Learning Design Parameters to Build Application Customizable Network-on-Chips for FPGAs

by

Gurshaant Singh Malik

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2024

© Gurshaant Singh Malik 2024

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Samuel Bayliss Fellow, Software Engineering, AMD
Supervisor(s):	Nachiket Kapre Assoc. Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
Internal Member:	Mark Aagaard Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
Internal Member:	Andrew Morton Assoc. Professor, Teaching Stream, Dept. of Electrical and Computer Engineering, University of Waterloo
Internal-External Member:	Troy Vasiga Associate Professor, Teaching Stream, Cheriton School of Computer Science, Associate Dean, Undergraduate Admissions and Outreach, University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Gurshaant Singh Malik is the sole author of this thesis, which was written under the supervision of Prof. Nachiket Kapre. The Butterfly Fat Tree work of Chapter 4 was written in close collaboration with Prof. Kapre. The latency and backlog analysis presented in section 5.4 was developed by Prof. Rodolfo Pellizzoni and Tushar Garg. The implementation of the HopliteBuf+BP hybrid NoC on the Zynq SoC in Chapter 5 was carried out by Ian Elmor Lang.

Abstract

We can exploit configurability of Field Programmable Gate Arrays (FPGA) and maximize the performance of communication-intensive FPGA applications by designing specifically customized Network-on-Chips (NoCs) using Machine Learning (ML). As transistor density growth stalls, NoCs play an increasingly critical role in deployment of FPGA applications for modern-day use cases. Unlike Application-Specific Integrated Circuits (ASICs), FPGA configurability allows the design of application-aware NoCs that can outperform statically configured NoCs in terms of both performance and efficiency. Conventional NoC design process is typically centered around universally-sound one-size-fits-all NoC design decisions and does not take the underlying application into account. In contrast, we present application aware designs that *learn* their NoC parameters by casting the NoC design space as a function of application performance using ML algorithms.

Complex and non-obvious relationships between the large search space of NoC parameters and performance of the underlying FPGA application necessitates a more efficient approach than manual hand-tuning or brute force. Modern ML algorithms have demonstrated a remarkable ability to generalize to complex representations of the world by extracting high-order, non-linear features from complex inputs. In this thesis, we identify 1) NoC topology, 2) Flow control and 3) Regulation rate, as the key NoC design variables that have the strongest influence on application performance and leverage two primary ML methodologies in this thesis: 1) Stochastic Gradient Free Evolutionary Learning and 2) Gradient based Supervised Learning.

First, we present NoC designs based on Butterfly Fat Tree (BFT) topology and light weight flow control. These BFT-based NoCs can customize their bisection bandwidth to match the application being routed while providing features such as in-order delivery and bounded packet delivery times. We present the design of routers with 1) latency-insensitive interfaces, coupled with 2) deterministic routing policy, and 3) round-robin scheduling at NoC ports. We evaluate our NoC designs under various conditions to deliver up to 3× lower latency and 6× higher throughput. We also *learn* the routing policy on a per-switch basis in an application-aware manner using Maximum Likelihood Estimation, decreasing latencies by a further ≈1.1–1.7× over the static policy.

Second, we overcome the pessimism in routing analysis of timing-predictable NoCs through the use of a "hybrid" application-customized NoCs. HopliteBuf NoCs leverage stall-free FIFOs as a measure of flow control under token bucket regularization. The static analysis, in the worst-case, can deliver very large FIFO size and latency bounds. Alternatively, HopliteBP uses light-weight backpressure as flow control under similar injection

regulation. But, it suffers from severely pessimistic static analysis due to propagation of backpressure to other switches. We show that a hybrid FPGA NoC that seamlessly composes both design styles on a per-switch basis, delivers the best of both worlds. We learn, specifically for the application being routed, the switch configuration through a novel evolutionary algorithm based on Maximum Likelihood Estimation (MLE). We demonstrate ≈1–6.8× lower routing latencies and ≈2–3× improvements in feasibility, while only consuming ≈1–1.5× more FPGA resources.

Third, we further improve routability of a workload on the hybrid Buf-BP Hoplite NoC by learning to tune regulation rates for each traffic trace. We model the regulation space as a multivariate gaussian distribution. We capture critical dependency between parameters of the multivariate distribution using Covariance Matrix Adaptation Evolution Strategy (CMA-ES). We also propose *nested* learning, that learns switch configurations and regulation rates in-tandem, and further lower cost-constrained latency by ≈1.5× and accelerate rates by ≈3.1×.

Finally, we propose a Graph Neural Network (GNN) based framework to accurately predict NoC performance in *sub-second* latencies for a variety of FPGA NoC designs and applications. Application-aware NoC design can include thousands of incremental updates to the NoC design space, with each step requiring performance evaluation of NoC configuration using slow and expensive conventional tooling. This presents a bottleneck in the adoption of application-aware FPGA NoC design. Instead of spending up to tens of wall clock minutes simulating the NoC design for each step, we present a GNN based framework to encode any FPGA NoC and any FPGA application into graphs. We create a dataset, consisting of over 1.5 million samples, to train GNNs to predict NoC routing latencies. GNNs accelerate benchmarking run-times by up to ≈148× (≈506× on GPU) while preserving accuracies as high as 97.2%.

Through this work, we observe that application-aware NoCs designed using ML algorithms such as MLE and CMA-ES can decrease routing latency by ≈2.5–10.2×, increase workload feasibility by ≈2–3×, increase injection rates by up to ≈3.1×. By leveraging GNNs trained using supervised learning, we can accelerate design time of such NoCs by up to ≈4.3×.

Acknowledgements

I stand on the shoulder of giants

As I attempt to do justice in my feeble attempt to acknowledge and thank everyone who has helped me in this journey, I have 2 main thoughts racing through my brain:

1. How lucky am I that I have had the absolute highest privilege of generally being surrounded by the brightest and kindest people one could ever ask for? Folks who have been patient in my failures, rejoiced in my success and inspired me to better different aspects of my character.

2. No matter what I write below, it won't come close to capturing or framing my gratefulness for this journey and the people who made it so satisfying.

First, I would like to thank Kinjal: my wife, my best friend, my confidant and my biggest pillar of support. Thank you for your grace. Thank you for your patience. Thank you for that brightest smile that never fails to cheer me up. Thank you for listening to my rants. Thank you for giving me the best possible advice I could get. Thank you for always saying the right things, even when I did not know I needed to hear them.

Second, I would like to thank my parents, without whose unwavering support and sacrifices, I probably would not have achieved a fraction of what I set out to do. I am grateful to have grown up in an environment where I could push myself but never be afraid of failures. Where success was not a question of *if* I can do it but *how* I can do it.

Third, I would like to express my heartfelt gratitude to my supervisor, Prof. Nachiket Kapre. I am thankful for his guidance and patience. His insights and feedback over the past seven years have been pivotal in my growth as a better thinker and writer. I am thankful for his support as I ventured into unfamiliar research territory and his belief in me when I experienced setbacks in terms of paper rejections. It has been an absolute privilege to work with him.

Fourth, I would like to thank Prof. Chris Eliasmith, Dr. Peter Blouw, Dr. Daniel Rasmussen, Dr. Xuan Choo, Dr. Travis DeWolf, Dr. Trevor Bekolay, Dr. Eric Hunsberger and Peter Suma for their support in allowing me to dive deep and wide into the field of Machine Learning. Many foundational concepts that I continue to leverage in my career were learnt thanks to the patience and encouragement of these kind folks. I will be always grateful for their support.

A special thanks to Ben Morcos for being a constant source of good vibes and introducing me to the kind folks at ABR. I have now known Ben for over 7 years, at WATCAG and beyond. His calm demeanour, amazing work ethic have and continue to inspire me to be a better version of myself. Finally, a very big shoutout to Prarthana, Amit, Michael, Reza, Yuba and Indrani. I truly cherish their friendship and they made Kitchener-Waterloo a place worth calling home.

Dedication

To Kinjal, Mom and Dad.

Table of Contents

Examining Committee ii

Author's Declaration iii

Statement of Contributions iv

Abstract v

Acknowledgements vii

Dedication ix

List of Figures xv

List of Tables xx

1 Introduction 1

1.1 Organization and Research Themes 3

1.1.1 List of Publications 6

2 Background: Network on Chips 8

2.1 History of NoCs 8

2.2 Basics of NoC Architecture 10

2.3 Encoding Applications into Traffic Traces . . . 14
2.4 Butterfly Fat Tree based FPGA NoC . . . 16
2.5 Hoplite FPGA NoC . . . 20
2.6 Survey of Relevant FPGA NoCs . . . 25

3 Background: Machine Learning **27**
3.1 Gradient based Supervised Learning with Graph Neural Networks . . . 28
3.1.1 Basics of Supervised Learning with Neural Networks . . . 29
3.1.2 Graph Neural Networks . . . 33
3.2 Stochastic Gradient Free Evolutionary Learning . . . 36
3.2.1 Bayesian Statistics . . . 37
3.2.2 Covariance Matrix Adaptation Evolution Strategy . . . 39
3.2.3 Maximum Likelihood Estimation . . . 40

4 Butterfly Fat Tree FPGA Overlay NoCs **43**
4.1 Introduction . . . 44
4.2 Motivation . . . 45
4.3 BFTs with lightweight flow-control . . . 47
4.3.1 Designing the BFT Switch Micro-architecture . . . 47
4.3.2 FPGA Mapping . . . 50
4.4 Results . . . 51
4.4.1 Cost-Power-Performance Tradeoffs . . . 52
4.4.2 Understanding FPGA Mapping Trends . . . 54
4.4.3 Effect of Injection Rate of Synthetic Traffic . . . 56
4.4.4 Effect of System Size . . . 57
4.4.5 Routing Realistic Workloads . . . 59
4.4.6 Choosing a BFT that is best for you . . . 61
4.4.7 Analyzing BFTs with SRL FIFOs . . . 63
4.5 Conclusions . . . 65

5 HopliteBuf+HopliteBP: Evolving Application Customizable FPGA NoCs 66

5.1 Introduction . . . 67
5.2 Motivating Example . . . 69
5.3 Design of the HopliteBuf+BP Switch . . . 71
5.4 Latency and Backlog Analysis . . . 72
5.4.1 Flow Model . . . 73
5.4.2 Traffic regulation . . . 73
5.4.3 Injection Latency . . . 74
5.4.4 Analysis for FIFO Mode . . . 75
5.4.5 Analysis for Backpressure Mode . . . 76
5.4.6 Hybrid NoC Analysis . . . 76
5.5 Evolutionary Learning of Switch Configuration . . . 77
5.6 Evaluation . . . 81
5.6.1 Methodology . . . 81
5.6.2 Results . . . 82
5.6.2.1 Learning for Feasibility . . . 83
5.6.2.2 Learning for Worst-Case Latency . . . 85
5.6.2.3 Learning for Cost Constrained Worst-Case Latency . . . 86
5.6.2.4 Routing Real World FPGA Applications . . . 88
5.6.2.5 Analysing Efficiency of MLE . . . 89
5.6.2.6 Adapting MLE for NoCs with different topologies and parameters . . . 90
5.6.2.7 Comparing an ASIC implementation with Timing Predictable NoCs . . . 92
5.7 Conclusions . . . 93

6 Learning Injection Rates and Switch Configuration 94
6.1 Introduction . . . 94
6.2 Motivating Example . . . 96
6.3 Evolutionary Learning of Regulation Rates . . . 98
6.3.1 Learning Regulated Injection Rates . . . 99
6.3.1.1 Univariate Rate Learning . . . 99
6.3.1.2 Multivariate Rate Learning . . . 100
6.3.2 Nested Learning . . . 102
6.4 Evaluation . . . 105
6.4.1 Methodology . . . 105
6.4.2 Results . . . 106
6.4.2.1 In-Tandem Learning Regulation Rates and Switch Configurations . . . 106
6.4.2.1.1 Routing RANDOM, LOCAL and REAL Benchmarks: . . . 107
6.4.2.1.2 Analyzing Learnt Rates: . . . 111
6.4.2.2 Learning Regulation Rates for Vanilla Hoplite NoCs . . . 112
6.4.2.2.1 Analyzing Distribution of Latency Improvement: 112
6.4.2.2.2 Routing REAL Benchmarks: . . . 113
6.4.2.2.3 Understanding Multivariate Rate Learning: . . . 114
6.5 Conclusions . . . 116

7 GraphNoC: Graph Neural Networks for Accelerating NoC Evaluation 117
7.1 Introduction . . . 118
7.2 Background . . . 119
7.2.1 Alternatives to RTL Simulation . . . 119
7.2.2 Motivating Graph Neural Networks for NoCs . . . 121
7.2.3 Motivating Example: GNNs vs Conventional NN Models . . . 122

7.3 The GNN based GraphNoC framework . . . 123
7.3.1 Encoding NoC and Application into Graphs . . . 123
7.3.1.1 Layer 1: Encoding FPGA NoC Switches into Graphs . . . 124
7.3.1.2 Layer 2: Encoding NoC Topology into Graphs . . . 125
7.3.1.3 Layer 3: Encoding Application Traffic into Graphs . . . 125
7.3.2 Message Passing Graph Neural Networks . . . 125
7.4 Constructing the GraphNoC Dataset . . . 127
7.5 Experimental Setup and Training Recipes . . . 129
7.6 Evaluation . . . 130
7.6.1 Measuring Accuracy of GraphNoC GNN Models . . . 130
7.6.2 Benchmarking Throughput of GraphNoC GNN Models . . . 131
7.6.3 Application-Specific FPGA NoC's Design with GraphNoC . . . 134
7.6.4 Building Foundation GNN Models with GraphNoC . . . 137
7.7 Conclusion . . . 138

8 Conclusion **139**
8.1 Summary of Contributions . . . 139
8.2 Future Work . . . 142

References **146**

List of Figures

2.1 Visual representation of some common NoC Topologies. 11

2.2 Different BFT topologies benchmarked in this work. The placement and type of switches influences the routing bandwidth available to different packets based on their position in the NoC. 17

2.3 Micro-architecture of BFT t and pi switches. 18

2.4 Visual representation of the routing path of a packet for a BFT NoC. . . . 19

2.5 Visual Representation of localized deflections at a `pi` switch. 20

2.6 2D unidirectional torus topology of the Hoplite NoC. 20

2.7 Block diagram of the original deflection based Hoplite switch. 21

2.8 Visual representation of the DOR routing policy of Hoplite 22

2.9 Block diagram of the original deflection based Hoplite switch with shared East exit. 22

2.10 Block diagram of the HopliteBuf switch. 23

2.11 Block diagram of the HopliteBP switch. 24

3.1 Visual representation of a Linear layer. 30

3.2 Visual representation of a 2D convolution layer. 31

3.3 A representative example of an input graph and operations of a GNN layer. Operations related to f_n and f_e correspond to examples of message passing for node and edge attributes respectively. The incoming ----→ line shows the updated attributes of the example node and edge after the message passing operation. 34

3.4 A representative example of a sample space consisting of complimentary hypothesis and evidence sets. The **green boxes** corresponds to all samples satisfying hypothesis H_1, with **darker green boxes** further showing evidence E_2. Similarly, the **orange boxes** corresponds to all samples satisfying hypothesis H_2, with **darker orange boxes** further showing evidence E_2. . 38

4.1 High-Level Picture of the Butterfly Fat Tree (BFT) NoC. Configurable bisection bandwidth N^p with p=Rent parameter. Deterministic $A \rightarrow B$ path highlighted. Parametric pipelining of long wires in upper tree stages. . . . 44

4.2 Pruning the *pi* switch to enforce deterministic routing policy and save multiplexing cost. 48

4.3 Livelock in BFTs: mixing deflection and determinism. 49

4.4 Lightweight Latency-Insensitive Interface integration with the BFT t switch. Each input port has a cheap shadow register (single FF, or an SRL32 absorption FIFO). 50

4.5 FPGA LUT cost-performance trends for different NoCs. 16 clients, RANDOM traffic at 100% injection. 53

4.6 FPGA Dynamic Power-performance trends for different NoCs. 16 clients, RANDOM traffic at 100% injection. 54

4.7 FPGA frequency characterization as a function of system size and datawidth. 55

4.8 Sustained Throughput trends for different NoCs for 16 clients and for RANDOM and LOCAL traffic patterns. 56

4.9 NoC packet latency trends for 64 client NoCs using RANDOM traffic pattern. 57

4.10 PE scaling impact on throughput and worst-case latency for RANDOM and real workloads (geomean). 59

4.11 Throughput-Latency trends for REAL SpMV traffic. 60

4.12 Throughput-Latency trends for REAL Graph traffic. 61

4.13 Selecting the BFT that is right for you. Evaluating FPGA LUT cost and Dynamic Power tradeoffs for different system sizes, RANDOM traffic pattern. 62

4.14 FPGA LUT cost and Dynamic Power tradeoffs for different NoC configurations. 64 clients, RANDOM traffic at 100% injection rate 64

5.1 Interacting network flows on a 4×4 NoC arranged to show the benefits of mixing FIFO and Backpressure switches. 70

5.2 Block diagrams of HopliteBuf+BP design. 71

5.3 FPGA NoC Switch Configuration for a 3×3 NoC for a set of nine flows (shown in leftmost subfigure) with identical but four combinations of rate ρ and block size b characteristics. At low rates and block sizes, most switches tend to be cheap HopliteBuf ■ mode of operation, while a growing number convert to expensive HopliteBP □ mode of operation with increasing rate+size. 78

5.4 Evolution of switch type probabilities for a 2×2 NoC using Maximizing Likelihood Estimation (MLE). One MLE iteration is shown which has a candidate size C=8 (each of the 8 shown 2x2 Hoplite NoC configurations is a candidate) and elite size λ=4. Only ■ marked NoC configurations are used to generate the new Bernoulli distribution B^{g+1}. 80

5.5 The complete MLE algorithm is shown for a regulation rate of 0.5 and a block size of 16. Switch colors that settle for white are Backpressure switches while those that are Black are FIFO-based. 81

5.6 Feasible set of RANDOM and LOCAL flowsets on system sizes 3×3–7×7 (*across columns*) and data block sizes 1–16 (*across rows*). Note that x and y axes are shared vertically and horizontally respectively. 84

5.7 Worst-case latency as a function of regulation rate for 100 synthetic LOCAL flowsets for various data block sizes. 86

5.8 LUT Cost-Worst Case Latency tradeoffs for a 4×4 NoC with 0.17 regulation rate for 100 synthetic RANDOM and LOCAL flowsets across various block sizes. 87

