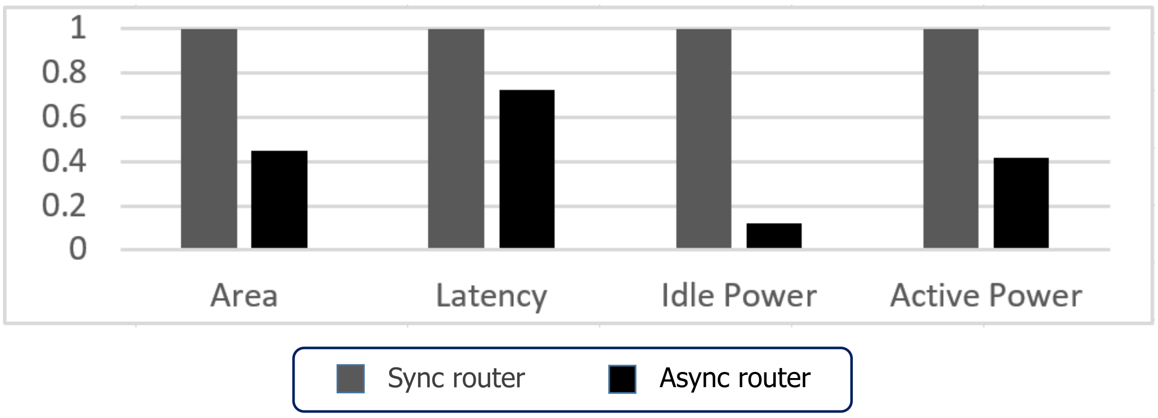

Figure 5.6: Baseline asynchronous vs. synchronous router: basic comparison for 2 VCs

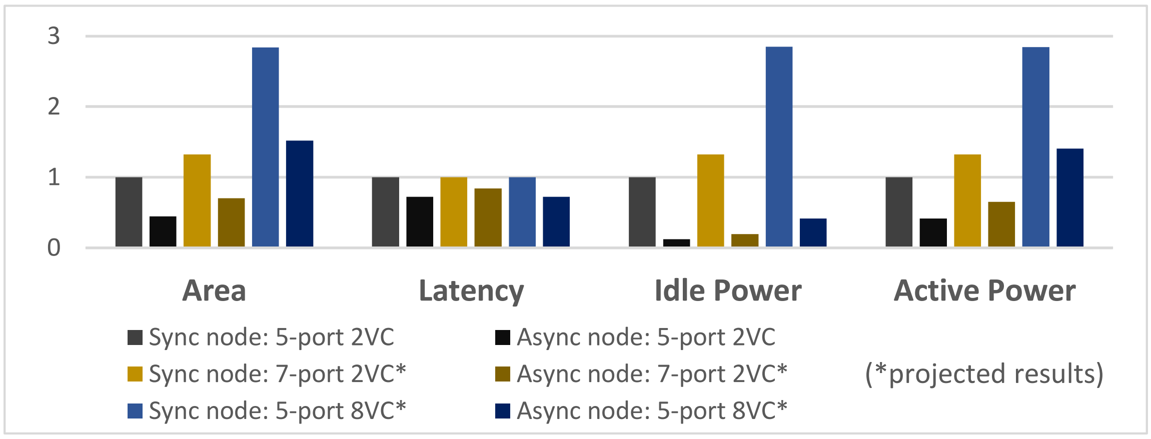

Figure 5.7: Asynchronous vs. synchronous router: projected results

5.3 New Multicast Approach

The new multicast asynchronous NoC is now introduced for 2D-mesh topology, that loosely builds on the previous unicast solution. The new approach enhances the widely-used tree multicast method using a novel replication strategy and router architecture. This section presents an overview of this approach, including its protocol and ability to avoid deadlocks.

5.3.1 Tree-Based Parallel Multicast

Figure 5.8 shows a simple view of the new router micro-architecture and illustrates how it supports tree-based multicast. The figure only shows one IPM in detail with new enhancements for parallel

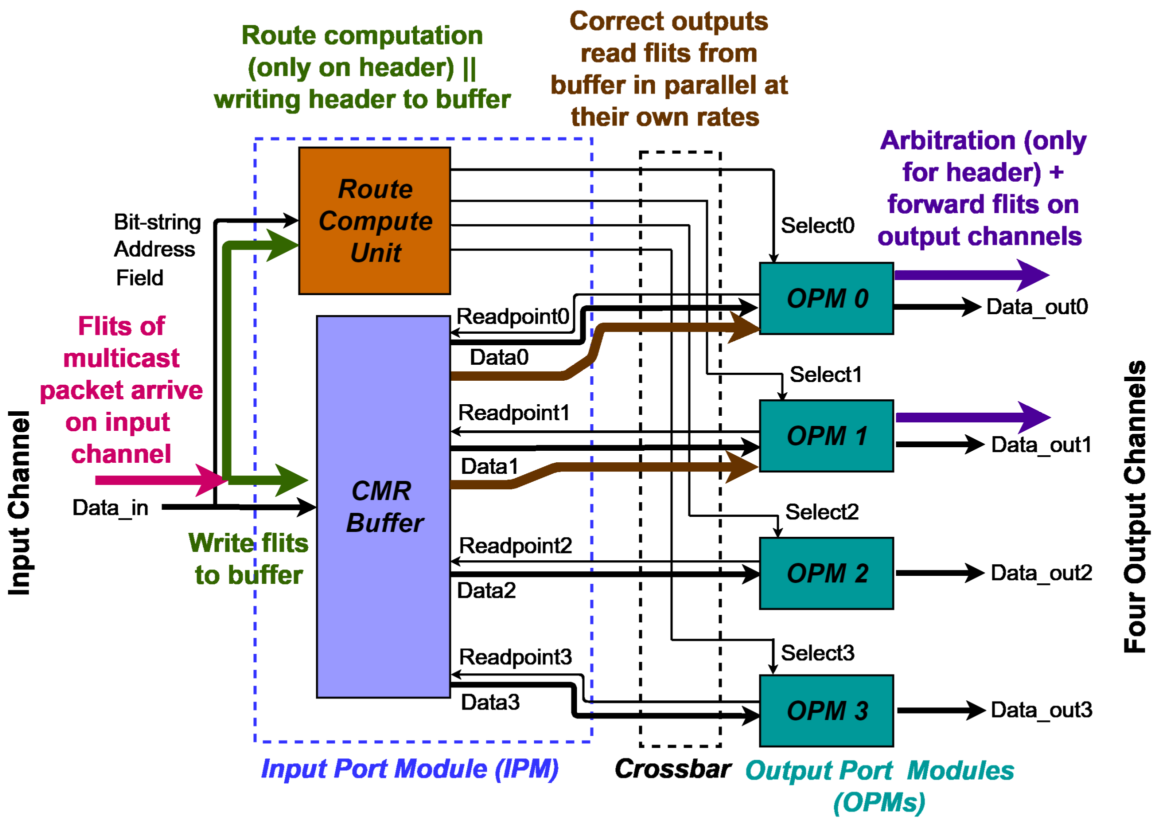

Figure 5.8: New router architecture and tree-based multicast operation

multicast, connected to all four OPMs using a partial crossbar. The IPM consists of a single CMR buffer, which connects both to the input channel and to four OPMs through the crossbar; and a parallel Route Compute Unit that also connects to the input channel and identifies the correct OPMs for routing. As shown in the given example, the flits of a multicast packet are replicated and forwarded to two output channels in parallel based on the route computation. These new copies will again follow a tree-based path through the network, performing multiple replications until they reach the destinations.

5.3.2 Continuous-Time Replication Strategy

The new replication strategy routes the set of flits in a multicast packet through the distinct output ports in parallel, and at each output's own rate. Unlike synchronous schemes, this replication and forwarding is performed in continuous time, accommodating the individual read rates of the different OPMs, and not discretized to clock cycles.

In more detail, the flits of a multicast packet stored in sequence in a single shared input buffer are read by multiple OPMs. The OPMs read these flits independently and at their own rates using decoupled read pointers, not bound to any discrete clock cycle. If an output port gets congested, others can continue reading the flits of the stored packet. Due to the continuous-time read operation, the congested output can resume reading the next flits as soon as it becomes available rather than waiting for the next clock cycle. Hence, unlike synchronous, this design can handle subtle variations in network congestion. Each OPM stalls once it has read all the currently stored flits. A stored packet becomes stale after all the required outputs have read all its flits, including the tail.

Figure 5.8 shows the top-level view of the continuous-time multi-way read (CMR) buffer used to implement the new replication strategy. The buffer has a single write port to store new incoming flits, and four read ports, which can be accessed by multiple OPMs in parallel using decoupled read pointers. The CMR buffer is an extension of the baseline single read-ported buffer [66], but with the new multiple read capability and distributed read control.

5.3.3 Route Computation and Buffering Policy

To remove the overhead of buffering a new header flit, route computation on the header is performed in parallel, using a separate small dedicated Route Compute Unit. When a header flit arrives, it is rapidly forked on two paths: to Route Compute and to the CMR Buffer. The former stores the header only, in a one-place buffer, and holds it for the packet lifetime. The latter is written into the circular FIFO, as part of the datapath, where it can be read by the different OPM units. *Bit string addressing* is used, which is a common efficient multicast encoding [88], [193], [105]. The address field has a single bit for each node in the network. The bit is 1 if the node is a destination, otherwise it is 0. Route computation is performed only on the packet header; after address decode, for energy efficiency, the unit is disabled. XY routing is used, to forward the multicast flit on the appropriate output ports, which ensures a deadlock-free protocol.

The IPM's buffering policy is specifically designed for the continuous-time replication strategy. A new packet can only be buffered when the entire previously-stored packet has been completely routed. The advantage of this *packet-based buffering policy* is to allow each OPM to freely read stored flits, without complex checks and any safety violations. In particular, the Route Compute

Unit only opens after a complete packet has been processed. If multiple flits of the next packet have already arrived, the unit will not sample its header address. A special synchronization is enforced, to ensure all OPM's have read the tail flit before sending the final packet-based upstream acknowledgment. The buffer size is therefore set equal to the worst-case packet size handled by the NoC. While the buffer can be under-utilized for shorter packets, an extra regular circular FIFO can also be added on the input channel to increase capacity. This approach also simplifies the flow control, since the buffer will never risk overflow.

5.3.4 Simulation of Multicast Routing

The basic operation of the new router is now presented, highlighting the interesting aspects of the continuous-time replication strategy, route computation, and packet-based buffering. Without loss of generality, a simple multicast scenario is considered: a 3-flit multicast packet is routed through two output channels of the new router.

As shown in Figure 5.9(a), first, the header flit of the packet arrives on the input channel. In step 2, two operations occur concurrently: the complete header flit is stored inside the CMR buffer, and the addressing of the header gets stored in a small buffer inside the route computation unit, where it is used for rapid address decoding. In step 3, an ack is sent upstream so the body flit can arrive. This ack also *disables the route computation unit* for the rest of the packet i.e. holds the head flit and blocks the remaining flits, improving energy efficiency. Concurrent to the ack, the two selected OPMs (0 and 1) start reading the header flit from the buffer using their individual read pointers in continuous time, handling any sub-cycle timing differential between the two read operations. The header is finally sent out in parallel through the two OPMs at their own rates.

As shown in the Figure 5.9(b), next, the body flit arrives on the input channel. In step 2, different from the header, the body flit is only stored in the CMR buffer, while the route computation unit remains deactivated. In step 3, two operations happen concurrently: an ack is sent upstream so the final tail flit can arrive, and the correct OPMs can now read the stored body flit. To demonstrate the independent read operation feature of the continuous-time replication strategy, assume that OPM 1 is congested and gets stalled, while OPM 0 is still available. As the OPMs can read the CMR buffer independently using individual read pointers, OPM 0 reads the body flit and sends it out on its output channel, while OPM 1 remains stalled.

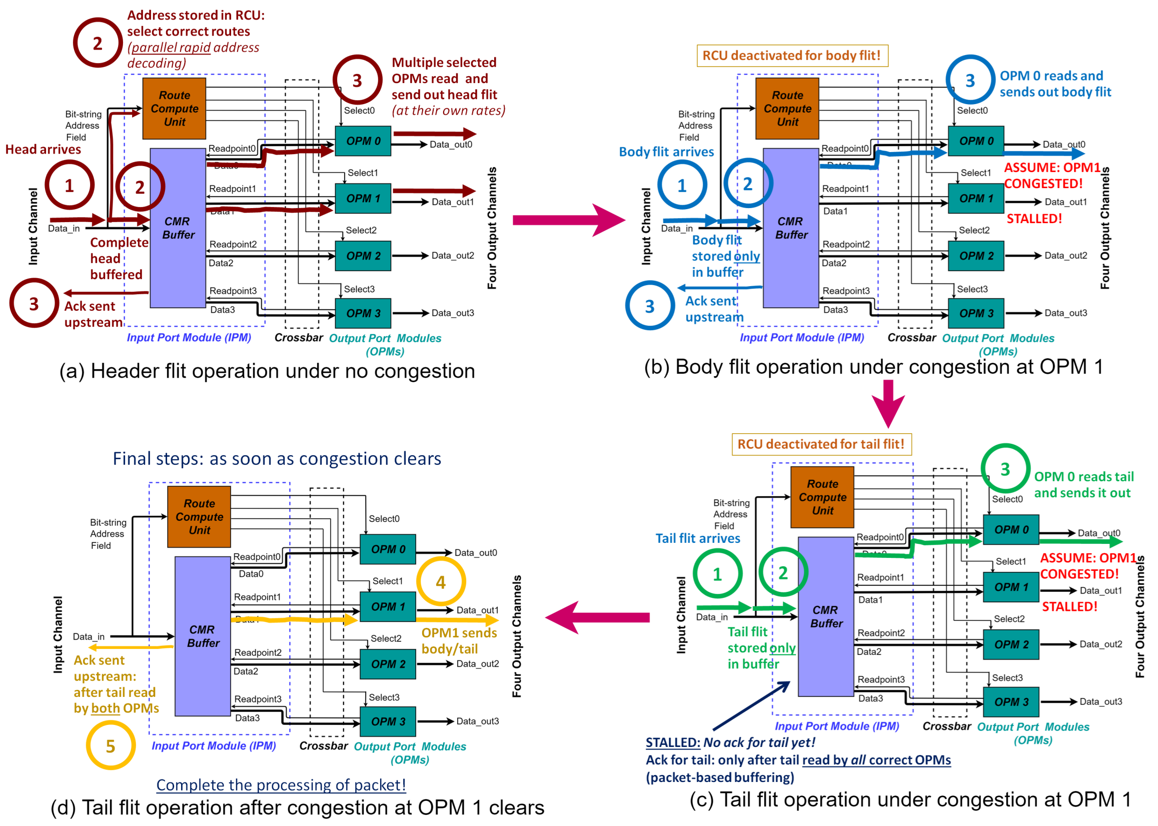
Address stored in RCU: select correct routes (parallel rapid address decoding)
Multiple selected OPMs read and send out head flit (at their own rates)
Head arrives
Complete head buffered
Ack sent upstream
Route Compute Unit
CMR Buffer
Bit-string Address Field
Data_in
Input Channel
Readpoint0
Data0
Readpoint1
Data1
Readpoint2
Data2
Readpoint3
Data3
Select0
Select1
Select2
Select3
OPM 0
OPM 1
OPM 2
OPM 3
Data_out0
Data_out1
Data_out2
Data_out3
Four Output Channels
Input Port Module (IPM)
Crossbar
Output Port Modules (OPMs)
(a) Header flit operation under no congestion
RCU deactivated for body flit!
OPM 0 reads and sends out body flit
Body flit arrives
Body flit stored only in buffer
ASSUME: OPM1 CONGESTED!
STALLED!
(b) Body flit operation under congestion at OPM 1
RCU deactivated for tail flit!
OPM 0 reads tail and sends it out
Tail flit arrives
Tail flit stored only in buffer
STALLED: No ack for tail yet!
Ack for tail: only after tail read by all correct OPMs (packet-based buffering)
(c) Tail flit operation under congestion at OPM 1
Final steps: as soon as congestion clears
OPM1 sends body/tail
Ack sent upstream: after tail read by both OPMs
Complete the processing of packet!
(d) Tail flit operation after congestion at OPM 1 clears

Figure 5.9: Continuous-time replication strategy operation for a 3-flit packet: highlighting interesting aspects of parallel route computation, independent read operations, and packet-based buffering

In Figure 5.9(c), the final tail flit arrives on the input channel and gets stored only in the CMR buffer (steps 1 and 2), while the route computation unit remains deactivated. Interestingly, no ack is sent upstream yet after storing of the tail flit as a different ack protocol is followed for the tail to enable packet-based buffering (Section 5.3.3): *ack is sent for tail only after it has been read by all the required OPMs.* This conservative protocol makes the final ack as the acknowledgment of the completion of routing of a multicast packet through all the required outputs, enabling safe processing of the packet without interference from another packet. In step 3, showing the independent read operation, the available OPM 0 reads the tail flit from the CMR buffer using its read pointer, while OPM 1 is still stalled. Since, the tail flit has not been read completely yet, no ack is sent upstream.

Figure 5.9(d) shows the final steps in the tail flit's processing, after OPM 1 becomes available.

In step 4, as soon as the congestion clears at OPM 1, without waiting for a clock cycle, OPM 1 reads the stored body and tail flits, using its read pointer, and sends them out on its output channel. Since the tail flit has now been read by both outputs, the final ack is sent upstream in step 5, which completes the processing of the current packet, and also re-enables the route computation unit for the next packet header.

5.3.5 Resource-Dependent Deadlock Avoidance

The new replication strategy not only has a simple operation but also entirely avoids resource-dependent deadlocks within a router. This section presents the deadlock scenario that can occur for multicast transmission, and how the new strategy avoids this issue. For further exploration, two alternative approaches are also presented to avoid multicast deadlocks, but are shown to be infeasible due to major limitations, which, however, are not seen in the proposed strategy.

Deadlocks using multicast protocols. A potential deadlock may occur due to a cyclic dependency between multi-flit multicast packets arriving at different inputs. This dependency arises when these multicast packets arrive almost simultaneously and are intended for the same output ports but acquire different subsets of these outputs. A deadlock can occur if a conservative multicast routing protocol is followed, where no body flits can be routed until the header has been routed through all required outputs.[1] The new replication strategy eliminates this deadlock, since it allows a packet to be routed through any OPM, and the OPM is released, regardless of the status of other OPMs.

Figure 5.10 shows a simple example of a resource-dependent deadlock in a conservative multicast routing protocol. Without loss of generality, a four-port router is considered. The router receives two multi-flit multicast packets: *A* and *B* almost simultaneously on *east* and *west* IPMs. Both these packets are intended for *north* and *south* OPMs. Assuming packet *A* header occupies the *north* OPM, while the packet *B* header occupies the *south* OPM. A circular dependency has occurred: packet *A* requires access to *south* for its complete routing, but is occupied by *B*, similarly, *B* requires access to *north,* which is occupied by *A*. Since, a conservative protocol is followed and none of the headers have been completely routed, the remaining flits are stuck and a deadlock has occurred.

[1]Deadlock will even occur in a more relaxed protocol, where body and tail flits can be sent freely on each OPM, but no OPM can be released until full multicast is complete.

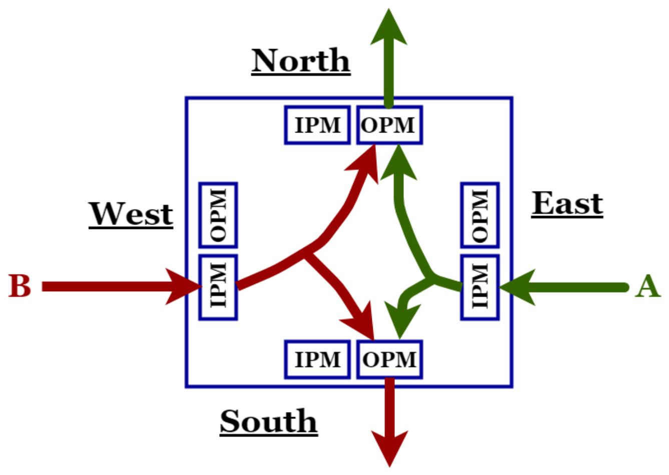

Figure 5.10: Cyclic dependency between multicast packets

Deadlock avoidance in proposed approach. The new replication strategy breaks this cycle. The packets can be completely routed through each acquired OPM, releasing these outputs for the other packet to acquire. In the scenario of Figure 5.10, packet *A* will be completely routed through *north,* releasing this OPM for *B* to acquire. Similarly, *B* will release *south* for *A*. Hence, both packets are completely routed through the required outputs.

Alternative approaches for deadlock avoidance. There can be two other ways to avoid resource-dependent deadlocks that can occur for multi-flit multicast packets: *a centralized arbitration approach,* and a *flit-level distributed arbitration approach* which was used in [164]. However, both these approaches suffer from major performance and correctness issues and are therefore not viable solutions.

Centralized arbitration approach. As shown in the Figure 5.11, in this scheme, multiple packets, arriving at different input ports, compete for the complete control of the router using a centralized arbiter. The winning packet is then routed through its required output ports, which do not perform any further arbitration. After the packet has been completely routed through all the necessary outputs, the central arbiter is released so that one of the other packets can now acquire the router.

Since, only a single packet can be routed through the router at a time, resource-dependent deadlock is avoided in this approach as there can be no dependency between multiple packets. For example, in Figure 5.11, the headers of the competing packets *A* and *B* arrive almost simultaneously at the input ports *east* and *west*, respectively, both intended for *north* and *south*, and compete with

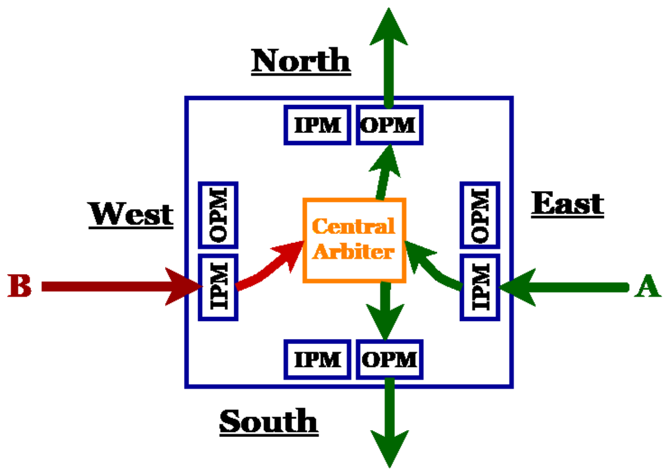

Figure 5.11: Centralized arbitration: an alternative approach to avoid resource-dependent deadlocks for multicast

each other to gain control of the router through the central arbiter. The central arbiter first grants access to the header *A*, which is then routed through both the outputs *north* and *south*. Similarly, the remaining flits of *A* are also routed through these outputs. After routing of the tail, the central arbiter is released and can now be acquired by *B*, hence, no cyclic dependency.

This approach is simple, however, it can incur serious performance overheads. In this case, a normally parallel 5-port router is effectively transformed into a serial router, which can degrade the system-level throughput considerably and is therefore not a viable solution to avoiding deadlocks.

Flit-level distributed arbitration approach. This scheme uses the similar distributed arbitration at each output port as shown in Figure 5.10, however, the arbitration is now performed for every flit, and not just the header. After the winning flit is routed through an output port, the port is released and can now be acquired by the flit of another competing packet (assuming fair arbitration). Therefore, an output port is not held for the lifetime of a packet.

This scheme can avoid resource-dependent deadlocks [164]. Releasing the output port after routing each flit breaks the cyclic dependency between multiple packets. For example, in Figure 5.10, after the header for packet *A* acquires the *north* port, and *B* header acquires the *south* port, each of these output ports are released after the respective headers are routed and can now be acquired by the other packet's header. Hence, breaking the cyclic dependency and allowing for complete routing of the two packets without deadlock.

This approach, however, suffers from two major limitations concerning performance and correctness. In terms of performance, arbitrating and releasing an output port for every flit can have significant impact on throughput, compared to the standard wormhole routing employed in the proposed strategy, where the arbitration is only performed for the header and the output port is allocated for the entire packet so that the body/tail flits are simply fast forwarded after arrival. In terms of correctness, this scheme can result in an out-of-order delivery of flits to the local cores, which is erroneous. Ordering of flits can be guaranteed by tagging each flit with an ID and then performing a correct re-ordering at the local NI before delivering to the cores. However, this technique can have extra hardware overhead, such as a re-order buffer, as well as lower coding efficiency due to added ID bits in each flit. Given the associated prohibitive performance and correctness issues, this deadlock-avoidance approach is impractical.

5.4 Design Details: New Input Port Module (IPM)

The detailed design of the new IPM is now presented, which supports the continuous-time replication strategy, along with the details on its route computation unit and the CMR buffer.

5.4.1 IPM Structure and Operation

The structure and operation of the IPM is first described, before going into the design details of its units.

5.4.1.1 Structure

As shown in the Figure 5.12, the IPM has one input channel that connects to the upstream router and four output channels towards the OPMs through a crossbar. There are three components in the new IPM: a Route Computation Unit (RCU), the CMR buffer and four Address Modifier Units (AMUs) on each output direction. The RCU stores only the header addressing and selects the correct OPMs for routing. The CMR buffer, however, stores all the flits, which can be accessed by the OPMs concurrently using its four decoupled read ports. Finally, an AMU is present at each output of the CMR buffer, which modifies the header address such that there is always a unique path for the multicast packet to reach each destination, preventing sending multiple copies of the packet to the

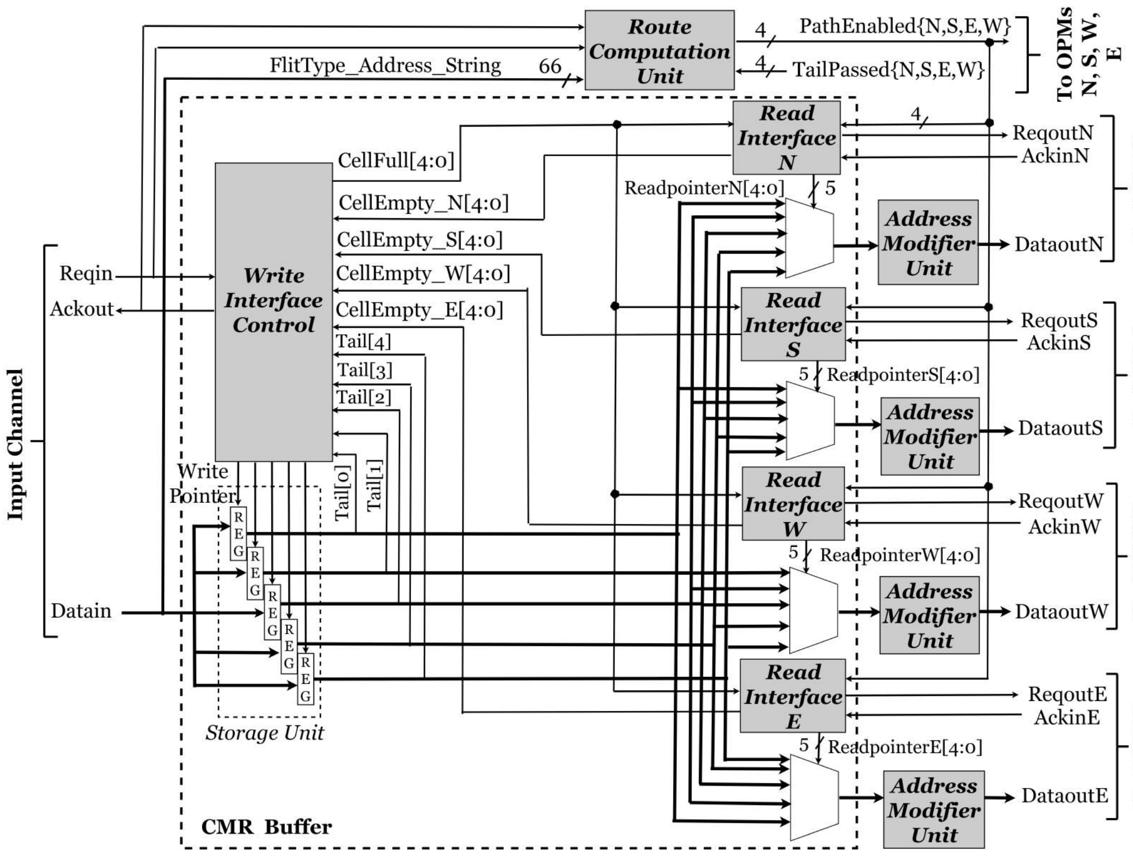

Figure 5.12: New IPM micro-architecture with CMR buffer

same destination through different paths.

5.4.1.2 Operation

When a new packet header arrives, it is stored in the CMR buffer and in parallel its address is stored in a small buffer in the RCU to start route computation. After the header is stored in the CMR buffer, it is speculatively broadcast to all read interfaces. The write interface, next, generates an *Ackout* on the input channel, which is used to (a) advance the buffer's write pointer, and (b) close the buffer in the RCU, disabling it for the remainder of the packet to save energy. A similar write protocol is followed for the body flits until the tail arrives and is stored. The *Ackout* for the tail is sent only after the tail has been read by all the correct OPMs of the buffer. This *Ackout* also re-activates the RCU.

Read operations on the buffer are performed in parallel to the write. The header is first speculatively read out of all the read interfaces, with AMU address modifications, and sent to all the

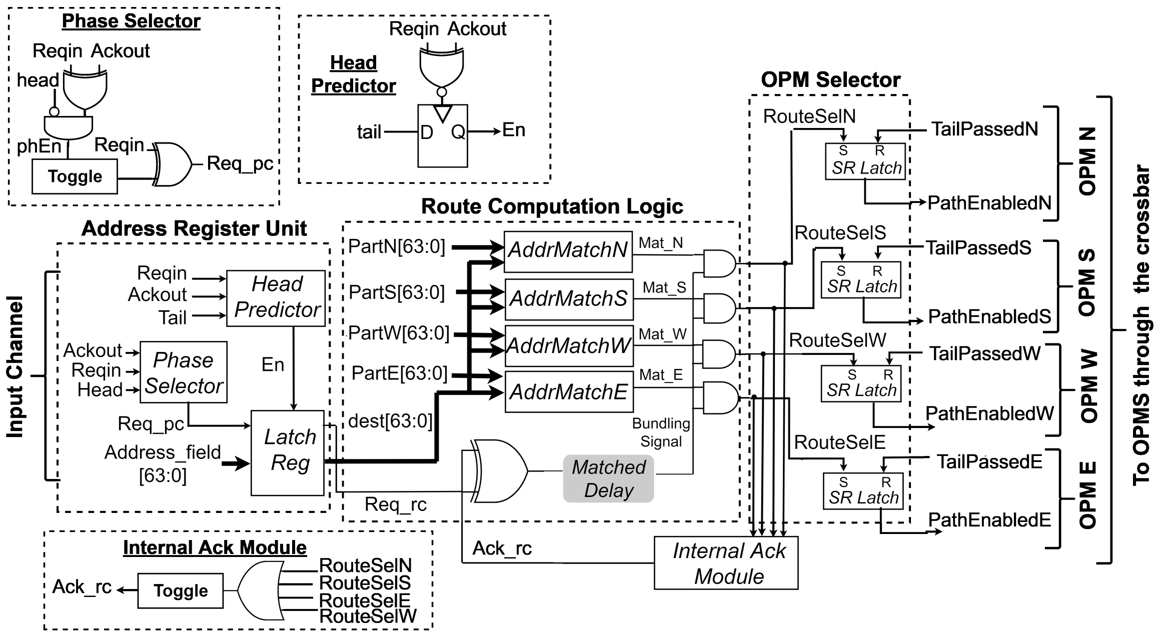

Figure 5.13: New route computation unit architecture

OPMs along with *Reqouts*. After the RCU finishes the route computation, the correct OPMs are selected using *PathEnabled*, which are also used by the read interfaces to throttle the copies on the wrong paths. Each read interface on the correct paths receives an *Ackin* from its OPM after the header has been routed through the OPM, and advances its individual read pointer. Similar read operations are performed for the body and tail flits, where each OPM on the correct path reads these flits independently.

5.4.2 Route Computation Unit (RCU)

5.4.2.1 Structure of RCU

The RCU has one input channel and four output channels connected to the OPMs through the crossbar (Figure 5.13). It has three main components: an address register to store the addressing of the header, a route computation logic to decode the addressing, and an OPM selector that selects the correct output ports for the lifetime of the packet.

5.4.2.2 Operation of RCU

The RCU operates only on the arrival of a header flit. When a packet header arrives on the input channel, its address field is stored in a normally-transparent latch register with the correct phase of its bundling request. The *head predictor* controls this register: it is closed immediately after an *Ackout* is sent on the input channel for the header and reopens after the *Ackout* for tail, anticipating the next header. The stored header address is then used to compute the correct routes for the packet by the *route computation logic.*

Route computation algorithm: The correct routes are selected by the *route computation logic* based on a network partitioning approach. Four 64-bit partition bit-strings are used corresponding to each output direction, which have a bit for every node: 1 if the node is reachable through this direction using XY routing, 0 otherwise. A bit-wise AND is then performed between the destination bit-string and each of these partition strings: if the result is non-zero then that direction is selected for routing as there is at least one destination reachable through this direction.

Finally, based on the computed routes from the *route computation logic,* the *OPM selector* then generates the correct *PathEnabled* signals for the packet lifetime, which are deasserted only after the appropriate *Tailpasseds* are received.

5.4.3 CMR Buffer

The micro-architecture and operation of the new CMR buffer is presented, followed by the detailed designs and functionality of its write and read control interfaces.

5.4.3.1 Structure of the CMR buffer

As shown in Figure 5.12, the CMR buffer consists of one *write interface* and four decoupled *read interfaces*. The buffer can handle up to 5-flit packets.

The *write interface* has one input channel and 5 output channels, corresponding to each cell of its storage unit. The input *Datain* is broadcast to all the storage cells with its handshaking signals connected to the *write interface* control. Each cell has an output datapath and a bundling *CellFull* broadcast to all the *read interfaces.* The *write interface* control selects the correct storage cell using a 1-hot write pointer. This control also gets 2-phase *CellEmpty* signals from each *read* *interface*, which provide the status of the read operations on individual cells, and the *Tail* flags from its datapath to determine which cell has the tail flit.

Each *read interface* is connected to the *write interface* and to an OPM through the AMU on datapath. This interface receives the data along with the bundled *CellFull* requests from each cell of the *write interface* and sends back 2-phase *CellEmpty* status signals. Each *read interface* has its own control and a 1-hot read pointer, which selects the correct cell to read using a MUX. The *read interface* also gets *PathEnabled* signals from the RCU to determine if the interface is on the correct path or not. The *read interface* control generates *Reqouts* to the OPM and receives *Ackin*.

5.4.3.2 Operation of the CMR buffer

A packet header first arrives on the input channel and is written to a cell pointed by the *write interface*. Next, the bundling *CellFull* request for this cell along with the data are speculatively broadcast to all the *read interfaces*. This broadcast is followed by an *Ackout* on the input channel, which then increments the write pointer. A similar write protocol is followed for the body and tail flits. However, the *Ackout* for the tail is generated only after it has been read by all the read interfaces (i.e. toggle on all the *CellEmpty* signals), both speculatively and non-speculatively.

The read interfaces of the buffer have a uniform operation for every flit. A cell is read if the read pointer is pointing to it, and its *CellFull* signal is toggled. In this case, the cell data is read out along with its *Reqout*. There are two types of reads: correct if the interface is on the right path, or incorrect. With a correct read, an *Ackin* is received from the OPM, which then increments the read pointer and toggles the corresponding *CellEmpty*. However, with an incorrect read, no *Ackin* is received. This case is detected using the *PathEnabled* signals from the RCU, and the speculative *Reqout* is canceled by toggling it again. This second toggle on *Reqout* also increments the read pointer and toggles the corresponding *CellEmpty* signal. Due to this uniform operation for both speculative and non-speculative reads, all the read pointers point to exactly the same location after all the flits of a packet are read, leading to an overall simple operation.

5.4.3.3 Write interface control structure and operation

Figure 5.14 shows the design of the write interface control, which connects to the input channel on the left and to all four read interfaces on the right.

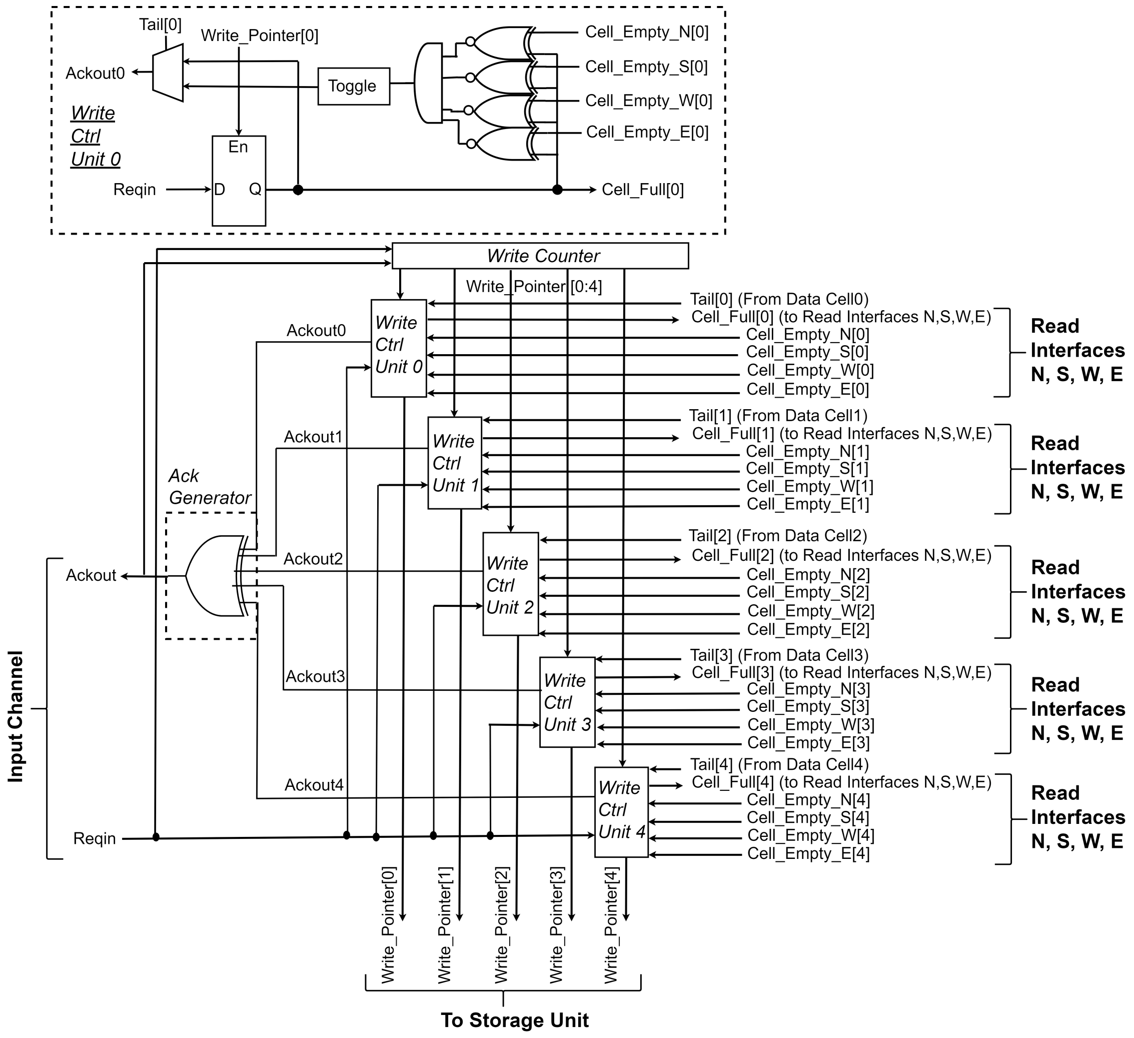

Figure 5.14: CMR buffer: write interface control

Structure. There are three main components of this interface: one write counter, five write control units corresponding to each buffer cell, and an ack generator. The write counter controls the write pointer, which selects the correct cell for storing the incoming flit. The write control unit detects writing of a new flit and generates a *CellFull* bundling request, which is speculatively broadcast to all the *read interfaces*. This unit also generates an internal ack following the storage of a flit. The ack generator merges internal acks from write control units to generate the final *Ackout* on the input channel, completing the handshaking.

Operation. A new header flit first arrives on the input channel, and its *Reqin* is broadcast to all the write control units. The write control unit, selected by the write pointer (a 1-hot code), generates a *CellFull* signal that signifies completion of the storing of the new flit. Next, an internal ack is generated by the control unit right after the *CellFull*. Finally, an *Ackout* is generated on the input channel by the ack generator after merging the internal ack outputs of the write control units. Toggling of the *Ackout* also increments the write counter, which advances the write pointer to point to the next cell of the storage unit. Similar operations happen for body/tail flits with one important distinction for the tail flit: the write control unit generates the internal ack only after the tail flit has been read by all the OPMs, i.e., all the *CellEmpty* signals have been received from the read interfaces, following a packet-based buffering protocol.

5.4.3.4 Read interface control structure and operation

Figure 5.15 shows the design details of a read interface control that connects to the CMR buffer's write interface on the left and to the north OPM on the right.

Structure. There are four main components of this interface: one read counter, five read control units corresponding to each buffer cell, a request generator, and an internal ack generator. The read counter controls the read pointer, which selects the correct cell for reading. The read control unit forwards the bundling request of the stored flit to the request generator, and also acknowledges the write interface by toggling its *CellEmpty* signal after a completed read operation. The request generator sends the final request to the OPM, derived from the request outputs of the read control units. The internal ack generator generates an ack for both speculative read, if the interface is on an incorrect path of a flit, or a non-speculative read if it is on the correct path. This internal ack is used to advance the read pointer and generate *CellEmpty* status signals.

Operation. The read control interface has a uniform operation for all flits of a packet. Initially, the read pointer, which uses a 1-hot code, selects the tail cell of the buffer. When a *CellFull* request, corresponding to any stored flit, arrives for this cell from the write interface, the selected read control unit will generate an internal request. This request will also be generated in another case, where *CellFull* has already arrived to a read control unit, and this unit gets newly selected by the read counter. After an internal request is generated, the req generator merges this request with the others and generates a *ReqX*, which is sent to the OPM. Before the *ReqX* is forwarded, a phase correction might be required if the read interface is on the incorrect path (speculative read case), as determined by the routing information from the RCU. In this case, a toggle is first generated

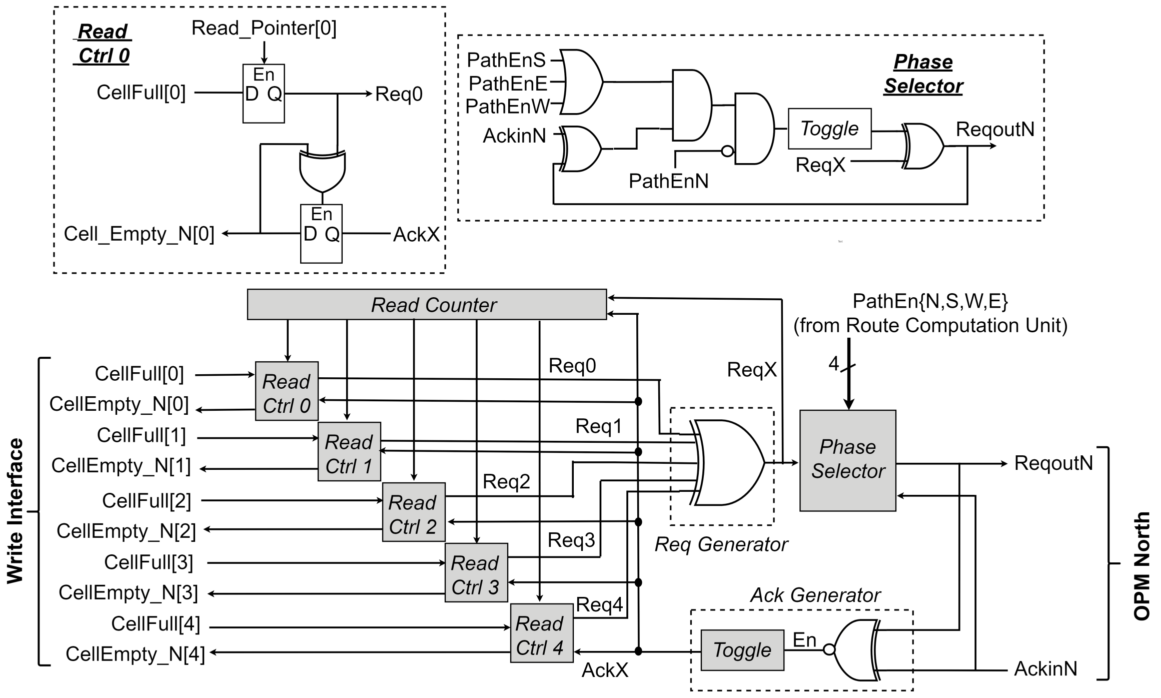

Figure 5.15: CMR buffer: read interface control

on the *Reqout*, but it is quickly canceled by the phase selector by toggling the *Reqout* again. This cancellation operation does not lead to any malfunctions as the small pulse on the *Reqout* is blocked by the normally-opaque input D-latches in the corresponding OPM, which has not been activated for this packet.

Finally, the ack generator generates an internal ack for both speculative and non-speculative read cases. In case of a speculative read, the second toggle on *Reqout* generates the *AckX*. However, for a non-speculative read, *Ackin* received from the correct OPM, showing that the flit has been routed through the OPM, generates the *AckX*. *AckX* is then used to perform two operations concurrently: (i) send a *CellEmpty* to the write interface through the selected read control unit, and (ii) increment the read counter i.e., the read pointer.

5.4.4 Address Modifier Unit (AMU)

Before sending the header to each OPM, a small adjustment is required to its addressing. The header address must be modified to guarantee that a unique path exists for the multicast packet to each destination, preventing sending multiple copies of the packet to the same destination through

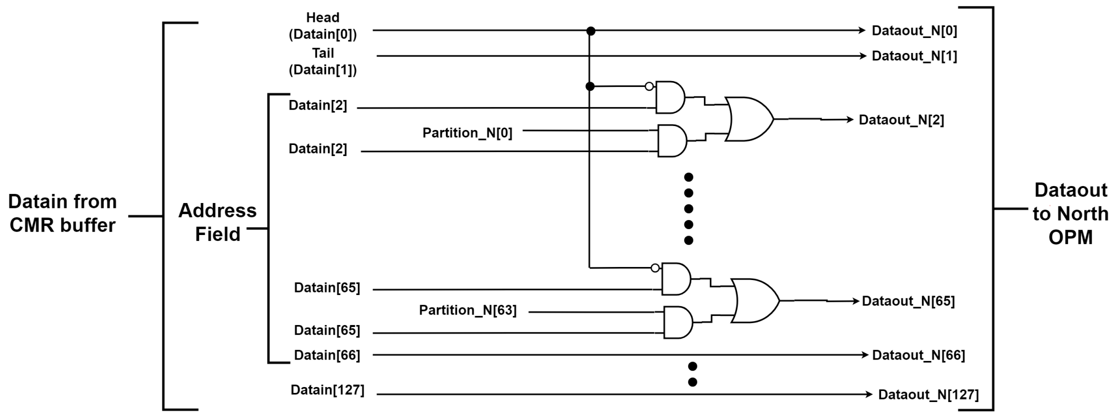

Figure 5.16: Address modifier unit (AMU)

different paths. The AMU performs this perturbation only on the header address, keeping the data bits of the header, and the complete body and tail flits unchanged.

As shown in Figure 5.16, the address modification is implemented using a network partitioning approach. In this approach, four 64-bit partition bit-strings are used, corresponding to each output direction, which have a bit for every node: 1 if the node is reachable through this direction using XY routing, 0 otherwise. At each AMU, a bit-wise AND is then performed between the header address bit-string and the corresponding partition bit-string; the result is a new address bit-string with some destinations masked that are not reachable by XY routing through this direction.

5.5 Experimental Setup and Node-level Results

The *new parallel multicast* and the *baseline serial multicast networks* are implemented and evaluated at post-mapped, pre-layout level. The experimental framework and node-level evaluations in terms of latency, area and energy are now presented, followed by network-level results in the next section.

5.5.1 Experimental Framework

5.5.1.1 Experimental setup

Two 8x8 2D-mesh networks, *new* and *baseline*, are implemented in Nangate 45 nm technology. One five-slot buffer is used per input port of the routers: the *baseline* uses a single read-ported circular buffer, while the *new network* uses a CMR buffer. The datapath width used is 128 bits. Router nodes are first technology mapped to the Nangate standard cell library using the Cadence Virtuoso tool. Next, the Spectre simulator is used to extract gate-level models, i.e. determine rise/fall delays for every I/O path of each gate, at a typical process corner. The channel length and delay are borrowed from the Freescale PowerPC e200z7 floorplan: 1 mm and 100 ps [105]. The networks are implemented in Structural Verilog using the extracted models.

An asynchronous NoC simulator, previously developed for unicast [90], was extended to support multicast. The simulator uses a Programming Language Interface (PLI) to connect a technology mapped network to a C-based test environment. This environment was significantly enhanced to support injection and validation of routing of multicast packets. Packet headers are injected following an exponential distribution. Long warmup and measurement phases are also used based on a procedure similar to [41]. In addition, asynchronous network energy is measured using two steps: (a) annotate the precise switching activity of every wire during a benchmark run, (b) run Synopsys PrimeTime using the recorded activity to compute energy.

5.5.1.2 Benchmarks

Most of the experiments are conducted on 8 synthetic benchmarks, comprising both unicast and multicast benchmarks.

The unicast benchmarks contain the widely-used communication patterns, common in real applications, resembling both uniform and congestion traffic scenarios [41].

The multicast benchmarks strongly align with real applications such as cache coherency and neural networks, which have been shown to exhibit multicast traffic. Cache coherence traffic, e.g., due to protocols such as token coherence and region-based coherence, can contain 5-10% of total injected traffic as multicast [88]. On the other hand, spiking neural networks can contain only multicast or broadcast traffic; the former is seen in random neural network (RNDC), and the latter in a Hopfield network [188]. The synthetic multicast benchmarks, created in this chapter, contain such real multicast patterns, and also provide an opportunity for a thorough evaluation by having more control on parameters, such as injection rate.

There are 3 unicast benchmarks and 5 multicast benchmarks. The unicast benchmarks are: (1) *uniform random*; (2) *bit complement*; and (3) *hotspot10*, which is similar to *uniform random* but the middle four routers have 10% higher probability of receiving packets than others. The multicast benchmarks are: (4) *multicast5* and (5) *multicast10*, where 5% and 10%, respectively, of total injected packets are multicast to random destinations and rest of the traffic is uniform random unicast, (6) *multicast-static*, where 16 sources are statically fixed to do random multicast, while remaining perform uniform random unicast; (7) *all-multicast* and (8) *all-broadcast*, where all sources only perform random multicast or only broadcast.

5.5.2 Node-Level Results

Before presenting the network-level results, node-level comparisons are first performed between the *baseline* and the *new router* in terms of three important cost metrics: area, latency, and energy. These node-level results are important as they form the basis for the later network-level performance and energy.

5.5.2.1 Node-Level Area

As shown in Table 5.1, the *new router* has almost 2× more area than the *baseline router*. This overhead is expected and primarily due to the new parallel multicast capability. However, this area overhead will be amortized when considering the total network-level area, including link area. [1] As shown later, the simpler *baseline* incurs significant performance and energy overheads for multicast traffic, justifying the need for this parallel multicast capability.

It it also useful to get a rough area comparison of the new asynchronous router with a state-of-the-art single-cycle synchronous multicast router. The latter has an area of $227\,230\,\mu\text{m}^2$, using 6 VCs per port and 10 total buffer slots, each with a 64-bit datapath [149]. The total buffering of this router is similar to our new router (one 5-slot buffer per port with a 128-bit datapath). Therefore, the two routers are estimated to have similar buffer area, which typically dominates the router area.

[1]It would be interesting to also analyze how this area overhead will scale with adding VCs. The area overhead of the new router with VCs over the baseline with VCs will be similar to the current 2× overhead, which does not include VCs in both the designs. This outcome is due to the use of multi-switch architecture with replicated crossbars for these projected VC-based routers, rather than the multi-stage architecture with shared crossbar, which is less efficient for asynchronous designs [89]. Since the switch area is the dominating factor in multi-switch architectures, the areas of the two routers will scale linearly w.r.t. the switch area as the numbers of VCs are increased.

Metrics	***Baseline Node***	***New Node***
Node area	24 962 μm^2	54 083 μm^2
Node latency (header)	833 ps	693 ps
Node latency (body)	602 ps	636 ps

Table 5.1: Node-level area and latency comparisons

Both the routers use 45 nm technology, but the synchronous one is laid out in a different process and also uses lookahead optimization, which should not have considerable impact on area. Overall, the new asynchronous router is estimated to have roughly $4\times$ lower area than the synchronous one.

5.5.2.2 Node-level latency

As shown in Table 5.1, node latency is measured for both header and body flits. Header latency is critical, since these flits incur the most overhead setting up the path, and they typically define the packet latency. Interestingly, unlike synchronous, header and body latency can be different in an asynchronous NoC as the header takes longer time due to address decoding and arbitration/channel acquisition, while the body/tail flits can propagate faster after this pre-allocation.

In case of the header, latency of *new router* is 16% lower than the *baseline*. The *baseline router* performs buffering and route computation serially, while the *new router* performs buffering in parallel with route computation. This parallelism significantly improves header latency for the *new router*, recouping any overheads due to a more complex route computation that handles multicast addressing.

For body flits, however, both have almost the same latency as they have similar forward critical paths.

5.5.2.3 Node-level energy

Figure 5.17 presents the node energy results for the *baseline* and the *new* routers considering all possible transmission scenarios.

Four simple benchmarks were created for this experiment that capture all possible unicast and multicast transmissions across a node, and their impact on energy as the degree of multicast in-

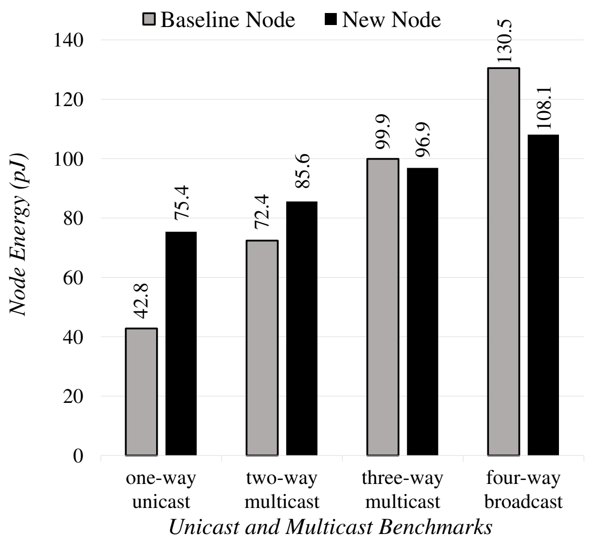

Figure 5.17: Node-level energy results for different unicast and multicast transmission scenarios

creases. All of these benchmarks comprise of 5-flit packet transmissions: (i) *one-way unicast:* single unicast packet is sent from an input port of a router to an output port, (ii) *two-way multicast:* in case of the *new router,* single multicast packet is injected at an input port, however, since the *baseline* performs unicast-based serial multicast, this multicast packet is divided into two unicast packets and injected serially, (iii) *three-way multicast:* for the *new router,* single multicast packet is injected at an input port, but for the *baseline*, this packet is divided into three unicasts and injected serially, and (iv) *four-way broadcast:* in this final transmission, single broadcast packet is injected at an input port of the *new router,* while the same packet is divided into four unicasts and injected serially in case of the *baseline.*

As shown in the Figure 5.17, there are two important observations for the four different transmission benchmarks:

- *Individual node trends: energy vs. multicast degree.* Considering the energy trends of a single node as the number of outputs in the multicast transmissions increase, the *new router* shows significantly lower energy overheads than the *baseline*. The *baseline* shows 69.04% higher energy for *two-way multicast* over *one-way unicast,* 38.02% higher energy for *three-way multicast* over *two-way,* and 30.61% more energy for *four-way* than *three-way.* In contrast, these overheads for the *new router* are significantly lower: 13.51%, 13.09%, and 11.57%

respectively. The benefits seen by the *new router* are due to the use of parallel multi-way replication of a single packet, which is much more energy-efficient than serial routing of multiple packet copies for each multicast transmission, as employed by the *baseline.* The latter approach leads to extra switching activity in the IPM due to buffering of each additional packet copy and performing route computation for this copy.