5.9 Cumulative distribution of improvements of worst-case latency MLE NoC over HopliteBuf and HopliteBP 88

5.10 Latency Scaling of a 4x4 NoC for Graph/ SpMV traces 89

5.11 Solution quality and time taken by MLE optimizer across flowsets mapped to a 5×5 NoC with 0.1 regulation rate 90

5.12 Analyzing MLE's ability in optimising for the parameter design space of a Butterfly Fat Tree crossbar NoC. 91

5.13 Average Latency vs Rate. Phase and SurfNoC from [1] 93

6.1 Interacting network flows on a 4×4 NoC arranged to show the benefits of learning regulation rates for each traffic trace 97

6.2 Evolution of a 2D gaussian distribution when learning regulation rates for a 2x2 HopliteBP NoC with 2 traces of data block size 1 in a RANDOM flowset. Note the convergence of the distribution over epochs as CMA-ES progressively learns better solutions. This is a toy example for illustrative purposes. We evaluate our design on larger NoC sizes in section 6.4 103

6.3 QoR achieved by univariate and multivariate rate learning strategies, compared for data block sizes of 4, 8 and NoC sizes of 4x4, 5x5 for 10 different RANDOM traces. 108

6.4 QoR achieved by univariate and multivariate rate learning strategies, compared for data block sizes of 4, 8 and NoC sizes of 4x4, 5x5 for 10 different LOCAL traces. 109

6.5 QoR achieved by univariate and multivariate, compared for data block sizes of 1,2 and 4 (columns) and NoC sizes of 3 and 4 (rows) for REAL traces. . 110

6.6 Range (as whisker plot) and average rates of learnt multivariate rates on a *per trace basis* along with univariate rates for REAL benchmarks for a NoC of size 4x4, over a range of data block sizes (burst). 111

6.7 Latency improvements of multivariate learning over univariate learning, for vanilla HopliteBuf (fifo) and HopliteBP (bp) NoCs of size 4x4, routing RANDOM, LOCAL and REAL traces. 112

6.8 Univariate and multivariate rate learning compared for vanilla HopliteBuf and HopliteBP NoCs of size 4x4 while routing REAL traces of data block size of 1,2 and 4. "NF" tagged flowsets could not be routed feasibly for both learning types. 114

6.9 Chronological updates to mean and standard deviation (SD) of a multivariate gaussian distribution for a 4x4 HopliteBP NoC routing the roadnet benchmark with a data block size of 1. Note the 3 stages of learning: 1. Rate Halving, 2. Univariate and finally the dominant 3. Multivariate. Also note the 0 SD for rate halving. 115

7.1 Foundation of Message Passing GNNs. 121

7.2 Loss (**Y axis**) over epochs (**X axis**) for **ANN** vs **GNN** 122

7.3 Hoplite [2] and BFT [3] switches encoded as graphs. 124

7.4 Encoding a 3×3 Hoplite NoC and application being routed. Notice the different edge types: port routes (——›), switch (‹·····›), topology (⟶) and application (⟶). 126

7.5 Design of GNN models: GraphSage, followed by residual application of classification and regression MLPs. 126

7.6 Throughput (**Y axis**, log_2 scale) as a function of batch size (**X axis**) for GNNs and conventional tools for HopliteBuf/BP NoCs. 132

7.7 Learnt NoC's latency (**Y axis**) vs seconds spent (**X axis**) completing MLE loop for HopliteBuf/BP NoC. 136

7.8 Distribution of ground truth vs latency predictions. 138

8.1 A Visual representation of the hard but configurable Versal NoC, re-drawn from [4]. 144

List of Tables

2.1 Resource utilization of Hoplite based switches on Xilinx Virtex-7 for 32b datawidth and 32-deep SRL32 FIFOs. . . . 24

4.1 BFT t and pi switch cost (32b data + 4b addr). . . . 51

5.1 Resource utilization of Hoplite based switches on Xilinx Virtex-7 for 32b datawidth and 32-deep SRL32 FIFOs. . . . 72

5.2 Application Performance over block size of 1,4 for `Graph/SpMV` traces. (`F`: FIFO, `B`: Back-Pressure, `M`: MLE) . . . 89

5.3 Area in mm^2. MLE NoCs' area calculated using Intel 45 nm process ($3.3x10^6$ transistors/mm^2) . . . 92

7.1 Comparing alternatives to RTL based simulations. . . . 120

7.2 Seeding GraphNoC dataset with 3 different NoCs. . . . 129

7.3 Authoring and training recipes of GNN models. . . . 129

7.4 Measuring accuracy of differently sized GNNs against GraphNoC dataset for HopliteBuf/BP hybrid NoCs. . . . 130

7.5 Best achieved throughput, across all batch sizes. . . . 133

Chapter 1

Introduction

Network-on-Chips (NoCs) play an increasingly critical role in at-scale millions of devices deployment of complex FPGA applications for modern-day use cases such as AI inference, DNN training, time series forecasts, high frequency trading, among others. Transistor density growth is expected to stall with the end of Moore's law [5] but we can continue to reap scaling benefits by integrating increasing chiplets via 2.5D and 3D stacking. Furthermore, the at-scale deployment of these modern day applications impose strict Service Level Agreements in terms of latency and throughput expectations. NoCs can offset the limitations of transistor packing by providing fabric-spanning routing support to increasingly exotic designs that tightly couple highly modular FPGA sub-systems or chiplets with strictly choreographed data-flow requirements. NoCs now play a central role in enabling hardware designs for ASICs and FPGAs that implement the most disruptive applications of Machine Learning [6, 7], Autonomous Driving [8], Neuromorphics [9, 10] etc. With each new generations of FPGAs steadily increasing Look-Up Table (LUT) resources [11, 12, 13] and embedding hardened blocks for in-fabric compute (HBMs [14], DSPs [15]), NoCs are poised to be the critical piece of the design puzzle; enabling high utilization of these powerful FPGAs by providing *last mile connectivity* across the FPGA fabric. The latest family of AMD and Achronix FPGA devices ship with a dedicated hard NoC that is user exposed.

Application-customized NoC designs can outperform traditionally configured NoCs in terms of routing performance, utilization and efficiency. As opposed to traditional NoC design loops that are centered around universally-sound one-size-fits-all principles, application-customized NoC design loops *tune* their NoC parameters by taking into account the application being routed. NoC design parameters (switch types, regulation rate etc) can be *"learnt"* using Machine Learning (ML) algorithms by casting the NoC design space as a function of the quality of result of the application, like latency and throughput.

As evidenced through this work, application-customized NoC designs, when compared to their traditional design counterparts, decrease routing latency by ≈2.5–10.2×, increase workload feasibility by ≈2–3×, increase injection rates by up to ≈3.1× while preserving the features of the original static NoC design. Since the application-customized NoC design space is guided by the Quality of Result (QoR) of the application, such designs are better poised to achieve pareto efficiency against their traditional counterparts for any application workload. Supporting routing requirements for the the next generation of disruptive FPGA applications should encourage us to rethink traditional NoC design process; by incorporating application customization into the NoC design search space.

A NoC design search space is composed of several design-time variables ranging from NoC-level categories like topology to switch-level decisions like flow control. In this thesis, we focus our efforts around identifying and tuning NoC parameters that most impact FPGA application performance: 1) NoC topology (Chapter 4), 2) Flow control (Chapter 5) and 3) Regulation rate (Chapter 6). We cast these NoC parameters as tunable variables that can be adapted to maximize performance of the underlying FPGA application. In this thesis, the adaptation is done via a combination of static classification (Chapter 4) as well as learning the best configuration using ML algorithms. ML algorithms have advanced significantly in their ability to learn complex patterns from data by leveraging increased computational resources to extract higher-order non-linear relationships from input features. We cast NoC configuration parameters as variables that can be learnt by ML algorithms with the goal to maximize the given QoR function of the underlying FPGA application. We use stochastic gradient free evolutionary learning algorithms like Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) and Maximum Likelihood Estimation (MLE) to learn NoC design parameters to maximise the performance of the underlying FPGA application.

However, the iterative nature of the NoC design pipeline can mandate expensive, lengthy and frequent simulations to capture key QoR metrics, which are then used to refine the NoC design search space. As a result, in Chapter 7, we propose a Graph Neural Network (GNN) based framework to *infer*, instead of simulate, NoC performance for a given application. Our aim with this chapter is to train GNN models to *accurately* predict NoC performance in *sub-second* latencies, as opposed to tens of wall clock minutes for simulations. We also release a NoC performance dataset, composed of over 1.5 million graphs of 3 different FPGA NoCs and 200 different FPGA applications, that can be used by other users to train their own GNN models using our proposed framework. We use this dataset to train powerful GNNs that accelerate NoC benchmarking and application-specific NoC design by up to 148× (506× using GPU) with prediction accuracies up to 97.2%.

1.1 Organization and Research Themes

We now explain the organization of chapters in the rest of this thesis. We also list down the main contributions for Chapters 4, 5 and 6.

- **Chapter 2: Background on NoCs**

  In this chapter, we provide the required prerequisite information about NoCs that is key to understanding the work in the rest of this thesis. We introduce the basics of NoC architecture and cover the NoC designs that have been utilized in the rest of the thesis.

- **Chapter 3: Background on Machine Learning Techniques**

  In this chapter, we provide the required prerequisite information about the machine learning algorithms that are leveraged in this thesis to design application customizable NoCs. We cover 2 categories of algorithms: 1) Gradient based supervised learning, and 2) Gradient-free optimizations. For gradient based supervised learning, in preparation for Chapter 7, we first cover the basics of supervised learning of Neural Networks, followed by a deep-dive into Graph Neural Networks. In preparation for Chapters 5 and 6, we discuss Bayesian Statistics, Covariance Matrix Adaptation and Maximum Likelihood Estimation.

- **Chapter 4 and 5: Designing Application Customizable BFT Topologies**

  In Chapter 4, we present a low cost Butterfly Fat Tree (BFT) based FPGA NoC that supports customization of its bisection bandwidth to the application being routed. We also devote a section of Chapter 5 to tune the upstream routing data-paths of the 4-ary `pi` switches of BFTs on a per-switch basis to maximise routing performance of the FPGA application. The main contributions of this work are:

  - The NoC offers application customizability to its bisection bandwidth by connecting together switches of different arity. We benchmark 4 different topologies with a rich diversity of application traces for a combination of application performance, NoC area and NoC power scenarios.
  - We introduce a backpressure-based flow control interface that is coupled with deterministic routing and fair arbitration to provide bounded and in-order routing of packets.

- We benchmark the NoCs for a range of synthetic and realistic workloads. We measure our flow-controlled BFT NoC to outperform other NoCs (Hoplite, Deflection BFT, CMU CONNECT etc) by 1–6×.
- For application customized upstream routing paths, we measure routing latencies latencies that are ≈1.1-1.7× lower than the vanilla versions that do not take the specific application into account.

- **Chapter 5: Learning Hoplite Switch Configurations**

  In this chapter, we *learn* the configuration between buffer based or back pressure based flow control for the Hoplite NoC, on a *per switch* basis. We learn this configuration to maximise the QoR of the specific application being routed. The main contributions of this chapter are:

  - To learn switch configuration, we present a novel evolutionary algorithm based on Maximum Likelihood Estimation (MLE) that allows us to maximise the QoR of the application being routed in a gradient free manner.
  - We benchmark the application customized NoCs generated by MLE to have ≈2–3× better feasibility and ≈1–6.8× reduced latency.
  - We propose a framework to dynamically reconfigure a live NoC on a Pynq-Z1 by running the MLE algorithm on the Software layer of the Pynq-Z1 while the NoC routes the application running on the FPGA layer.

- **Chapter 6: Nested Learning of Regulation Rates along with Switch Configuration**

  In this chapter, we further extend the learnable parameters for the Hoplite NoC by tuning the regulation rates of each traffic trace in the application to maximise QoR. We also propose *nested* learning where the switch configurations (from Chapter 5) are learnt alongside the regulation rates to maximise application QoR. The main contributions of this chapter are:

  - To learn regulation rates, we model parameters space as a multivariate gaussian distribution. With an aim to maximise application QoR, the regulation rates are learnt on a per-trace basis using Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES).
  - We develop a nested learning strategy where, for a given application workload, the switch configurations (from Chapter 5) and regulation rates (this chapter) are learnt together.

- We benchmark our algorithms for a variety of real and synthetic applications. We observe ≈1.4× lower cost-constrained latency, ≈3.1× faster rates when learning switch configurations and regulation rates together.

- **Chapter 7: Framework and Dataset to Predict NoC Performance using Graph Neural Networks**

  From previous chapters, we observed that designing application-specific NoCs can often involve 100s of rounds of tuning, with each round requiring benchmarking of 100s of intermediate NoC designs. These thousands of rounds of benchmarking can often be slow and expensive, presenting a critical bottleneck in designing application specific NoCs and harnessing the reconfigurability of FPGAs. In a quest to remove this bottleneck, we create a Graph Neural Network (GNN) based framework to predict NoC performances instead of slow and expensive RTL simulations or analytical simulations. The main contribution of these chapter are:

  - We present a framework to encode any FPGA NoC and any FPGA application into graphs. This framework can encode any FPGA NoC, regardless of topology or architecture, and supports both soft and hard FPGA NoCs.
  - For the Hoplite and BFT NoCs of Chapters 4, 5, and 6, we curate a dataset of over 1.5 million unique samples of NoC and application pairs.
  - We train powerful GNN models using our curated dataset to predict routing performance for these NoCs in place of traditional RTL simulations or analytical simulations.
  - We quantify our trained GNNs for prediction accuracy and speed-ups. We observe up to ≈148× (≈506× on GPU) speedups compared to traditional benchmarking methods, while achieving up to 97.2% prediction accuracy.

- **Chapter 8: Conclusion and Future Work**

  Finally, we conclude this thesis by presenting some future directions to build on top our research contributions. We primarily focus on expanding our work to accelerate FPGA application design as a whole. Our first line of thinking presents research ideas that explore using machine learning to design and tune different components of the FPGA design pipeline, that have typically been governed by heuristics. Our second line of thinking pitches using Graph Neural Networks to accelerate the time it takes to go from RTL to an FPGA bitstream, a lengthy process consisting of net-list placement and timing closure.

1.1.1 List of Publications

The research work associated with Chapter 4 ([3]), Chapter 5 ([2]), Chapter 6 ([16]) and Chapter 7 ([17]) is shared in their complete form in this document. The contents of these chapters have been published in peer-reviewed conferences and journals. For each chapter, I list down its corresponding reference below:

1. Chapter 4:

   G. S. Malik and N. Kapre, "Enhancing Butterfly Fat Tree Nocs for FPGAs with Lightweight Flow Control," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 154–162, 2019

2. Chapter 5:

   G. Malik, I. E. Lang, R. Pellizoni, and N. Kapre, "Learn the Switches: Evolving FPGA NoCs with Stall-Free and Backpressure Based Routers," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 18–25, IEEE, 2020

3. Chapters 5 and 6:

   G. Malik, I. E. Lang, R. Pellizzoni, and N. Kapre, "HopliteML: Evolving Application Customized FPGA NoCs with Adaptable Routers and Regulators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 4, pp. 1–33, 2022

4. Chapters 7:

   G. S. Malik and N. Kapre, "GraphNoC: Graph Neural Networks for Application-Specific FPGA NoC Performance Prediction," in *2024 International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2024

The motivation to use gradient-free learning, as a means to learn parameters in a gradient free manner, was born out of auxiliary work I co-authored during this thesis study and is listed below.

5. Learning the Partitioning Layouts for Systolic Arrays:

   L. C. Chan, G. Malik, and N. Kapre, "Partitioning FPGA-Optimized Systolic Arrays for Fun and Profit," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, pp. 144–152, IEEE, 2019

6. Distributing Evolutionary Strategies across Compute Nodes:

   G. S. Malik, L. Petrica, N. Kapre, and M. Blott, "DarwiNN: Efficient Distributed Neuroevolution under Communication Constraints," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pp. 141–142, 2020

Chapter 2

Background: Network on Chips

The advent of multi-core processors over the past two decades ago has positioned Network-on-Chips (NoCs) as first-class citizens for modern-day hardware designs. Over successive generations of HW life-cycles, the progressively increasing number of processing cores has mandated a proportional increase in inter-core communication bandwidth to maintain efficient core utilization and meet application Service Level Agreements (SLA). This escalating demand has underscored the critical need for a scalable and high-performance interconnection fabric. NoCs fulfill this requirement of providing high bandwidth and last mile connectivity by carefully scheduling and managing packet traffic over a shared interconnect overlay, while exhibiting superior asymptotic area-scaling when compared with point-to-point connections. This has effectively positioned NoCs as a crucial architectural component of the modern-day hardware design puzzle, evident across their integration into a broad spectrum of computing platforms (GPUs, TPUs, FPGAs, ASICs, etc) and applications (High-Performance Computing, Neuromorphics, Artificial Intelligence, etc).

2.1 History of NoCs

We now provide a brief history of NoCs. The advent of multi-chassis interconnect networks, used in routers, workstations and supercomputers, exposed the limitations of pin-limited physical wire connections around scalability and bandwidth. Virtual Channels [20] were developed to facilitate deadlock-free routing in interconnection networks, particularly in scenarios where physical wiring resources are constrained. As data payload sizes increased, wormhole routing [21] emerged as a viable solution to maintain low latency and high throughput for extended packet transmissions. Although Network-on-Chips drew

upon principles established in prior multi-chassis interconnection networks, the design constraints encountered in on-chip interconnects exhibited different characteristics, with plentiful and high bandwidth wiring available on-chip being a key differentiator. However, as the computing paradigm witnessed a shift towards multi-core chips and architectures, previously plentiful on-chip wiring saw significant pressures on demand; bigger caches and increasing number of PEs competing with the NoC interconnects for the same wire real-estate.

Dedicated point-to-point connections between on-chip PEs exhibited unfavorable asymptotic scaling trends for wiring resources, rendering this approach impractical beyond a small node count. The simplest form of on-chip networks, Buses and Crossbars, were suitable for low core counts. However, buses suffered from limited scalability due to traffic saturation, high power consumption, and arbitration latency. Crossbars, while addressing bandwidth issues, also scaled poorly due to their large area footprint and high power consumption. To mitigate these limitations, [22] made the case for being judicious in the use of on-chip wiring resources by smartly designing NoC topology and flow-control methods to best exploit communication patterns and available on-chip resources. During this period, key architectural techniques like wormhole routing [21] also became standard features in NoC implementations [23]. This is evidenced by their adoption in numerous proposed architectures and commercially available chips, including MIT RAW [24], Tilera TILE64 [25], Intel's 80-core TeraFLOPS processor [26], Austin TRIPS system [27], as well as the SPIN network [28]. To further enhance the bandwidth capabilities of wormhole flow control, some designs, such as Intel TeraFLOPS [26] and Austin TRIPS [27], leveraged multiple virtual channels. Conversely, other architectures, like RAW [24] and Tilera TILE64 [25], employed a strategy of utilizing multiple on-chip networks.