- *Pairwise node trends: baseline vs. new approach.* Comparing the two nodes for each transmission scenario, the *baseline* shows moderate to significant (18.30-76.19%) energy improvements for *one-way* and *two-way* scenarios, but the *new router* demonstrates 3.06-17.18% better energy for *three-way* and *broadcast.* Energy overhead for the *new node* in case of one-way and two-way multicasts is due to its higher complexity than the *baseline*, but which is required for parallel multicast support. In contrast, for multicasts intended for more number of outputs, the *baseline's* routing of multiple packet copies for a single multicast leads to significantly higher switching activity than *new*, recouping any overheads due to the *new router's* higher complexity.

5.6 Network-level results

Network latency, throughput and energy results are now presented for the *new* and the *baseline networks*, considering both multi-flit and single-flit packets. In addition, a new metric, *multicast delivery time,* is also introduced and evaluated for multi-flit packets. Furthermore, an interesting case study is performed to measure the critical latency and energy of an isolated multi-flit multicast packet. Finally, careful analytical comparisons are made to three different synchronous NoCs.

In the above experiments, a multi-flit packet consist of 5 flits, which is a general size for various parallel computing applications. Single-flit is also common for multicast control packets in cache coherency, e.g. write-invalidates [1], and in neural network applications, where the packet size can be as small as 8 bits [196].

Benchmarks are considered in three distinct categories: *unicast, a mix of unicast and multicast,* and *only multicast/broadcast.* Unicast benchmarks show the impact of the new parallel multicast capability on basic unicast performance and energy. The mix of two types of traffic is common in cache coherence[1] and therefore is also evaluated. Finally, all multicast/broadcast represents an extreme scenario, common in emerging applications like neuromorphic computing.

5.6.1 Multi-Flit Network-Level Results

This section presents the network-level results for multi-flit packets, in terms of latency, multicast delivery time, throughput and energy. An interesting evaluation of an isolated single multicast packet for latency and energy is also performed.

5.6.1.1 Network latency

For multi-flit packets, network latency is measured for all unicast and multicast benchmarks. For unicast and mix of multicast/unicast benchmarks, latency is measured both at a fixed rate as well as at varying injection rates. For extreme multicast benchmarks, latency is only measured at a fixed injection point.

Unicast latency. Figure 5.18 shows the average latency results for all three unicast benchmarks: *Uniform Random, Bit Complement,* and *Hotspot10.* Latency is measured for header at 25% of saturation load of *baseline*. This injection rate is high enough to show interesting benchmark characteristics with network largely uncongested. The *new network* shows moderate latency reductions from 6.1% (*uniform random*) to 14% (*hotspot10*). This important result shows that the *new network* not only supports parallel multicast but also improves unicast latency.

Figure 5.19 shows the average latency results for the same three unicast benchmarks but now at a varying injection rate, covering a complete range from zero-load to saturation. For all three benchmarks, the *new network* achieves significantly lower latency than the *baseline* for low to moderate injection rates, while incurring overheads close to the saturation. The latency improvements seen

[1]Cache coherence traffic may also include aggregation patterns (many-to-1 or many-to-few), where multiple processing nodes send back individual ACKs to a single node or a small number of nodes. The proposed network can handle this traffic as these ACKs are sent using unicast packets, which can then be aggregated at the network interface attached to the receiving router, followed by delivering the combined packet to the processor. This approach, although simple, can have energy overheads due to increased network utilization. These overheads can be minimized using an intermediate ACK aggregation protocol: instead of aggregating at the destination, ACKs can be coalesced at the intermediate routers, and then the combined ACK forwarded to the next router, followed by further aggregation, until the destination is reached [118].

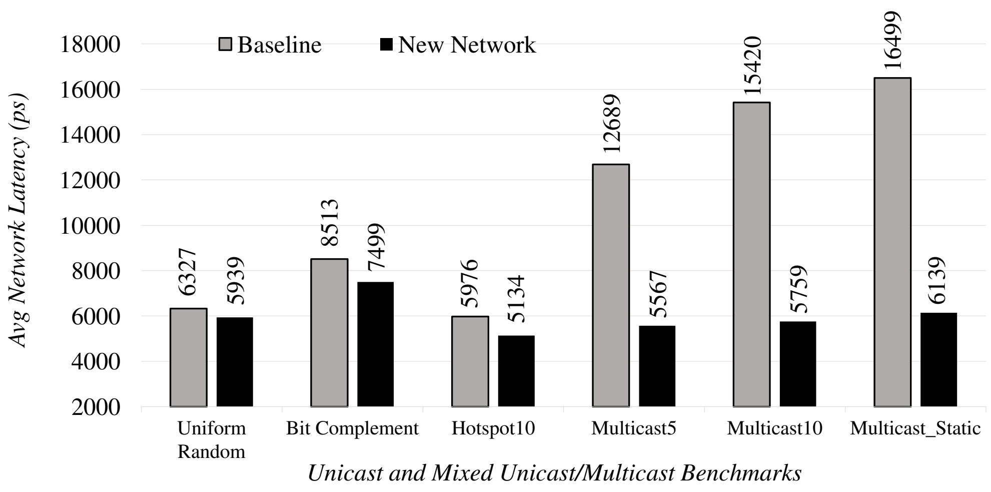

Figure 5.18: Network latency for unicast and mixed unicast/multicast benchmarks at 25% saturation load of *baseline*

by the *new network* for low to moderate injection rates are due to the critical path optimization in its router node that performs buffering of the header in parallel with the route computation rather than serially as in the *baseline.* However, the *new network* saturates earlier than the *baseline* and incurs latency overheads at high-traffic load as the *new network* uses packet-based buffering policy. In this policy, ack to upstream for the tail flit of a packet has a conservative synchronization, which can be sent only after the tail has been routed on the output channel. This protocol incurs significantly longer delay than the flit-based buffering in the *baseline,* where ack can be sent right after the tail is buffered. At high-traffic load, for the *new network,* this long wait for ack to upstream can become a bottleneck for routing of the next packet, hence, degrading network latency.

Mixed unicast/multicast latency. Figure 5.18 also shows the average network latency results for the three mixed unicast/multicast benchmarks: *Multicast5, Multicast10,* and *Multicast-Static.* Latency is measured at 25% saturation load of the *baseline.* The latency for a multicast packet is measured from the time of injection of header (or first header for *serial baseline*) to the arrival of the last header at any destination. As shown, the *new network* achieves significantly lower latency (56.13-62.65%) than the *baseline.* These results show that parallel replication strategy of the *new* network considerably outperforms the serial unicast-based multicast approach of the *baseline.* In case of the *baseline*, the time from injection of the first header to the arrival of the last header at any destination can be very long due to the serial injection and routing of each unicast copy.

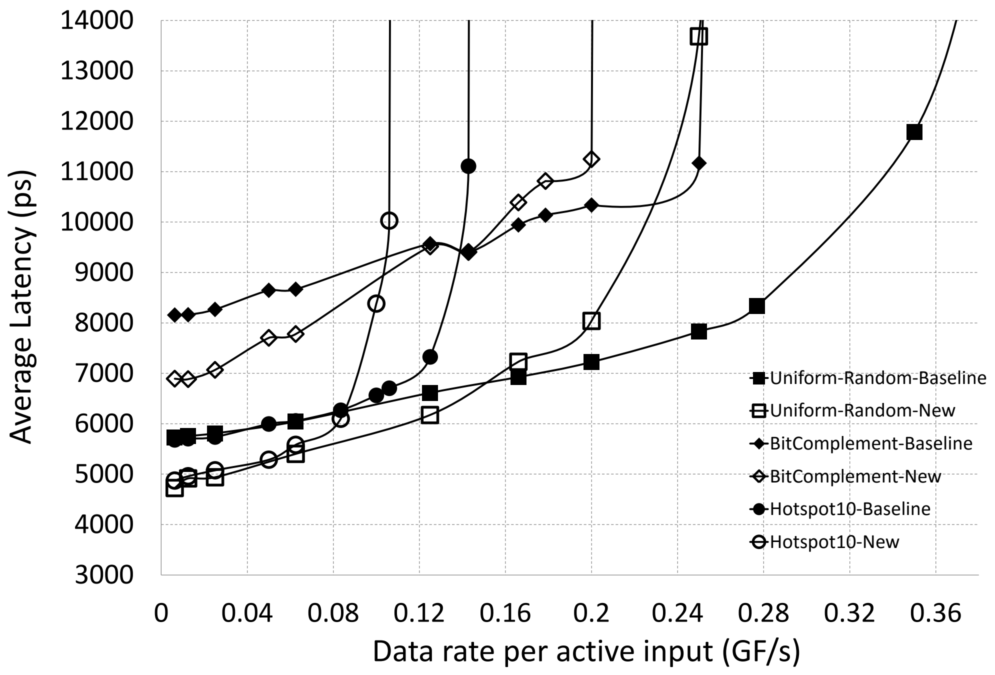

Figure 5.19: Unicast network latency at varying injection loads

	Baseline	*New Network*
all-multicast	60842 ps	7872 ps
all-broadcast	307108 ps	9744 ps

Table 5.2: Zero-load latency for *all-multicast* and *all-broadcast*

Figure 5.20 shows the average latency results for the same mixed unicast/multicast benchmarks at a varying injection rate, covering a complete range from zero-load until saturation. There are three important trends visible from these results: (i) latency magnitudes, (ii) shape of curves, and (iii) points of saturation. The *baseline* incurs significant latency overheads: for each benchmark, the gap between the curves increases significantly as injection rate increases. Also, note the steep rise in latency for *baseline* as the number of multicast packets injected increase. The *baseline* curves are steeper for a higher degree of multicast (*multicast10* and *multicast-static*). Finally, the *baseline* saturates very early compared to the *new network*, showing the effectiveness of parallel multicast.

Extreme multicast latency. Finally, zero-load latency is measured for two extreme cases, as shown in Table 5.2: *all-multicast* and *all-broadcast*. Even at zero-load, with a single packet injected

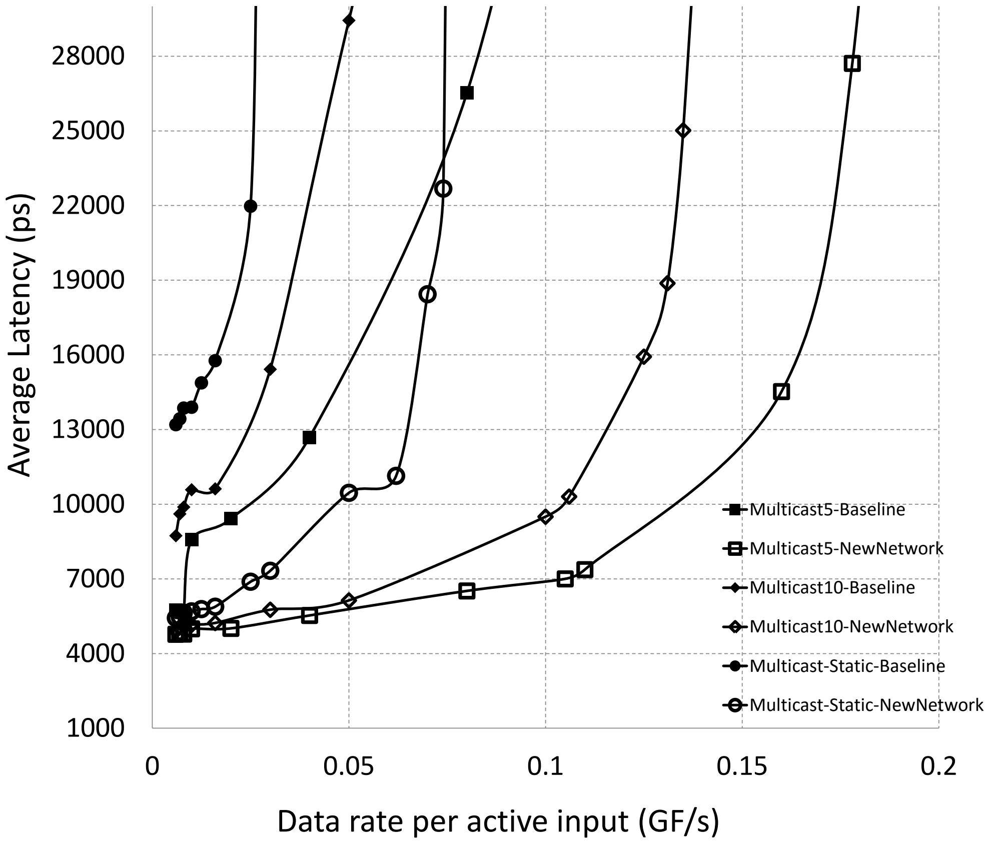

Figure 5.20: Mixed unicast/multicast latency at varying injection loads

from each source, the latency for *serial baseline* is severely degraded for both benchmarks, showing that the *baseline* is not a practical solution to support traffic comprising only multicast and broadcast. In contrast, the *new network* handles this traffic well and achieves 87.0% and 96.8% latency improvements for *all-multicast* and *all-broadcast*, respectively.

5.6.1.2 Multicast delivery time

While the previous sub-section evaluates latency, which is the time to deliver the last header flit of a multicast packet, this section measures delivery time of complete multicast packet copies, i.e. tail flits. This metric is important as it shows how fast entire multicast packet copies can be delivered to the destination computing nodes, which can then start processing the packets. Such delivery time has a direct impact on the overall application execution time. This metric is also critical to evaluate the effectiveness of the continuous-time replication strategy, which allows the copies of a multicast

	Multicast5	*Multicast10*	*Multicast-Static*	*All-Multicast*	*All-Broadcast*
	Min. Multicast Delivery Time (ps)				
Baseline	11485.75	11890.53	14989.86	9927.83	7618.36
New	5117.31	5105.14	5354.43	5094.67	4801.10
	Avg. Multicast Delivery Time (ps)				
Baseline	43672.76	37179.04	30492.23	36481.54	188076.02
New	9036.59	9099.94	8498.87	8160.25	8575.39
	Max. Multicast Delivery Time (ps)				
Baseline	73665.08	63090.93	51697.08	63694.01	309975.88
New	13773.72	14043.22	12595.75	11241.57	13109.37

Table 5.3: Minimum, average, and maximum multicast delivery time for different benchmarks, averaged over the total number of multicast packets injected for each benchmark.

packet to be routed to different destinations at varying speeds.

Table 5.3 compares the *baseline* and *new network* for three different multicast delivery times: minimum, average, and maximum. These metrics are measured for both mixed unicast/multicast benchmarks, and the extreme all-multicast and all-broadcast. For the mixed unicast/multicast benchmarks, these metrics are measured at 25% of the saturated load of *baseline*, while for the extreme cases, zero load is used.

For a multicast packet, three different delivery metrics are explored. The first is the *minimum multicast delivery time,* defined as time from injection of the header (or first header for the baseline) to the delivery of the first tail to any destination. *Average multicast delivery time* is the average of the delivery times of all the multicast copies to their respective destinations. *Maximum delivery time* is a dual of the minimum delivery time, and is defined as the time from injection of the header to the last tail delivery to any destination. While the min time reflects the fastest a multicast packet can be delivered to any destination, the max time shows the slowest delivery time, perhaps due to network congestion. The average case is also important to get an overall sense of the multicast packet delivery and its impact on application performance.

As shown in the table, the *new network* consistently outperforms the serial *baseline,* for all three metrics, with improvements ranging from 36.9% to 95.7%. These three metrics truly capture the effect of the new continuous-time replication strategy, which allows copies of a multicast packet to be routed to different destinations, in parallel, and at varying speeds, where some copies traversing less congested regions are delivered very fast (e.g. min time of 5105 ps for *Multicast10*), while others can take longer time (max time of 14043 ps for *Multicast10*). The *baseline*, however, suffers major overheads due to the serial injection/routing of each unicast copy of a multicast packet, leading to severe congestion, which degrades not only the avg/max multicast delivery times but also the minimum (11890 ps for *Multicast10*).

5.6.1.3 Output throughput at saturation load

Figure 5.21 shows the output throughput results for all benchmarks at each network's saturation load.

For unicast, the *new network* incurs an overhead of 13.3% (*hotspot10*) to 30.9% (*uniform random*). This overhead is the result of packet-based buffering in the *new network,* which leads to longer delay in acknowledging to the upstream router after the tail flit is stored in the current router, compared to the flit-based buffering used in *baseline.*

However, for multicast, the *new network* shows significant improvements: 25.6% (*multicast5*) to 88.2% (*multicast-static*) and 170.2% (*all-broadcast*) higher throughput. In summary, more gains are seen for benchmarks with a higher multicast amount as the parallel multicast capability provides a higher threshold for saturation than serial.

5.6.1.4 Total network energy

Figure 5.22 shows total network energy results for all benchmarks.

For unicast, the *new network* incurs 2× overhead over the *serial baseline* due to the extra instrumentation for parallel multicast.

However, for multicast benchmarks, the routing of several unicasts for each multicast in *serial baseline* leads to higher resource utilization than the *new network.* Interestingly, this low utilization in the *new network* overcomes the complexity overhead as multicast amount increases, showing a

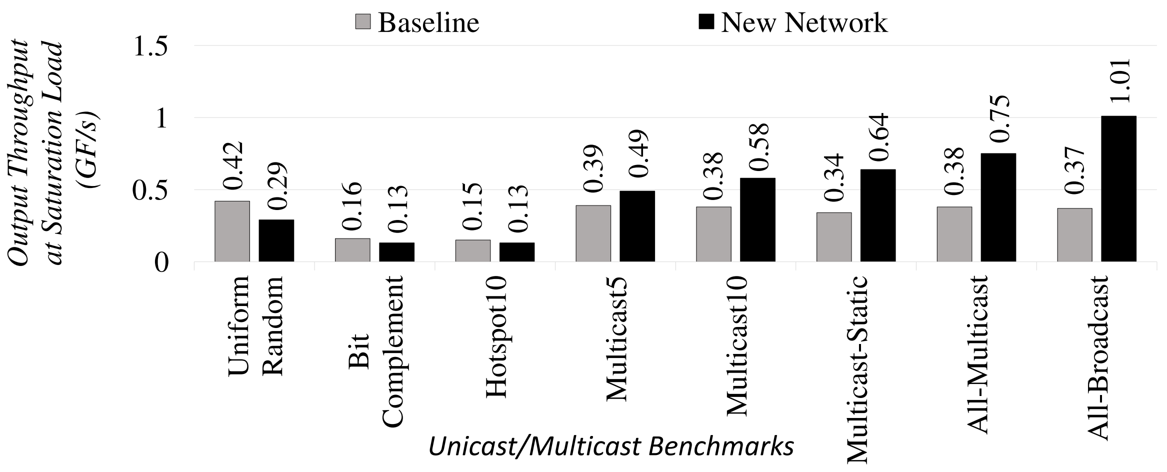

Figure 5.21: Output throughput at saturation load

28.7% degradation for *multicast5* to 2.7-57.2% lower energy than *baseline* for benchmarks with higher multicast portion.

Overall, in terms of multi-objective design space exploration for multicast traffic, the *new network* has a clear advantage over *baseline*. As one important example, for mixed traffic like *multicast10*, the *new network* shows significant improvements in latency and throughput, with slightly lower energy, but with an expected area overhead.

5.6.1.5 Isolated multicast case study

Finally, limited experiments are conducted to measure latency and energy of a single isolated multicast packet. These results are critical to give insight into the actual delivery time and energy of a single multicast packet, not averaged with other traffic.

A head-to-head comparison is performed, while varying the number of destinations and the location of the packet source. Four multicast benchmarks are created with a single 5-flit packet: (i) *short-multicast*: to 2 immediate neighboring destinations, (ii) *typical-multicast*: to 30 destinations uniformly distributed across network, (iii) *extreme-multicast*: to 62 destinations, and (iv) *broadcast*. Two different source locations are considered: corner and center, covering a range of possible multicasts.

As shown in Figure 5.23, *new network* achieves 72.9% (*short-multicast*) to 95.2% (*broadcast*) lower latency for a corner source. The absolute latency for both networks increases with the longest

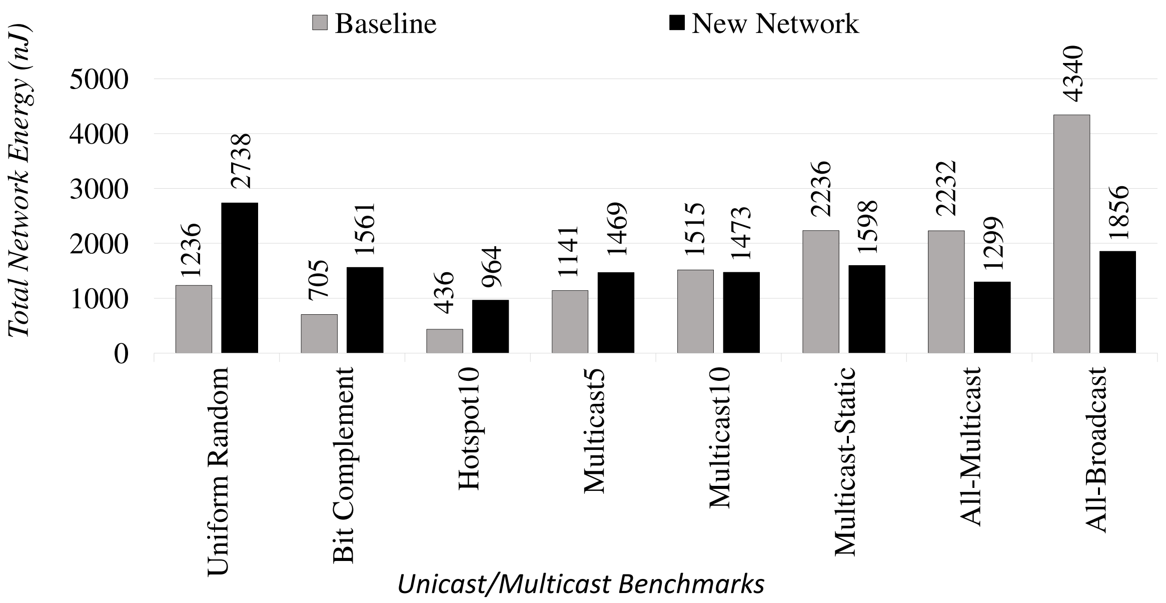

Figure 5.22: Total network energy results measured at 25% saturation load of *baseline*

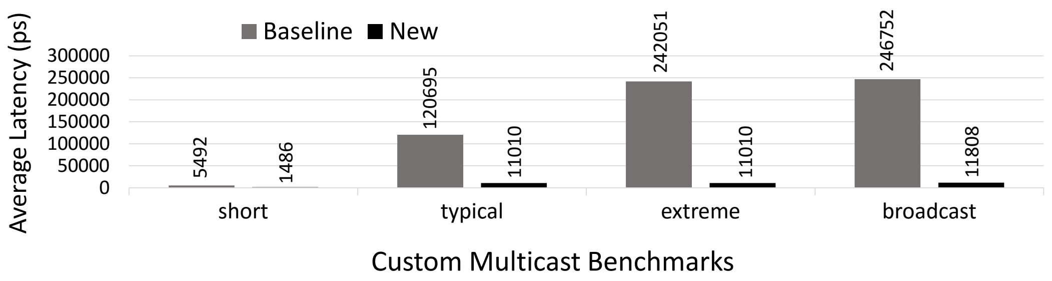

Figure 5.23: Isolated multicast case study: network latency for corner source

path length: 2 hops in *short-multicast*, 13 hops in *typical/extreme multicast* and 14 hops in *broadcast*. The poor *baseline* results (and up to 37× worse than *new* for center source as shown in Figure 5.24) are due to the long delay from injection of first header to the arrival of last header.

For energy comparison, as shown in Figure 5.25 and Figure 5.26, for both sources, the *new network* incurs 59.8% overhead over *baseline* for *short-multicast*. In contrast, in case of the corner source, for *typical-multicast* and *broadcast*, with more destinations, the *new network* achieves 29.6% to 58% lower energy, respectively, due to the lighter traffic in parallel multicast. Similar results are shown for the center source.

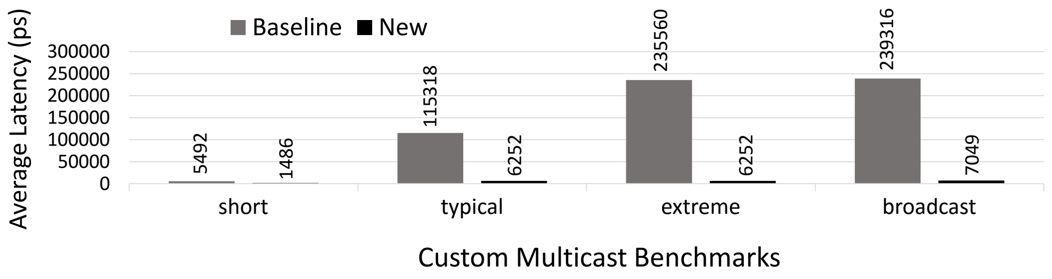

Figure 5.24: Isolated multicast case study: network latency for center source

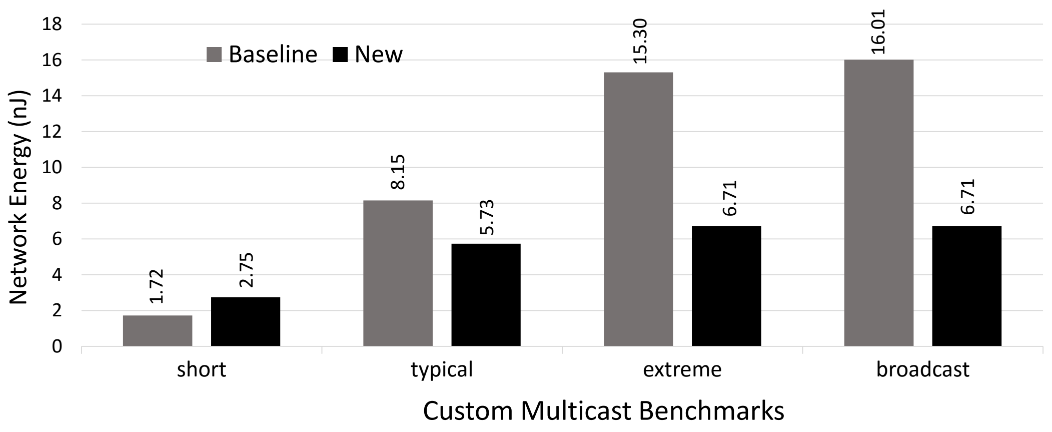

Figure 5.25: Isolated multicast case study: network energy for corner source

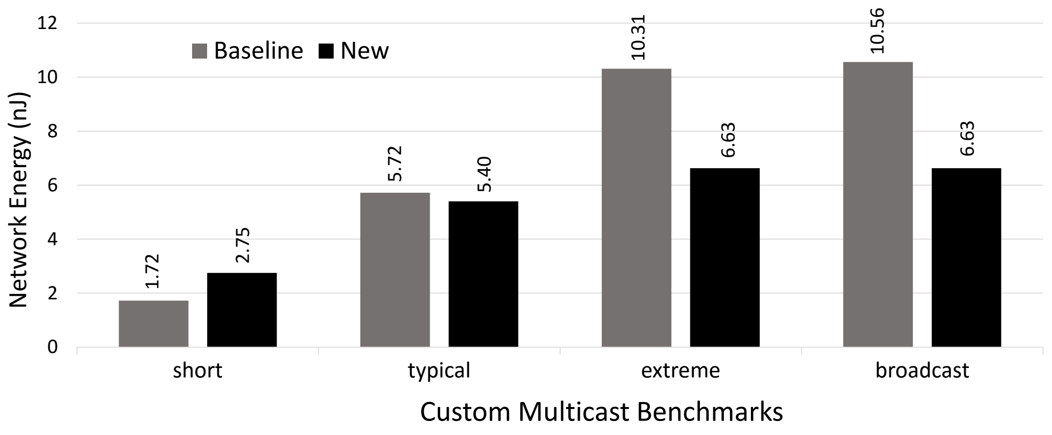

Figure 5.26: Isolated multicast case study: network energy for center source

5.6.2 Single-Flit Network-Level Results

While the network-level evaluation of the routers on steady-state traffic of multi-flit packets is quite useful, it is also interesting to compare the operation of the routers on single-flit packets, which are often part of typical applications; and can dominate traffic in control-oriented transmissions. A limited set of experiments measures performance and energy of single-flit packets for two benchmarks: *uniform random* and *multicast10*.

5.6.2.1 Network latency

Figure 5.27 shows the network latency results for both benchmarks, measured at 25% of the saturation load of the *baseline.* For the unicast *uniform random*, as expected, the *new network* achieves 11.6% lower latency than *baseline*. This result is due to the critical path optimization in the *new* router that performs buffering of the header in parallel to the route computation. For *multicast10*, the *new network* achieves 33.6% lower latency than the *baseline* due to its efficient parallel multicast capability.

5.6.2.2 Output throughput at saturation load

In terms of saturation throughput (Figure 5.28), the *new network* incurs a 57% degradation over the *baseline* due to its conservative write protocol performed at a packet granularity. However, for *multicast10*, the *new network* achieves almost the same throughput as the serial *baseline*, where the *new network's* parallel multicast capability recoups some of the overheads due to its packet-based buffering protocol.

5.6.2.3 Total network energy

Figure 5.29 shows the total network energy results for the two benchmarks in case of single-flit packets. The trends are similar to the multi-flit packets: (i) for unicast *uniform random,* the *new network* incurs almost 2X overhead due to the more complex router design, but which is required for parallel multicast capability, and (ii) for *multicast10*, the *new network* achieves a 2.52% lower energy than the *baseline* as the latter leads to extra network utilization due to routing of multiple unicast packets for each multicast packet. For single-flit traffic also, like multi-flit packets, this

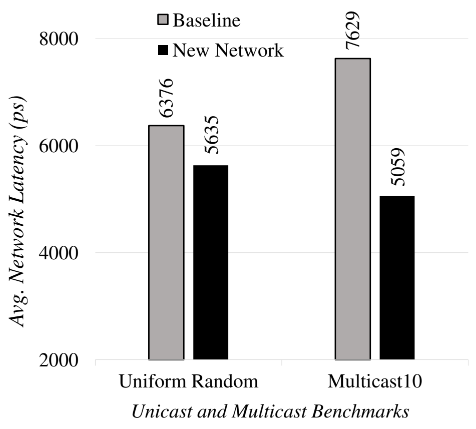

Figure 5.27: Network latency for single-flit traffic at 25% saturation load of *baseline*

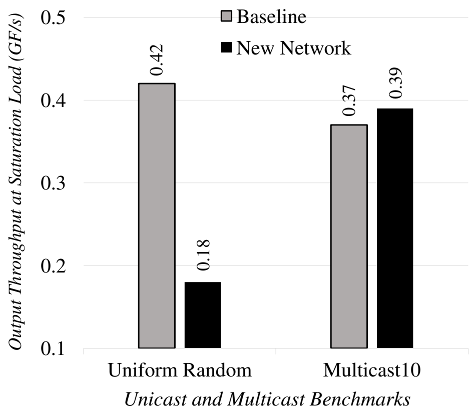

Figure 5.28: Saturation throughput for single-flit traffic

improvement is expected to increase for benchmarks with higher amount of multicast

5.6.3 Analytical comparison with state-of-the-art synchronous multicast NoCs

While the previous comparisons only use an asynchronous NoC baseline, it is also important to compare the the proposed asynchronous NoC with state-of-the-art synchronous multicast NoCs.

Three synchronous multicast NoCs are considered: (i) a high-performance tree-based multicast [149], (ii) multicast SMART NoC [105], and (iii) a hypothetical ”reduced-overhead” multicast SMART NoC. The first NoC supports highly-efficient multicast using single-cycle routers, where a flit traverses both router and link in a single clock cycle. The second NoC extends the original unicast-only SMART NoC [104] to perform multicast. This multicast SMART NoC is aggressive and achieves full-chip broadcast in just 2 cycles for an 8x8 2D mesh, where multiple routers can

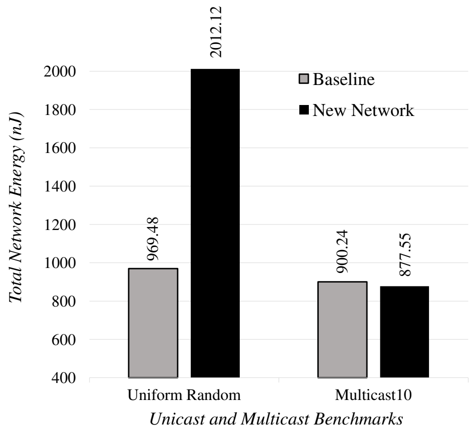

Figure 5.29: Total network energy for single-flit traffic at 25% saturation load of *baseline*

be bypassed in a single cycle, using an early arbitration and channel pre-allocation approach. The third NoC does not exist, and is a hypothetical potential future version of a recent unicast-only low-overhead SMART NoC [31], which currently only supports unicast, but now imagined to be extended to support multicast for the purpose of comparisons.

The aim of this analysis is to provide relatively normalized comparison of these three synchronous NoCs with the proposed asynchronous NoC. While the focus is mainly on network latency, switch-level area, channel overhead, and protocol operation are also compared when possible. To compare latency, the benchmark considered is *all-broadcast,* where each source sends every packet to all destinations, and single-flit packets are used. This benchmark was used as its latency can be easily extracted for the synchronous NoCs from the corresponding papers, while new experiments were performed for the proposed asynchronous NoC. All three NoCs are at the same pre-layout level of implementation, target 8x8 2D mesh, and use similar 45 nm technology, although in different processes, which, however, should not have considerable impact on latency comparison. Furthermore, these synchronous NoCs also use VCs, where tree-based multicast [149] uses 6 VCs per port but with almost the same total amount of buffering as the proposed asynchronous NoC, while the multicast SMART NoC [105] uses 12 VCs per port but the amount of buffering is different. Although the proposed asynchronous NoC does not use VCs, it can be extended to include VCs with minimal latency overhead, as discussed later.

(i) High-performance tree-based multicast [149]. For zero-load network latency, for *all-broadcast,* the synchronous NoC reported 11500 ps, while the asynchronous NoC achieved 9102.9 ps, i.e.

20.8% lower latency. Additionally, at a switch-level comparison, the router of this synchronous NoC was shown to have a $4\times$ area overhead over the new asynchronous router (see Section 5.5.2), which can potentially also lead to lower total power for the asynchronous NoC.

(ii) Multicast SMART NoC [105]. The multicast SMART NoC is compared in terms of network latency as well as area and energy costs. This design showed a zero-load network latency of 5600 ps for *all-broadcast* traffic, which is significantly better than the asynchronous NoC latency of 9102.9 ps. This result is expected due to the aggressive optimization of bypassing multiple routers in a single cycle in the SMART NoC, while the proposed NoC is the first basic solution to support multicast in asynchronous NoCs, and does not include such optimizations.

Moreover, in terms of other costs such as area, energy and power, the SMART NoC is expected to have substantial overheads compared to the proposed NoC. There are two primary reasons: use of an expensive multicast protocol, and hardware overheads due to the extra monitoring network. In terms of protocol, the SMART NoC performs multicast using either a full-chip broadcast and then dropping packets at non-destinations or using a serial unicast-based approach, both of which are very power-hungry operations. In terms of hardware overhead, switch-level area is not explicitly provided by the paper, however, this multicast SMART NoC uses almost the same monitoring network as the original unicast SMART NoC [104], where each monitoring link is wide (2-4 bits) and a large number of such links (24) emanating from each router (see Section 5.1). As a result, the monitoring network requires dozens of extra wires. In contrast, the proposed asynchronous NoC targets the precise destinations in parallel, without packet dropping or unicast, and entirely avoids the use of an extra monitoring network. Therefore, while the SMART NoC achieves better latency than the proposed NoC, it is expected to incur significant overheads for other costs.

(iii) Reduced-overhead SMART multicast NoC. As a final analytical comparison, we hypothesize potential future research from the synchronous NoC community, where the multicast SMART NoC [105] is combined with the reduced-overhead techniques proposed for SMART NoCs [31]. The design for this hypothetical NoC does not exist, but an extrapolated comparison is being performed in terms of network latency. This NoC can be imagined as an extension of a recent unicast-only SMART NoC but with an optimized low-overhead monitoring network [31], to perform multicast using the above SMART multicast approach. Such an extension, however, is non-trivial. Although the unicast reduced-overhead SMART NoC achieved $8\times$ reduction in the monitoring

network wiring overhead of the original unicast SMART NoC [104], it still leaves extra wiring compared to the proposed asynchronous NoC that does not use monitoring network. Since the unicast reduced-overhead SMART NoC had similar latency as the original unicast SMART NoC, we extrapolate that a future solution of the multicast reduced-overhead SMART NoC will also have a similar latency as the multicast SMART NoC. Therefore, such an extrapolated multicast NoC might achieve better network latency than the proposed asynchronous NoC, but is still expected to incur significant energy/power overheads due to the limitations of the SMART multicast protocol as highlighted in the above paragraph.

Finally, although the synchronous routers for the above NoCs use VCs, while the asynchronous router does not, the latter can be extended to include VCs without much latency overhead. An existing work added VCs to a unicast-only asynchronous router using a simple approach with minimal overheads [89], as also described in Section 5.2.2. A similar approach can be used to extend the proposed multicast-enabled asynchronous router to include VCs as well. These projected asynchronous routers will use a multi-switch architecture with replicated crossbars, which has been shown to be more efficient than the shared crossbar architecture for asynchronous designs [135]. An advantage of the multi-switch architecture is minimal latency overhead for VC management (simple addition of a demux and an arbiter on the forward datapath of the non-VC design) [89]. In particular, a head-to-head comparison of the baseline asynchronous router with a leading synchronous router from AMD, both with 2 VCs, implemented in 14 nm technology, showed that the asynchronous router achieved 55% lower latency and 28% smaller area. Similar improvements were also estimated for this asynchronous router with 8 VCs. Therefore, overall, the new asynchronous NoC also, after addition of VCs, is expected to have minimal overheads in terms of latency.

5.7 Conclusions

This chapter proposes a new parallel multicast asynchronous NoC with 2D-mesh topology. A novel *continuous-time replication strategy* is proposed, where the flits of a multicast packet are routed through the distinct outputs of the router according to each output's own rate, in parallel and in continuous time. A new *continuous-time multi-way read (CMR) buffer* is also proposed to enable this strategy. For multicast benchmarks, the new parallel multicast network achieved significantly better network latency and saturation throughput over a serial baseline, with reductions in energy for benchmarks with a high multicast portion. In addition, in an analytical comparison with three synchronous multicast NoCs, the proposed asynchronous NoC is expected to show some estimated cost benefits. While a thorough evaluation has been performed using synthetic benchmarks, that contain common traffic patterns from cache coherence and neural networks, this work will be extended to target real applications. Moreover, future work also includes using the proposed NoC to build a GALS system and performing system-level evaluation, and supporting many-to-1 traffic more efficiently.

Chapter 6

Synthesizing Asynchronous NoCs on FPGAs: a Systematic Methodology

6.1 Introduction

While the previous chapters focused on only pre-layout implementation of asynchronous NoCs, this chapter targets a complete physical design on FPGAs for a more realistic evaluation. Recent advancements in FPGA technology have led to a wide-ranging use of these devices for not only rapid prototyping but also being deployed for applications such as hardware-assisted acceleration of machine learning algorithms. However, there has been only very limited research on how to implement asynchronous NoCs on modern commercial FPGAs. This chapter takes on this challenge and makes an important advancement in the field of asynchronous NoCs.

A systematic CAD methodology is proposed for mapping asynchronous NoCs on FPGAs. The target asynchronous NoCs use bundled-data design style instead of quasi-delay-insensitive style, where the former may lead to more efficient design but involves modest timing constraints for correct operation [147]. For bundled-data NoCs, the proposed methodology targets a two-fold challenging goal: the final implementation on FPGA should achieve both high performance, and must also satisfy all timing constraints, which includes bundling constraints in the datapath as well as any relative timing constraints in the control path. This approach uses a semi-automated tool flow, where automation is used to achieve high-performance mapping, while checking that all timing constraints are satisfied, is performed manually. The latter manual operations can also be easily automated. In addition, the tool flow only uses the existing *Xilinx Vivado tool set,* leading to ease of implementation and convenience for the designers. *To the best of our knowledge, this is the first systematic methodology for efficiently mapping asynchronous NoCs on commercial FPGAs.*

Asynchronous NoCs also use some special asynchronous components, which are not used in synchronous designs, and must be correctly mapped on FPGAs. These components are: the C-element, used for storage, and the mutex for arbitration. This work introduces a comprehensive guide on how to efficiently and safely map these elements on FPGAs. Only the standard Xilinx Vivado tool is used for mapping, combined with some small manual interventions, which may be needed to meet the timing requirements for some of these components. Moreover, the latter can be easily automated using simple scripts.

There are unique challenges with mapping these special components on FPGAs, which are addressed. The C-element is an asynchronous state machine, which must be implemented in a glitch-free manner. To this end, not just a robust but also a highly-efficient implementation of the C-element was achieved. On the other hand, the mutex is a continuous-time arbiter, which is implemented using analog circuits. However, since no such implementation is possible for FPGAs, an efficient digital standard-cell mutex is used for synthesis on FPGAs, and its final implementation was stress tested to ensure its reliable operation. Finally, a high-performance 4-input arbiter, which is a critical component of asynchronous NoCs and built using the mutex and the C-element, was also synthesized and validated on FPGAs.

Two bundled-data asynchronous NoC switches were synthesized using the proposed tool flow on a commercial FPGA, to demonstrate not only the effectiveness of the flow, but also significantly advancing the field of asynchronous NoCs. Both asynchronous switches are highly-efficient: one performs only unicast [66] and has been used as an effective baseline throughout this thesis, while the other also supports lightweight multicast [17], which was proposed in Chapter 5. The implementations of these switches were compared to a high-performance single-cycle synchronous switch that only supports unicast, and has been used in accelerator-based systems for image processing applications [121]. The target FPGA is *Xilinx Virtex 7* in *28 nm*, which has been commercially deployed in line cards as well as in portable RADAR systems.

6.2 Implementing Asynchronous Circuits on FPGAs: Related Work

As an alternative to commercial synchronous FPGAs, custom asynchronous FPGAs have been developed to implement asynchronous circuits, which, however, are not mainstream and therefore not the focus of this work. One of the earliest works, from 1992, is the *MONTAGE asynchronous FPGA,* which had custom functional units and routing structures that could handle both bundled-data and QDI circuits [75]. The early 2000s also saw some interesting asynchronous FPGAs, that had 'island-style' architectures [199], [183], [82], which is commonly used in the commercial synchronous FPGAs. While some of these works could support both bundled-data and QDI circuits [82], others were only able to handle QDI circuits [199], [183]. Although there has been considerable research on developing asynchronous FPGAs, these devices are not mainstream like the commercial synchronous FPGAs from Xilinx and Altera, which are seeing a major interest for several applications from Microsoft, Amazon, etc. (see Section 1.5.2). Given the increasing importance of the synchronous FPGAs, there is a need for a systematic tool flow to map asynchronous NOCs on these devices. Therefore, this work targets these commercial FPGAs only.

Early works have a narrow focus and only target synthesizing small asynchronous components and designs on FPGAs. Some works focus on special components, such as a C-element [77], or a mutual exclusion element [137]. The former only presents one possible implementation of the C-element, which uses 4 gates. In contrast, the current work performs thorough exploration of various implementations and selects the most efficient and robust one. The latter mutex implementation is very complex, consisting of 4 FFs and 2 gates, as opposed to a much more efficient mutex implementation in the current work. Other works have targeted small designs, such as a simple pausable clock generator [64].

There has been only limited research on synthesizing asynchronous/GALS NoCs on commercial FPGAs. Earlier works target a quasi-delay insensitive (QDI) design style with one simple timing assumption that all wire forks must be isochronic [194], [157], while recent ones use bundled-data [110], [97]. The former NoCs use delay-insensitive data encoding [147], and therefore do not have any timing constraints but can incur large area/power overheads. These NoCs also use expensive FF- or latch-based mutex implementations. The latter bundled-data NoCs rely on non-trivial timing constraints, but an implementation methodology is largely missing.

Recently, two CAD flows have been proposed to map bundled-data circuits on FPGAs [180],

[207], which, however, do not target NoCs. Both flows have several limitations: (i) they do not address how to handle the special asynchronous components of NoCs; (ii) they focus only on correctness, while performance optimization, which is critical, is not targeted, in contrast, to the proposed approach; (iii) one approach requires a set of six specialized and custom tools, which are used in addition to the standard FPGA synthesis tool [180]. A preferable alternative is to only use the FPGA synthesis tool, as in the proposed work, exploiting the recent advances; and (iv) the second approach focuses only on *click-based asynchronous pipelines,* that employ high-overhead FF-based datapaths and control to simplify automation [207]. However, this flow may not be used for the majority of more efficient bundled-data NoCs based on lightweight Mousetrap pipelines and use normally-transparent single-latch registers with different timing constraints [83], [89].

Given the limitations of the above approaches, there is a need for a comprehensive as well as a systematic CAD methodology to map bundled-data asynchronous NoCs on FPGAs. This methodology should not only target correctness but also achieve a high-performance implementation. Moreover, the special asynchronous elements, which form the critical components of these NoCs, must also be mapped in an efficient and safe manner.

6.3 Mousetrap Pipeline and Timing Requirements of Bundled-Data Circuits: A Brief Background

A brief recap on Mousetrap pipelines is presented, as these pipelines are the simplest, yet an important example of bundled-data circuits, and they also form the basis of the asynchronous NoC routers in this chapter. The Mousetrap pipeline will be used in the next section to demonstrate the proposed CAD methodology for FPGAs. In addition, the timing requirements associated with bundled-data circuits, and in particular, the Mousetrap pipelines are also revisited, which must be satisfied when mapping these circuits on FPGAs.

6.3.1 Mousetrap Pipeline

As discussed in Section 2.4 of Chapter 2, Mousetrap is a high-performance pipeline that uses a two-phase handshaking protocol and single-rail bundled data encoding. Figure 6.1 shows a 3-stage Mousetrap pipeline. Its operation is based on a capture-pass protocol, using normally-transparent

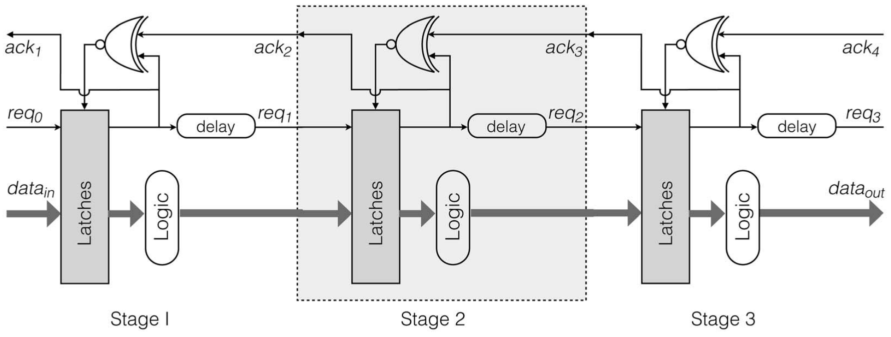

Figure 6.1: A 3-stage Mousetrap pipeline

latches, as covered in more details in Section 2.4.

6.3.2 Timing Requirements of Bundled-Data Circuits

Satisfying all timing constraints is an important step for mapping bundled-data circuits correctly on FPGAs. As discussed before in Section 2.4.3 of Chapter 2, to ensure correct operation of the bundled-data circuits, two types of timing constraints must be satisfied: (i) *bundling constraint in datapath: req* must transition after data is stable, and (ii) *relative timing constraints (or RTCs)* in control: delay across one path should be less or greater than the other. Mousetrap exhibits both bundling constraints as well as relative timing constraints such as data protection.

6.4 A CAD Methodology for Bundled-Data Asynchronous Circuits

A systematic methodology is proposed to synthesize bundled-data asynchronous circuits on commercial FPGAs. This approach largely uses the standard synthesis tool flow for both mapping as well as validation of the final implementation. The tool flow for implementation and validation is presented, followed by illustrating the methodology using an example of a Mousetrap pipeline.

6.4.1 Tool Flow

The basic strategy of the tool flow is first presented, followed by the detailed steps involved in the flow.

6.4.1.1 Basic strategy of the tool flow

The tool flow not only targets a high-performance mapping of a bundled-data circuit but also makes sure that it is robust. As a result, there are two phases in the flow: one that achieves high performance, and the other that makes sure all timing constraints are satisfied. In this flow, a design is first mapped focusing only on maximizing the performance, entirely ignoring the robustness aspect, in a *performance-oriented mapping stage.* Next, the flow enters a *robustness-oriented mapping stage,* which takes a set of bundling and relative timing constraints, supplied by the user for the input design, and checks to make sure these constraints are satisfied. If a subset of these constraints are not satisfied, then small incremental delay insertions are performed in the initial design on the offending paths of each of these constraints, thereby concluding this stage.

The updated design is then re-synthesized using the performance-oriented stage, followed by re-checking the implementation for timing constraints. The two stages are iterated until all timing constraints are satisfied, and the result is an efficient and safe mapping on the FPGA. More details are presented below.

Performance-oriented mapping stage. In the performance-oriented mapping stage, the synthesis tool tries to find the implementation with the best performance by executing the standard synthesis steps under some performance constraints.

This stage takes the gate-level RTL of a design as input. In this design, first, all the critical control and datapaths that govern latency are determined, in a *critical path determination step.* For example, in Mousetrap, such paths include the forward request paths and the forward datapaths.

Next, performance constraints are imposed on these critical paths in the form of max delay constraints - this step is referred to as *performance constraints application step.* The max delay values for these paths are intuitively decided based on the gates involved: these values are picked to be somewhat relaxed for these paths, and not overly aggressive.

However, a problem arises, as max delay constraints can only be applied in a synchronous setting, i.e. between clocked registers. For asynchronous designs, this limitation is overcome by using hypothetical clock boundaries. In particular, enable signals of the registers in the asynchronous design can declared as 'fake' clocks. Such declarations are performed only for the registers that are on the critical paths, in a step called *fake clock declarations.*

The gate-level RTL, the set of max delay constraints, and the fake clock declarations are then used in the *implementation step.* This step only uses the FPGA synthesis tool to perform the following automated steps: logic synthesis, placement and routing.

The resulting implementation is then checked to see if all the max delay constraints are satisfied in the *performance constraints checking step.* There can be two outcomes: (i) all performance constraints are satisfied or (ii) some performance constraints are not satisfied. In the former case, the tool flow tries to find a better mapping with a higher performance: if possible, the performance constraints are tightened using smaller max delays (*tighten max delay constraints step*), followed by re-implementation under the new constraints. In the latter case, the max delay constraints are relaxed to allow the tool to meet timing in the *relax max constraints step,* and then the design is re-implemented. After each re-implementation, the performance constraints checking is again performed, and based on the outcome, max constraints are relaxed or tightened. This complete mapping stage, therefore, follows an iterative process to get a high-performance implementation on FPGAs, where all the max delay constraints satisfied and no further slack optimization is possible.

Robustness-oriented mapping stage. The high-performance implementation from the previous stage may not satisfy all the timing constraints, and therefore a robustness-oriented stage is required to make sure all these constraints are satisfied.

First, as a pre-processing step, all the bundling and relative timing constraints are determined in the initial gate-level RTL design in a *timing constraints enumeration step.* The next steps are then performed on the implementation obtained from the previous stage to check and make sure all these constraints are satisfied.

A *path delay extraction step* is first used on the previous implementation, where the delays across all the paths involved in all the timing constraints are extracted using the synthesis tool.

These path delays are then used to check if all the timing constraints are satisfied by a safe margin of more than 300 ps in a *timing constraints checking step*. If all the constraints are met then no further action is required and the final high-performance and robust implementation is obtained.

However, if a subset of constraints are not satisfied by a margin more than 300 ps, then an additional step, called *adding delay* is used, where the offending paths of these constraint are slowed down by adding localized delay lines in the initial gate-level RTL, keeping the rest of the design unmodified. These delay values are picked intuitively based on the differences by which the constraints were not satisfied.

The updated RTL is then re-synthesized using the performance-oriented stage, followed by again checking if all the timing constraints are satisfied in the robustness-oriented stage. The two stages are repeated until all timing constraints are satisfied, and a high-performance and a safe mapping is obtained on the FPGA.

6.4.1.2 Tool flow details

Figure 6.2 shows the steps involved in this tool flow using the Xilinx Vivado tool set. The performance-oriented stage uses an automated flow, while the robustness-oriented stage is performed manually. Although the latter is currently done manually, this flow can be automated in the future.

Performance-oriented mapping stage. The steps involved in the performance-oriented mapping stage are shown as gray boxes in the Figure 6.2. In this stage, the Xilinx Vivado tool takes the gate-level RTL model of a hazard-free bundled-data design as the input. This RTL model also contains *DONT_TOUCH* attributes to disable any logic manipulations that can introduce hazards. In Step 1a, *critical paths determination* is performed to get all critical control and datapaths of the design. In Step 1b of *performance constraints application step,* max delay constraints are applied to all critical paths using *set_max_delay*. In parallel, Step 1c (*fake clock declarations*) is performed, where the enable signals of all registers on the critical paths are declared as 'fake' clocks using the *create_clock* construct. Next, the *implementation Step 1d* takes the RTL, the max delay constraints, and the fake clock declarations to perform the following: (i) logic synthesis that maps the design to the appropriate LUTs and FLOPs, (ii) placement that places these cells on to the FPGA, and (iii) routing interconnect wires between these cells. The resulting implementation is then checked to see if all the max delay constraints are satisfied in the Step 1e of *performance constraints checking.* If so, then in Step 1f, if possible, these constraints are further tightened to get a better performance, called the *tighten max delay constraints.* The design is re-implemented under these new constraints using Step 1d, followed by rechecking in Step 1e. However, if in Step 1e, some of these constraints are not satisfied, then Step 1g of *relaxing max delay constraints* is used, and the design is again implemented, then rechecked in Step 1e. These steps are iterated until the best high-performance mapping is obtained with all the performance constraints satisfied.