The increasing need for scalable, low-latency, and high-bandwidth communication within complex systems-on-chip (SoCs) and chiplet-based architectures has been the predominant driving force for innovation in modern-day NoC design. Specifically, the exponential growth in artificial intelligence training and inference workloads has placed unprecedented demands on interconnects within AI accelerators. NoCs are now optimized to handle the massive data movement required for training and inference by providing high-bandwidth, low-latency communication between the numerous processing elements, DRAM and HBM memory controllers, cache-based sub-systems and network interfaces. In GPUs, like NVIDIA's A100 [29], the NoC connects Streaming Multiprocessors (SMs), each containing multiple CUDA cores, to the memory controllers and L2 cache using a hierarchical NoC with a crossbar within each SM and a mesh connecting the SMs, memory controllers, and L2 cache. This NoC implementation supports access to up to 80 GB of High Bandwidth Memory (HBM) [30] at a rate as high as 2 TB/s [29]. Tensor Processing Units

(TPUs) [31], such as Google's TPU V4 [32], now utilize a 3D mesh NoC to interconnect the Matrix Multiply Units (MXUs), which are specialized for tensor operations, with up to 32 GB of HBM2 [33]; enabling efficient computation rate of up to 275 teraflops [34] for deep learning based training and inference workloads.

In the context of FPGAs and the design considerations for a NoC implementation, a key distinction lies in the inherent availability of routing resources. Unlike ASICs, where on-chip wiring constitutes a critical design constraint, FPGAs possess a pre-fabricated, abundant programmable logic composed of dedicated routing channels and LUTs. Consequently, the challenge of on-chip wiring of ASIC NoCs, is somewhat mitigated in the FPGA domain. This abundance of wiring resources, relative to the available LUTs for logic implementation, alters the trade-offs typically considered in NoC design; strategies focused on conserving wiring resources of [22] may not be as directly applicable for FPGA NoCs. Instead, targeting the scarcer LUT resources might be more applicable for FPGAs. Buffered FPGA NoCs increase design complexity by consuming significant amount of LUT resources, in-turn consuming significant energy and leaving few LUT left-overs for FPGA application implementation. As a result FPGA NoCs ushered in small and lightweight form-factor [35, 36] that leveraged buffer-less deflection based routers inspired from groundwork laid in [37, 38]. However, these deflection-based NoCs work by intentionally mis-routing packets to manage contention. In the worst case, these deflections can go on forever, making them unsuitable for real-time applications where worst-case bounds must be computable to meet strict Service Level Agreements (SLA) of the application. To mitigate this and have light-weight buffer-less NoCs support real-time applications, latest iteration of these FPGA NoCs [39, 40, 41] have adapted the theory of network calculus [42] to estimate bounds on delay on a per-link and global basis purely as a function of the NoC's routing policy and application flow-set.

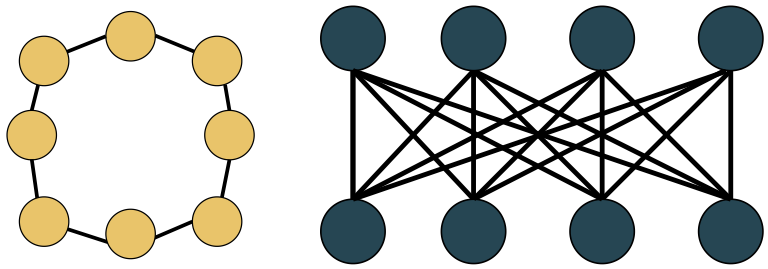
We now present the various aspects of NoC design that serve as central links to our work in the rest of the thesis' chapters.

2.2 Basics of NoC Architecture

We now discuss the design variables that are central to the NoC designs and algorithms presented in this thesis.

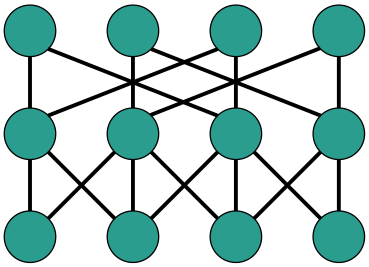
- **NoC Topology**: The layout of the switches and connections between them determines the topology of the NoC, as illustrated in Figure 2.1. These topologies not

only influence packet routing paths but also directly impact the bisection bandwidth [43] of the NoC, a measure of the minimum bandwidth between two halves of the network. Evolution of NoC topology has been driven by increasing performance and bandwidth demands in hardware implementations [44]. Initially, traditional interconnects like buses and crossbars were deployed, but their scalability limitations with growing core counts [45] prompted the shift to regular topologies such as meshes and tori due to their relative simplicity and scalability [46]. As application requirements diversified, more complex topologies emerged, including irregular and hierarchical structures like Butterfly Fat Trees (explained in detail later in section 2.4), as well as 3D meshes [34], addressing specific needs like those of FPGAs and ASICs like GPUs and TPUs. Of note, the topology of the NoC is closely related to the placement of the Processing Element (PE) endpoints, which is more constrained in an FPGA fabric compared to ASICs. As a result, FPGA NoC topologies exhibit more structure to compose with the FPGA macro placement as opposed to the blank canvas nature of ASIC placements.

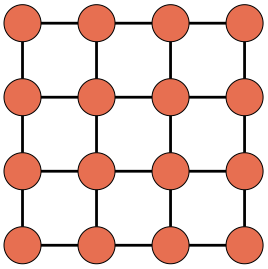


(a) Ring

(b) Crossbar



(c) Fat Tree



(d) Mesh

Figure 2.1: Visual representation of some common NoC Topologies.

- **Routing Algorithm**: As a packet arrives at the input port of a switch, its output port is chosen by the routing algorithm. Although the NoC typically only consists of a single shared overlay of switches, there can be many different paths (sequence of switch hops) between two endpoints. A real life analog of this is the different sets and sequences of roads and turns we can take as we travel to our destination. Further, similar to the real life example, routing algorithms are motivated to also minimize travel time and contention. NoCs make use of different routing algorithms to best suit their topology and flow control. Routing algorithms can generally be classified into 2 broad categories: Deterministic and Adaptive routing. In the case of deterministic routing, all packets between the same source and node PE will traverse the same path. For example, Hoplite is a 2D directed torus and uses Dimension Ordered Routing (DOR), where packets travel horizontally (W→E) before making just a single turn (W→S) to travel vertically (N→S) to avoid deadlock. Another example is address-based routing, where the routing path is determined as a function of the source and destination PE. An example of a NoC targeted in this thesis that uses address-based routing is the Butterfly Fat Tree (BFT) NoC where packets climb the tree until they reach the first available downstream path to the destination endpoint. At this point, the packets turn and descend the tree to reach the destination. In the case of adaptive routing, in contrast to deterministic routing, a packet's exact path is influenced by the traffic conditions in the NoC which can help minimise contention but at a great cost. It introduces significant implementation overhead to maintain a stateful model to track the status and position of flits over the shared interconnect. In this thesis, we focus on light-weight NoCs that predominantly make use of deterministic routing. Its important to also note that while it might be tempting to design a routing algorithm to always take the shortest path across the NoC, such a philosophy does not take contention and the resultant flow control into account.

- **Flow Control**: Packets travel through the NoC via hops along the shared overlay of switches. Sometimes, two or more packets at the same switch might contend for the same output port to hop to the next switch. In such scenarios, flow control decides the fate of the packet that loses this round of contention. Flow control manages the second order effects of contention since the losing packet can itself influence the routing of other packets and so on. Flow control complexity is generally proportional to the number of ports, virtual channels and cross-bar complexity. For example, deflecting the losing packet to any other available output port consumes very few LUT resources to implement but sacrifices important routing features like live-lock freedom (packets keep bouncing around the NoC forever), in-order packet delivery (first-in-first-out for a stream of packets) while also creating contention in other

parts of NoCs. On the other end, absorbing losing packets into an infinite depth FIFO, theoretically, can self contain any effects of contention, but at great LUT cost and infinite routing latency. However, both buffered and buffer-less flow control implementations that employed a single queue at each input suffered from Head of Line (HOL) blocking affects where the packet at the front of this queue is blocked and cannot proceed to the next stage due to congestion or resource unavailability; thus preventing all subsequent packets in that same queue from moving forward. To mitigate HOL, Virtual Channels (VC) [20] were introduced. VCs splice the traffic across distinct queues within the router, with multiple VCs sharing the underlying physical link between two routers. Consequently, if a packet occupying a specific VC is blocked, other packets can still utilize the physical link via alternative VCs. This functionality enhances physical link utilization and subsequently improves overall network throughput.

- **Switch I/O Arity**: It refers to the number of input and output ports of a switch. For example, the standard Hoplite NoC [35] has 3 input (North, West, PE) and 3 output ports (South, East, PE). Pi switches of flow-controlled BFTs of Chapter 4 ([3]) and deflection routed BFTs of [36] have 4 input and 4 output ports (Left, Right, U0, U1). The number and type (input vs output) of ports influence the topology, performance and complexity of the overall NoC design. While increasing the I/O ports of a switch naturally increases its routing bandwidth, an excessive increase is accompanied by a disproportionate increase to switch area, implementation complexity and performance penalties. Note that FPGAs are primarily constrained by LUT resources, while ASICs are limited by wiring costs. Hence, while increasing the I/O arity of a switch will result in an increased wiring and implementation costs for both FPGA and ASIC NoCs, an FPGA NoC will incur a disproportionately larger implementation penalty by being forced to spend precious LUTs to support the required increase in complexity of accompanying arbitration, flow-control and routing policies.

- **Regulators**: As discussed, NoCs route data traffic over a shared interconnect of switches. To ensure quality of service guarantees, we can use regulation of traffic injection into the NoC fabric. Failing to do so in a timely manner can overwhelm the NoC, resulting in cascading impacts to routing bandwidth and culminating in congestion or starvation. Regulators play this important role of limiting injection into the NoC. A few examples of regulators include the leaky bucket, token bucket and sliding window regulators. The leaky bucket operates on a simple principle: it's analogous to a bucket with a hole at the bottom, releasing packets at a fixed, constant

rate. This mechanism effectively smooths out traffic, preventing sudden spikes, but it comes at the inability to replicate bursts of traffic injection into the NoC. In contrast, the token bucket regulator employs a more dynamic approach. It uses tokens, which are generated at a constant rate. Each packet requires a certain number of tokens to be transmitted. This allows for bursty traffic as long as there are enough accumulated tokens, offering greater flexibility. Sliding window regulation in NoCs manages the amount of data in-flight to prevent network congestion. It uses a window of sequence numbers, allowing the sender to transmit multiple packets without waiting for individual acknowledgments. As the receiver acknowledges packets, the window slides forward, enabling further transmission. In this thesis, in Chapter 5 ([2]) and Chapter 6 ([16]), we employ the token bucket regulator which is characterized by two parameters: regulation rate and burst size. Regulation rate controls the amounts of packets injected per cycle and burst size determines the maximum consecutive packets the regulator can inject into the NoC. We discuss the token bucket regulator in greater detail in section 5.4.2. We explore learning of regulation rates for an application in Chapter 6 ([16]).

- **Arbitration**: Flow control determines *what* happens to the losing packet in case of a contention. Arbitration rules decide *which* packets win the contention. Broadly speaking, arbitration can resolve contention by ranking on packet properties or port properties. For example, First-In-First-Out would grant higher priority to the older packet (ranking on packet). Conversely, packets trying to turn (W→S) are given highest priority regardless of packet properties in the case of the HopliteRT [39] NoC to deliver livelock freedom (ranking on port). Arbitration algorithms should be fair in assigning priorities to ensure all packets get serviced. A lack of fairness can cause packets to perpetually lose out in arbitration contests, which in turn can lead to a mixture of deadlock (packet stuck at a switch) and livelock (packet moving but never reaching destination) scenarios.

2.3 Encoding Applications into Traffic Traces

A collection of individual traffic traces characterize the application being routed. Each of these individual traffic traces captures the unique communication pattern to send data from one endpoint to the other. Multiple traffic traces can be composed between the same two endpoints to capture different behaviors. The NoC designs and algorithms presented in this thesis have been tested on `synthetic` and `real` application traffic traces. In this thesis, we define the following entries to compose a traffic trace.

- **Source-Destination Pairs**: A traffic trace is characterized by the source generating the packets and the destination to which these packets are headed. This is represented as a tuple of endpoints (`source`, `destination`), where the `source` is the index of the NoC endpoints into which the packets would be injected and the `destination` is the index of the NoC endpoint which the packet will exit the NoC. Each traffic trace is assumed to be independent of others; operating under the assumption of either continuous traffic streams or bulk-synchronous parallel processing where traffic can be arbitrarily mixed. The execution of workloads across multiple processors will necessitate modeling dependency between traffic traces to ensure both consistency and correctness of execution. However, a comprehensive modeling of inter-trace dependencies falls outside the defined scope of this thesis.

- **Injection Rate**: The rate at which the packets are ideally injected into the NoC. However, the real injection rate is often determined by the combination of regulator state and NoC bandwidth availability.

- **Data Block Size**: The consecutive number of packets the traffic trace will send at any particular time. The real data block size is instead determined by the regulator; burst size, for example, in the case of token bucket regulators.

In addition to the NoC architecture (like topology, flow control etc), an FPGA application's routing performance attributes like throughput and latency are determined by its load factor, which is quantified as the amount of interference between individual traffic traces over the shared NoC. We now explain in detail the `synthetic` and `real` traces that have been used to benchmark the various NoC designs presented in this thesis.

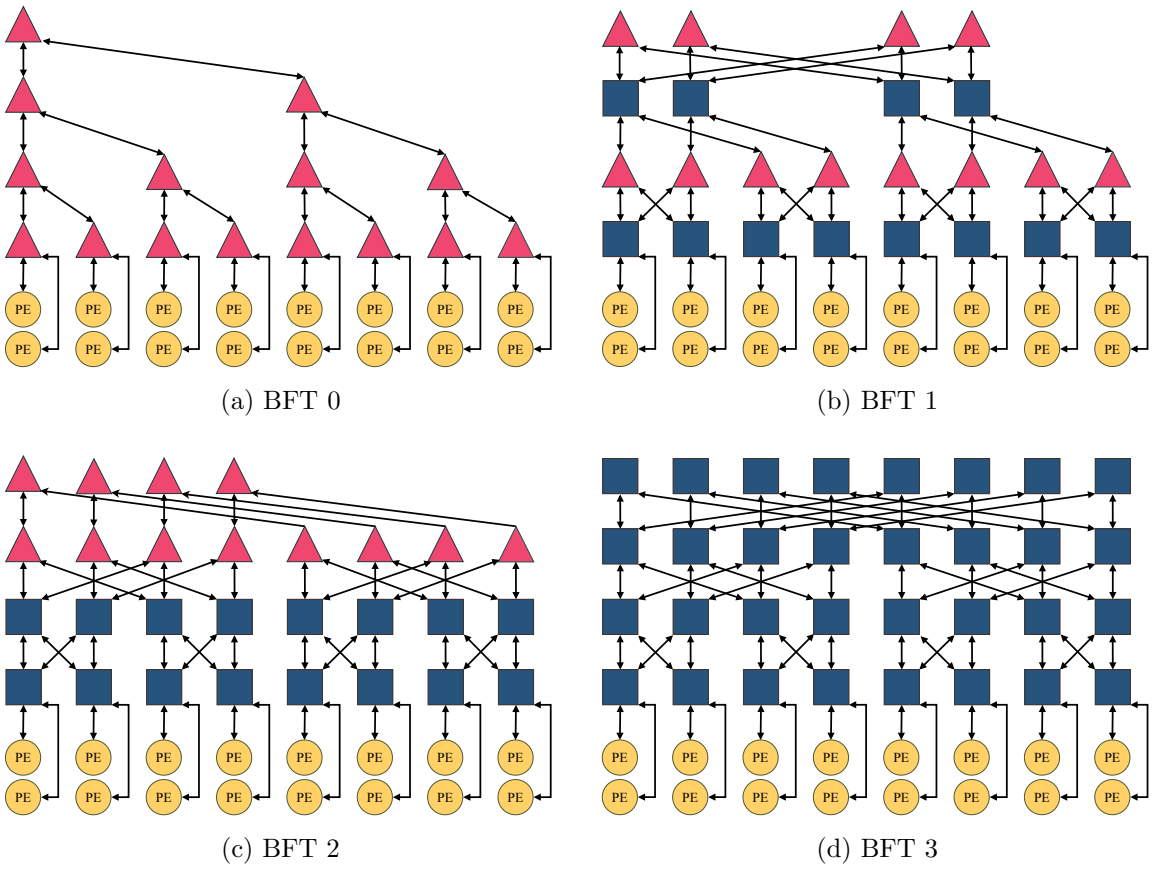
`Synthetic` traces have been generated artificially to capture a wide range of workloads, contention and communication patterns. They also allow us to carry out ablation studies on the performance on NoC by generating and testing the NoC on traces with a particular character. For example, we can generate artificial traces where the the communication workload is biased toward bursty but sporadic communication patterns. For `RANDOM` application traces, a source endpoint's destination is generated randomly by sampling a uniform distribution. The application is compiled by performing such uniform sampling for every NoC endpoint. `LOCAL` application traces are aimed at testing nearest-neighbor style workloads. For each NoC source endpoint, the uniform sampling is limited to the top-K neighboring endpoints. K can be varied to control the level of *long distance* vs *short distance* traffic in the NoC.

`Real` traces, on the other hand, are extracted from real life applications and allow us to test the NoC on consumer-grade as well as enterprise-grade workloads. This is

especially useful for system designers to ensure that the designed NoCs can work under a variety of real life workloads, as well as generating the required confidence to deploy the designed NoC to at-scale, production scenarios. Throughout this thesis, we use a variety of `real` benchmarks, ranging from workloads related to applications like Sparse matrix vector multiplication (`SpMV`) [47] to analysis of graphs using the `SNAP` dataset [48]. These real life workloads exist along the entire spectrum of routing difficulty, with load factors varying from 37% to 92%.

2.4 Butterfly Fat Tree based FPGA NoC

Most modern NoC topologies are either 2D meshes or tori due to the geometric simplicity of mapping to a 2D VLSI substrate. However, these topologies are unable to scale bisection bandwidth to meet differing application requirements. Instead of making all chip-spanning NoC links wider to boost throughput, Butterfly Fat Trees (BFTs) offers customizable bisection bandwidth to the communication workload with configurable multi-level NoC topology. Through proper selection of switch richness at different levels of the tree we can tune the network to any bisection bandwidth $O(N^p)$ where p is the Rent parameter [49] of the architecture. BFTs have a long history [50] [51] and have found use in traditional ASIC designs [52] [53] [54], Multi-Processor SoC designs [55] [56] [57] and off-chip networks in the data-center [58] [59]. We evaluate the topologies shown in Figure 2.2 for our work in Chapter 4.



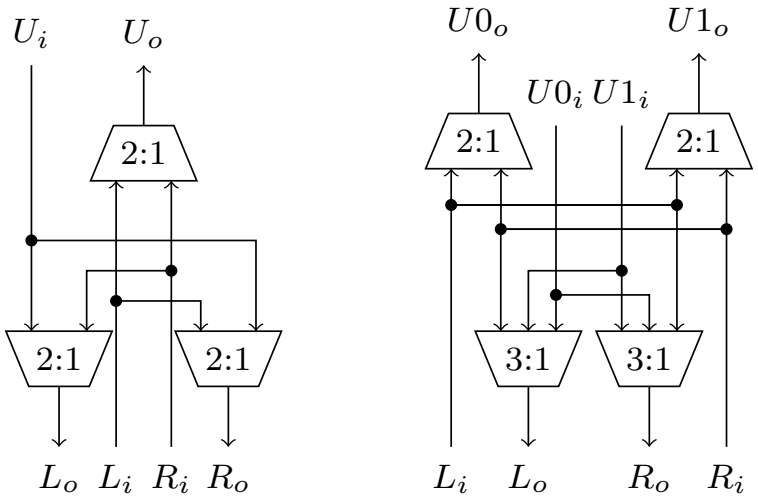
(a) BFT 0 (b) BFT 1

(c) BFT 2 (d) BFT 3

Figure 2.2: Different BFT topologies benchmarked in this work. The placement and type of switches influences the routing bandwidth available to different packets based on their position in the NoC.

Switch Micro-architecture: A popular version of 2-ary BFT NoCs is constructed from two types of building blocks: *t* and *pi* switches as shown in Fig 2.3. A *t* switch has two ports to/from the lower levels of the tree and only a single uphill port. A *pi* switch has two ports from the lower levels of the tree, and two ports to the upper levels of the tree, thereby preserving total bandwidth.

Bandwidth Customization: Fat trees allow the network bisection of the NoC ($O(N^p)$) to be adjusted as required to reflect Rent properties (p) of the communication workload.



(a) t switch (b) pi switch

Figure 2.3: Micro-architecture of BFT t and pi switches.

We can consider two extreme configurations of a BFT to better understand the tunable range of bisection bandwidth configurations available to us. On one hand, a binary tree (**BFT0**), configured by simply choosing t switches at all levels at a cost of $O(N)$ switches, can only sustain nearest-neighbor style lightweight communication (bisection bandwidth of $O(1)$). On the other hand, a multi-stage crossbar (**BFT3**), configured by choosing only pi switches at all levels at a cost of $O(N \times log(N))$, supports all-to-all communication patterns (bisection of $O(N)$) in a cost effective manner. By choosing the right mix of t and pi switches, we can construct the right NoC to suit application demands, a key requirement of FPGA workloads. In this thesis, we will refer to the (t-pi-t-pi) topology as **BFT1** and the (t-t-pi-pi) topology as **BFT2**. We evaluate these topologies for performance in greater detail in Chapter 4.

Routing Policy: For packet-switched operation, the destination address and payload are part of a single indivisible unit of communication. The routing algorithm used on BFTs [36] is straightforward and can be separated into three phases: (1) climbing, (2) turning, and (3) descent. As shown in Figure 2.4, during the climbing phase, packets can take **any** available uphill port until they reach the common ancestor of both source and destination clients. The level of the NoC that represents the common ancestor is the bit position of the common prefix of both addresses. The packet must turn here and has only one choice of outgoing port. During descent, at level i of the NoC, extract bit i of the destination

address and turn left (0) or right (1). Again the downhill path is fixed based on destination address and no routing freedom is available during descent.

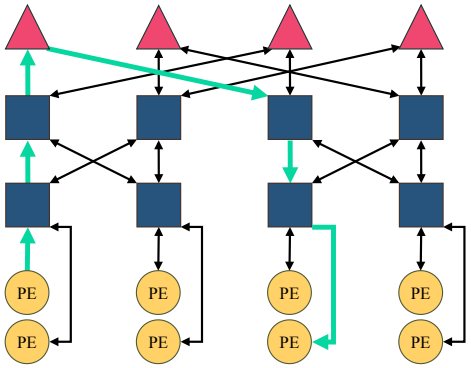


Figure 2.4: Visual representation of the routing path of a packet for a BFT NoC.

Deflection based Flow Control: Deflection based BFTs of [36] use a light-weight mechanism of localized deflections that intentionally misroute a packet towards its own output ports before being deflected back for another attempt at a successful exit. Deflection based flow control can be implemented without a significant increase in FPGA resources; the fan-in of multiplexers of Figure 2.3 increases by a count of one. This still allows a tightly packed LUT based implementation of these switches: a `t` switch can be implemented using 3×5-LUTs and a `pi` switch can be implemented using 4×6-LUTs. We show this deflection based flow control with an example in Figure 2.5. For a `pi` switch with 2 packets (**yellow**, **green**) attempting to exit to the same output port, arbitration designates **yellow packet** as the higher priority packet and is thus allowed to route to the destination port. The **green packet**, as the losing packet, is misrouted towards it own output port. In the next cycle, this **green packet** is deflected back towards the original input port and allowed to make an attempt to exit at its destination port. We discuss the impact of localized deflections on in-order delivery, traffic hot-spots and live-locks in greater detail in Chapter 4.

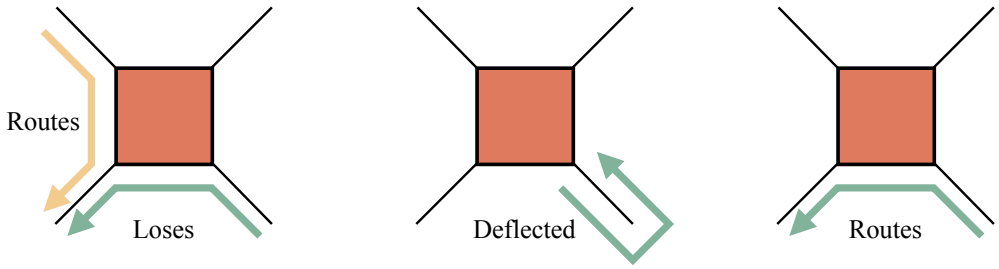


Figure 2.5: Visual Representation of localized deflections at a `pi` switch.

2.5 Hoplite FPGA NoC

The Hoplite [35] FPGA NoC is a LUT-optimized network-on-chip architecture targeting fracturable (a K LUT can operate as 2 K-1 LUTs) Xilinx FPGAs for high-speed, low-cost operations. Hoplite is characterized by its frugal use of FPGA resources, resulting in high implementation clock frequencies. While different flow control variants of Hoplite switches exist to address a wide variety of application use-cases, all of them use a 2D unidirectional torus topology, as shown in Figure 2.6.

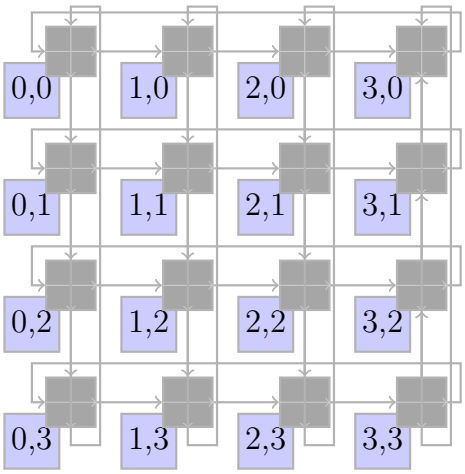


Figure 2.6: 2D unidirectional torus topology of the Hoplite NoC.

Switch Micro-architecture: The Hoplite switch of [35] is characterized by its low IO port counts and emphasis on directional routing. As a result, as shown in Figure 2.7, it is

composed of only 2 dedicated input (West W and North N) and 2 dedicated output (East E and South S). The PE port is the only bi-directional port in the switch. The hoplite switch enforces a strict priority order between packets vying for the same destination port, based on the input ports at which packets arrive. The only time packets can contend for the same port are turning packets from West to South vying for the same S port with vertical packets travelling from North to South. In such cases, vertical packets are always given priority over the turning packets. Flow control of losing packets depends on the exact implementation and is discussed in greater detail below.

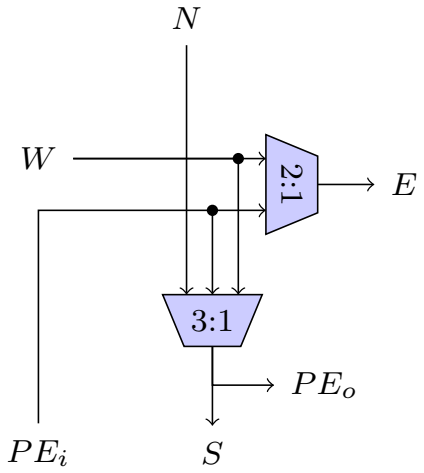


Figure 2.7: Block diagram of the original deflection based Hoplite switch.

Routing Policy: All hoplite variants use Dimension Ordered Routing (DOR). As shown in Figure 2.8, packets first must travel horizontally using the **W** $\rightarrow$ **E** route until they reach their destination column. At which point, they turn into the vertical column using the **W** $\rightarrow$ **S** datapath to reach their destination using the **N** $\rightarrow$ **S** datapath.

Deflection based Flow Control: The original switches of Hoplite from [35] uses deflections as the method of flow control. Specifically, packets that are trying to turn from West $\rightarrow$ South are intentionally misrouted to the East port instead. As packets follow Dimension Ordered Routing, an unlucky packet flit (individual packet as part of a stream of packets) might forever be stuck in the horizontal ring, leading to live-locks. The muxes of the deflection based switch can be efficiently mapped to a single fracturable 6-LUT for one bit of switching datapath, as shown in Figure 2.9. While the lightweight hoplite switches are an attractive proposition for area sensitive applications, possibility of routing

livelock and unbounded routing latencies are significant limitations, making it unsuitable for systems requiring guarantees on worst-case packet latencies.

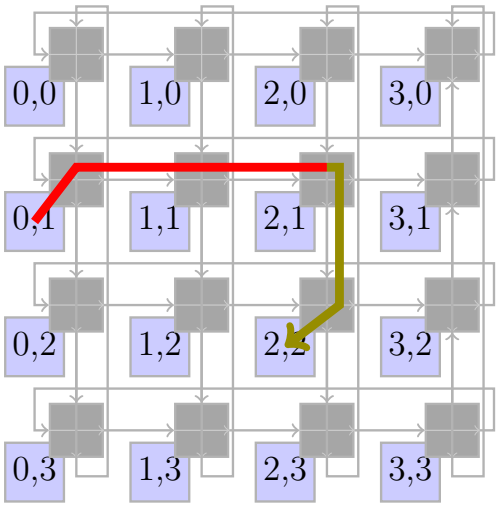


Figure 2.8: Visual representation of the DOR routing policy of Hoplite

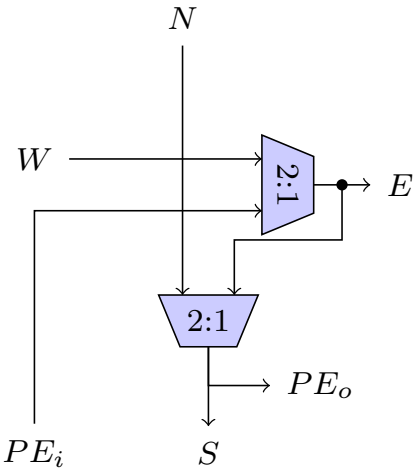


Figure 2.9: Block diagram of the original deflection based Hoplite switch with shared East exit.

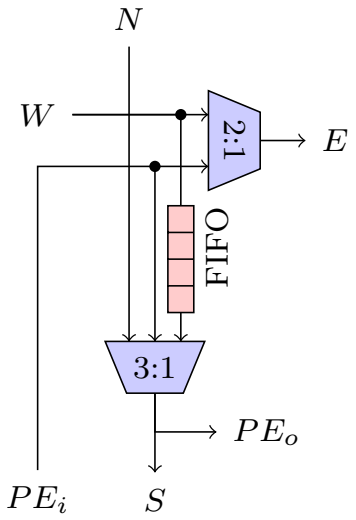


Figure 2.10: Block diagram of the HopliteBuf switch.

Stall-Free FIFO based Flow Control: HopliteBuf (**FIFO** in Figure 2.10) [60] is a variant of Hoplite that introduces stall-free FIFOs on the turns in the NoC. It buffer packets turning from $W \rightarrow S$ if a $N \rightarrow S$ packet is present in that cycle. Following DOR routing policy, it allows N packets to travel S while W packets must wait for an empty cycle to progress further. The 2:1 East mux chooses between W, and PE packets while the South mux chooses between W' (FIFO output), N, and PE packets. It is worth noting that while the FIFO is only available for $W \rightarrow S$ packets, we also need to hold back $PE \rightarrow S$ packets if a N packet is present, as the PE port has the lowest priority for injecting a packet; the same holds for a $PE \rightarrow W$ packet if a higher priority $W \rightarrow E$ is present. PE packets have lowest priority amongst all packets and must wait for traffic to clear on the target exit mux. Rather than implementing additional, LUT-expensive FIFOs, a regulator compatible PE interface that can be stalled is assumed; such that the PE is prevented from injecting a packet in any clock cycle when a higher priority packet is present. Packet delivery to a PE is not stallable, thus allowing unrestricted N→S traffic. The south mux needs to select between three inputs: N, FIFO output and PE_i. As a result, the HopliteBuf design cannot exploit the same degree of LUT compactness as deflection based switches. To realize these FIFOs, SRL32 (32-bit shift register implemented using a LUT) primitive are leveraged to re-purpose LUTs as Memory elements.

Backpressure based Flow Control: HopliteBP (**Backpressure** in Figure 2.11) [41] supports lightweight backpressure in the horizontal ring. $N \rightarrow S$ packets have the highest

priority; thus foregoing a need for backpressure along the vertical dimension. Only packets turning $W \rightarrow S$ in the network may be subject to contention and thus require a backpressure interface. Shadow registers on the W port are inserted which provide an ability to store stalled packets *in place* and propagate the backpressure control (BP_0) upstream in the opposite direction of packet flow [3]. For HopliteBP, the DOR routing logic must be adapted to account for the presence of backpressure signals. The control logic and shadow registers are more expensive to implement on the FPGA than SRL-based FIFOs. They result in a 1.2–1.8× increase in LUT and FF usage over HopliteBuf switches, summarised in Table 2.1

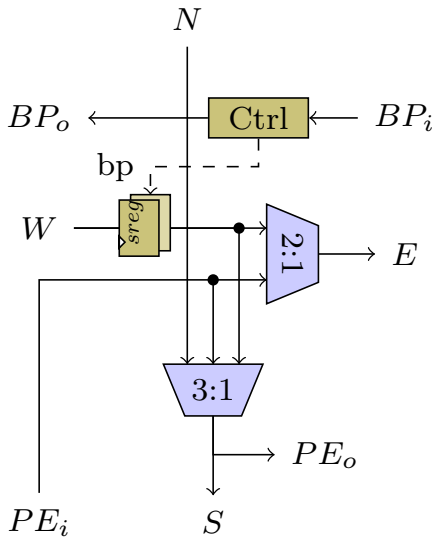


Figure 2.11: Block diagram of the HopliteBP switch.

	Hoplite	HopliteBuf	HopliteBP
LUTs	59	161	189
FFs	86	91	167

Table 2.1: Resource utilization of Hoplite based switches on Xilinx Virtex-7 for 32b datawidth and 32-deep SRL32 FIFOs.

We now explain in detail the primary FPGA NoCs leveraged in this work to design application-specific NoC designs.

2.6 Survey of Relevant FPGA NoCs

FPGA-based NoCs differ from traditional ASIC NoCs in their configurability and adaptability, tailored for the programmable nature of FPGAs. In recent years, the latest generations of AMD Versal, Intel Agilex and Achronix Speedster family of FPGAs have seen a hybridization of traditional soft NoCs and ASIC NoCs in the form of Hard FPGA NoCs. Similar to other hardened FPGA components like DSPs and HBMs, hard FPGA NoCs utilize dedicated, prefabricated circuitry optimized for performance and power efficiency. This hardened implementation results in higher clock speeds, lower latency, and improved area and power efficiency compared to their soft counterparts. However, this performance advantage comes at the cost of flexibility; critical variables of the NoC architecture, like topology, flow-control and routing algorithms, are fixed.

In contrast, soft NoCs are implemented using the programmable logic resources of the FPGA. This approach offers significantly greater flexibility, allowing designers to customize and reconfigure the NoC architecture as needed. Soft NoCs are well-suited for applications requiring adaptability, experimentation, or prototyping. As a result, in contrast to hardened NoCs, soft NoCs can be easily designed and tuned specifically for the FPGA application at hand. This can optimize available NoC routing bandwidth, in turn minimising costs and maximising application performance. This is achieved by identifying design parameters of soft FPGA NoCs that have a tangible effect on NoC routing bandwidth. Subsequently, these parameters can be optimized and learnt as part of the NoC design loop, such that critical NoC performance indicators (routing latency, throughput, etc) are optimized specifically for the given FPGA application.

This thesis presents novel soft NoC designs and algorithms to customize soft NoCs for the application to be routed. We benchmark these new NoC designs and learning algorithms against a wide variety of NoCs. Although every competing NoC design has been discussed in detail in its relevant chapters, we provide an executive summary for each below.

- **CONNECT**: Introduced in [61], CONNECT is a fully featured NoC framework that provides a wide variety of topology, virtual channels and flow control options. CONNECT can also guarantee in-order packet delivery and bounded routing latencies. But this flexibility comes at the cost of expensive FPGA LUT requirements as well as low implementation frequencies (resulting in low throughput). This expensive implementation of the NoC itself in turn results in a low resource environment (both FPGA LUTs and timing slack) for the actual FPGA application to be implemented. This is in stark contrast. Finally, the Fat Tree topology option in CONNECT is

limited in flexibility; it does not allow for mixing switches with different arity at different levels. Similar to deflection routed Hoplite and BFTs, the CONNECT NoC has been studied as a comparison point to benchmark the Flow Controlled Butterfly Fat Trees of chapter 4.