Robustness-oriented mapping stage. The steps involved in the robustness-oriented stage are shown as blue boxes in the Figure 6.2. This stage takes the performance-oriented mapping from the

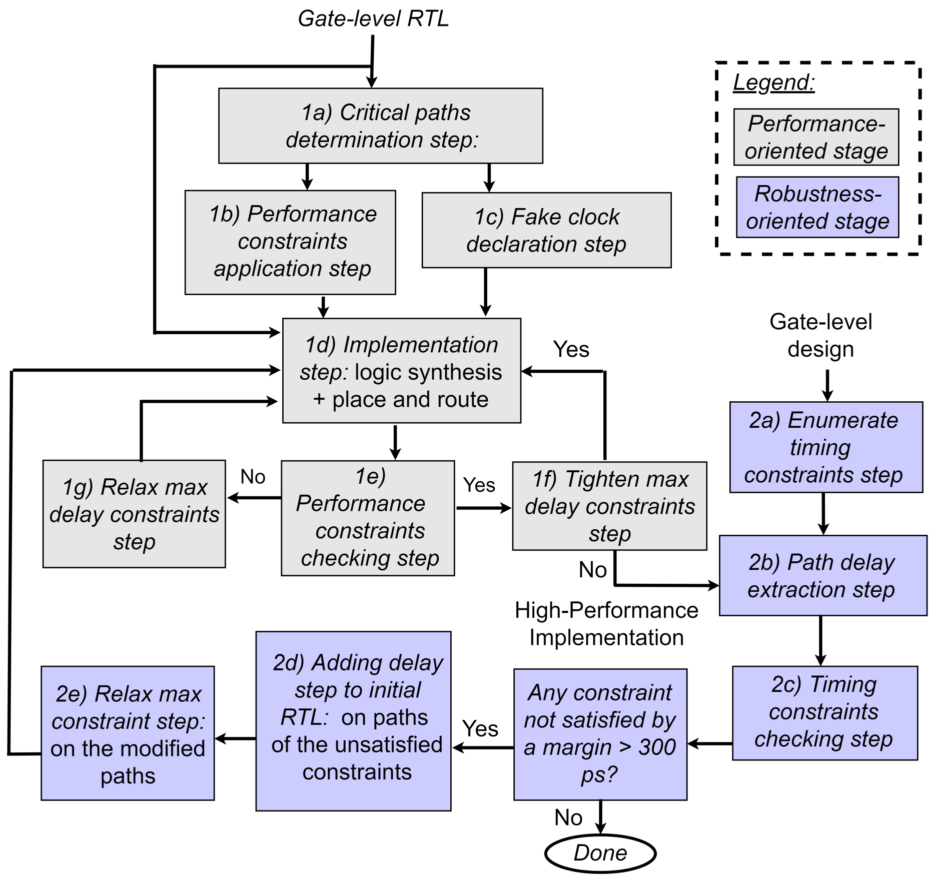
Gate-level RTL
Legend:
Performance-oriented stage
Robustness-oriented stage
1a) Critical paths determination step:
1b) Performance constraints application step
1c) Fake clock declaration step
1d) Implementation step: logic synthesis + place and route
Yes
Gate-level design
2a) Enumerate timing constraints step
1g) Relax max delay constraints step
No
1e) Performance constraints checking step
Yes
1f) Tighten max delay constraints step
No
2b) Path delay extraction step
High-Performance Implementation
2e) Relax max constraint step: on the modified paths
2d) Adding delay step to initial RTL: on paths of the unsatisfied constraints
Yes
Any constraint not satisfied by a margin > 300 ps?
2c) Timing constraints checking step
No
Done

Figure 6.2: Tool flow for implementing bundled-data asynchronous circuits on FPGAs

previous stage and follows these steps in Vivado to make sure all the timing constraints are satisfied. In the first pre-processing Step 2a, all the bundling and relative timing constraints are determined and enumerated in the initial gate-level RTL design. Next, in Step 2b of *path delay extraction*, the delays of the paths, involved in all the timing constraints, in the implementation from the previous stage are extracted. This delay extraction is performed using *get_timing_paths*. Using these path delays, all the timing constraints are then checked to see if they are satisfied by more than 300 ps in Step 2c of *timing constraints checking*.[1] If so, then the procedure terminates. If, however, a constraint is not satisfied, then Step 2d of *adding delay* is used to enable meeting of this constraint, where a delay line, comprising buffer LUTs, is added to the offending path in the initial RTL model.

[1]For a Virtex 7, fabricated in 28 nm, 300 ps is equivalent to adding a buffer LUT delay, and can be considered a safe margin.

Since, an extra delay is added, max delay constraints for these paths are slightly relaxed in Step 2d of *relax max delay constraints,* followed by re-implementing the design using the performance-oriented stage, and rechecking the timing constraints in the robustness-oriented stage. The two stages are iterated until all the timing constraints are satisfied, yielding a final high-performance and robust implementation on the FPGA.

6.4.2 Validation Approach

The implemented asynchronous design is validated using a synchronous test wrapper, exploiting the existing Xilinx Vivado validation tools. This wrapper involves both injecting test inputs to the asynchronous design-under-test (DUT) as well as monitoring its internal signals and outputs. A hierarchical validation approach is used where the DUT is implemented in isolation first, followed by locking its implementation on FPGA (using *lock_design*) and then interfacing with the test wrapper. An advantage of this approach is that it isolates the DUT from the test circuits, so any change to the synchronous test wrapper will not impact the DUT. Validation is performed at two levels: (i) post place-and-route back-annotated timing simulation, and (ii) emulation of the the implemented hardware on FPGA, where the synchronous wrapper uses a *Xilinx integrated logic analyzer (ILA) hardware core* to monitor the DUT's internal wires and outputs during emulation.

6.4.3 Tool Flow Illustration: A Mousetrap Pipeline

A Mousetrap pipeline is used to illustrate the CAD methodology presented in Section 6.4.1, as it is a simple example of bundled-data circuits, as well as forms the basis of the routers. As described earlier, the tool flow iteratively goes through two stages to find the best mapping on FPGAs: performance-oriented stage and robustness-oriented stage, both of which are now illustrated using the Mousetrap example.

Performance-oriented mapping stage. Figure 6.3 shows how the performance-oriented stage is used to synthesize a Mousetrap pipeline. A gate-level RTL model of the Mousetrap pipeline is input to this stage. This model describes the behavior of each of the components of the pipeline: the XNOR gates and the latch-based registers, along with the *DONT_TOUCH* primitives. In Step 1a, the forward critical control and datapaths, responsible for latency of the design are determined as shown in the Figure. Next, in Step 1b, max delay constraints are applied on all these critical

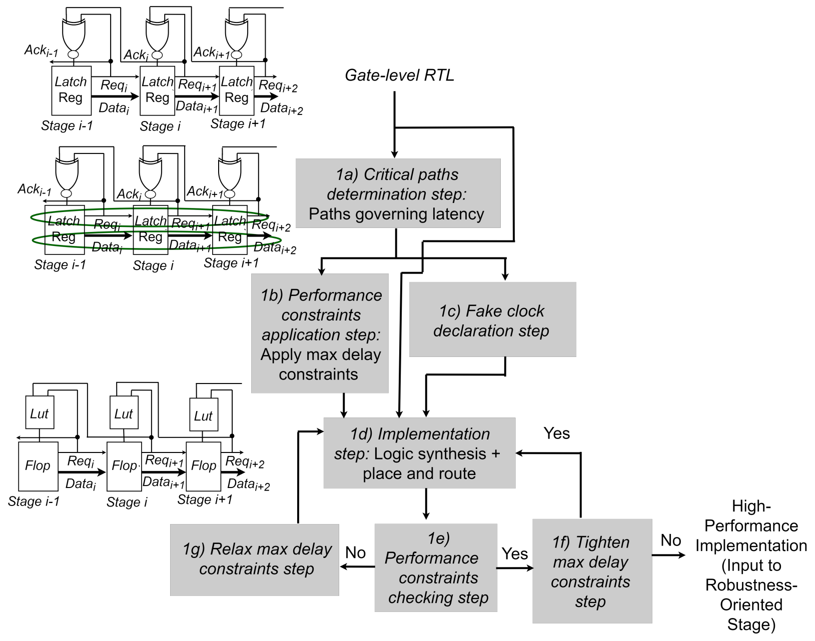
Gate-level RTL
1a) Critical paths determination step: Paths governing latency
1b) Performance constraints application step: Apply max delay constraints
1c) Fake clock declaration step
1d) Implementation step: Logic synthesis + place and route
1e) Performance constraints checking step
1f) Tighten max delay constraints step
1g) Relax max delay constraints step
Yes
No
High-Performance Implementation (Input to Robustness-Oriented Stage)
Latch Reg
Flop
Lut
Stage i-1
Stage i
Stage i+1

Figure 6.3: Illustrating the proposed tool flow of performance-oriented stage on a Mousetrap pipeline example

paths. The figure shows an example of max delay constraints for critical req/data paths between stage i and stage $i+1$. In parallel, in Step 1c, register enables of all the latch registers on the critical paths (en_{i-1}, en_i, en_{i+1}) are declared as fake clocks. In Step 1d, gate-level RTL, max delay constraints, and fake clock declarations, are then used to implement the Mousetrap design following the standard synthesis steps: logic synthesis, which maps the registers to FLOPs that implement the latch functionality and maps the XNORs to 2-input LUTs that performs the XNOR operation, followed by place and route. The iterative procedure (involving steps 1d, 1e, 1f and 1g) is then followed to tighten the max delay constraints if possible, and re-implementing the design, until no further tightening is possible, and a high-performance mapping of the Mousetrap pipeline is obtained with all the max delay constraints satisfied.

Robustness-oriented mapping stage. Figure 6.4 shows the robustness-oriented mapping stage. This stage takes the performance-oriented implementation of the Mousetrap, and manually checks and makes sure all timing constraints are satisfied. First, as a pre-processing step, all the bundling timing constraints and relative timing constraints, such as data protection, in the initial gate-level RTL design of Mousetrap are determined and enumerated in Step 2a. The Step 2b takes the high-

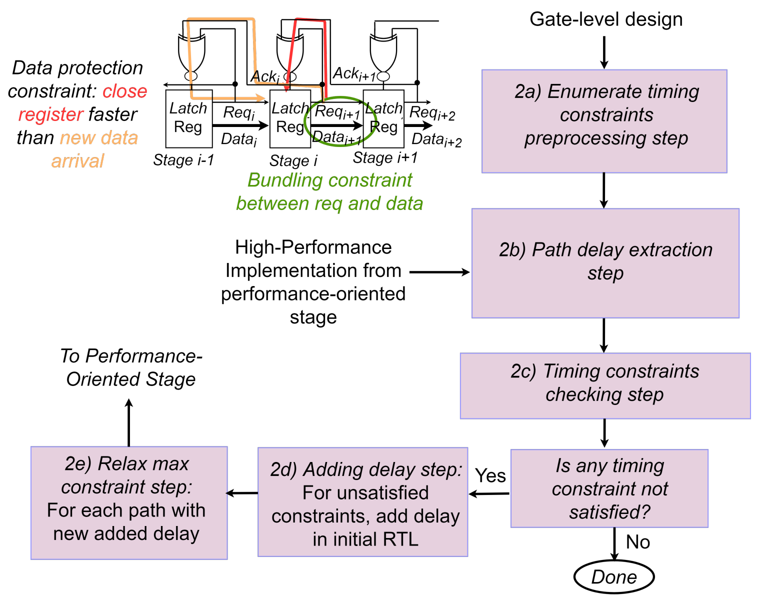
Data protection constraint: close register faster than new data arrival
Latch Reg
Req$_i$
Data$_i$
Ack$_i$
Ack$_{i+1}$
Req$_{i+1}$
Data$_{i+1}$
Req$_{i+2}$
Data$_{i+2}$
Stage i-1
Stage i
Stage i+1
Bundling constraint between req and data
Gate-level design
2a) Enumerate timing constraints preprocessing step
High-Performance Implementation from performance-oriented stage
2b) Path delay extraction step
2c) Timing constraints checking step
Is any timing constraint not satisfied?
Yes
No
Done
2d) Adding delay step: For unsatisfied constraints, add delay in initial RTL
2e) Relax max constraint step: For each path with new added delay
To Performance-Oriented Stage

Figure 6.4: Illustrating the proposed tool flow of robustness-oriented stage on a Mousetrap pipeline example

performance Mousetrap implementation from the previous stage as input, and the delays across the various timing paths involved in all the timing constraints are extracted in Vivado. For example, the datapath delays between any two registers, delays of the req paths between any two registers, delays across each of the the backward acknowledge paths, and delays across the self-closing paths of the different latch registers. Next, in Step 2c, all these path delays are used to check if all the bundling constraints, and relative timing constraints are satisfied by a margin more than 300 ps. If a constraint is not satisfied, for example, a bundling constraint between stage i and stage $i+1$, then in Step 2d, a matched delay line, comprising appropriate buffer LUTs, must be added to the req path in the initial RTL, as shown in the figure. Similarly, all the other timing constraints are checked, and delay is added if needed. In Step 2e, the max delay constraints across the paths that involved adding delay are relaxed slightly, and the design is then re-implemented using the performance-oriented stage.

The two stages are iterated until the final high-performance mapping with all the timing constraints satisfied is achieved.

6.5 Synthesis of Special Asynchronous Components on FPGAs

Implementation of the special asynchronous elements on FPGAs is now presented, which form the critical components of asynchronous NoCs: the C-element, the mutex, and a 4-input arbiter. These elements are implemented largely following the standard FPGA synthesis tool flow.

6.5.1 C-Element

A C-element is an asynchronous state-holding component, discussed in detail in Section 2.3.1 of Chapter 2. Synthesizing a C-element on FPGAs, both efficiently and safely, is a challenging task with only limited research that has tackled this problem. In this work, three different standard-cell designs of the C-element are explored for mapping on to FPGAs, and the best design in terms of performance, resource utilization, and robustness is selected.

Figure 6.5 shows the three designs for a C-element: using AND/OR gates, AOI222 complex gate, and a D-latch based design. Since, the C-element is an asynchronous state machine, the two combinational designs require the output of the C-element to be fed back to provide state information. In contrast, the latch-based design does not require any feedback, but it can be very slow. However, the latch-based design is more robust than the former combinational ones as these require an extra timing constraint for glitch-free operation: after any change in the output, the feedback input must arrive before the primary inputs change.

All three designs are synthesized on the FPGA using the standard FPGA synthesis tool flow. For each design, a gate-level RTL model is the input to the tool, which is then used to perform the following automated steps: logic synthesis, placement, and routing to get the final physical design on the FPGA. In more details, the input RTL model also contains *DONT_TOUCH* directives to disable any logic manipulations that can introduce hazards. Xilinx Vivado tool is used for implementation, which takes the RTL model as input, and first performs logic synthesis, mapping the gates/latches to LUTs or FLOPs, respectively. The placement step is then performed, which places these cells on the FPGAs, followed by the routing step, which routes the appropriate interconnections between these cells.

Figure 6.5 shows the FPGA mappings obtained for each of the three C-element designs, which are analyzed in terms of resource utilization, performance and robustness to select the best one. The

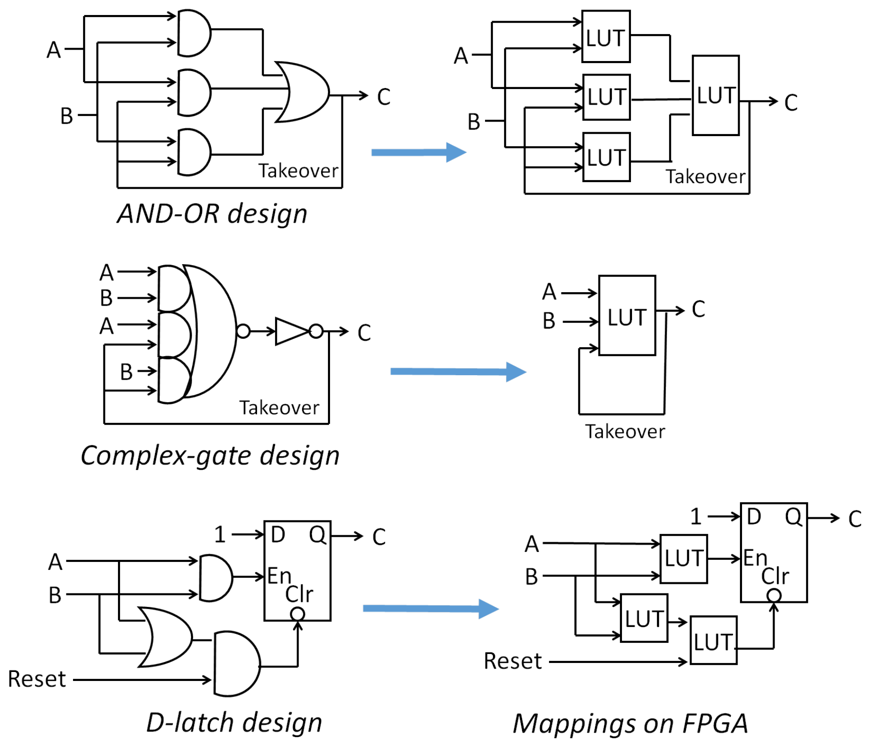

Figure 6.5: C-element: Standard-cell designs and FPGA mappings

FPGA considered is Xilinx Virtex 7 in 28 nm. As shown in the figure, the AND-OR design leads to a mapping with 4 LUTs and has a post place-and-route latency of 434 ps. The complex-gate design, on the other hand, only uses 1 LUT and has a latency of just 43 ps due to the absence of any other LUT or wire delays. The latch-based design is the most expensive, and uses 3 LUTs and 1 FLOP with a latency of 778 ps, which is large due to the FLOP on the critical path that has a delay of around 330 ps. Therefore, in terms of latency and resource utilization, the complex-gate design that leads to a 1-LUT mapping is the best. This design is also very robust as the feedback wire delay from output to input is small (159 ps), and will arrive significantly faster than the new primary inputs after any change in the output, leading to a glitch-free operation. Given these results, the 1-LUT mapping is picked out of the three implementations.

Each of the three final implementations of the C-element were exhaustively validated to make sure they operate correctly. The validation was performed using a synchronous test wrapper, following the similar methodology as in Section 6.4.2, at two levels: (i) post place-and-route back-annotated timing simulation, and (ii) emulation of the using the integrated logic analyzer (ILA). All

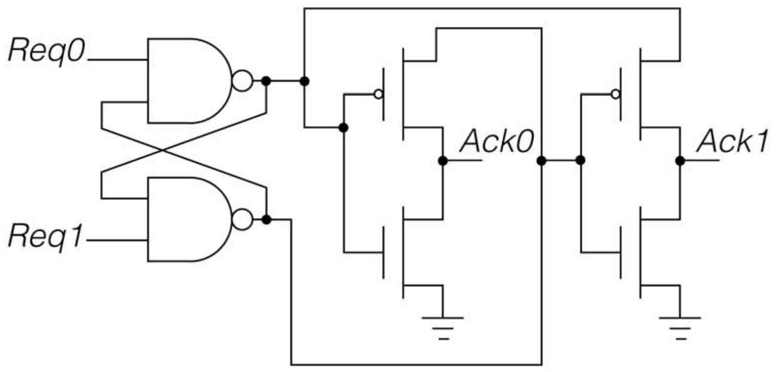

Figure 6.6: The analog mutex

three implementations of the C-element were validated at both these levels using a wide-range of inputs.

6.5.2 Mutex

Unlike the synchronous arbiters, a mutex performs arbitration between two requests in continuous time, and therefore, faces unique challenges for its correct and efficient implementation on FPGAs. While a synchronous arbiter arbitrates based on input arrival order per discrete clock cycles, an asynchronous mutex must resolve arrival order in continuous time. The correct mutex is implemented using analog circuits [147], as shown in Figure 6.6, and also discussed in Section 2.3.2 of Chapter 2. This mutex has two stages: an arbiter stage that mediates between the two requests using cross-coupled NANDs and an analog filter. The filter keeps the grant outputs low until the arbitration is resolved, and cleanly asserts exactly one grant after the arbitration is complete. However, since an analog implementation is not possible on FPGAs, a digital mutex, with some inherent mean time between failure (MTBF), must be used for mapping. The next sub-sections present the structure/operation of this digital mutex, any requirements for its correct and high-performance operation, followed by details on the mapping approach, and exhaustive validation to ensure its reliable operation.

6.5.2.1 Structure and operation of the digital mutex

As shown in the Figure 6.7, the digital mutex, similar to the analog mutex, also has two stages: an arbiter, and a filter [68]. The arbiter stage performs arbitration between the two incoming requests, designed using cross-coupled NAND gates. The filter stage, on the other hand, is used to keep both the grant outputs deasserted while the arbitration is being resolved, and only after the arbitration is complete, it allows exactly one of the grant outputs to be cleanly asserted high. The filter stage is designed using digital AND gates, each of which has two inputs: an intermediate grant request (*mid0/mid1* through the inversion), and a filter enable (*mid0/mid1* without inversion). If a filter AND is enabled, then the corresponding grant output is low, otherwise, the grant output is an inversion of the intermediate grant request.

There can be three important operating scenarios for the mutex: no contention with only one request arriving, basic contention with one request arriving well before the other, and extreme contention, where both requests arrive very close to each other. The digital mutex, like the analog one, uses a four-phase handshaking protocol.

In a simple case of no contention, during the set phase, an assertion on a single *req* causes the corresponding mid output of the arbiter level to be asserted low, which was initially high, followed by assertion on the grant output of the filter stage. In the reset phase, deassertion of the *req* deasserts the mid wire to high, leading to deassertion of the grant.

During basic contention, assuming *req0* arrives well before *req1, mid0* output of the arbiter level is asserted low first, followed by three operations in parallel: blocking of *req1* (keeping mid1 deasserted high) in the arbiter level, asserting grant0 output of the filter, as well as masking of mid1 in the filter level, i.e., keeping grant1 low. After the *req0* releases the mutex, *req1* will be granted next.

In the extreme contention case, when both requests arrive very close to each other, the arbiter stage can become metastable, where the internal mid values can exhibit either oscillations or intermediate voltage levels. As an example of the oscillations, after the arrival of the requests, the outputs of the arbiter stage will both go low. As a result, both 'A' and 'B' inputs will go low, flipping the outputs of the NANDs, making them high again. This cycle continues leading to oscillations. During this metastability, the filter safely keeps the grant outputs low. The metastability will be resolved eventually, with one request blocking the other, followed by a clean assertion on exactly one of the

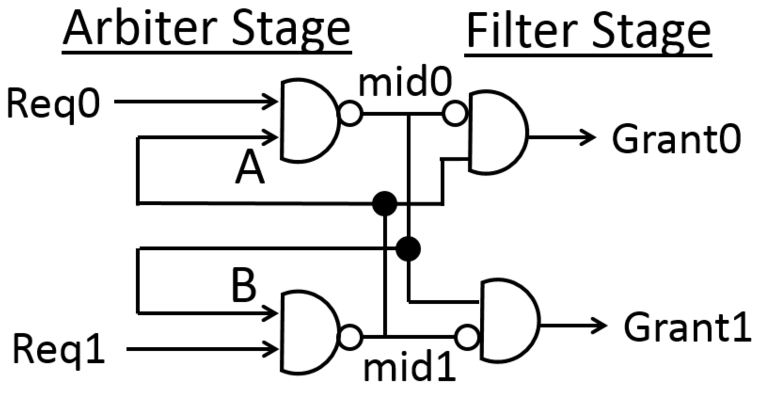

Figure 6.7: The digital standard-cell mutex

grant outputs.

6.5.2.2 Requirements for correct and high-performance operation of the digital mutex

For the digital mutex to operate correctly as well as achieve high performance in case of contention, two issues must be addressed. First, for *high performance,* a request must be able to quickly block a competing request to achieve fast arbitration resolution time. Second, for *correctness,* while the arbitration is being performed, the digital filter stage must keep both grant outputs low, and assert exactly one of the outputs only after the arbitration has been resolved. If the filter stage is not effective then it can lead to unwanted glitches on the outputs of the mutex.

The focus for achieving high performance is on the arbiter stage. A simple timing requirement must be satisfied in the arbiter stage to enable quick blocking of a competing request, leading to fast resolution time. In particular, the wire delays of the two feedback wires, i.e. *mid0* to *A* and *mid1* to *B,* must be small. If these wire delays are large, and assuming the two requests arrive close to each other, where *req0* arrives slightly before *req1,* then *mid0* is asserted low first but due to the long wire delay, *req1* is not blocked on time, resulting in assertion of mid1, potentially leading to metastability. On the other hand, if the two feedback wires have very small delays, then the assertion on *mid0* can quickly block *req1,* which can either lead to a runt pulse on mid1 or keeps mid1 at deasserted high, potentially leading to a very fast resolution of arbitration in favor of *req0.*

The focus of the correctness issue is on the filter stage. Contrary to the analog mutex, where the filter stage performs a stable masking of any internal glitches while resolving arbitration, the filter stage of the digital mutex is unstable and can get transiently disabled during the internal oscillations. Therefore, to improve its filtering capability, an electrical intervention in the form of large inertial delay is required at the inputs of the filter stage. To this end, large input capacitances can be used. For example, if two requests arrive close to each other, oscillations can occur on *mid0* and *mid1,* and while the arbitration is being resolved, *mid0* going low blocks mid1 at the filter stage from going to the grant1 output, and vice versa. However, *mid0* can become deasserted high shortly after, removing the blocking on *mid1,* which can then allow the value of mid1 to get out on grant1, even before the arbitration has resolved. To enable the filter stage to perform consistent and robust filtering, the controlling inputs of its AND gates (inputs with the inversions) must have high input capacitance, which can block internal fast oscillations or any intermediate voltage levels, that can occur while arbitration is being resolved, hence keeping the grant outputs safely low during this time.

6.5.2.3 Mapping of digital mutex on FPGAs

The above requirements on correctness and high performance are used to guide the mapping of the mutex on FPGAs. These requirements can be simply met by manually forcing the placement of its components on certain locations. First, each gate of the mutex must be mapped to a LUT. For fast arbitration resolution, both the arbiter LUTs should be placed symmetrically in the same CLB such that the wire delays from *mid1-A* and *mid0-B* will be small. For correctness, the filter stage must have a high input capacitance. Since, no such high-capacitive elements exist on FPGAs, long wires can be used between the arbiter stage and the filter as such wires can exhibit high capacitance, both due to the wire length as well as the presence of buffers [113], [35]. Hence, the LUTs of the filter stage must be placed in the CLB next to the arbiter CLB.

The implementation procedure for the digital mutex largely uses the standard synthesis flow. In more details, the Xilinx Vivado tool is used, which takes two inputs: (i) a gate-level RTL model with *DONT_TOUCH* directives forcing each gate to be mapped to a LUT, and (ii) a set of constraints to manually force these LUTs to be placed at the above discussed locations on the FPGA. The tool then performs the following automated synthesis steps: logic synthesis for mapping to LUTs, followed by placement that uses the above location constraints to place the LUTs, and finally routing the interconnection wires between these LUTs. It is important to note that even though the locations of LUTs are manually specified, these locations on the CLBs are relative and any two CLBs can be chosen on the FPGA, and therefore, mapping of several instances of this mutex can be automated using simple scripts that specify these relative placements. The final implementation is then checked for any relative timing constraints, which were inherently satisfied with good timing margins.

The final mutex implementation operates with equalized paths and achieves high performance. In this implementation, the wire delays between the cross-coupled LUTs of the arbiter (*mid0-B* and *mid1-A* delays) are small and almost the same for fast and fair arbitration resolution: $\sim 114ps$ on Xilinx Virtex 7 in 28 nm. Also, longer wires are used between the arbiter and the filter stage for high-capacitive filtering, with delays of $\sim 350ps$. The resulting mapping is high-performance and balanced with a latency of nearly $\sim 445ps$ between the *reqs* and the corresponding *grants.*

6.5.2.4 Validation of mapped mutex on FPGA

Since a digital mutex is mapped on the FPGA, it is important to stress test its final implementation to check its reliability. Similar to Section 6.4.2, a hierarchical validation approach is used, where the mutex is first implemented in isolation and its implementation is locked on the FPGA, followed by interfacing it with a synchronous test wrapper.