- **AXI-S Interconnect**: A commercial soft IP offered by Xilinx, the AXI-S Interconnect can be used to connect a handful of sub-systems via its monolithic router blocks. Packets are routed over a single stage crossbar composed of multiplexer cascades. Due to its simple crossbar design, the AXI-S Interconnect suffers from steep drops in implementation frequency with increase in endpoint and thus providing very low absolute throughput. Furthermore, only a maximum of 16 endpoints can be supported, which limits the scalability of this crossbar design. The AXI-S Interconnect does offer in-order packet delivery, bounded latencies and out-of-box AXI protocol compatibility. We use the AXI-S Interconnect as a commercial benchmark against the Flow Control Butterfly Fat Trees of 4.

- **SurfNoC**: Introduced in [62], SurfNoC divides the application traffic into domain exclusive waves. These waves can travel unimpeded through the NoC, without any contention. However, intra-domain contention is still possible and can impose long wait times for losing packets to ride another *same-domain* wave. Further, the division of the application traffic into domains is not application customizable. We compare SurfNoC against an ASIC implementation of NoCs generated by the MLE algorithm in chapter 5.

- **PhaseNoC**: Introduced in [63], PhaseNoC also attempts to schedule traffic into waves like SurfNoC [62]. Naturally, it still struggles with intra-domain contention. Similar to SurfNoC, selection of traffic into domains is not application customizable in the case of PhaseNoC as well. PhaseNoC also uses multiplexing at the virtual channel level but this results in increased router complexity. Similar to SurfNoC, we also compare PhaseNoC against an ASIC implementation of NoCs generated by MLE in chapter 5.

Chapter 3

Background: Machine Learning

Modern machine learning algorithms have enabled systems to generalise to progressively more complex and abstract representations of the world; in-turn tackling increasingly challenging tasks and making more accurate predictions. This has been made possible by their increasing ability to extract higher order non-linear features from complex inputs using millions of parameters. They do so by leveraging massive amounts of data and the power of modern day computing. Modern machine learning algorithms have enabled at-scale and performative implementations of: language understanding; speech transcription, autonomous driving, systems and micro-architecture design, to name a few. Due to this massive increase in their representative power, machine learning models have the capacity to match and even surpass human level cognition, thus potentially unlocking copious amounts of productivity in almost every domain.

For developing application-specific NoC designs, certain components of the NoC architecture are exposed as learnable parameters that are tuned specifically for the underlying application such that the application's performance is maximised. However, learning these parameters presents a unique set of challenges. First, an exhaustive search quickly becomes impractical for larger NoC sizes with the search space exploding in size. For example, for learning switches of Chapter 5 ([2]), the search space grows to the tune of 2^P, where P is the size of the Hoplite NoC. Additionally, the absence of a definitive ground truth prevents us from leveraging supervised learning to learn a mapping between NoC parameters and application characteristics. Hence, application-specific NoC designs require leveraging of learning algorithms capable of identifying optimal NoC parameters in absence of available gradients. As a result, we leverage Stochastic and Gradient-Free Evolutionary Learning based algorithms for designing application-specific NoCs in this thesis. Finally, leveraging gradient-free algorithms mandates simulation of thousands of intermediate NoC designs,

which can be slow and expensive using traditional tools like RTL simulation and analytical modeling. As a result, we leverage Gradient-based Supervised Learning with Graph Neural Networks to accelerate the benchmarking of NoC designs. We now explain these algorithms in sections 3.1 and 3.2 below.

3.1 Gradient based Supervised Learning with Graph Neural Networks

Supervised learning is focused around building machine learning systems that can learn and generalise to unseen inputs for a certain task by learning from the ground truth input-output pairings provided whilst training. Supervised regression problems learn to map inputs to continuous outputs whereas supervised classification problems generate discrete class labels as outputs.

We now briefly explain the history of supervised learning with artificial neural networks. This field has a storied past spanning several decades, going from the implementation of a single perceptron in the 1950s [64] all the way to the multi-billion parameter models of today [65] that can extract highly complex and non linear features over long tail temporal dependencies. The first attempt at an *intelligent* neuron was made in 1958 by the design of a single layer perceptron which implemented a linear function of weighted sum over the input [64]. The perceptron was eventually used to create multi-layer artificial neural networks, which expressed more fidelity in the outputs due to their richer representation capacity. The back-propagation algorithm [66] was instrumental in increasing the representative capacity of these artificial neural networks by extracting the gradient of the errors compared to the ground truth with respect to the parameters of the neural network. These gradients were then used to train the parameters of the multi layer models. Over time, enabled by the confluence of modern computing resources and ample data, modern day supervised neural network models have evolved to generate feature maps with deep convolutions [67], make predictions over sequential data with Long Short-Term Memory (LSTM) Recurrent Neural Networks [68] as well as generate attention masks between temporal features with self attention networks [69].

In the sections below, we first introduce the basics of supervised learning in section 3.1.1; concepts that are a prerequisite for training any neural network in a supervised fashion. Then we introduce the architecture of Graph Neural Networks in section 3.1.2, to be used in creating neural network models in chapter 7.

3.1.1 Basics of Supervised Learning with Neural Networks

Architecture of the model: The architecture lays out the topology of the neural network model being constructed. Specifically, the number of layers being used (VGG-16 has 16 layers, for example), the type and size of each layer (2D convolutions, fully connected, self attention etc), and the connection between them (feed-forward, recurrent, residual etc). Finally, the locations and types of non linear activations (ReLU, Tanh etc) applied to some layers of the neural network. We now explain some aspects of the architecture in greater detail below.

- **Neural Network Layer**: A neural network layer is a collection of operations that apply linear transformations on the given set of inputs. A neural network is generally comprised of multiple such layers stacked sequentially, summarised below in Equation 3.1, where f_{l+1} is the mathematical description of layer l+1, o^{l+1} is output of layer l+1, o^l is output of preceding layer l. Observe that output of previous layer o^l becomes the input to next layer l+1.

$$o^{l+1} = f_{l+1}(o^l) \tag{3.1}$$

  The most ubiquitous layer is called the linear layer that implements the operation described in Equation 3.2 below, where W and B are the learnable parameters of the layer, X is the input to the layer, O refers to the size of the layer output, I refers to the size of the input. We visualize this layer in Figure 3.1.

$$f_{linear} = W \cdot X + B; W \in R^{(O,I)}, X \in R^I, B \in R^O \tag{3.2}$$

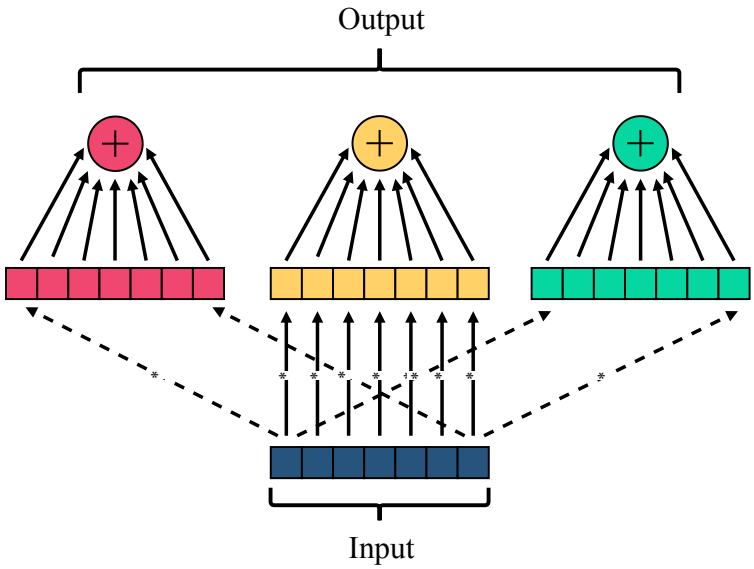


Figure 3.1: Visual representation of a Linear layer.

Convolution layers are also another commonly leveraged layers to make up a neural network. Convolution layers are adept at finding spatial relationships between data points of an input tensor. Convolutions do this by extracting higher order features across various localized points of focus in an input tensor. We formalize this in Equation 3.3 for 2D convolution. While convolutions can be scaled out to support higher dimensions, the underlying principles governing the behaviour remain the same. 2D convolutions consists of 2D learnable weight parameters; referred to as W and of size K $\times$ K. They are also commonly referred to as filters or kernels in literature. Its referred as W in Equation 3.3. The input X is a 2D tensor of size A $\times$ A. The Kernel W *sweeps* over an input patch of the kernel's size, referred to as P in equation below. For the given input patch P and kernel W, each cell is multiplied and added up together to produce a corresponding cell of the output, as shown in Figure 3.2.

$$P^{s,q} = X[s][q] : X[s+W][q+W]$$
$$O[s][q] = \sum_{i=0}^{K-1}\sum_{j=0}^{K-1} W[i][j] \cdot P^{s,q}[i][j]; 0 \leq s \leq A-K-1, 0 \leq q \leq A-K-1 \tag{3.3}$$

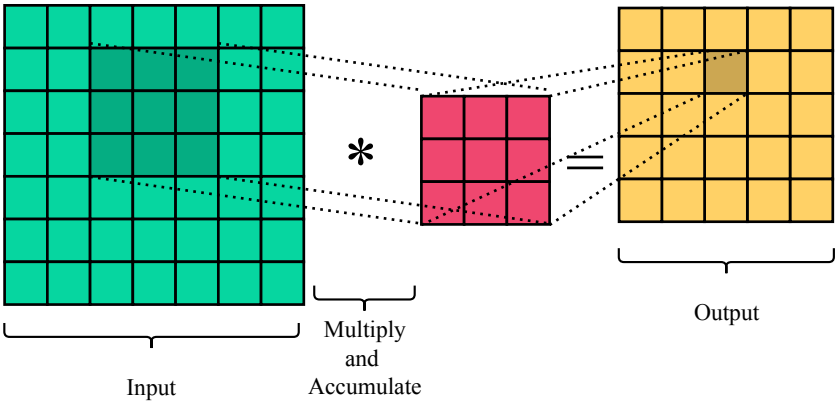


Figure 3.2: Visual representation of a 2D convolution layer.

- **Activation Functions**: Activation functions transform the output of a neural network layer and are not learnt as part of the neural network's training. Activation functions are typically non-linear in nature; critical in bestowing neural networks with the power to model complex, non-linear relationships in the data. This is crucial for solving real-world problems that do not confirm to linear representations. Without activation functions transforming the intermediate outputs between layers of neural networks, the neural network would at-best only be able to represent linear relationships. We now explain the activation function that is used in Chapter 7 to train Graph Neural Networks. `ReLU` is a non-linear activation function that zeros out all negative input values while acting as a simple pass-through for non-negative values, as shown in Equation 3.4. The non-linearity at the inflection point of 0 helps provide high fidelity signals while calculating gradients during the back-propagation phase of training the neural network. This in turn allows the neural network to converge quicker and model complex non-linear relationships between the predicted output and given input.

$$ReLU(X) = max(0, X) \tag{3.4}$$

Objective/ Loss Function: When training the neural network, the loss function is used to calculate the margin of error between the predictions of a neural network and the ground truth. The loss is then the source of gradients calculated for each parameter of the neural network using back-propagation and chain rule of differentiation. The magnitude of the loss function quantifies the margin of error between the model prediction and the actual ground truth. A loss of zero implies complete overlap between the 2 entities.

The exact loss function used depends on the nature of the problem being learnt. Generally speaking, neural network modeling problems can be broadly categorised into 2 categories: 1) Regression and 2) Classification. Regression-based problems refer to predicting continuous output values. Classification-problems refers to cases where neural networks are trained to predict a categorical label instead. In this thesis, we train Graph Neural Networks to predict both continuous and class based variables.

We now present the 2 loss functions we have used in this work in Chapter 7. For predicting continuous values, we leverage the `Huber` loss function, as explained in Equation 3.5. `Huber` loss combines the best of Mean Square Error and Mean Absolute Error by taking into account degree of divergence of predicted output from the ground truth (captured by Δ). For categorical predictions, we leverage the `CrossEntropy` loss, which penalises the neural network on an exponential scale, based on the degree of divergence of the predicted class from the ground truth class, as shown in Equation 3.6.

$$L(x,y) = \begin{cases} 0.5 \cdot (x^2 - y^2), & \text{if } \mid x - y \mid < \Delta \\ \Delta \cdot (\mid x - y \mid - 0.5 \cdot \Delta) & \text{otherwise} \end{cases} \tag{3.5}$$

$$L(x,y) = \log(\frac{exp(x_y)}{\sum_{c=1}^{C} exp(x_c)}) \tag{3.6}$$

Optimizer: Whilst gradients (for every learnable parameter) are calculated using back-propagation, the optimizer is responsible for defining *how* these gradients are used to update the neural network. For example, a straightforward implementation of the optimizer will simply subtract the gradient from its parameter as shown in Equation 3.7 below (W: parameter, E: error, α: learning rate), whereas a more nuanced optimizer might take momentum (past trajectory) into account to vary the step size. For modern neural networks, various flavours of Adam [70] have proved to be popular for efficient convergence over a variety of tasks.

$$W = W - \underbrace{\alpha \cdot \frac{dE}{dW}}_{\text{step size}} \tag{3.7}$$

Dataset: The dataset is the foundational piece for any supervised training of a neural network. It is composed of `input-ground truth` pairs. The `input` is transformed by the many layers of the neural network and the final output is then contrasted against the `ground truth` to calculate the loss and train the parameters. A typical dataset can

contain anywhere from thousands to billions of these entries and varies between datasets and domains. It should be noted that the dataset is typically split into 3 sub-sets: training, validation and test sub-sets. The training sub-set is used to train the neural network. The validation and test sub-sets are never exposed to the neural network while training. Once trained (on the training sub-set), the validation sub-set is used to evaluate the model's performance on never before seen data and accordingly modify the architecture and hyper-parameters of the model. The training $\rightarrow$ validation loop continues until target validation performance is achieved. Subsequently, the test sub-set is then used to benchmark the model and report as final performance since this sub-set is not visible either during training or validation stages.

3.1.2 Graph Neural Networks

Graphs based representations are ubiquitous in almost all domains; molecules in chemistry, interactions in a social network being a few examples. Graph based representations are not just limited to entities naturally composed as graphs; any set of entities and interactions between them can be expressed as graphs. As a result of their ability to capture any kind of relations (or lack thereof) between entities, graphs have been progressively gaining popularity as a universal data structure to express inputs for a neural network model. However, traditional neural network layers (like convolutions, Multi Layer Perceptrons, etc) are only compatible with highly structured and regular data. As a result, traditional neural network architectures are not compatible with graph based inputs out of the box. Although obtuse tensor manipulation for one specific neural network architecture might result in limited success, such a strategy is neither generalisable nor scalable to other network architectures or dynamic graphs. Graph Neural Networks (GNNs) address this very problem by directly operating on graph data. GNNs have already been applied to many applications, including (but not limited to) drug discovery [71], particle physics [72], cyber security [73] etc.

We now provide an overview of the architecture of GNNs. We first discuss the composition of a graph input tensor and then subsequently discuss the application of GNN layers with message passing on this graph tensor. For this discussion, we treat Figure 3.3 as an anchor example that we refer to throughout this discussion.

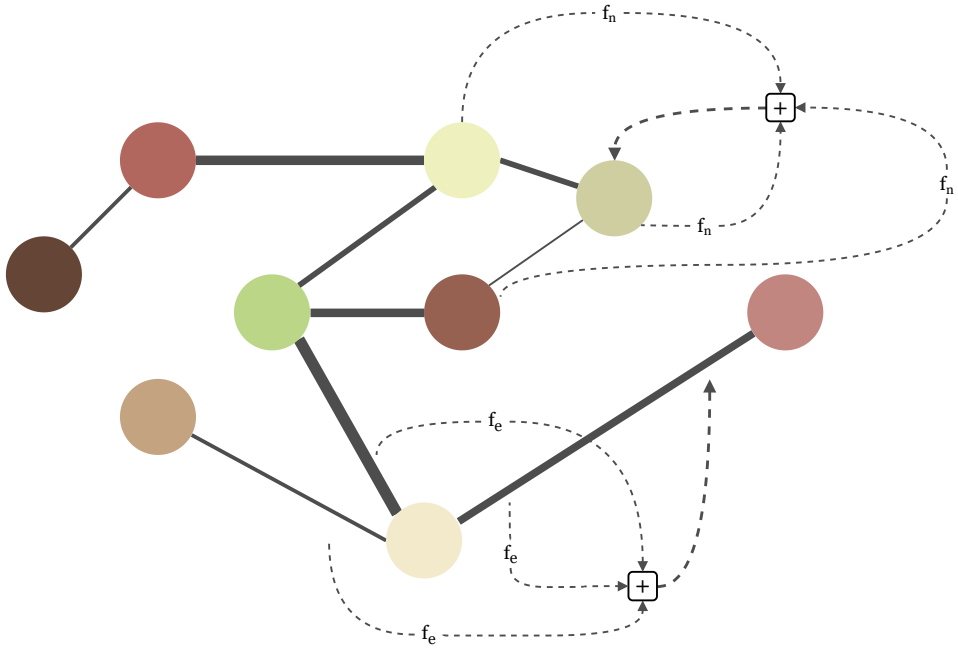


Figure 3.3: A representative example of an input graph and operations of a GNN layer. Operations related to f_n and f_e correspond to examples of message passing for node and edge attributes respectively. The incoming ----→ line shows the updated attributes of the example node and edge after the message passing operation.

Composition of Input Graphs: GNNs accept graphs as inputs. We now explain how an input graph is composed. A graph is composed of nodes and related nodes are connected together via edges. In Figure 3.3, (●, ●, ●, etc) are nodes and connected together by edges (——, ▬, ——, etc). Each node and edge can be encoded with attributes that capture an intrinsic property. For example, in the case of a molecule, the node attribute might encode the type of atom and the edge attribute might encode the type of bond. In Figure 3.3, the colors of nodes represent different node attributes and the thickness of edges represent different edge attributes. Note that although Figure 3.3 creates an example with undirected edges, a graph can have a mix of directed and undirected edges.

Graph Invariance: With respect to Graph Neural Networks, graph invariance ensures that the output of a GNN, for a given input graph, does not change regardless of the ordering and indexing of the input graph's nodes. For example, if an input graph is rotated or the indexing order of the nodes changed, the GNN should still be be able to extract the same meaningful representations from the graph. To express connectivity

between nodes, it is tempting to use a traditional adjacency matrix. However, adjacency matrices are not graph invariant. An Adjacency matrix represents a graph by explicitly encoding connectivity between nodes based on indices. Swapping rows and columns of the adjacency matrix will change the matrix even though the underlying graph remains the same. Additionally, adjacency matrices are memory inefficient when representing large and sparsely connected graphs. Adjacency lists, composed of tuples of connected nodes, address the graph invariance and memory inefficiencies of the matrix approach by encoding each node's connectivity as a list of its neighbor nodes. The order of neighbors within that list does not impact the underlying graph structure. Hence, they are now the standard data structure to express graph connectivity. We refer the reader to [74] for more details on adjacency lists.

GNNs with Message Passing: The simplest implementation of a GNN could simply transform the nodes and edges of Figure 3.3 by operating upon each node and edge attribute individually using a neural network layer (MLPs, convolutions, residual connections etc). A few sequential layers of such isolated operations could then extract higher order features from each node and edge. While valid, such a formulation does not take into account graph connectivity at all. Graph connectivity encodes critical information about relationships between nodes and edges. As a result, inability to operate upon this information can drastically reduce the representational capacity of the GNNs themselves. The message passing operation [75] of GNNs addresses this exact problem. Instead of simply updating an attribute in isolation, the updated attributes of the neighbors are pooled together to form the updated attributes. For example, in Figure 3.3, a node's ( ) attributes are operated upon using the differentiable layer f_n. The neighbouring nodes (,) are also operated upon using the same f_n. Then to form the updated attribute for node ( ), the updated attributes of neighbour nodes (,) are all pooled together. We use the summation operator to pool together attributes in Figure 3.3. Different pooling operators like mean, min, max etc exist. Similarly, the edge attributes are updated using a layer (f_e in Figure 3.3) and pooling together updated edge attributes. Each node and edge in the input graph tensor goes through this transformation (update attributes→pool) and is described as a single layer of GNN. The output graph tensor of the current layer becomes the input graph tensor to the next layer. We now formalise the operations of a GNN layer below:

$$h_i = \alpha(\gamma(h_i, \Omega_{j \in N_i} \phi(h_i, h_j))) \tag{3.8}$$

where h_i are the attributes of a node/edge i. h_j are the attributes of the neighbour j. N_i is the set of all neighbours. ϕ is a differentiable operator (f_n or f_e in Figure 3.3). Ω is

the pooling operator (like sum, max, min etc). γ is another differentiable function. Finally, α is a linear or non linear activation function. A GNN can be composed of multiple layers, with each layer defined by a different set of differentiable functions γ and ϕ, as well as different pooling operators Ω.

3.2 Stochastic Gradient Free Evolutionary Learning

Gradient free solvers can lead to faster discovery of solutions with superior quality when applied to a certain class of problems where the gradients (toward global optimum) can be noisy, unavailable or expensive to compute. These solvers, also used in this thesis in Chapter 5 ([2]) and Chapter 6 ([16]), use an iterative and gradient free process of evolution (mutation $\rightarrow$ selection $\rightarrow$ update) to bias the parametric distribution towards the set of best performing candidates. Evolutionary learning strategies have been used in robotic applications [76], to design hardware for neural network inference [18], place macro blocks in hardware designs [77], train reinforcement learning policies [78], design and even train deep neural networks [79, 19]. Due to their ability to converge towards global optimums without gradients, such solvers are a superior alternative to back-propagation based methods when optimising problems with intractable gradients and high dimensional feature spaces.

In this thesis, we use 2 such gradient free solvers: 1. Maximum Likelihood Estimation (MLE) in Chapter 5 ([2]) to learn switch configurations and 2. Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) in Chapter 6 ([16]) to learn regulation rate for application traces. Both problems (switch learning and rate learning) do not have obvious gradients available to guide the search towards a set of globally optimum parameters. The choice to use MLE and CMA-ES is motivated by matching the complexity of the problem to the algorithm used. For example, MLE is used to learn switches modeled as a simple Bernoulli distribution whereas CMA-ES is used to learn rates modeled as a far more complex multivariate Gaussian distribution. As discussed above, these algorithms are an ideal candidate to solve such problems; MLE (switch learning) and CMA-ES (rate learning) improve worst case latencies by up to $\approx$6.8$\times$ and $\approx$1.5$\times$, over their respective baselines.

Covariance Matrix Adaptation belongs to a class of algorithms based in Evolutionary Strategy while Maximum Likelihood Estimation can be classified as a statistical inference algorithm. Before discussing these algorithms in detail, we first introduce the concept of Bayesian Statistics to ground the discussion in bayesian mathematics in section 3.2.1. We then present Covariance Matrix Adaptation Evolution Strategy in section 3.2.2 and Maximum Likelihood Estimation in section 3.2.3 below.

3.2.1 Bayesian Statistics

Bayesian statistics defines a statistical model of conditional probabilities where the updates to the model are rooted in observational data using Bayes theorem [80]. Specifically, Bayesian statistics quantifies the conditional probability model as the degree of confidence in a hypothesis (H) given a particular evidence (E). This differs starkly from the frequentist interpretation of probability, where the probability of hypothesis is derived by counting the frequency of events in which the hypothesis is true, with the number of trials $\rightarrow \infty$.

We now explain the Bayes theorem with an example motivated from [81]. Our goal in this example is to estimate the probability that a particular hypothesis (H) is true given some evidence (E). For simplicity we limit both hypothesis and evidence to be binary classes, as shown in Equations 3.9 and 3.10 respectively.

$$\begin{aligned} P(E_1) &= \neg P(E_2) \\ P(E_2) &= \neg P(E_1) \\ P(E_1) + P(E_2) &= 1 \end{aligned} \tag{3.9}$$

$$\begin{aligned} P(H_1) &= \neg P(H_2) \\ P(H_2) &= \neg P(H_1) \\ P(H_1) + P(H_2) &= 1 \end{aligned} \tag{3.10}$$

Bayes theorem helps calculate the probability that a particular hypothesis (H1) is true given a particular evidence (E2). Mathematically, it is coined as $P(H_1 \mid E_2)$. We explain the formulation with the help of Figure 3.4 where our total sample size is T. Out of them, $P(H_1)$ is the proportion of T where the hypothesis H_1 is true and $P(H_2)$ is the proportion of T where the hypothesis H_2 is true. Further, within H_1, $P(E_2 \mid H_1)$ is the proportion of samples showing evidence E_2. Similarly, within H_2, $P(E_2 \mid H_2)$ is the proportion of samples showing evidence E_2. As a result, we can simply brake down our original goal of finding $P(H_1 \mid E_2)$ as the ratio of samples within H_1 showing evidence E_2, divided by the total number of samples showing evidence E_2. We show this formulation below in Equation 3.11.

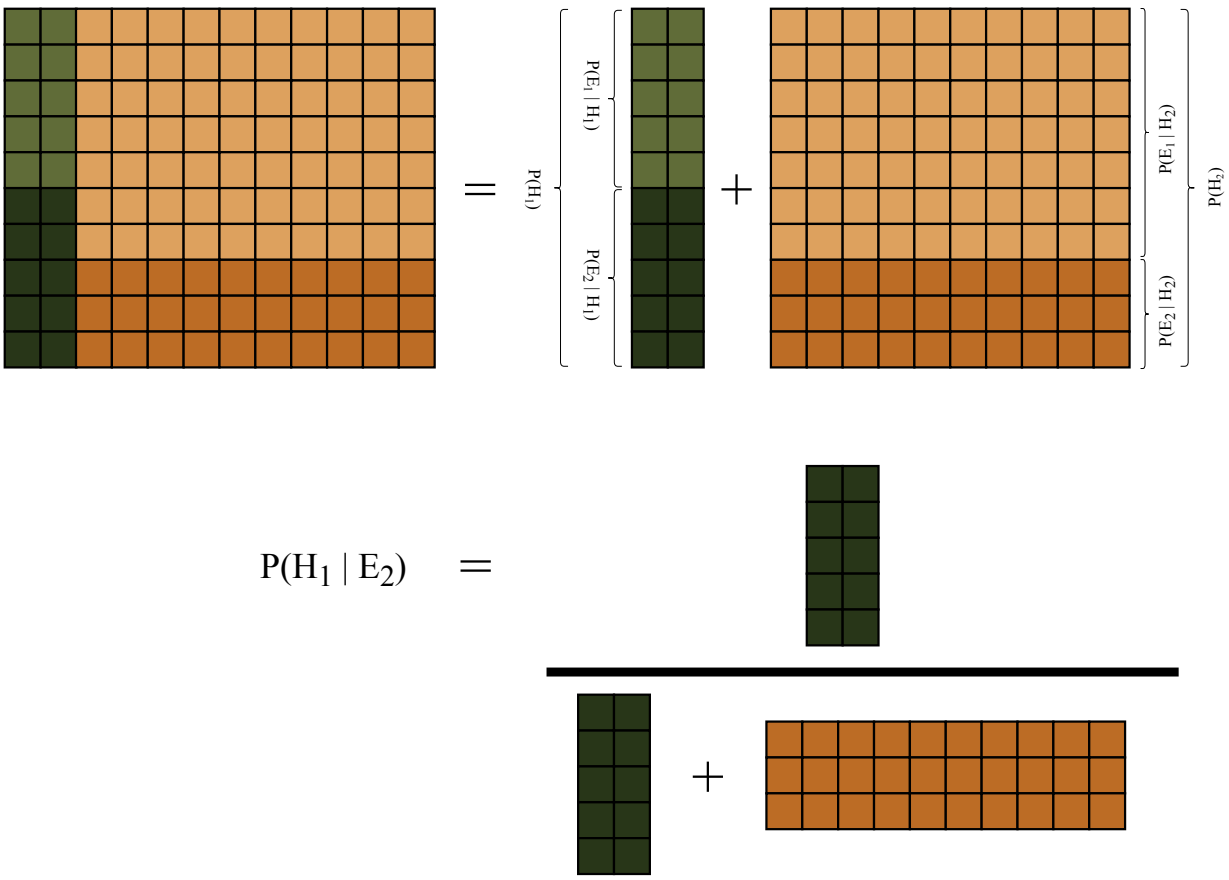


Figure 3.4: A representative example of a sample space consisting of complimentary hypothesis and evidence sets. The **green boxes** corresponds to all samples satisfying hypothesis H_1, with **darker green boxes** further showing evidence E_2. Similarly, the **orange boxes** corresponds to all samples satisfying hypothesis H_2, with **darker orange boxes** further showing evidence E_2.

$$
\begin{aligned}
P(H_1 \mid E_2) &= \frac{T \cdot P(H_1) \cdot P(E_2|H_1)}{T \cdot P(H_1) \cdot P(E_2|H_1) + T \cdot P(H_2) \cdot P(E_2|H_2)} \\
P(H_1 \mid E_2) &= \frac{P(H_1) \cdot P(E_2|H_1)}{P(H_1) \cdot P(E_2|H_1) + P(H_2) \cdot P(E_2|H_2)}
\end{aligned}
\tag{3.11}
$$

The denominator in Equation 3.11 is simply the probability of E_2 being true. As a result, the final form of Bayes theorem is Equation 3.12 below.

$$P(H_1 \mid E_2) = \frac{P(H_1) \cdot P(E_2|H_1)}{P(E_2)} \tag{3.12}$$

3.2.2 Covariance Matrix Adaptation Evolution Strategy

Covariance Matrix Adaptation Evolution Strategy (CMA-ES) is a gradient free and stochastic solver for non-convex (neither convex nor concave) and continuous objective functions. The ability to manipulate continuous parameters allows optimization of NoC design parameters that are continuous in nature. For example, we optimize the regulation rate of each traffic trace in an application specific manner in Chapter 6.

CMA-ES projects the parameters of the objective function as a multivariate gaussian distribution with mean $\mu \in \mathbb{R}^N$ and covariance matrix $\sigma \in \mathbb{R}^{N,N}$ where N is the dimension of the search space. Convergence using CMA-ES is achieved over a number of *evolution* steps (also called *generations*), where the parameters of the gaussian distribution are updated in each step using the equations below:

$$\mu^{g+1} = \mu^g + \alpha\{\sum_{i=0}^{\lambda-1} w_i(x_{i:C}^{g+1} - \mu^g)\} \tag{3.13}$$

$$\sigma^{g+1} = (1 - c_1 - c_\lambda \sum_{i=0}^{C-1} w_i)\sigma^g + \underbrace{c_1 e^{g+1} e^{g+1^T}}_{\text{rank one update}} + \underbrace{c_\lambda \sum_{i=0}^{C-1} w_i y_{1:C}^{g+1} y_{1:C}^{g+1^T}}_{\text{rank } \lambda \text{ update}} \tag{3.14}$$

Equations 3.13 and 3.14 list down the functions to update the mean and covariance parameters respectively of the multivariate gaussian distribution. We now unpack these equations.

For generation (or step) g, the CMA-ES solver first samples the multivariate gaussian distribution (with mean μ^g and covariance σ^g) C times to generate C different set of potential parameters. In line with the genetic inspired naming convention, these C candidates are also called the *population* of *generation* g. Each candidate is tested on the objective function being optimised and the entire population is performance ranked. The top λ performing candidates in generation g are used to evolve the mean μ^{g+1} and covariance σ^{g+1} for the next generation $g+1$. The mean of the multivariate distribution (Equation 3.13) is updated as a weighted w_i sum of best performing candidates $x_{i:C}^{g+1}$ in generation

$g+1$. The $\alpha\{\sum_{i=0}^{\lambda-1} w_i(x_{i:C}^{g+1} - \mu^g)\}$ looks very similar to the traditional back-propagation step with α being the learning rate.

The covariance matrix σ^{g+1} (Equation 3.14) is updated as a recombination of evolution path e (sequence of steps the solver took over the past horizon), current estimated covariance y^{g+1} ($y_{1:C}^{g+1} = \frac{x_{i:C}^{g+1} - \mu^g}{s}$; s is step size) and current covariance σ^g. The step pertaining to rank one update is able to exploit correlations between generations to counter strong biases induced by estimated covariance when a generation has a very small population size. On the other hand, rank λ update can be especially important to extract information from the entire population when the population is large enough for the distribution to remain fairly unbiased. A weighted balance of both can be then controlled by learning rates c_1 for rank 1 updates and c_λ for rank λ updates. We refer the reader to [82] for the complete set of steps that make up the CMA solver.

It is important to note that CMA-ES is not particularly efficient at solving discrete problems where, instead of predicting a set of continuous parameters, we need to find a discrete set of parameters. This is evidenced by CMA-ES not being the most efficient solver for the discrete problem of switch discovery in Chapter 5 ([2]). This can be attributed to the mandated sectioning of the continuous multivariate gaussian distribution into non-overlapping sub spaces for discrete optimization problems. Hence, a solver like Maximum Likelihood Estimation of section 3.2.3 is needed to generate categorical labels for discrete optimization problems.

In this work, we leverage CMA-ES to learn regulation rates for maximising routing performance of hybrid HopliteBuf/BP NoCs for the given FPGA application.

3.2.3 Maximum Likelihood Estimation

A discretized search space is characterized by each parameter in the search space belonging to a distinct set of classes. For example, the switch learning problem of Chapter 5 ([2]) involves learning switch classification between HopliteBuf and HopliteBP modes of operations, for each switch in the NoC. Likelihood based statistical inference is a promising approach that allows for sampling, and based on observed data, evolving discrete probability distributions. The goal of likelihood based statistical inference is to find the best parameters of a pre-assumed distribution that fits the observed data.

Maximum Likelihood Estimation (MLE) based solvers generate solutions by sampling discrete probability distributions. The type of the probability distribution should reflect the solution type being searched for and can greatly influence the convergence effort. For

example, a Bernoulli distribution is best composed for a binary classification problem. Once the probability distribution is selected, MLE tunes the parameters of the distribution such that the probability of generating the best performing solutions is maximised. By casting the problem onto discrete probability distributions best reflecting the problem, MLE based solvers do not require any artificial chunking of the solution space, and are by design more compatible with discretized search spaces (when compared to a continuous solver like CMA-ES). Note that while Maximum Likelihood Estimation can help find the distribution parameters that best fit the observed data, it does not answer whether the chosen distribution is the best option to fit the data.

Similar to CMA-ES, MLE solvers converge towards the global optimum through a series of *evolution* steps with each evolution step marked by sampling of the current generation of probability distribution to produce a population of candidates. The best performing candidates of a generation are then used to seed the distribution parameters of the next generation. We formalise and explain each step of the process below.

$$L(X_\lambda^g|\theta^{g+1}) = P(x_1^g, x_2^g ... x_\lambda^g|\theta^{g+1}) \tag{3.15}$$

$$ln\{L(X_\lambda^g|\theta^{g+1})\} = ln\{P(x_1^g, x_2^g ... x_\lambda^g|\theta^{g+1})\} \tag{3.16}$$

$$\frac{d[ln\{L(X_\lambda^g|\theta^{g+1})\}]}{d\theta^{g+1}} = \frac{d[ln\{P(x_1^g, x_2^g ... x_\lambda^g|\theta^{g+1})\}]}{d\theta^{g+1}} = 0 \tag{3.17}$$

Equation 3.15 defines the likelihood of the best performing samples set $X_\lambda^g = (x_1^g, x_2^g ... x_\lambda^g)$ being generated by sampling of the discrete probability distribution parameterised by θ^{g+1} for generation $g+1$. The best performing set is selected by generating C candidates by sampling the probability distribution using parameters θ^g. Each of the discrete candidate solutions are tested for performance on the objective function and ranked. The λ top performing candidates X_λ^g are then selected for defining the likelihood function of Equation 3.15. Joint probability for a large λ can be numerically unstable so we apply logarithmic transformation to the likelihood function in Equation 3.16. Finally, given we are trying to maximise the likelihood of the best performing solutions being generated when sampled in generation $g+1$, we can differentiate the function with respect to θ^{g+1} in Equation 3.17. Section 5.5 reintroduces these equations from the perspective of using a Bernoulli distribution to learn switch configurations of the Hoplite NoC.

We use Maximum Likelihood based Estimation for the following use-cases in this work:

- Chapter 5: Designing application specific HopliteBuf/BP NoC designs where the switch configuration is learnt on a per-switch basis.
- Chapter 6: Learning the upstream routing paths for Butterfly Fat Trees, optimized to maximise performance for the given FPGA application.