Figure 6.8 shows the setup used to exhaustively validate the mapped mutex using a synchronous test wrapper during emulation. Two extreme cases are considered: requests arriving simultaneously and near simultaneously (3 ps apart). The outputs of the mutex are checked to see if the arbitration is resolved cleanly without glitches. A specialized *glitch catcher* is used on each output: the glitches are very short transients (oscillation period $\sim 300ps$) and cannot be directly detected by the logic analyzer as its sampling frequency is small. A FF-based glitch catcher is used, clocked by the negative edge of the grant output, with its D input tied to 1. If a glitch occurs while resolving arbitration, the glitch catcher output, which is normally reset, will become 1, which can then be detected by the ILA during its sampling window. A similar latch-based glitch catcher was also used, which can be more sensitive than the FF-based; the same result is achieved from both the catchers. An exhaustive testing was performed by running 10000 samples of each of the above cases: no glitches were observed and the mutex was shown to perform correctly. Although such extreme cases are very rare in a NoC of a real system, this test gives strong confidence that the

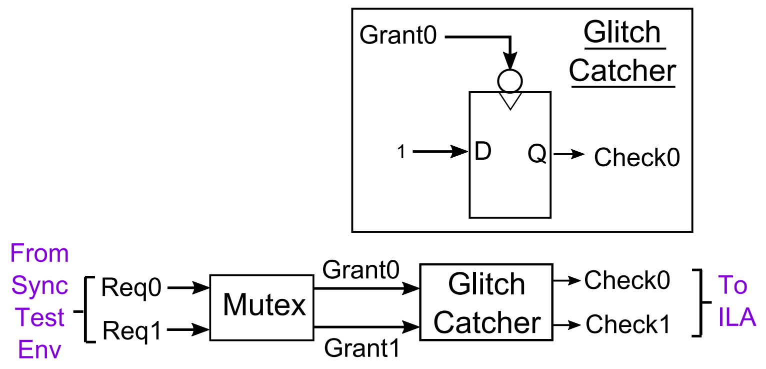

Figure 6.8: Validation setup for mapped mutex

MTBF of the mapped design will be very high for more realistic traffic.

6.5.3 4-Input Arbiter

The 4-input arbiter is a critical component of the asynchronous NoC routers, and therefore must be synthesized correctly and efficiently.

As shown in Figure 6.9, and covered in more detail in Section 2.3.3 of Chapter 2, the arbiter takes four input requests and has grant output corresponding to each request. This design achieves high performance using three mutexes in parallel [135]: left and right mutexes arbitrate between the requests *req0/req1* and *req2/req3,* respectively, while the middle mutex arbitrates between the two pairs. The final decision on which request to grant access is made using a merge of the individual arbitration results of the mutexes, performed using C-elements.

A hierarchical approach is used to synthesize the arbiter. In this approach, each of the three mutexes are first implemented in isolation, using the approach of Section 6.5.2.3, followed by synthesizing the remaining design and interfacing with the mutexes. The three mutexes, after implementation, are placed and locked side-by-side on the FPGA to achieve a similar symmetrical structure of Figure 6.9, which will minimize different wire delays, leading to an overall high-performance and robust implementation. The remaining design is synthesized using largely the standard synthesis flow but with small manual interventions, performed to optimize the latency of the mapped design. In this method, the synthesis tool takes the gate-level RTL model of the remaining design and performs the following automated steps: logic synthesis, followed by placement of LUTs on

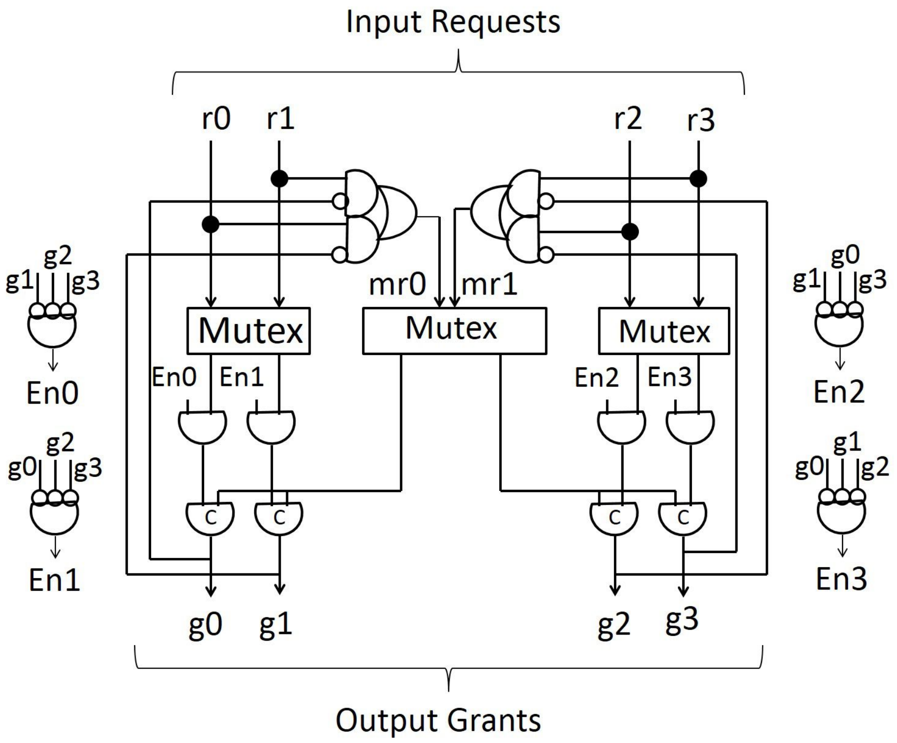

Figure 6.9: A 4-input arbiter

the FPGA, but where the locations of mapping of the LUTs, that are part of the critical paths, are manually specified so as to achieve low latency, and finally the routing step.

The Xilinx Vivado tool is used to synthesize the remaining arbiter. In more details, the tool gets two inputs: (i) a gate-level RTL model of the remaining arbiter design, with *DONT_TOUCH* primitives to disable any logic manipulations that can introduce hazards, forcing each gate to be mapped to a LUT, and (ii) a set of constraints to manually force the LUTs on the critical paths (such as the C-elements and the ANDs) to be placed at specific locations on the FPGA, such that the resulting wire delays between these LUTs will be minimized and the post place-and-route latencies from each *req* to the corresponding *grant* will be small. The latter enables a high-performance implementation. Finally, the Xilinx tool performs logic synthesis on the input RTL, and places the LUTs according to the location constraints, followed by the routing step. The final implemented design is checked for relative timing constraints, which are inherently satisfied with good timing margins.

The resulting implementation is high-performance. In addition, the implementation also achieves very balanced latencies between the requests and their corresponding grants: 1.8 ns, 1.8 ns, 1.7 ns, and 1.9 ns, respectively on Xilinx Virtex 7 in 28 nm.

The final mapped design was also exhaustively validated to ensure correct operation. Similar hierarchical validation approach was followed as for the above components: the arbiter was implemented in isolation, followed by locking its implementation on the FPGA, and then interfacing with a synchronous test wrapper. Validation was performed using both the post place-and-route timing simulation, and a logic analyzer based emulation, for variety of test scenarios.

6.6 Case Study: Asynchronous NoC Routers

To demonstrate the effectiveness of the proposed tool flow, as well as to advance the field of asynchronous NoCs, two highly-efficient asynchronous 5-port routers are synthesized on commercial FPGAs. The two implementations will also be used in a head-to-head comparison with a synchronous router in the next section. One of these asynchronous routers only supports unicast [66], which has been used in several works to achieve high performance at low overheads [89], [134], while the other router also supports lightweight multicast [17], proposed in Chapter 5. Structure and operation of these routers are briefly recapped, followed by the details on how these routers are implemented on FPGAs.

6.6.1 A Brief Recap: Unicast-Only and Multicast Asynchronous Routers

As described in Chapter 5, these routers have two main components: input port modules (IPMs) and output port modules (OPMs), connected using a shared crossbar. The IPM performs route computation on an incoming packet and selects the correct output port, while the OPM arbitrates between packets from the four IPMs and selects the winner for routing on the output channel. The two routers have different IPM designs but use the same OPM: the IPM of the multicast router supports extra replication capability and uses a different multicast addressing scheme than the unicast one, while the OPM for both the routers performs the same arbitration operation. The structure and functionality of these components are briefly presented. The IPMs and the OPMs are based on the Mousetrap pipeline.

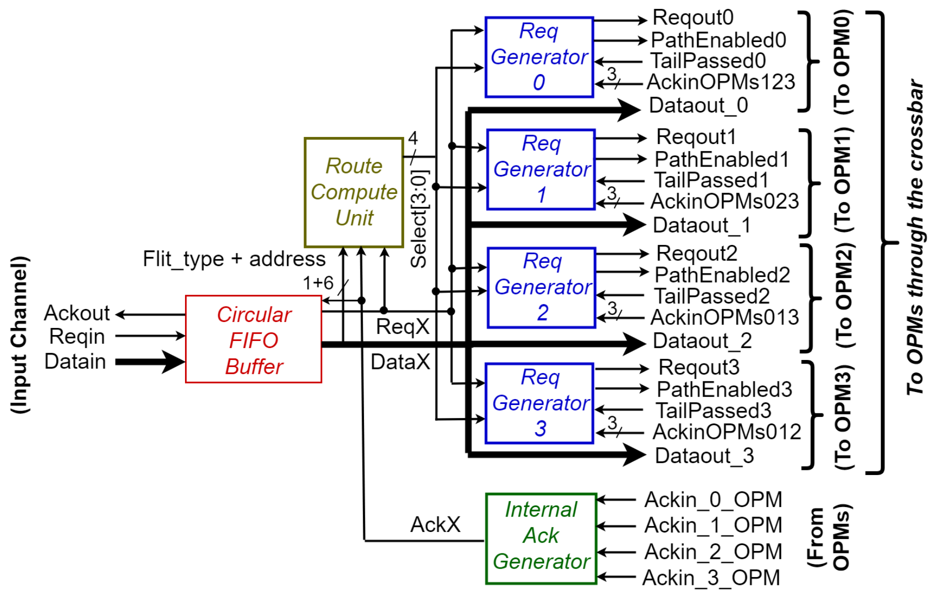

Figure 6.10: Asynchronous unicast router input port module

Unicast IPM. Figure 6.10 shows the micro-architecture of the IPM. There are four main components: circular buffer, route computation unit, request generators, and an internal ack generator. The buffer stores the incoming flits in single latch-based registers. The route computation unit reads the destination address of the header from the buffer then selects the correct request generator based on XY routing. The selected request generator then activates the correct OPM for the packet lifetime. The internal ack generator advances the read pointer in the circular buffer, after handshaking is complete between the IPM and the OPM for each flit.

Unicast/Multicast OPM. Figure 6.11 shows the micro-architecture of the OPM. There are five main components: a 4-way arbiter, four normally-closed input register for each IPM, a data mux, normally-transparent output register, and an ack generator. The arbiter arbitrates between packet headers from the four IPMs and selects the winner. The winning input channel of the OPM is allocated for the entire packet lifetime by enabling the selected input register, and selecting the correct data stream in the data mux. The winning header is then routed through the normally-transparent output register, followed by sending ack to the correct IPM by the ack generator. The trailing body/tail flits are then fast forwarded through the path pre-allocated by the header without performing arbitration. Finally, after the tail has been routed, as detected by the tail detector, a special end-of-packet ack (*tailpassed*) is sent to the correct IPM, along with the ack from the ack

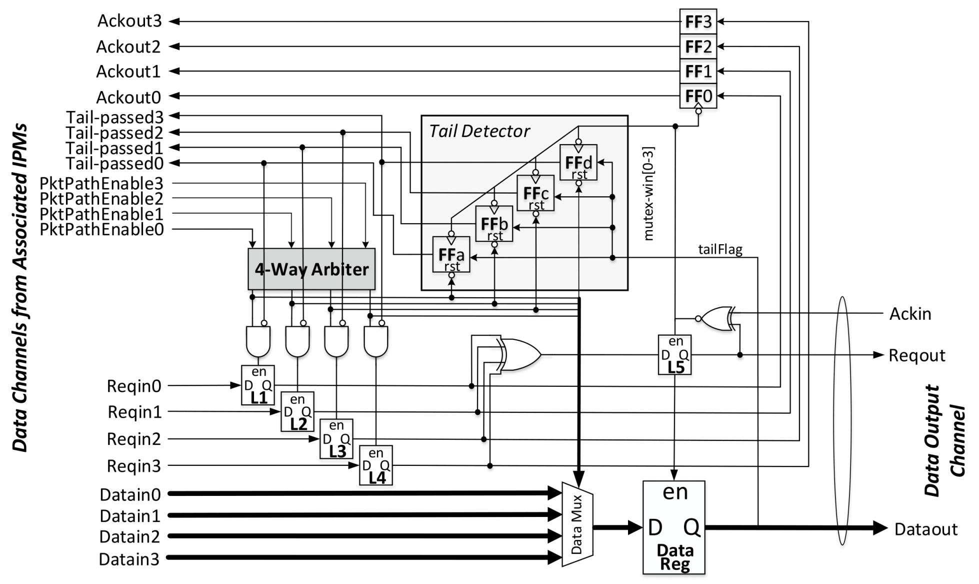

Figure 6.11: Asynchronous unicast/multicast router output port module

generator. The correct IPM releases the OPM after receiving the *tailpassed* corresponding to the tail, which then resets the entire OPM.

Multicast IPM. Figure 6.12 shows the micro-architecture of the multicast IPM. This IPM has three main components: a continuous-time multi-way read (CMR) buffer, route computation unit (RCU), and an address modifier unit (AMU). The CMR buffer stores all the flits of a packet in a single write interface, which can be accessed by multiple OPMs in parallel using the buffer's four decoupled read interfaces. On the other hand, the RCU only stores the header addressing, performs address decoding on the multicast bit-string address, and selects the correct OPMs for routing for the packet lifetime. To achieve low header latency, route computation on header is performed in parallel to its buffering in the CMR buffer, as opposed to serially in the unicast-only router. Finally, an AMU is used on each of the four outputs of the CMR buffer, which modifies the multicast header addressing, to guarantee that there is always a unique path for the multicast packet to reach each destination.

6.6.2 Implementing Asynchronous Routers on FPGAs

A hierarchical implementation approach is used to synthesize each of the above asynchronous routers on FPGAs. In this approach, the arbiters of a router's OPMs are first synthesized in isolation,

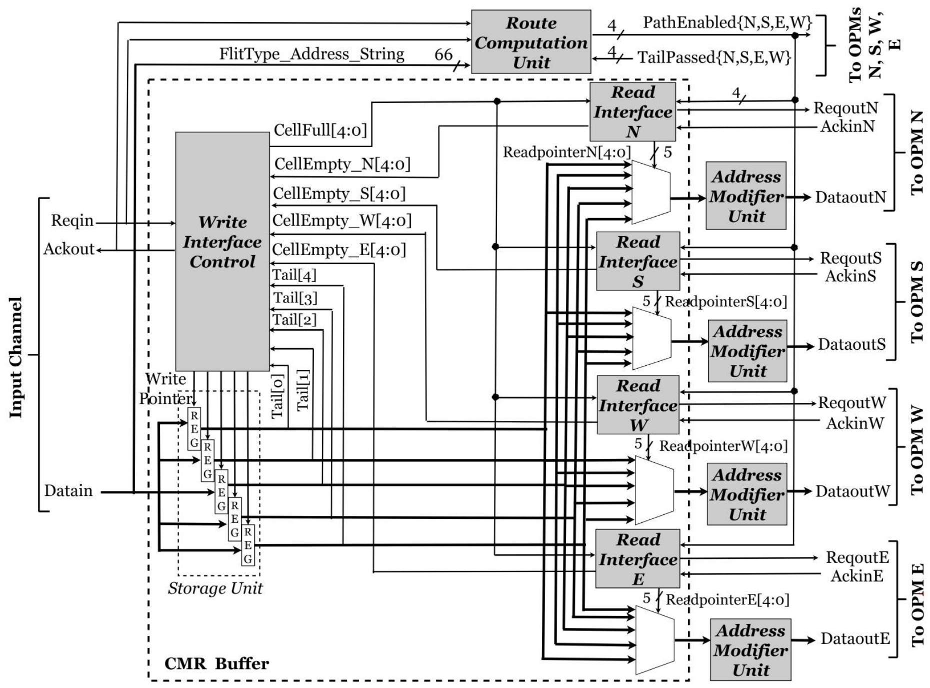

Figure 6.12: Asynchronous multicast router input port module

using the approach from Section 6.5.3, and then placed and locked on the FPGA, followed by synthesizing the remaining router design. The remaining design is synthesized using the proposed tool flow for bundled-data circuits from Section 6.4. The tool flow iterates through the performance-oriented mapping stage and the robustness-oriented stage to yield a final high-performance and robust router implementation on the FPGA.

The performance-oriented mapping stage for the router follows the steps shown in the gray boxes of the Figure 6.2. A gate-level RTL model of a hazard-free router design, keeping the arbiters as black boxes, is input to this stage. This RTL model also contains *DONT_TOUCH* directives to disable any logic manipulations that can introduce hazards. Step 1a of *critical paths determination* is performed to extract all data and control paths that are critical for both the header and body latencies from each IPM to each OPM. In Step 1b, *set_max_delay* constraints are applied to all these critical paths. In order for the max delay constraints to be applied, the *Fake clock declarations step* (Step 1c) is used to declare the latch register enable signals of all the registers on these critical paths (for example, in the input buffers and the OPMs) as 'fake' clocks using the *create_clock* construct. Next, the *implementation step* (Step 1d) is performed: logic synthesis, and place and route. The resulting implementation then undergoes an iterative process (involving steps 1d, 1e, 1f, and 1g) to check if all max delay constraints were met, and if possible perform slack optimization, followed by re-implementation or if not all constraints are met then relax the unsatisfied constraints and re-implement the design. The result of this stage is a high-performance mapping with all the max delay constraints satisfied.

The robustness-oriented stage follows the blue boxes in the Figure 6.2. First, as shown in Step 2a, all the bundling and relative timing constraints are enumerated in a pre-processing step. Next, the high-performance router implementation from the previous stage is input to the Step 2b of *path delay extraction,* where *get_timing_paths* is used to extract the delays across the paths involved in these timing constraints. Step 2c then checks if all the bundled-data and relative timing constraints are satisfied by more than 300 ps. In the router implementation, all the relative timing constraints were met by a margin more than 300 ps, however, some of the bundled-data constraints were not satisfied. So, the *adding delay step* (Step 2d), in the input RTL, appropriate delay lines, comprising buffer LUTs, are added to the request paths of these unsatisfied bundling constraints, to match the corresponding datapaths. Finally, in Step 2e, the max delay constraints for these paths were slightly relaxed, followed by re-implementation using the performance-oriented stage.

The two stages are repeated until the final high-performance and robust implementation of the router, with all timing constraints satisfied, is obtained on the FPGA.

The final implementations were stress-tested using different traffic patterns, e.g. uniform, hotspot, multi-way transmissions, and various packet sizes. Both the routers were shown to operate correctly.

6.7 Experimental Results

The asynchronous switches are implemented using the proposed methodology, and are compared with a state-of-the-art synchronous switch. The experimental setup, followed by the results, in terms of different metrics such as resource utilization, performance, and energy, are presented in this section.

6.7.1 Experimental Setup

The final implementations of the two asynchronous switches, one that only handles unicast and the other that also supports multicast, are compared with a high-performance synchronous switch. The latter is a single-cycle switch, which performs all operations in parallel: buffer write, route computation, arbitration and even link traversal, and also includes aggressive optimizations such as lookahead routing, which are not used in the asynchronous switches. This synchronous switch only supports unicast and has been used in efficient accelerator-based systems for image processing applications [121]. The two asynchronous switches are referred to as *Uni-Async*, and *Multi-Async,* respectively, and the synchronous switch is called *Uni-Sync.* All three switches use the same 34-bit datapath with no virtual channels. Each input port is buffered with a FIFO queue of depth 5. The asynchronous switches are implemented using the proposed methodology, as in Section 6.6.2, while the synchronous switch is implemented using the standard Vivado Design Flow. The target FPGA is a Xilinx Virtex 7 with speed grade -2, implemented in 28 nm technology.

All the performance and power evaluations are performed at a post place-and-route level. Vivado Post-Implementation Timing Simulator is used to get the switch latency while routing a single flit through a switch from an input port to an output port. A 2-step procedure is used for accurate switch energy/power measurements in Vivado: (i) for each switch, record the precise switching activity of its wires during a simulation while routing a single 5-flit packet through a switch, but isolating the switch from any other traffic injection circuits or any clock buffers, and (ii) perform *report_power* in Vivado using the recorded activity and the post place-and-route netlist.

6.7.2 Results

6.7.2.1 Resource utilization on FPGA

Figure 6.13 shows the number of LUTs/FLOPs used on the FPGA for each of the three switches.

Uni-Async takes 28% more LUTs and 15.7% more FLOPs than *Uni-Sync.* However, due to the predominantly latch-based asynchronous design, the majority of the FLOPs for *Uni-Async* are used as latches (1120/1210). On the other hand, the *Uni-Sync* only uses flip-flops. The use of latches in former may lead to overall better performance and reduced switching energy. Moreover, the future generations of FPGAs might use explicit single latches that are smaller than the FLOPs, which can

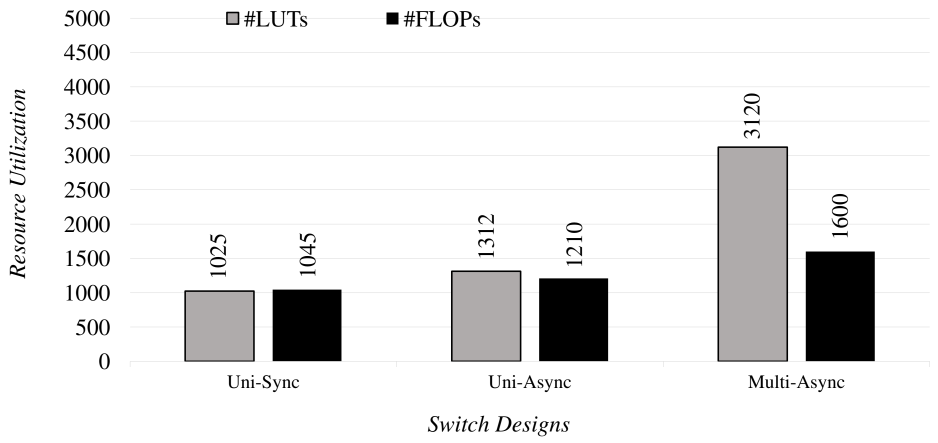

Figure 6.13: Resource utilization for the three switch designs

then lead to overall area reduction for the asynchronous switch.

Further, *Multi-Async* uses significantly more number of LUTs/FLOPs than the other unicast switches. This result is expected as the multicast switch supports an additional capability and will require extra instrumentation to achieve high-performance parallel multicast. However, similar to the unicast asynchronous switch, *Multi-Async* is also predominantly latch-based design, where 1355 out of 1600 FLOPs are latches, which can lead to improved performance and energy, compared to the synchronous switch that only uses flip-flops.

6.7.2.2 Switch latency

Figure 6.14 shows the switch latency results for routing the header and the body flits through each switch. While the synchronous switch shows a fixed latency for each flit traversal, the asynchronous switches incur different latency based on the type of the flit. In addition, for asynchronous switches, the latency is measured as the average of latencies between different input/output pairs, although these latencies show very slight variation from each other.

Comparing *Uni-Async* and *Uni-Sync,* the former shows 75% longer header latency, but achieves 30.7% lower body latency than the latter. For the header, *Uni-Async* performs all critical operations in series: buffering, route computation and arbitration, while the synchronous switch performs all these operations in parallel, and hence achieves significantly better latency. However, improved header latency results are expected for the asynchronous switch after incorporating optimizations such as lookahead optimizations. Further, for the body flits, the routing of the previous header

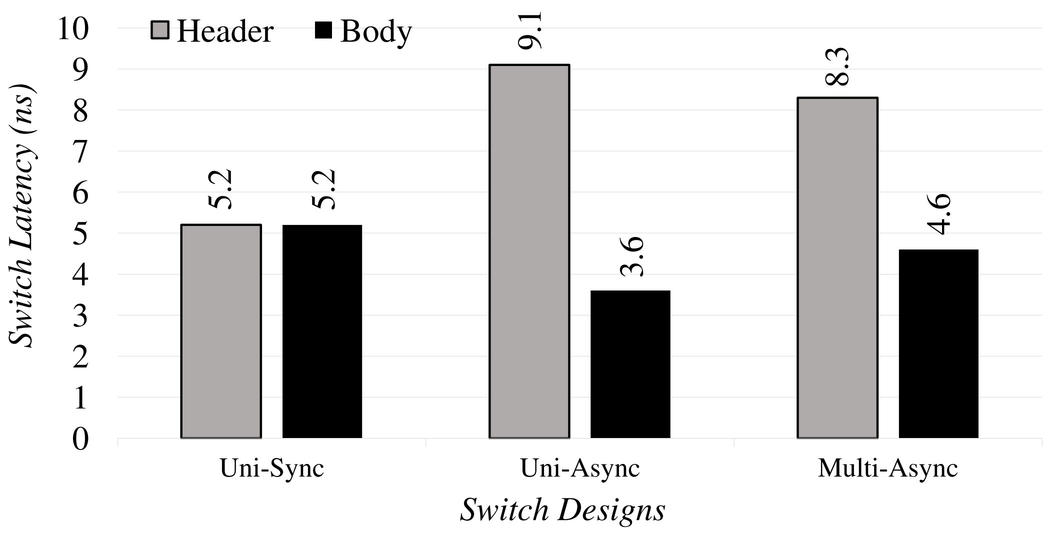

Figure 6.14: Switch latency for the header and body flits

through the asynchronous switch pre-allocates the path, such that the trailing body flits are simply fast forwarded using this path. The critical path for the body flits through the asynchronous switch is very simple and only involves latches, rather than FFs in the synchronous design, leading to considerably better latency for these flits. Moreover, this improvement in body latency can lead to overall system performance improvement, particularly for loosely-coupled accelerators that leverage long packets (1000s of body flits) to fill in their local memories and perform batch computation [121].

Multi-Async showed 59.6% higher header latency than *Uni-Sync*, which is moderately less expensive than its unicast counterpart, but still achieved 11.5% lower body latency than *Uni-Sync.* For the header, *Multi-Async* performs the buffering and route computation operation in parallel, which leads to 8.8% lower latency than the unicast asynchronous switch. Both these operations are still followed by a serial arbitration for *Multi-Async,* which leads to overheads compared to the *Uni-Sync.* However, for body flits, which are again fast forwarded in *Multi-Async,* the critical path involves latches rather than FFs, and is moderately faster than *Uni-Sync* despite the added new components such as a simple address modifier unit for multicast.