Chapter 4

Butterfly Fat Tree FPGA Overlay NoCs

FPGA overlay networks-on-chip (NoCs) based on Butterfly Fat Tree (BFT) topology and lightweight flow control can outperform state-of-the-art FPGA NoCs, such as Hoplite and others, on metrics such as throughput, latency, cost and power efficiency, and features such as in-order delivery and bounded packet delivery times. On one hand, lightweight FPGA NoCs built on the principle of bufferless deflection routing, such as Hoplite, can deliver low LUT cost implementations but sacrifice crucial features such as in-order delivery, livelock freedom, and bounds on delivery times. On the other hand, capable conventional NoCs like CONNECT provide these features but are significantly more expensive in LUT cost. Butterfly Fat Trees with lightweight flow control can deliver these features at medium cost while providing bandwidth configuration flexibility to the developer. We design FPGA-friendly routers with (1) latency-insensitive interfaces, coupled with (2) deterministic routing policy, and (3) round-robin scheduling at NoC ports to develop switches that take 311-375 LUTs/router. We evaluate our NoC under various conditions including synthetic and real-world workloads to deliver resource-proportional throughput and latency wins over competing NoCs, while significantly improving dynamic power consumption when compared to deflection-routed NoCs. We also explore the bandwidth customizability of the BFT organization to identify best NoC configurations for resource-constrained and application-requirement constrained scenarios.

4.1 Introduction

Overlay NoCs complement hardened NoC resources by providing much-needed configurability and *last-mile connectivity* options to the FPGA developer. The design of efficient overlay NoCs can also help us customize the programmable nature of FPGA resources to different FPGA application requirements. Overlay NoCs such as Hoplite [83] demonstrate a radically different approach towards NoC design for FPGAs built around deflection routing principles that are low-cost and significantly better in cost and throughput compared to contemporary alternatives such as CMU CONNECT [61] and Penn Split-Merge [84] designs. However, deflection-routed NoCs are prone to livelock, unable to deliver packets in-order, and impose reordering cost on the destination. Deflection-Routed Butterfly Fat Trees [36] do marginally better by allowing the bisection bandwidth of the NoC to be customized to match application requirements, but otherwise suffer from rest of the compromises of a deflection-routed design.

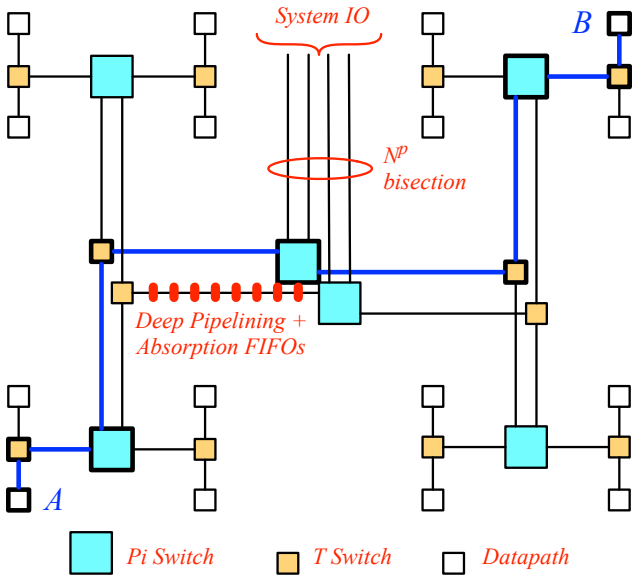


Figure 4.1: High-Level Picture of the Butterfly Fat Tree (BFT) NoC. Configurable bisection bandwidth N^p with p=Rent parameter. Deterministic $A \rightarrow B$ path highlighted. Parametric pipelining of long wires in upper tree stages.

In this work, we show how to design an in-order, low-cost, FPGA-friendly Butterfly Fat

Tree (BFT) NoC that supports customization of bisection bandwidth, without suffering the compromises of a deflection-routed NoC, while exploiting FPGA-specific features. As shown in Figure 4.1, we can configure the bisection bandwidth of the NoC by choosing a combination of *pi* (■) and *t* (■) switches. This is in stark contrast to the Fat Tree topology option in CMU CONNECT [61] that does not provide this parameterization and has a rigid organization. We add deterministic routing support to the Fat Tree routing algorithm to ensure packet flow between a source-destination endpoint takes identical paths. This is highlighted by a candidate path from A→B in Figure 4.1 which takes that strict path even when the *pi* switch hops provide alternatives on the uphill climb. We adopt a lightweight latency-insensitive interface on the NoC ports to (1) ensure packets are delivered in-order, (2) do not suffer the penalty of deflections, and (3) are AXI-S compatible. We augment the latency-insensitive interfaces with absorption FIFOs to allow deep pipelining in the upper stages of the tree, illustrated as ⊢⊣⊢⊣ in Figure 4.1.

They key contributions of this work include:

- Design and RTL implementation of BFT NoC routers with latency-insensitive AXI-S I/O, deterministic routing function, round-robin arbitration, LUT-fracturing friendly mapping, and NoC assembly with configurable bisection bandwidth tuning and optional pipelining support.
- Performance evaluation of the different NoC configurations under synthetic and real workloads. Measurement and analysis of sustained throughput, average latency, worst-case latency, power reduction due to avoided deflections.
- FPGA floorplanning and mapping characterization of resource cost, frequency, and dynamic power.

4.2 Motivation

Modern FPGA IO interfaces include high-bandwidth DDR4 memories (21.3 GB/s), HBM memories (460–480 GB/s), along with next-generation network ports (100–400 Gbps). Besides matching the high data rates of these system-level interconnect, we often require guarantees on ordering, latency, and synchronization on these communication channels. These growing connectivity requirements have long suggested a need for low-cost, high-bandwidth overlay NoCs [83] or hardened NoCs [85] for FPGA chips. In fact, the HBM AXI IP [14] for the Xilinx VU37P integrates two 16×16 256b AXI crossbars to ensure

uniform access to the HBM memory stack. Beyond one-off solutions for such interfaces, we need FPGA overlay and hard NoCs to support intra-FPGA traffic patterns.

Deflection-routed NoCs such as Hoplite [83] [86] and BFTs [36] have upended conventional wisdom in the design and engineering of FPGA overlay NoCs. We refer the reader to sections 2.4 and 2.5 for background information on these NoCs. These NoCs are low-cost, FPGA-friendly solutions that emphasize economy of LUT usage while sacrificing other features of the communication network. Fully-featured NoC frameworks like CONNECT [61] provides developers with a host of choices including topology, buffering, virtual channel support, flow control options, and others to match developer requirements but do so at significant LUT expense. Off-the-shelf IPs such as the Xilinx AXI4-Stream Interconnect exploit mux cascades in Xilinx logic fabric to deliver monolithic AXI-compatible router blocks but have limited scalability (16-endpoints) and poor scheduling.

- **Implementation Cost**: Deflection-routed NoCs are significantly cheaper to realize on FPGAs and even on ASICs [87]. By avoiding the use of buffers, Hoplite and Deflection-routed BFTs offer a cheaper alternative to classic buffered NoCs. They are easier to pipeline owing to fewer logic stages in the internal crossbar and can easily run at high speeds subject to appropriate pipelining. However, deflections keep the NoC switching activity high by forcing the packets to keep traveling in the network.

- **Performance**: Deflection routing offers high absolute throughput when considering implementation frequency (GB/s) but low relative throughputs (packets/cycle). They also suffer from higher worst-case routing latency due to the overhead of deflections. In contrast, conventional NoCs like CONNECT supports higher packet throughput (packets/cycle) but suffers on absolute throughput metrics (GB/s) due to significantly lower implementation frequency (f_{max}). The Xilinx AXI4-Stream Interconnect IP only scales to 16 endpoints but offers competitive throughput.

- **Packet Delivery**: Deflection routers are unable to offer in-order packet delivery due to the unpredictable order of deflections. Deflection routers misroute packets along unintended directions to manage contention without the need for buffering. It is possible to provide ordering in presence of deflection [88] at the cost of a significant reduction in throughput. Stateful streaming FPGA applications require ordered packet delivery and ordered DMA transactions on external DRAMs. Deflection routed NoCs are also susceptible to **livelocks** where unlucky packets keep getting misrouted perpetually. While this behavior is very uncommon, even a possibility

of a livelock makes the design of coherence protocols, implementation of synchronization operations on FPGAs, and realization of real-time applications with strict timing guarantees difficult. Conventional buffered NoCs like CONNECT, and Xilinx AXI4-Stream Interconnect IP can provide in-order delivery at great cost.

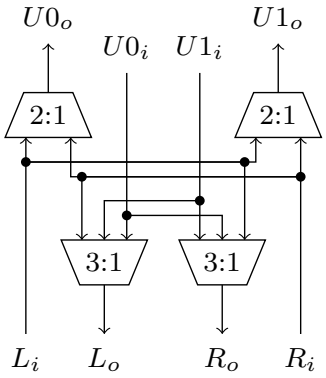
4.3 BFTs with lightweight flow-control

Butterfly Fat Trees offer a configurable platform for the design of FPGA-friendly NoCs. Conventional buffered variants offered by CONNECT are too large and slow, while bufferless deflection-routed variants are cheaper but suffer from livelocks, lower packet throughputs and absence of bounds on packet delivery. *Can we retain the features of a buffered NoC while approaching the low cost potential of bufferless deflection-routed NoCs?* Let us find out by first understanding the root causes of problems on deflection routed BFTs. Then we will walk through the proposal for rectifying these limitations with our BFT NoC design that employs lightweight flow control.

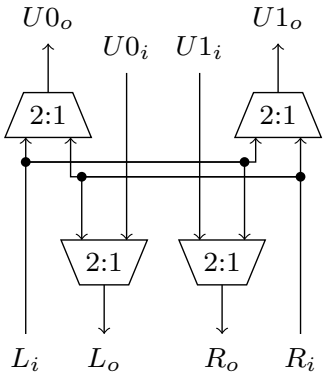
4.3.1 Designing the BFT Switch Micro-architecture

The broad design goals for the switch include in-order delivery, livelock freedom, and bounded routing times.

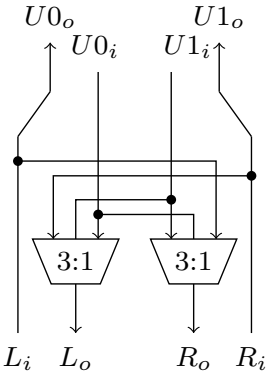
Quest for Deterministic Routing As discussed in Chapter 2.4, the key feature of the classic BFT routing policy allows packets routing via *pi* switches to choose uphill routes as required, depending on local congestion. This baseline switch, shown in Figure 4.2 (a) costs us 2 `2:1` multiplexers (uphill ports), and 2 `3:1` multiplexers (downhill ports). Even if the IO ports are buffered [89], the NoC still delivers packets out-of-order as the uphill ports still provide two choices at each climb. This generous routing freedom can lead to packet reordering that is typically handled with reassembly buffers at the endpoints. If the ports are bufferless [36], with a suitable choice of routing algorithm, the NoC switches are low-cost but, as discussed previously, guarantee neither packet ordering nor livelock freedom. Instead, we can strategically limit this freedom of path selection while distributing bandwidth fairly across the NoC. For data-center BFT networks, deterministic routing can be enforced [90] by scattering packets based on destination address. This is applicable to larger-scale networks as the cost of switch has already been paid and no optimization of switch richness is necessary. Instead, for an efficient FPGA mapping, we investigate LUT-friendly deterministic routing choices as shown in Figure 4.2.



(a) Orig. *pi* switch, all turns allowed.



(b) Downhill locked $U0 \rightarrow L$, $U1 \rightarrow R$.



(c) Uphill locked $L \rightarrow U0$, $R \rightarrow U1$.

Figure 4.2: Pruning the *pi* switch to enforce deterministic routing policy and save multiplexing cost.

- We can limit routing freedom by pruning the downhill muxes and forcing packets to strictly take $U0 \rightarrow L$ and $U1 \rightarrow R$ paths as shown in Figure 4.2b. This reduces the LUT mapping cost of the switch to 2 `2:1` multiplexers (uphill ports) and 2 `2:1` multiplexers (downhill ports).
- Alternatively, we can restrict uphill packets to take $L \rightarrow U0$ and $R \rightarrow U1$ paths as shown in Figure 4.2c. This will eliminate the uphill multiplexers entirely. We now only need to pay the cost of 2 `3:1` multiplexers for the downhill ports.

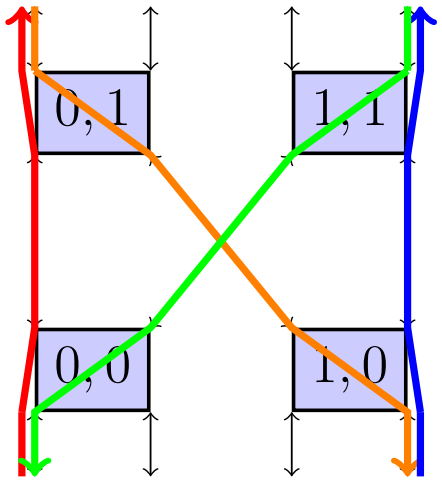


Figure 4.3: Livelock in BFTs: mixing deflection and determinism.

By restricting routing freedom, packets take deterministic paths along the tree and packets for a particular source-destination pair will travel along identical routes in the tree.

However, if we add this restriction to deflection-routed BFTs, *the NoC loses livelock freedom*. A concrete example of this manifestation is shown in Figure 4.3. In this example, the red path is forced to take $L \rightarrow U0$ route at 0,0 and blue path must take $R \rightarrow U1$ at 1,0 thereby creating a cyclic dependency. The orange and green downhill paths also have a fixed route during BFT descents. The four co-dependent flows can lead to livelock. Deflection-routed BFTs force packets to return along the arriving links and create a local dependency on the output resource. When we enforce uphill routing restrictions, packets no longer have the ability to route around congestion. Recall, that during BFT routing, downhill paths are already constrained to a specific path based on the destination address. If a set of packet flows request resources in a cyclic fashion, they are now susceptible to a livelock that previously did not exist under the relaxed routing policy of the BFT NoC.

Quest for Low-Cost Handshaking: The key to breaking cyclic livelocks is to eliminate the dependency on the deflected outgoing port. Instead of sharing the outgoing port for both deflected packets and nominally routed packets, we choose to use a lightweight latency-insensitive interface for handling contentions, as shown in Figure 4.4. With latency-insensitive protocol, the contending packet is stored in a shadow register at the input port itself rather than deflecting it on the output port. This removes the dependence on the output port and eliminates livelock. This makes the NoC handshake-driven (**valid, back-**

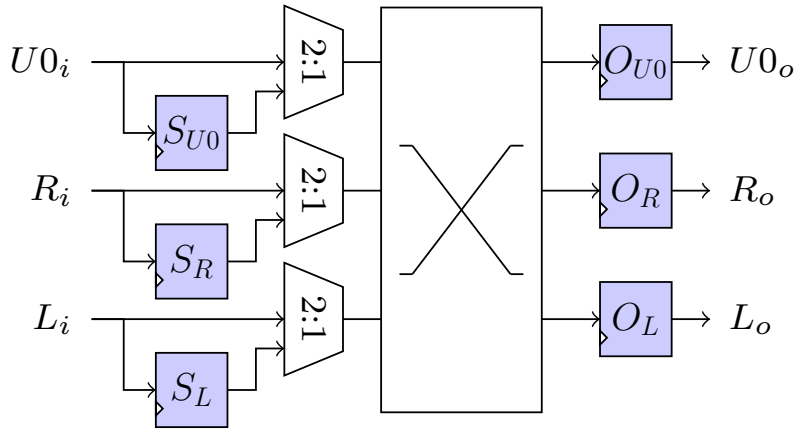


Figure 4.4: Lightweight Latency-Insensitive Interface integration with the BFT t switch. Each input port has a cheap shadow register (single FF, or an SRL32 absorption FIFO).

pressure protocol) rather than built on deflection-routing principles. Previous embodiments of this handshake [91] suffered from some key limitations. They were (1) mapped to cheap SRL FIFOs but limited by the logic cost of complex flow control, (b) throttled throughput by 50% to simplify the control logic for handshaking, and (c) compromised frequency of the mapped design in the quest for low LUT footprint. Neither of these alternatives are effective for NoCs where the cost of every extra LUT matters. We adapt the latency insensitive interface to compose seamlessly with the multiplexing stage of our internal switching crossbar to lower overheads. A side-effect of the adoption of handshaking, combined with the proposed deterministic routing, also enforces ordering guarantees on the packets in addition to livelock-free deterministic routing.

Round-Robin Arbitration: The final piece of the puzzle is the choice of arbitration policy. We manage contentions using a fair, work-efficient arbiter that provides service to contending packets in a round-robin fashion. Since packets are not deflected, they just stay in the shadow registers, service is guaranteed and no local livelocks are possible. Conventionally deflection-routed NoCs [36] can suffer from local livelocks due to mismatch between pipelining latency on the link and service cycle period of the round robin arbiter.

4.3.2 FPGA Mapping

We now investigate the FPGA LUT cost and implementation frequency of the different BFT router variants in Table 4.1 for the Xilinx UltraScale+ VU9P FPGA with Vivado 2017.4 running synthesis, placement and routing. In contrast to the Deflection BFT

router [36], we introduce a lightweight shadow register module and a different round robin arbiter. For the *pi* switch proposed in this work with deterministic uphill paths, we are able to save the cost of the muxes but still need to pay for the latency insensitive interface blocks. Thus, our proposed *t* and *pi* switches are 1.5–3× larger than their vanilla deflection-oriented counterparts but provide 1.) livelock-free routing, 2.) in-order delivery, and 3.) a many-fold increase in throughput, latency and power performance, as illustrated in Section 4.4. We also augment our lightweight router with deep pipelining along the links supported with lightweight 32-deep absorption SRL FIFOs. This further increases area cost by 1.5× and drops frequency slightly. Later, we see in Section 4.4, that the designer may choose between wire-rich or LUT-rich alternatives for a given workload.

In contrast to a recent [92] partial-reconfiguration friendly implementation of a 32-node 64b deflection-routed BFT taking 29K LUTs on a Xilinx ZU9EG MPSoC, our topology takes ≈30K LUTs on a VU9P FPGA and provides multiple extra benefits in exchange for the modest increase in LUTs and identical wiring requirement.

Table 4.1: BFT *t* and *pi* switch cost (32b data + 4b addr).

Architecture	**Kind**	**LUTs**	**FFs**	**Freq.** (MHz)
Deflection BFT [36]	*t*	122	146	1176
	pi	207	145	965
This work	*t*	311	216	780
	pi	375	288	881
This work + SRL FIFOs [91, 89]	*t*	530	282	751
	pi	548	378	875

4.4 Results

In this section, we report the outcomes of the experimental evaluation of the our Flow-controlled BFT FPGA overlay NoCs and other contemporary NoCs under different traffic patterns. We quantify metrics such as throughput, latency (source queuing+inflight), resource cost and power. We aim to establish the trade-offs between the different NoCs and how to choose between them under various constraints.