6.7.2.3 Switch energy

Figure 6.15 shows the energy-per-packet results for four different transmission scenarios: one unicast and three multicast. In the unicast transmission, a 5-flit packet is sent from one input port to

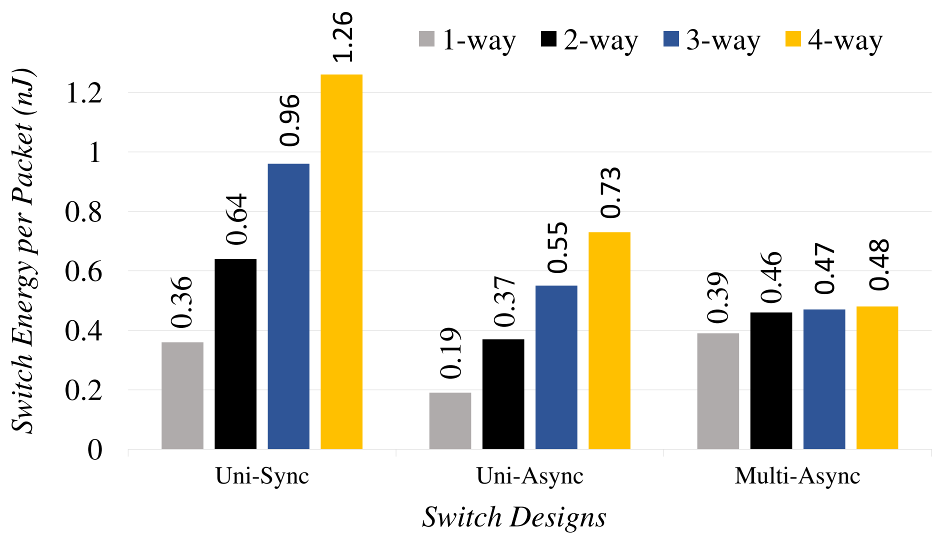

Figure 6.15: Switch energy per packet for different unicast and multicast transmissions

one output port. While in the multicast scenarios, the packet is sent from one input to multiple output ports. Unicast is the most common traffic pattern, and it is important to evaluate the energy consumption for each of the three routers for this pattern. It would also be interesting to see how the *Multi-Async* energy for transmitting a unicast packet compares with the energy of the other unicast routers, given the additional logic for multicast in the former. Evaluating energy for multicast transmission is also very important given the significance of supporting this traffic efficiently for applications such as cache coherency as well as emerging areas of neuromorphic computing. In the absence of multicast support, *Uni-Async* and *Uni-Sync* switches serially inject and route multiple unicast copies for each multicast packet, while *Multi-Async* routes the single multicast packet through multiple output ports in parallel.

For all these transmissions, *Uni-Async* achieves significantly lower energy than the synchronous switch, in the range of 42-47.2%. This result is due to the absence of the clock switching energy in the former. The clock net in the synchronous switch has a large fanout to nearly 1000 FFs, which can lead to a lot of extra switching activity than asynchronous.

Similarly, even though *Multi-Async* is more complex than *Uni-Sync*, it still has almost the same energy under a unicast transmission, and achieves significantly lower energy (28.1-61.9%) for multicast transmissions. For the latter, the absence of clock energy and the parallel routing of a multicast packet in *Multi-Async* leads to lower energy than *Uni-Sync*. For each multicast transmission, *Uni-Sync* serially routes multiple copies leading to a significant extra switching activity than *Multi-Async,* which always routes a single packet for these transmissions.

In addition, to isolate the effect of adding the new multicast capability in *Multi-Async,* the energy for this switch is compared with *Uni-Async.* The energy comparisons for these two switches for various transmissions depend on two major factors: (i) the difference in the design complexity of the two switches, and (ii) the serial vs. parallel handling of multicast transmissions. For unicast scenario, since *Multi-Async* is significantly more complex than *Uni-Async*, the former incurs almost $2\times$ overhead. For 2-way multicast transmission, this overhead is reduced to 24.3% as the unicast switch serially injects and routes two copies for a single multicast packet, which leads to extra switching activity compared to *Multi-Async*, recouping some of the complexity overheads of the latter However, for the other 3-way and 4-way transmissions, the extra switching activity due to routing of multiple packet copies becomes a major overhead for *Uni-Async,* leading to 17-52% higher energy than *Multi-Async.*

6.7.2.4 Idle power

While the previous energy results consider routing activity, it is interesting to evaluate these routers under no activity, i.e. in terms of idle power. As expected, the asynchronous routers achieve significantly lower idle power, 75% and 25% than the synchronous router, respectively, due to the absence of any clocking activity in the former (*Uni-Async:* 2 mW, *Multi-Async:* 6 mW, *Uni-Sync:* 8 mW).

6.8 Conclusions

This chapter makes significant advances in the field of asynchronous NoCs by proposing a CAD methodology for implementing bundled-data asynchronous NoCs on commercial FPGAs. The proposed tool flow not only achieves a high-performance mapping on FPGAs but also makes sure that all timing constraints are satisfied for correctness. For ease of mapping and compatibility with synchronous tools, only existing Xilinx tool is used for implementation and validation of these NoCs. In addition, as asynchronous NoCs also use some special asynchronous components such as the C-element, and the mutex; a comprehensive guide is introduced to map these elements efficiently and safely on the FPGAs. To demonstrate the effectiveness of the proposed tool flow, two highly-efficient asynchronous NoC routers are synthesized on Xilinx Virtex 7 in 28 nm, where one only supports unicast and the other also handles multicast. The former achieved significant energy and idle power improvements with some performance benefits over a high-performance synchronous router. Interestingly, similar benefits were also shown by the multicast asynchronous router, despite the extra instrumentation for multi-way transmissions.

As a future work, the proposed tool flow will be used to implement complete asynchronous NoCs. These NoCs which will then be used to interface synchronous cores, accelerators, and memories, building an entire GALS system on FPGAs. This system will be evaluated for wide-ranging applications, such as image processing and convolutional neural networks, and compare its performance and power with a synchronous system.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis makes significant contributions in the field of asynchronous NoCs, advancing the state-of-the-art considerably. The main focus of the thesis is on supporting a new type of traffic pattern: *multicast,* where a source sends a packet to arbitrary number of destinations. Multicast is common in parallel computing applications such as cache coherence and barrier synchronization, as well as in emerging areas of neuromorphic computing. However, there has been only limited research on supporting this important capability in asynchronous NoCs. This thesis proposes new techniques and architectures to support efficient multicast in asynchronous NoCs, targeting a variant mesh-of-trees topology and a 2D-mesh topology. *To the best of our knowledge, these are the first general-purpose asynchronous NoCs to support multicast.*

Additionally, for a more realistic analysis and evaluation, and also to further advance the field of asynchronous NoCs, a systematic CAD methodology is introduced to synthesize these NoCs on FPGAs. A challenging two-fold goal is targeted for the final implementation: it must be highly robust, and achieves high performance. *To the best of our knowledge, this is the first systematic methodology to efficiently map asynchronous NoCs on commercial FPGAs using existing synchronous tools.*

The asynchronous NoCs targeted in this thesis use single-rail bundled data encoding and two-phase handshaking protocol. This combination can lead to cost-effective NoC solutions [89], [83], [66], but also rely on modest timing constraints for correct operation.

Overall, this thesis makes the following three contributions.

(i) Multicast in variant mesh-of-trees NoCs using local speculation. For variant MoT, a novel strategy, *local speculation,* is introduced for high-performance and low-overhead multicast. In this strategy, a packet (unicast or multicast) is always broadcast at a fixed subset of speculative switches in the network. To restrict the distance traveled by any redundant packets to small local regions, these packets are throttled by neighboring non-speculative switches, hence, limiting the penalties of speculation to minimal impact on congestion and power. Speculative switches are very simple and fast as they do not perform route computation or channel allocation. Non-speculative switches perform throttling with almost no hardware overhead. This mix of speculative and non-speculative switches leads to a new hybrid network architecture. For multicast traffic, significant performance benefits with small power savings are obtained by the new hybrid network over a fully non-speculative baseline. Interestingly, similar improvements are also shown for unicast. This latter result makes local speculation suitable for not just multicast-enabled NoCs but also for unicast-only NoCs to achieve high performance with low power/area overheads.

While introduced for the NoC domain, the local speculation based hybrid architecture paradigm can also be useful in other domains to achieve high performance at low overheads. For example, in many-accelerator heterogeneous SoCs, a fixed subset of the accelerators can be *approximate,* performing fast computation and are very simple, which can be used for applications that do not require very accurate results, such as in image processing. The other accelerators in the system can be accurate, to be used by applications that require full accuracy. Such a system can be asynchronous or synchronous, but the former can be a better option to exploit all the benefits of this hybrid architecture.

(ii) Multicast in 2D-mesh NoCs using a continuous-time replication strategy. For 2D mesh, a new replication strategy is introduced to achieve high-performance multicast while still maintaining low overheads. In this approach, the flits of a multicast packet are first stored in a single buffer at an input port, from where they are routed through the distinct outputs of the router according to each outputs own rate, in parallel and in continuous time. Unlike synchronous, this unique asynchronous approach, not discretized to clock cycles, can provide considerable performance benefits by accommodating subtle variations in network congestion and exploiting sub-cycle differences in interface operating rates. In addition, a new *continuous-time multi-way read (CMR) buffer* is introduced to enable the above strategy at low overheads. This buffer is a low-latency FIFO with multiple read controls that can be accessed in parallel. It is also a standalone architecture that can also be used for different applications. Exhaustive experiments on wide-ranging multicast benchmarks, show significant performance gains for the new approach, with small-moderate energy improvements over a baseline network that performs multicast using several unicasts, injected and routed serially. Interestingly, moderate latency improvements were also shown for unicast.

The continuous-time replication strategy and the CMR buffer can also be used for different applications to achieve high system performance. For example, the CMR buffer can be used to implement a shared memory, to be concurrently accessed by multiple accelerators/cores in a heterogeneous SoC. These units can exploit the decoupled read pointers of this memory to perform parallel read operations in continuous time, without waiting for any clock cycles, and exploiting any sub-cycle delay differentials in operating speeds.

(iii) A systematic methodology for synthesizing asynchronous NoCs on FPGAs. Finally, a comprehensive CAD tool flow is proposed for mapping asynchronous NoCs on modern commercial FPGAs. The target of this work is single-rail bundled-data asynchronous NoCs, which are shown to be highly-efficient but also involve complex timing constraints for correct operation. The proposed methodology not only makes sure that all timing constraints are satisfied but also achieves high performance for these NoCs. For ease of implementation and convenience for the designs, only the existing commercial synchronous FPGA synthesis tool is used. In addition, asynchronous NoCs also use some special asynchronous components such as a C-element, and a mutex; a systematic guide is also proposed on how to map these components, as well as a 4-input arbiter, both efficiently and safely. Two distinct bundled-data 5-port switches are synthesized using the proposed tool flow on Xilinx Virtex 7 in 28 nm: one that only supports unicast, and the other that also handles multicast (Chapter 5). Compared to a high-performance unicast-only synchronous switch, the unicast asynchronous switch achieved 47% lower energy and 75% lower idle power with some performance benefits. Interestingly, the multicast asynchronous switch also achieved similar benefits over the synchronous switch, despite the extra instrumentation for multicast support.

While introduced for asynchronous NoCs, the proposed CAD methodology and the guide to map asynchronous components are generic and can be used for different asynchronous designs and systems. In addition, the proposed validation approaches for asynchronous circuits can also be useful for testing the implemented systems on FPGAs.

7.2 Future Work

There are a few potential areas for future work.

Post-layout implementations of new multicast-enabled asynchronous NoCs. The new asynchronous NoCs proposed in Chapters 4 and 5 are pre-layout implementations. For a more realistic analysis, these NoCs can be implemented and evaluated at post-layout level.

Full-system ASIC implementation and evaluation. The proposed NoCs in Chapters 4 and 5 are evaluated only at the NoC-level. For more exhaustive system-level evaluation, complete GALS systems can be built, where multiple synchronous cores, operating at different clock speeds, are connected using these asynchronous NoCs through mixed-timing interfaces. These systems can then be evaluated using real applications running on these systems, and compared to fully-synchronous systems.

Evaluation using real traffic benchmarks. The NoCs in Chapters 4 and 5 are evaluated using only synthetic traffic. For more realistic evaluation and thorough analysis, application benchmarks such as SPLASH and PARSEC can be used to further evaluate these NoCs. These benchmarks model real-world multi-threaded behavior, including multicast traffic, and therefore, will be very useful for more accurate analysis.

Complete GALS systems on FPGAs for accelerating real applications. While the CAD methodology introduced in Chapter 6 has been demonstrated on mapping an asynchronous NoC switch on FPGAs, it can also be used to synthesize complete NoCs. Such NoCs can then be used to integrate multiple synchronous cores, memories and accelerators, building complete GALS systems on FPGAs. These systems can be evaluated in terms of performance, power, as well as scalability, and compared with other fully-synchronous systems on FPGAs for real applications such as image processing and convolutional neural networks.

Support for many-to-1 traffic. Another common traffic pattern seen in cache coherence traffic as well as neural networks is many-to-1 or many-to-few, also known as ACK aggregation. A potential future area of research can be to introduce new efficient methods to support this communication in asynchronous NoCs.

Bibliography

[1] S. Abadal, A. Mestres, E. Alarcón, A. Cabellos-Aparicio, and R. Martinez. Multicast on-chip traffic analysis targeting manycore NoC design. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 370–378, 2015.

[2] F. Akopyan, J. Sawada, A. Cassidy, R.A.-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, B. Taba, M. Beakes, B. Brezzo, J.B. Kuang, R. Manohar, W.P. Risk, B. Jackson, and D.S. Modha. TrueNorth: Design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34:1537–1557, 2015.

[3] R. Au. Intel FPGAs accelerate artificial intelligence for deep learning in Microsoft's Bing intelligent search. *https://newsroom.intel.com/editorials/intel-fpgas-accelerating-artificial-intelligence-deep-learning-bing-intelligent-search/*, 2018.

[4] P. Bahrebar and D. Stroobandt. The hamiltonian-based odd-even turn model for maximally adaptive routing in 2D mesh networks-on-chip. *Journal of Computers & Electrical Engineering*, 45:386–401, 2015.

[5] A.O. Balkan, M.N. Horak, G. Qu, and U. Vishkin. Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing. In *IEEE Symposium on High-Performance Internconnects (HOTI)*, pages 21–28, 2007.

[6] A.O. Balkan, G. Qu, and U. Vishkin. A mesh-of-trees interconnection network for single-chip parallel processing. In *International Conference on Application-Specific Systems, Architecture and Processors (ASAP)*, pages 73–80, 2006.

[7] A.O. Balkan, G. Qu, and U. Vishkin. Mesh-of-trees and alternative interconnection networks for single-chip parallelism. *IEEE Transactions on VLSI Systems*, 17:1419–1432, 2009.

[8] P. Beerel, G.D. Dimou, and A.M Lines. Proteus: an ASIC flow for GHz asynchronous designs. *IEEE Design and Test*, 28:38–51, 2011.

[9] P. Beerel and T.H.-Y. Meng. Automatic gate-level synthesis of speed independent circuits. In *International Conference on Computer-Aided Design (ICCAD)*, pages 581–587, 1992.

[10] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. F. Brown III, M. Mattina, C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - processor: a 64-core SoC with mesh interconnect. In *International Solid-State Circuits Conference (ISSCC)*, pages 88–89, 2008.

[11] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012: building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Design, Automation and Test in Europe DATE*, pages 983–987, 2012.

[12] L. Benini and G.D. Micheli. Networks on chip: A new SoC paradigm. *IEEE Transactions on Computers*, 35:70–78, 2002.

[13] B.V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. Chandrasekaran, J. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. Merolla, and K. Boahen. Neurogrid: a mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716, 2014.

[14] K.V. Berkeal, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *European Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[15] D. Bertozzi, G. Dimitrakopoulos, J. Flich, and S. Sonntag. The fast evolving landscape of on-chip communication - selected future challenges and research avenues. *Design Automation for Embbeded Systems*, 19:59–76, 2015.

[16] E. Bezati, S.C. Brunet, M. Mattavelli, and J.W. Janneck. Coarse grain clock gating of streaming applications in programmable logic implementations. In *IEEE Electronic System Level Synthesis Conference (ESLsyn)*, pages 1–6, 2014.

[17] K. Bhardwaj, W. Jiang, and S.M. Nowick. Achieving lightweight multicast in asynchronous NoCs using a continuous-time multi-way read buffer. In *International Symposium on Networks-on-Chip (NOCS)*, pages 6:1–6:8, 2017.

[18] K. Bhardwaj and S.M. Nowick. Achieving lightweight multicast in asynchronous networks-on-chip using local speculation. In *Design Automation Conference (DAC)*, pages 38:1–38:6, 2016.

[19] K. Bhardwaj and S.M. Nowick. A continuous-time replication strategy for efficient multicast in asynchronous NoCs. In *To appear in IEEE Transactions on VLSI Systems*, 2019.

[20] R. Bielby and G. Brown. Advantages of Xilinx 7 Series all programmable FPGA and SoC devices. *http://www.ni.com/white-paper/14583/en/*, 2017.

[21] E.E. Bilir, R.M. Dickson, Y. Hu, M. Plakal, D.J. Sorin, M.D. Hill, and D.A. Wood. Multicast snooping: a new coherence method using a multicast address network. In *International Symposium on Computer Architecture (ISCA)*, pages 294–304, 1999.

[22] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *Journal of ACM Computing Surveys*, 38:1–51, 2006.

[23] T. Bjerregaard and J. Sparsø. A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip. In *Design, Automation and Test in Europe (DATE)*, pages 1226–1231, 2005.

[24] P. Bogdan, T. Majumder, A. Ramanathan, and Y. Xue. NoC architectures as enablers of biological discovery for personalized and precision medicine. In *International Symposium on Networks-on-Chip (NOCS)*, pages 27:1–27:11, 2015.

[25] M. Branscombe. Why Microsoft has bet on FPGAs to infuse its cloud with AI. *http://www.datacenterknowledge.com/microsoft/why-microsoft-has-bet-fpgas-infuse-its-cloud-ai*, 2018.

[26] L.P. Carloni, P. Pande, and Y. Xie. Networks-on-chip in emerging interconnect paradigms: advantages and challenges. In *International Symposium on Networks-on-Chips (NOCS)*, pages 93–102, 2009.

[27] M.F. Chang, J. Cong, A. Kaplan, C. Liu, M. Naik, J. Premkumar, G. Reinman, E. Socher, and S. Tam. Power reduction of CMP communication networks via RF-interconnects. In *International Symposium on Microarchitecture (MICRO)*, pages 376–387, 2008.

[28] M.F. Chang, J. Cong, A. Kaplan, M. Naik, G. Reinman, E. Socher, and S. Tam. CMP network-on-chip overlaid with multi-band RF-interconnect. In *International Conference on High-Performance Computer Architecture (HPCA)*, pages 191–202, 2008.

[29] T. Chelcea and S.M. Nowick. Robust interfaces for mixed-timing systems. *IEEE Transations on VLSI Systems*, 12:857–873, 2004.

[30] G. Chen, M.A. Anders, H. Kaul, S.K. Satpathy, S.K. Mathew, S.K. Hsu, A. Agarwal, R.K. Krishnamurthy, V. De, and S. Borkar. A 340 mV-to-0.9 V 20.2 Tb/s source-synchronous hybrid packet/circuit-switched 16x16 network-on-chip in 22 nm Tri-Gate CMOS. *IEEE Journal of Solid-State Circuits*, 50:1444–1454, 2015.

[31] X. Chen and N.K. Jha. Reducing wire and energy overheads of the SMART NoC using a setup request network. *IEEE Transactions on VLSI*, 24:3013–3026, 2016.

[32] Y. Chen, T. Krishna, J.S. Emer, and V. Sze. Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks. *Journal of Solid-State Circuits (JSSC)*, 52:127–138, 2017.

[33] L. Cheng, M. Browning, P. V. Gratz, and S. Palermo. Energy-efficient optical broadcast for nanophotonic networks-on-chip. In *Optical Interconnects Conference*, pages 64–65, 2012.

[34] J.F. Christmann, E. Beigne, C. Condemine, N. Leblond, P. Vivet, G. Waltisperger, and J. Willemin. Bringing robustness and power efficiency to autonomous energy harvesting microsystems. In *International Symposium of Asynchronous Circuits and Systems (ASYNC)*, pages 62–71, 2010.

[35] J.A. Clarke, G.A. Constantinides, and P.Y.K. Cheung. On the feasibility of early routing capacitance estimation for FPGAs. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 234–239, 2007.

[36] K. Constantinides, S. Plaza, J.A. Blome, B. Zhang, V. Bertacco, S.A. Mahlke, T.M. Austin, and M. Orshansky. Bulletproof: a defect-tolerant CMP switch architecture. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 5–16, 2006.

[37] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Complete state encoding based on the theory of regions. In *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 36–47, 1996.

[38] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: A tool for manipulating concurrent specification and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D:315–325, 1997.

[39] D. Jeon *et al.* An energy efficient full-frame feature extraction accelerator with shift-latch FIFO in 28 nm CMOS. *IEEE Journal of Solid-State Circuits (JSSC)*, 49:1271–1284, 2014.

[40] W.J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *Design Automation Conference (DAC)*, pages 684–689, 2001.

[41] W.J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

[42] M. Daneshtalab, M. Ebrahimi, S. Mohammadi, and A. Afzali-Kusha. Low-distance path-based multicast routing algorithm for network-on-chips. *IET Computers & Digital Techniques*, 3:430–442, 2009.

[43] S. Das, J.R. Doppa, P.P. Pande, and K. Chakrabarty. Energy-efficient and reliable 3D network-on-chip (NoC): Architectures and optimization algorithms. In *International Conference on Computer-Aided Design (ICCAD)*, pages 1–6, 2016.

[44] S. Das, A. Fan, K.-N. Chen, C.S. Tan, N. Checka, and R. Reif. Technology, performance, and computer-aided design of three-dimensional integrated circuits. In *International Symposium on Physical Design (ISPD)*, pages 108–115, 2004.

[45] M. Davies, A. Lines, J. Dama, A. Gravel, R. Southworth, G. Dimou, and P. Beerel. A 72-port 10G ethernet switch/router using quasi-delay-insensitive asynchronous design. In *International Symposium of Asynchronous Circuits and Systems (ASYNC)*, pages 103–104, 2014.

[46] M. Davies, N. Srinivasa, T. Lin, G.N. Chinya, Y. Cao, S.H. Choday, G.D. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. Weng, A. Wild, Y. Yang, and H. Wang. Loihi: a neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38:82–99, 2018.

[47] B.K. Daya, C.O. Chen, S. Subramanian, W. Kwon, S. Park, T. Krishna, J. Holt, A.P. Chandrakasan, and L.S. Peh. SCORPIO: a 36-core research chip demonstrating snoopy coherence on a scalable mesh NoC with in-network ordering. In *International Symposium on Computer Architecture (ISCA)*, pages 25–36, 2014.

[48] M.E. Dean, T.E. Williams, and D.L. Dill. Efficient self-timing with level-encoded 2-phase dual-rail (LEDR). In *University of California/Santa Cruz Conference on Advanced Research in VLSI*, pages 55–70, 1991.

[49] S. Deb, A. Ganguly, P.P. Pande, B. Belzer, and D. Heo. Wireless NoC as interconnection backbone for multicore chips: Promises and challenges. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2:228–239, 2012.

[50] D. DiTomaso, A.K. Kodi, D.. Matolak, S. Kaya, S. Laha, and W. Rayess. A-winoc: adaptive wireless network-on-chip architecture for chip multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 26:3289–3302, 2015.

[51] R. Dobkin, R. Ginosar, and I. Cidon. QNoC asynchronous router with dynamic virtual channel allocation. In *International Symposium on Networks-on-Chips (NOCS)*, page 218, 2007.

[52] R. Dobkin, V. Vishnyakov, E. Friedman, and R. Ginosar. An asynchronous router for multiple service levels networks on chip. In *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 44–53, 2005.

[53] M. Ebrahimi, X. Chang, M. Daneshtalab, J. Plosila, P. Liljeberg, and H. Tenhunen. DyXYZ: Fully adaptive routing algorithm for 3D NoCs. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 499–503, 2013.

[54] M. Ebrahimi, M. Daneshtalab, P. Liljeberg, J. Plosila, J. Flich, and H. Tenhunen. Path-based partitioning methods for 3D networks-on-chip with minimal adaptive routing. *IEEE Transactions on Computers*, 63:718–733, 2014.

[55] V. Ekanayake, C. Kelly, and R. Manohar. An ultra low-power processor for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 27–36, 2004.

[56] R. Ezz-Eldin, M.A. El-Moursy, and H.F.A. Hamed. Asynchronous high throughput NoC under high process variation. In *International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 361–364, 2013.

[57] G. Faldamis, W. Jiang, G. Gill, and S.M. Nowick. A low-latency asynchronous interconnection network with early arbitration resolution. In *Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 329–336, 2014.

[58] R. Farah and H. Harmanani. A method for efficient NoC test scheduling using deterministic routing. In *International System-on-Chip Conference (SoCC)*, pages 363–366, 2010.

[59] D. Fick, A. DeOrio, G.K. Chen, V. Bertacco, D. Sylvester, and D. Blaauw. A highly resilient routing algorithm for fault-tolerant nocs. In *Design, Automation and Test in Europe (DATE)*, pages 21–26, 2009.

[60] R.M. Fuhrer, B. Lin, and S.M. Nowick. Symbolic hazard-free minimization and encoding of asynchronous finite state machines. In *International Conference on Computer-Aided Design (ICCAD)*, pages 604–611, 1995.

[61] S.B. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4:247–253, 1996.

[62] S.B. Furber, F. Galluppi, S. Temple, and L.A. Plana. The SpiNNaker project. *Proceedings of the IEEE*, 102:652–665, 2014.

[63] H.V. Gageldonk, K.V. Berkel, A. Peeters, D. Baumann, D. Gloor, and G. Stegmann. An asynchronous low-power 80C51 microcontroller. In *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 96–107, 1998.

[64] R. Gagne, J. Belzile, and C. Thibeault. Asynchronous component implementation methodology for gals design in fpgas. In *IEEE North-East Workshop on Circuits and Systems and TAISA Conference*, pages 1–4, 2009.

[65] J.D. Garside, S.B. Furber, S. Temple, and V. Woods. The Amulet chips: Architectural development for asynchronous microprocessors. In *International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 343–346, 2009.

[66] A. Ghiribaldi, D. Bertozzi, and S.M. Nowick. A transition-signaling bundled data NoC switch architecture for cost-effective GALS multicore systems. In *Design, Automation and Test in Europe (DATE)*, pages 332–337, 2013.

[67] G. Gill, S.S. Attarde, G. Lacourba, and S.M. Nowick. A low-latency adaptive asynchronous interconnection network using bi-modal router nodes. In *International Symposium on Networks-on-Chip (NOCS)*, pages 193–200, 2011.

[68] R. Ginosar. Handshake circuit implementations: slide 13. *http://slideplayer.com/slide/4906671/*, 2009.

[69] C.J. Glass and L.M. Ni. The turn model for adaptive routing. In *International Symposium on Computer Architecture (ISCA)*, pages 278–287, 1992.

[70] K. Goossens, J. Dielissen, and A. Radulescu. AEthereal network on chip: concepts, architectures and implementation. *IEEE Design and Test of Computers*, 22:414–421, 2005.

[71] P. Gratz, C. Kim, K. Sankaralingam, H. Hanson, P. Shivakumar, S.W. Keckler, and D. Burger. On-chip interconnection networks of the TRIPS chip. *IEEE Micro*, 27(5):41–50, 2007.

[72] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *Design, Automation and Test in Europe (DATE)*, pages 250–256, 2000.

[73] M. Halpern, Y. Zhu, and V.J. Reddi. Mobile cpu's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–76, 2016.

[74] D. Hand, M.T. Moreira, H. Huang, D. Chen, F. Butzke, Z. Li, M. Gibiluka, M.A. Breuer, N.L.V. Calazans, and P.A. Beerel. Blade - A timing violation resilient asynchronous template. In *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 21–28, 2015.

[75] S. Hauck, G. Borriello, S. Burns, and C. Ebeling. MONTAGE: an FPGA for synchronous and asynchronous circuits. In *International conference on Field Programmable Logic and Applications (FPL)*, pages 44–51, 1992.

[76] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, and D. Lindqvist. Network on chip: An architecture for billion transistor era. In *IEEE Nordic Microelectronics Conference (NORCHIP)*, pages 1–8, 2000.

[77] Q.T. Ho, J. Rigaud, L. Fesquet, M. Renaudin, and R. Rolland. Implementing asynchronous circuits on LUT based FPGAs. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 36–46, 2002.

[78] M.N. Horak, S.M. Nowick, M. Carlberg, and U. Vishkin. A low-overhead asynchronous interconnection network for GALS chip multiprocessors. *IEEE Transactions on Computer-Aided Design (TCAD)*, 30:494–507, 2011.

[79] J. Hu and R. Marculescu. Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures. In *2003 Design, Automation and Test in Europe (DATE)*, pages 10688–10693, 2003.

[80] J. Hu and R. Marculescu. DyAD: Smart routing for networks-on-chip. In *Design Automation Conference (DAC)*, pages 260–263, 2004.

[81] W. Hu, Z. Lu, A. Jantsch, and H. Liu. Power-efficient tree-based multicast support for networks-on-chip. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 363–368, 2011.

[82] N. Huot, H. Dubreuil, L. Fesquet, and M. Renaudin. FPGA architecture for multi-style asynchronous logic. In *Design, Automation and Test in Europe (DATE)*, pages 32–33, 2005.

[83] M. Imai, T.V. Chu, K.K., and T. Yoneda. The synchronous vs. asynchronous NoC routers: an apple-to-apple comparison between synchronous and transition signaling asynchronous designs. In *International Symposium on Networks-on-Chip (NOCS)*, pages 1–8, 2016.

[84] M. Imai and T. Yoneda. Improving dependability and performance for fully asynchronous on-chip networks. In *International Symposium of Asynchronous Circuits and Systems (ASYNC)*, pages 65–76, 2011.

[85] M. Imai and T. Yoneda. Multiple-clock multiple-edge-triggered multiple-bit flip-flops for two-phase handshaking asynchronous circuits. In *International Symposium on Circuits and Systems (ISCAS)*, pages 141–144, 2014.

[86] National Instruments. Advantages of Xilinx 7 series all programmable FPGA and SoC devices. *http://www.ni.com/white-paper/14583/en/#toc3*, 2017.

[87] S. Ishihara, M. Hariyama, and M. Kameyama. A low-power FPGA based on autonomous fine-grain power gating. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19:1394–1406, 2011.

[88] N.D.E. Jerger, L.S. Peh, and M.H. Lipasti. Virtual circuit tree multicasting: a case for on-chip hardware multicast support. In *International Symposium on Computer Architecture (ISCA)*, pages 229–240, 2008.

[89] W. Jiang, D. Bertozzi, G. Miorandi, S.M. Nowick, W. Burleson, and G. Sadowski. An asynchronous NoC router in a 14nm FinFET library: comparison to an industrial synchronous counterpart. In *Design, Automation and Test in Europe (DATE)*, pages 732–733, 2017.

[90] W. Jiang, K. Bhardwaj, G. Lacourba, and S.M. Nowick. A lightweight early arbitration method for low-latency asynchronous 2D-mesh NoC's. In *Design Automation Conference (DAC)*, pages 1–6, 2015.

[91] A.P. Johnson, R.S. Chakraborty, and D. Mukhopadhyay. A PUF-enabled secure architecture for FPGA-based IoT applications. *IEEE Transactions on Multi-Scale Computing Systems*, 1:110–122, 2015.

[92] J.W. Joyner, P. Zarkesh-Ha, and J.D. Meindl. A stochastic global net-length distribution for a three-dimensional system-on-a-chip (3D-SoC). In *IEEE International ASIC/SOC Conference*, pages 147–151, 2001.

[93] F. Karim, A. Nguyen, and S. Dey. An interconnect architecture for networking systems on chips. *IEEE Micro*, 22:36–45, 2002.

[94] A. Karkar, N. Dahir, R. Al-Dujaily, K. Tong, T.S.T. Mak, and A. Yakovlev. Hybrid wire-surface wave architecture for one-to-many communication in networks-on-chip. In *Design, Automation and Test in Europe (DATE)*, pages 1–4, 2014.

[95] A. Karkar, J.E. Turner, K. Tong, R. Al-Dujaily, T.S.T. Mak, A. Yakovlev, and F. Xia. Hybrid wire-surface wave interconnects for next-generation networks-on-chip. *IET Computers & Digital Techniques*, 7:294–303, 2013.

[96] E. Kasapaki and J. Sparso. Argo: A time-elastic time-division-multiplexed NoC using asynchronous routers. In *International Symposium of Asynchronous Circuits and Systems (ASYNC)*, pages 45–52, 2014.

[97] H. Katabami, H. Saito, and T. Yoneda. Design of a GALS-NoC using soft-cores on FPGAs. In *International Symposium on Embedded Multicore SoCs*, pages 31–36, 2013.

[98] S. Kilts. *Advanced FPGA Design*. Wiley-IEEE Press, 2007.

[99] J. Kim, W.J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable Dragonfly topology. In *International Symposium on Computer Architecture (ISCA)*, pages 77–88, 2008.

[100] M.M. Kim, K.M. Fant, and P. Beckett. Design of asynchronous RISC CPU register-file write-back queue. In *International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 31–36, 2015.

[101] M.A. Kinsy, M.H. Cho, T. Wen, G.E. Suh, M. Dijk, and S. Devadas. Application-aware deadlock-free oblivious routing. In *International Symposium on Computer Architecture (ISCA)*, pages 208–219, 2009.

[102] A.K. Kodi, A. Louri, and J.M. Wang. Design of energy-efficient channel buffers with router bypassing for network-on-chips (NoCs). In *International Symposium on Quality of Electronic Design (ISQED)*, pages 826–832, 2009.

[103] T. Krishna. Garnet2.0: a detailed on-chip network model inside a full-system simulator. *http://www.gem5.org/wiki/images/d/d4/Summit2017_garnet2.0_tutorial.pdf*, 2017.

[104] T. Krishna, C.-H. Chen, W.-H. Kwon, and L.-S. Peh. SMART: single-cycle multihop traversals over a shared network on chip. *IEEE Micro*, 34:43–56, 2014.

[105] T. Krishna and L.S. Peh. Single-cycle collective communication over a shared network fabric. In *International Symposium on Networks-on-Chip (NOCS)*, pages 1–8, 2014.

[106] T. Krishna, L.S. Peh, B.M. Beckmann, and S.K. Reinhardt. Towards the ideal on-chip fabric for 1-to-many and many-to-1 communication. In *International Symposium on Microarchitecture*, pages 71–82, 2011.

[107] M. Krstic, E. Grass, F.K. Gurkaynak, and P. Vivet. Globally asynchronous, locally synchronous circuits: overview and outlook. *IEEE Design and Test*, 24:430–441, 2007.

[108] A. Kumar, P. Kundu, A.P. Singh, L.S. Peh, and N.K. Jha. A 4.6 Tbits/s 3.6 GHz single-cycle NoC router with a novel switch allocator in 65 nm CMOS. In *International Conference on Computer Design (ICCD)*, pages 63–70, 2007.

[109] H. Kwon, A. Samajdar, and T. Krishna. Rethinking NoCs for spatial neural network accelerators. In *International Symposium on Networks-on-Chip (NOCS)*, pages 19:1–19:8, 2017.

[110] J. Lassen. FPGA prototyping of asynchronous networks-on-chip. *M.Sc. thesis, DTU, Kongens Lyngby*, 2008.

[111] S. Lee, S. Tam, I. Pefkianakis, S. Lu, M.F. Chang, C. Guo, G. R., C. Peng, M. Naik, L. Zhang, and J. Cong. A scalable micro wireless interconnect structure for CMPs. In *International Conference on Mobile Computing and Networking (MOBICOM)*, pages 217–228, 2009.

[112] W. Lee and G.E. Sobelman. Mesh-star hybrid noc architecture with CDMA switch. In *International Symposium on Circuits and Systems (ISCAS)*, pages 1349–1352, 2009.

[113] G. Lemieux, E. Lee, M. Tom, and A.J. Yu. Directional and single-driver wires in FPGA interconnect. In *International Conference on Field Programmable Technology (FPT)*, pages 41–48, 2004.

[114] M. Lewis, J. Garside, and L. Brackenbury. Reconfigurable latch controllers for low power asynchronous circuits. In *International Symposium of Asynchronous Circuits and Systems (ASYNC)*, pages 27–35, 1999.

[115] M. Li, Q. Zeng, and W. Jone. DyXY: a proximity congestion-aware deadlock-free dynamic routing method for network on chip. In *Design Automation Conference (DAC)*, pages 849–852, 2006.

[116] M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev. Asynchronous design using commercial HDL synthesis tools. In *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 1–12, 2000.

[117] A. Lines, P. Joshi, R. Liu, S. McCoy, J. Tse, Y. H. Weng, and M. Davis. Loihi asynchronous neuromorphic research chip. *International Symposium of Asynchronous Circuits and Systems (ASYNC)*, 2018.

[118] M. Lodde, J. Flich, and M. E. Acacio. Heterogeneous NoC design for efficient broadcast-based coherence protocol support. In *International Symposium on Networks-on-Chip (NOCS)*, pages 59–66, 2012.

[119] J. Luo, A. Elantably, V.D. Pham, C. Killian, D. Chillet, S.L. Beux, O. Sentieys, and I. O'Connor. Performance and energy aware wavelength allocation on ring-based WDM 3D optical NoC. In *Design, Automation and Test in Europe (DATE)*, pages 1372–1375, 2017.

[120] S. Ma, N.D.E. Jerger, and Z. Wang. Whole packet forwarding: Efficient design of fully adaptive routing algorithms for networks-on-chip. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 467–478, 2012.

[121] P. Mantovani, G.D. Guglielmo, and L.P. Carloni. High-level synthesis of accelerators in embedded scalable platforms. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 204–211, 2016.

[122] R. Marculescu, Y. Ogras, L.S. Peh, N.D.E. Jerger, and Y.V. Hoskote. Outstanding research problems in NoC design: system, microarchitecture, and circuit perspectives. *IEEE Transactions on Computer-Aided Design (TCAD)*, 28:3–21, 2009.

[123] A.J. Martin. *Programming in VLSI: from Communicating Processes to Delay-Insensitive Circuits*. Technical report, Department of Computer Science, California Institute of Technology, 1989.

[124] A.J. Martin, S. Burns, T.K. Lee, D. Borkovic, and P.J. Hazewindus. The design of an asynchronous microprocessor. In *Advanced Research in VLSI*, pages 351–373, 1989.

[125] A.J. Martin, M. Nystrom, and C.G. Wong. Three generations of asynchronous microprocessors. *IEEE Design and Test*, 20:9–17, 2003.

[126] M.M.K. Martin, M.D. Hill, and D.A. Wood. Token coherence: decoupling performance and correctness. In *International Symposium on Computer Architecture (ISCA)*, pages 182–193, 2003.

[127] H. Matsutani, M. Koibuchi, H. Amano, and T. Yoshinaga. Prediction router: Yet another low latency on-chip router architecture. In *International Conference on High-Performance Computer Architecture (HPCA)*, pages 367–378, 2009.

[128] P.B. McGee, M.Y. Agyekum, M.A. Mohamed, and S.M. Nowick. A level-encoded transitions signaling protocol for high-throughput asynchronous global communication. In *International Symposium of Asynchronous Circuits and Systems (ASYNC)*, pages 116–127, 2008.

[129] M. McKeown, A. Lavrov, M. Shahrad, P.J. Jackson, Y. Fu, J. Balkind, T.M. Nguyen, K. Lim, Y. Zhou, and D. Wentzlaff. Power and energy characterization of an open source 25-core

manycore processor. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 762–775, 2018.

[130] P. Merolla, J. V. Arthur, R. Alvarez-Icaza, J. Bussat, and K. Boahen. A multicast tree router for multichip neuromorphic systems. *IEEE Transactions on Circuits and Systems*, 61-I:820–833, 2014.

[131] P. Messina and S. Lee. The U.S. Exascale Computing Project. *https://exascaleproject.org/wp-content/uploads/2017/03/Messina_ECP-IC-Mar2017-compressed.pdf*, 2017.

[132] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *Design, Automation and Test in Europe (DATE)*, pages 1–6, 2004.

[133] D.A.B. Miller. Device requirements for optical interconnects to silicon chips. *Proceedings of the IEEE*, 97:1166–1185, 2009.

[134] G. Miorandi, M. Balboni, S.M. Nowick, and D. Bertozzi. Accurate assessment of bundled-data asynchronous NoCs enabled by a predictable and efficient hierarchical synthesis flow. In *International Symposium of Asynchronous Circuits and Systems (ASYNC)*, pages 10–17, 2017.

[135] G. Miorandi, D. Bertozzi, and S.M. Nowick. Increasing impartiality and robustness in high-performance n-way asynchronous arbiters. In *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 108–115, 2015.

[136] G. Miorandi, A. Ghiribaldi, S.M. Nowick, and D. Bertozzi. Crossbar replication vs. sharing for virtual channel flow control in asynchronous NoCs: A comparative study. In *International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6, 2014.

[137] S.W. Moore and P. Robinson. Rapid prototyping of self-timed circuits. In *International Conference on Computer Design (ICCD)*, pages 360–365, 1998.

[138] S. Moradi, N. Imam, R. Manohar, and G. Indiveri. A memory-efficient routing method for large-scale spiking neural networks. In *European Conference on Circuit Theory and Design (ECCTD)*, pages 1–4, 2013.

[139] M. Moreira, A. Neutzling, M. Martins, A. Reis, R. Ribas, and N. Calazans. Semi-custom NCL design wth commercial EDA frameworks: Is it possible? In *International Symposium of Asynchronous Circuits and Systems (ASYNC)*, pages 53–60, 2014.

[140] A. Moreno and J. Cortadella. Synthesis of all-digital delay lines. In *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 75–82, 2017.

[141] T.P. Morgan. Intel Knights Landing yields big bang for the buck jump. *https://www.nextplatform.com/2016/06/20/intel-knights-landing-yields-big-bang-buck-jump/*, 2016.

[142] S.T. Muhammad, M.A. El-Moursy, A.A. El-Moursy, and H.F.A. Hamed. Architecture level analysis for process variation in synchronous and asynchronous networks-on-chip. *Journal of Parallel and Distributed Computing*, 102:175–185, 2017.

[143] D.E. Muller and W.S. Bartky. A theory of asynchronous circuits. In *International Symposium on the Switching Theory in Harvard University*, pages 204–243, 1959.

[144] J. Navaridas, M. Luján, L.A. Plana, S. Temple, and S.B. Furber. On generating multicast routes for SpiNNaker. In *Computing Frontiers Conference (CF)*, pages 2:1–2:10, 2014.

[145] S.M. Nowick and D.L. Dill. Synthesis of asynchronous state machines using a local clock. In *International Conference on Computer Design (ICCD)*, pages 192–197, 1991.

[146] S.M. Nowick and D.L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14:986–997, 1995.

[147] S.M. Nowick and M. Singh. Asynchronous design - part 1: overview and recent advances. *IEEE Design & Test*, 32:5–18, 2015.

[148] S.M. Nowick and M. Singh. Asynchronous design - part 2: system and methodologies. *IEEE Design & Test*, 32:19–28, 2015.

[149] S. Park, T. Krishna, C.H. Owen Chen, B.K. Daya, A. Chandrakasan, and L.S. Peh. Approaching the theoretical limits of a mesh NoC with a 16-node chip prototype in 45 nm SOI. In *Design Automation Conference (DAC)*, pages 398–405, 2012.

[150] L.S. Peh and W.J. Dally. A delay model and speculative architecture for pipelined routers. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 255–266, 2001.

[151] M.D.V. Pena, J.J. Rodriguez-Andina, and M. Manic. The internet of things: the role of reconfigurable platforms. *IEEE Industrial Electronics Magazine*, 11:6–19, 2017.

[152] L.A. Plana and S.H. Unger. Pulse-mode macromodular systems. In *International Conference on Computer Design (ICCD)*, pages 348–353, 1998.

[153] S. Poddar, P. Ghosal, and H. Rahaman. Adaptive CDMA based multicast method for photonic networks on chip. In *International System-on-Chip Conference (SOCC)*, pages 298–303, 2015.

[154] J. J. H. Pontes, P. Vivet, and Y. Thonnart. Two-phase protocol converters for 3D asynchronous 1-of-n data links. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 154–159, 2015.

[155] G. Pouiklis and G. C. Sirakoulis. Clock gating methodologies and tools: a survey. *I. J. Circuit Theory and Applications*, 44:798–816, 2016.

[156] V. Puente, J. A. Gregorio, F. Vallejo, and R. Beivide. Immunet: A cheap and robust fault-tolerant packet routing mechanism. In *International Symposium on Computer Architecture (ISCA)*, pages 198–211, 2004.

[157] J. Quartana, S. Renane, A. Baixas, L. Fesquet, and M. Renaudin. GALS systems prototyping using multiclock FPGAs and asynchronous network-on-chips. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 299–304, 2005.

[158] A. Rahimi, I. Loi, M.R. Kakoee, and L. Benini. A fully-synthesizable single-cycle interconnection network for shared-L1 processor clusters. In *Design, Automation and Test in Europe (DATE)*, pages 491–496, 2011.

[159] D. Rahmati, H. Sarbazi-Azad, S. Hessabi, and A.E. Kiasari. Power-efficient deterministic and adaptive routing in torus networks-on-chip. *Microprocessors and Microsystems - Embedded Hardware Design*, 36:571–585, 2012.

[160] M. Renaudin and A. Fonkoua. Tiempo asynchronous circuits system verilog modeling language. In *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 105–112, 2012.

[161] J.J. Rodríguez-Andina, M.D. Valdes-Pena, and M.J. Moure. Advanced features and industrial applications of fpgas - a review. *IEEE Transactions on Industrial Informatics*, 11:853–864, 2015.

[162] D. Rostislav, V. Vishnyakov, E. Friedman, and R. Ginosar. An asynchronous router for multiple service levels networks on chip. In *International Symposium of Asynchronous Circuits and Systems (ASYNC)*, pages 1–10, 2005.

[163] P. Russell, J. Döge, C. Hoppe, T.B. Preußer, P. Reichel, and P. Schneider. Implementation of an asynchronous bundled-data router for a GALS NoC in the context of a VSoC. In *International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 195–200, 2017.

[164] F.A. Samman, T. Hollstein, and M. Glesner. Multicast parallel pipeline router architecture for network-on-chip. In *Design, Automation and Test in Europe (DATE)*, pages 1396–1401, 2008.

[165] F.A. Samman, T. Hollstein, and M. Glesner. Adaptive and deadlock-free tree-based multicast routing for networks-on-chip. *IEEE Transactions on VLSI Systems*, 18:1067–1080, 2010.

[166] K. Sankaralingam, R. Nagarajan, R.G. McDonald, R. Desikan, S. Drolia, M.S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S.W. Keckler, and D. Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *International Symposium on Microarchitecture (MICRO)*, pages 480–491, 2006.

[167] D. Seo, A. Ali, W. Lim, N. Rafique, and M. Thottethodi. Near-optimal worst-case throughput routing for two-dimensional mesh networks. In *International Symposium on Computer Architecture (ISCA)*, pages 432–443, 2005.

[168] A. Shacham, K. Bergman, and L.P. Carloni. Photonic network-on-chip for future generations of chip multiprocessors. *IEEE Transactions on Computers*, 57:1246–1260, 2008.

[169] L. Shang, L.S. Peh, and N.K. Jha. Dynamic voltage scaling with links for power optimization of interconnection networks. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 91–102, 2003.

[170] K.S. Shim, M.H. Cho, M.A. Kinsy, T. Wen, M. Lis, G.E. Suh, and S. Devadas. Static virtual channel allocation in oblivious routing. In *International Symposium on Networks-on-Chips (NOCS)*, pages 38–43, 2009.

[171] M. Singh and S.M. Nowick. MOUSETRAP: high-speed transition-signaling asynchronous pipelines. *IEEE Transactions on VLSI Systems*, 15:684–698, 2007.

[172] M. Singh, J.A. Tierno, A. Rylyakov, S.V. Rylov, and S.M. Nowick. An adaptively pipelined mixed synchronous-asynchronous digital FIR filter chip operating at 1.3 Gigahertz. *IEEE Transactions on VLSI Systems*, 18:1043–1056, 2010.

[173] B. Sinharoy, J.A. Van Norstrand, R.J. Eickemeyer, H.Q. Le, J. Leenstra, D.Q. Nguyen, B. Konigsburg, K. Ward, M.D. Brown, J.E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J.W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K.M. Fernsler. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development*, 59:2:1–21, 2015.

[174] C. Sitik, W. Liu, B. Taskin, and E. Salman. Design methodology for voltage-scaled clock distribution networks. *IEEE Transactions on VLSI Systems*, 24:3080–3093, 2016.

[175] R. Sivaram, C.B. Stunkel, and D.K. Panda. A reliable hardware barrier synchronization scheme. In *International Parallel Processing Symposium (IPPS)*, pages 274–280, 1997.

[176] W. Song, G. Zhang, and J.D. Garside. On-line detection of the deadlocks caused by permanently faulty links in quasi-delay insensitive networks on chip. In *Great Lakes Symposium on VLSI (GLSVLSI)*, pages 211–216, 2014.

[177] S. Stergiou, F. Angiolini, S. Carta, L. Raffo, D. Bertozzi, and G.D. Micheli. xpipesLite: A synthesis oriented design library for networks on chip. In *Design, Automation and Test in Europe (DATE)*, pages 1188–1193, 2005.

[178] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers. The wavescalar architecture. *ACM Transactions on Computing Systems*, pages 4:1–4:54, 2007.

[179] A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, and T. Nanya. TITAC-2: An asynchronous 32-Bit microprocessor based on scalable-delay-insensitive model. In *International Conference on Computer Design (ICCD)*, pages 288–294, 1997.

[180] K. Takizawa, S. Hosaka, and H. Saito. A design support tool set for asynchronous circuits with bundled-data implementation on FPGAs. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2014.

[181] M.B. Taylor, W. Lee, S.P. Amarasinghe, and A. Agarwal. Scalar operand networks: on-chip interconnect for ILP in partitioned architecture. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 341–353, 2003.

[182] P. Teehan, M. Greenstreet, and G. Lemieux. A survey and taxonomy of GALS design styles. *IEEE Design and Test*, 24:418–428, 2007.

[183] J. Teifel and R. Manohar. Highly pipelined asynchronous FPGAs. In *International Symposium on Field Programmable Gate Arrays (ISFPGA)*, pages 133–142, 2004.

[184] Y. Thonnart, E. Beigne, and P. Vivet. A pseudo-synchronous implementation flow for WCHB QDI asynchronous circuits. In *International Symposium of Asynchronous Circuits and Systems (ASYNC)*, pages 73–80, 2012.

[185] Y. Thonnart, P. Vivet, and F. Clermidy. A fully-asynchronous low-power framework for GALS NoC integration. In *Design, Automation and Test in Europe (DATE)*, pages 33–38, 2010.

[186] A.W. Topol, D.C. La Tulipe, L. Shi, D.J. Frank, K. Bernstein, S.E. Steen, A. Kumar, G.U. Singco, A.M. Young, K.W. Guarini, and M. Leong. Three-dimensional integrated circuits. *IBM Journal of Research and Development*, 50:491–506, 2006.

[187] S.H. Unger. *Asynchronous Sequential Switching Circuits*. New York, NY: Wiley, 1969.

[188] D. Vainbrand and R. Ginosar. Network-on-chip architectures for neural networks. In *International Symposium Networks-on-Chip (NOCS)*, pages 135–144, 2010.

[189] S.R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS. In *International Solid-State Circuits Conference (ISSCC)*, pages 98–589, 2007.

[190] P. Vivet, Y. Thonnart, R. Lemaire, C. Santos, E. Beigne, C. Bernard, F. Darve, D. Lattard, I. M.-Panades, D. Dutoit, F. Clermidy, S. Cheramy, A. Sheibanyrad, F. Petrot, E. Flamand, J. Michailos, A. Arriordaz, L. Wang, and J. Schloeffel. A 4x4x2 homogeneous scalable 3D network-on-chip circuit with 326 MFlit/s 0.66 pJ/b robust and fault tolerant asynchronous 3D links. *IEEE Journal of Solid-State Circuits*, 52:33–49, 2017.

[191] C. Wang, X. Li, P. Chen, A. Wang, X. Zhou, and H. Yu. Heterogeneous cloud framework for big data genome sequencing. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 12:166–178, 2015.

[192] H. Wang, L.S. Peh, and S. Malik. Power-driven design of router microarchitectures in on-chip networks. In *International Symposium on Microarchitecture*, pages 105–116, 2003.

[193] L. Wang, Y. Jin, H. Kim, and E.J. Kim. Recursive partitioning multicast: a bandwidth-efficient routing for networks-on-chip. In *International Symposium on Networks-on-Chip (NOCS)*, pages 64–73, 2009.

[194] X. Wang, T. Ahonen, and J. Nurmi. Prototyping a globally asynchronous locally synchronous network-on-chip on a conventional FPGA device using synchronous design tools. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, 2006.

[195] X. Wang, T. Ahonen, and J. Nurmi. Applying CDMA technique to network-on-chip. *IEEE Transactions on VLSI Systems*, 15:1091–1100, 2007.

[196] P. Warden. Why are eight bits enough for deep neural networks. *https://petewarden.com/2015/05/23/why-are-eight-bits-enough-for-deep-neural-networks/*, 2015.

[197] D. Wiklund and D. Liu. Switched interconnect for system-on-a-chip designs. In *IP 2000 Europe Conference (IP2000)*, pages 1–6, 2000.

[198] T.E. Williams and M.A. Horowitz. A zero-overhead self-timed 160-ns 54-b CMOS divider. *IEEE Journal of Solid-State Circuits*, 26:1651–1661, 1991.

[199] C.G. Wong, A.J. Martin, and P. Thomas. An architecture for asynchronous FPGAs. In *International Conference on Field Programmable Technology (FPT)*, pages 170–177, 2003.

[200] X. Xiang, W. Shi, S. Ghose, L. Peng, O. Mutlu, and N. Tzeng. Carpool: a bufferless on-chip network supporting adaptive multicast and hotspot alleviation. In *International Conference on Supercomputing (ICS)*, pages 19:1–19:11, 2017.

[201] J. Xue, A. Garg, B. Ciftcioglu, J. Hu, S. Wang, I. Savidis, M. Jain, R. Berman, P. Liu, M. C. Huang, H. Wu, E. G. Friedman, G. Wicks, and D. Moore. An intra-chip free-space optical interconnect. In *International Symposium on Computer Architecture (ISCA)*, pages 94–105, 2010.

[202] S. Yan and B. Lin. Design of application-specific 3d networks-on-chip architectures. In *International Conference on Computer Design (ICCD)*, pages 142–149, 2008.

[203] K.Y. Yun, P.A. Beerel, and J. Arceo. High-performance asynchronous pipeline circuits. In *International Symposium of Asynchronous Circuits and Systems (ASYNC)*, pages 17–28, 1996.

[204] C.A. Zeferino and A.A. Susin. Socin: A parametric and scalable network-on-chip. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*, page 169, 2003.

[205] G. Zhang, J.D. Garside, W. Song, J. Navaridas, and Z. Wang. Deadlock recovery in asynchronous networks on chip in the presence of transient faults. In *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 100–107, 2015.

[206] G. Zhang, W. Song, J.D. Garside, J. Navaridas, and Z. Wang. An asynchronous SDM network-on-chip tolerating permanent faults. In *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 9–16, 2014.

[207] J. Zhang, F. Guangbo, A. He, and H. Chen. From click based asynchronous design to Xilinx FPGA. In *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2018.

[208] Y. Zhang, T. Jeong, F. Chen, H. Wu, R. Nitzsche, and G.R. Gao. A study of the on-chip interconnection network for the IBM Cyclops64 multi-core architecture. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, 2006.

[209] Z. Zhang, A. Greiner, and S. Taktak. A reconfigurable routing algorithm for a fault-tolerant 2d-mesh network-on-chip. In *Design Automation Conference (DAC)*, pages 441–446, 2008.