Experimental Setup: For synthetic workloads, we investigate uniform random and locality-aware traffic patterns that send single-flit messages (1024 packets/client). We ex-

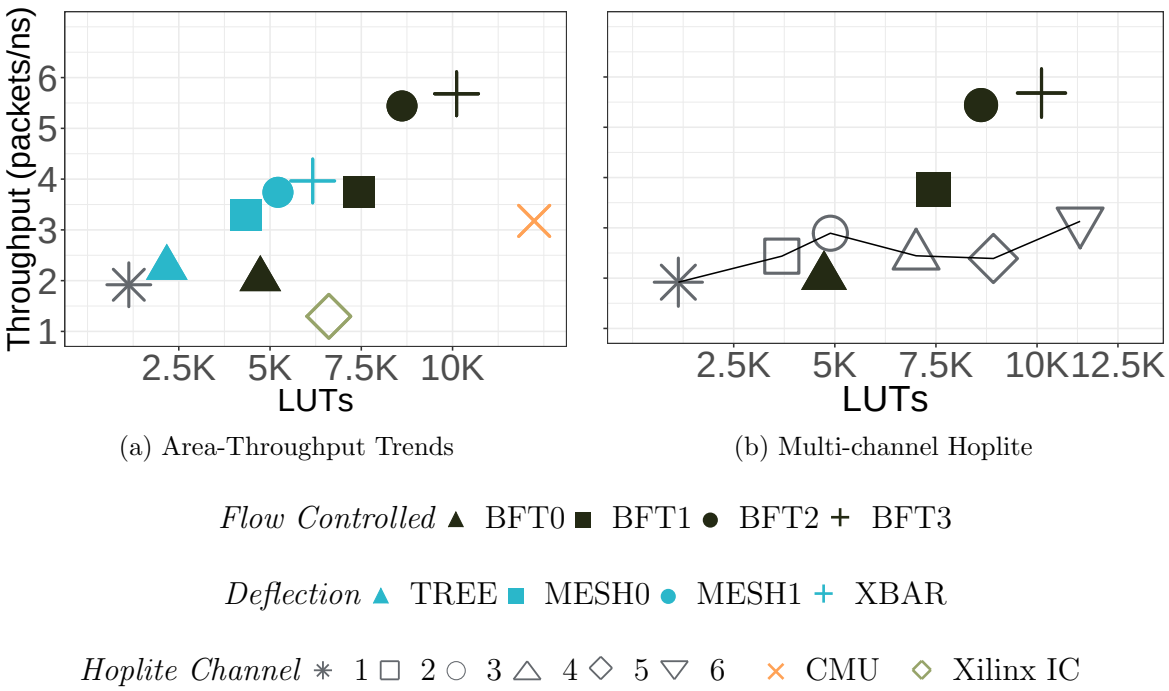
tract real-world traces from FPGA accelerators for Sparse matrix-vector multiplication, graph processing, and Token LU factorization graphs with 10K–1M messages/benchmark. We identify Hoplite, Deflection BFT, CMU CONNECT, and Xilinx AXI4-Stream Interconnect (IC) as the set of NoCs for head-to-head comparison across a range of system sizes from 2–64. For the Flow-controlled BFT, we choose four configurations (BFT0–3) between and inclusive of the binary tree (BFT0, all t switches) and multi-stage crossbar topologies (BFT3, all pi switches). We mimic similar four BFT configurations from the Deflection BFT work [36] (TREE, MESH0, MESH1, and XBAR), shown previously in Figure 2.2. The CMU CONNECT BFT is configured to have 8-deep buffers, peek flow control, and separable input-first round-robin arbitration to closely match our testing conditions. We run cycle-accurate simulations using a combination of `iverilog`, `verilator`, and `Xsim` as required by the different NoCs. All hardware mapping experiments targeted the Xilinx UltraScale+ VU9P FPGA (xcvu9p-flga2104) with a -3-e speed grade using Vivado 2017.4 under default synthesis and implementation settings. We floorplan the NoCs to span the chip dimensions and optimize layout for wiring distribution, allowing us to accurately measure the impact of the NoC, in isolation, on the placement and routing of FPGA resources.

4.4.1 Cost-Power-Performance Tradeoffs

In Figure 4.5a, we show sustained throughput (packets/ns) as a function of LUT utilization at 16 client system size. It is clear that Hoplite has the smallest LUT footprint of all NoCs, but the superior bisection bandwidth of the BFT NoCs help boost their sustained rates by as much as 3× over Hoplite. The BFT proposed in this work is more expensive than the deflection variant by 1.5–2× but delivers 1.5× better throughput in exchange. The CMU CONNECT NoC is significantly more expensive (1.5–3×) than the BFT NoCs and delivers 1.8× lower throughput, while the Xilinx IC has remarkably poor 6× less throughput than our BFTs.

It may seem tempting to exploit the low-cost nature of Hoplite to create replicas of the NoC and use it instead of paying for the more expensive BFT. In Figure 4.5b, we quantify the fallacy of this thinking. The baseline Hoplite design consumes 1.1K LUTs compared to the 10K LUTs of our fully-feature BFT. For RANDOM workload with 100% injection rate, we observe that multi-channel Hoplite with 3 channels is able to deliver a better throughput result compared to a BFT0 NoC. As we scale channel count, the Hoplite throughput saturates and is not competitive with rest of the BFT NoC configurations. This is for two reasons: (1) a drop in implementation frequency as we add more wires to the multi-channel NoC, and (2) ejection bottleneck where packets try to exit the NoC from

the multiple channels. Thus, a bisection configurable network like a BFT offers better use of wiring resources than naive replication of a low-cost topology.



(a) Area-Throughput Trends

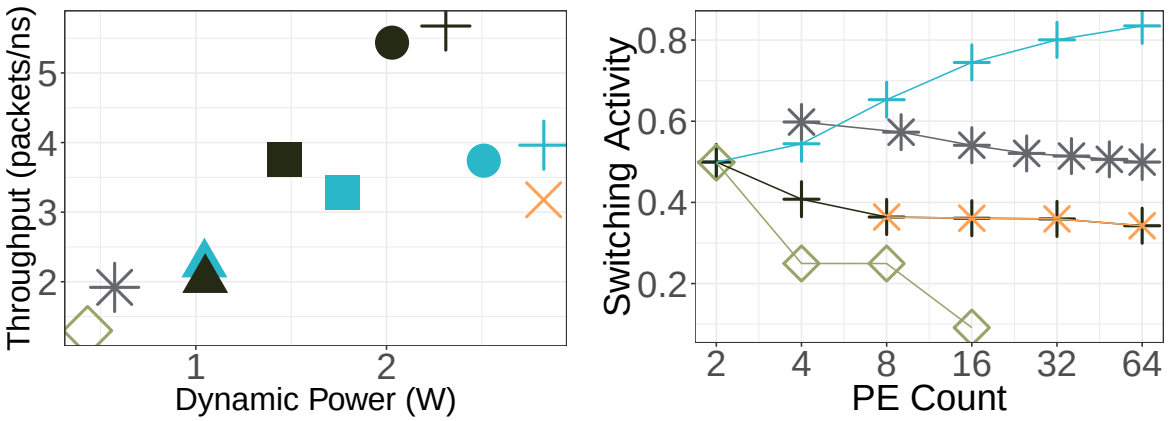
(b) Multi-channel Hoplite

Figure 4.5: FPGA LUT cost-performance trends for different NoCs. 16 clients, RANDOM traffic at 100% injection.

Next, we investigate power-throughput tradeoffs to primarily quantify the benefit of avoiding deflections. We expect the use of lightweight handshakes to eliminate the switching activity due to deflections. We compute dynamic power using `Xilinx Xpower` by supplying switching activity rates. We extract these switching activity rates from simulation data. In Figure 4.6a, for RANDOM workloads at 100% injection rate, we observe a simultaneous 1.1–1.2× reduction in dynamic power and 1.1–1.5× boost in throughput when comparing our BFTs with their deflection variants. While Hoplite has low dynamic power due to its dramatically lower LUT cost, it has high switching activity and struggles to meet the throughput capabilities of the BFT NoC.

In Figure 4.6b we measure switching activity as a function of system size for RANDOM workloads at 100% injection rate. As expected, deflection-based topologies like Hoplite and Deflection BFTs are characterized by high 60–80% activity rates. Our BFTs exhibits the

low switching rates of 40–50%. The Xilinx IC is a single-stage crossbar and thereby exhibits the lowest activity rates.



(a) Dynamic Power-Throughput

(b) Switching Activity-System Size

Flow Controlled ▲ BFT0 ■ BFT1 ● BFT2 + BFT3

Deflection ▲ TREE ■ MESH0 ● MESH1 + XBAR

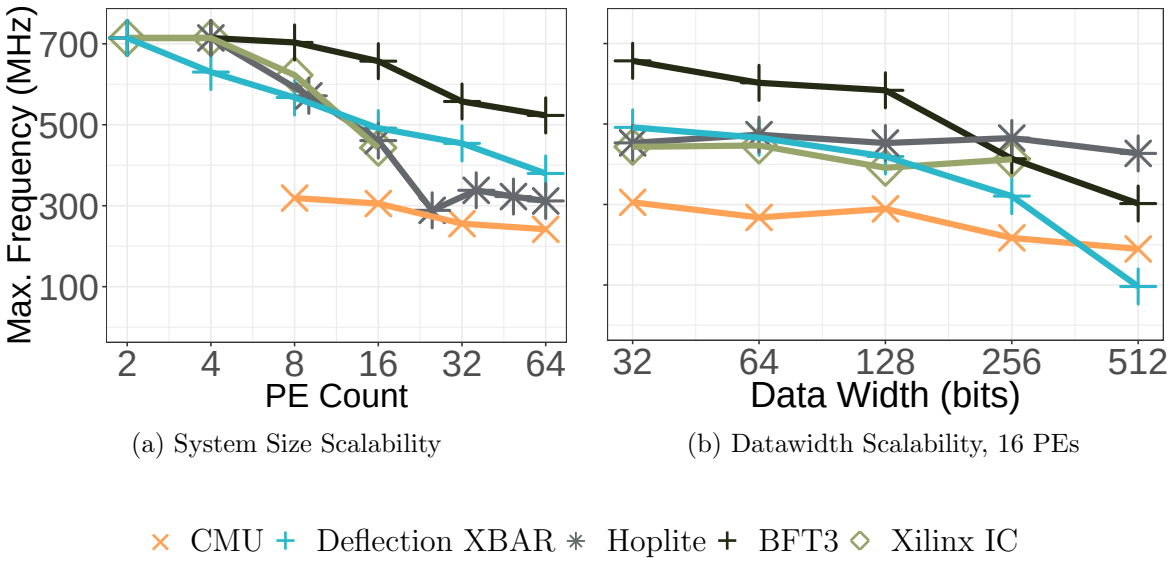
✳ Hoplite × CMU ◇ Xilinx IC

Figure 4.6: FPGA Dynamic Power-performance trends for different NoCs. 16 clients, RANDOM traffic at 100% injection.

4.4.2 Understanding FPGA Mapping Trends

To understand the source of these LUT cost efficiency trends, we now investigate the LUT implementation cost and operating frequency of the different NoC configurations.

In Figure 4.7a, we implement the different NoCs with 32b datawidth and scale PE counts. We observe that our BFT is the fastest NoC and runs as fast as 700 MHz (2–8 PEs) and degrades to 500 MHz (64 PEs). It is also better than the deflection variant of the BFT by ≈1.25×. The Hoplite frequency generally keeps pace with the BFT and only slows down at larger system sizes. With aggressive link pipelining even at large system sizes, it is possible to achieve frequency-parity between Hoplite and BFT topologies. The CMU interconnect is the slowest NoC in this comparison due to complex router micro-



(a) System Size Scalability

(b) Datawidth Scalability, 16 PEs

Figure 4.7: FPGA frequency characterization as a function of system size and datawidth.

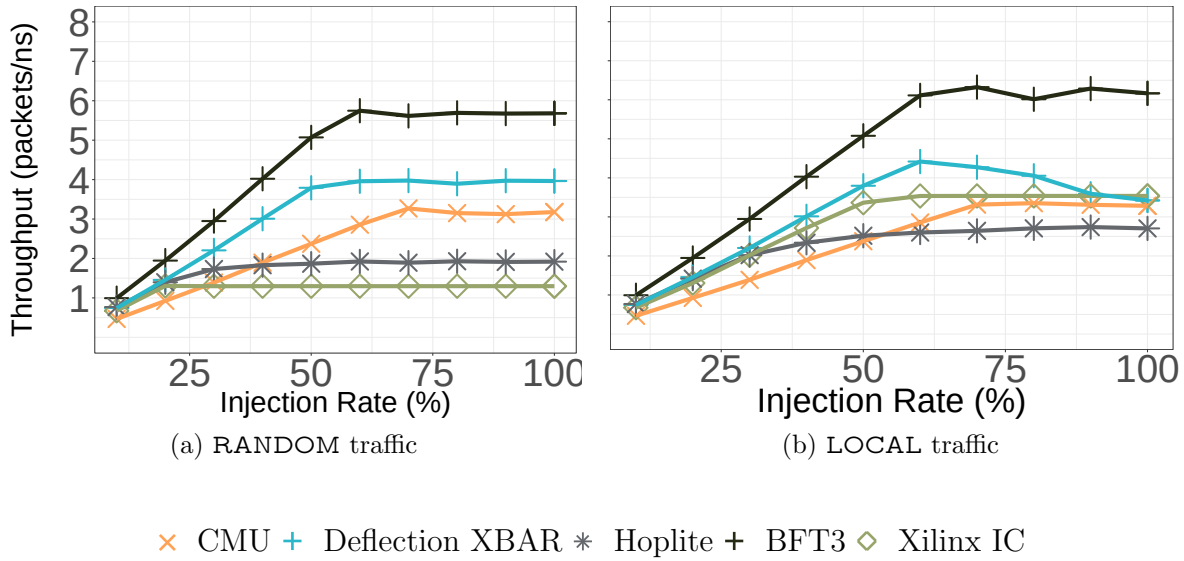
architecture. The Xilinx AXI4-Stream Interconnect is fast at small system sizes but its frequency drops to 400 MHz at 16 clients due to a simple, low-cost design.

In Figure 4.7b, we scale the datawidth of the NoC with 16 clients. As expected, all NoCs suffer a frequency drop with widening links consuming more FPGA resources. Below 128b links the BFT NoC proposed in this work offers the highest frequency across the set of NoCs. As we scale beyond 128b, the frequency of Hoplite and Xilinx AXI4-Stream interconnect is faster at around 500 MHz while the BFT drops to around 300 MHz. Note that with aggressive pipelining (extra register cost) of the links all NoCs can be made to achieve similar operating frequencies, conditional upon availability of registers in the FPGA fabric. Certain FPGA architectures, like Intel's Hyperflex based line of FPGAs, also provision for dedicated pipelining registers in the programmable fabric outside the standard set of registers, allowing for a 1.3-1.6× increase in implementation frequency at no extra cost [93]. While additional pipelining will not affect the cycles-based throughput of the NoC, it will regress the cycles-based routing latency since packets will have to hop over increased number of links to reach their destination PE.

4.4.3 Effect of Injection Rate of Synthetic Traffic

When considering synthetic traffic patterns, we must examine the effect of injecting packets into the NoC at varying rates and measure sustained rates and average latency at saturation.

For RANDOM (Figure 4.8a) and LOCAL (Figure 4.8b) workloads, on a 16 client system, the BFT NoC outperforms all other NoCs in our comparison by 1.5–2 × (worst case) and by 3–6× (best case). The key reason for this difference is the ability of our NoC to avoid the penalty of deflections (Hoplite, Deflection BFT), faster frequency (CMU CONNECT), and fairer arbitration (Xilinx AXI4-Stream Interconnect). As expected, the LOCAL traffic pattern suits the BFT architecture nicely as locality in the communication trace matches the hierarchical structure of the NoC.



(a) RANDOM traffic

(b) LOCAL traffic

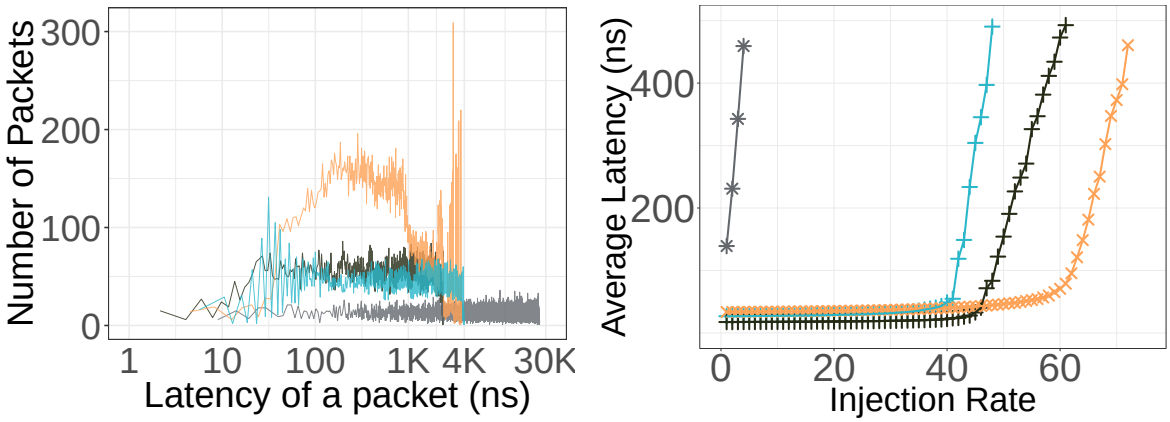
Figure 4.8: Sustained Throughput trends for different NoCs for 16 clients and for RANDOM and LOCAL traffic patterns.

In Figure 4.9a, we investigate the latency distribution trends of the different NoCs for 64 clients with RANDOM traffic and 100% injection rate. The long tail distribution of deflection-routed Hoplite NoC is clearly evident in this plot. We are able to marginally outperform the buffered CMU CONNECT NoC by 1.1× due to a faster implementation frequency of our design. We are also 1.7× better than a Deflection BFT.

In Figure 4.9b, we plot average latency (before network saturation) for varying injection rates under RANDOM traffic. As expected, deflection-routed NoCs like Hoplite saturate very quickly at 2% injection rate. The CMU CONNECT NoC delivers much better average latency behavior due to buffers. However, the BFT NoCs are ultimately superior due to richer wiring bandwidth of the upper stages.

4.4.4 Effect of System Size

In Figure 4.10, we quantify the effect of PE scaling on NoC throughput and worst-case latency for synthetic (RANDOM) and real (geometric mean of various benchmarks) workloads at 100% injection rate. The flow controlled BFT NoC is the clear winner with highest throughputs for both synthetic and real traffic and the lowest worst-case packet routing latencies.



(a) Latency distribution

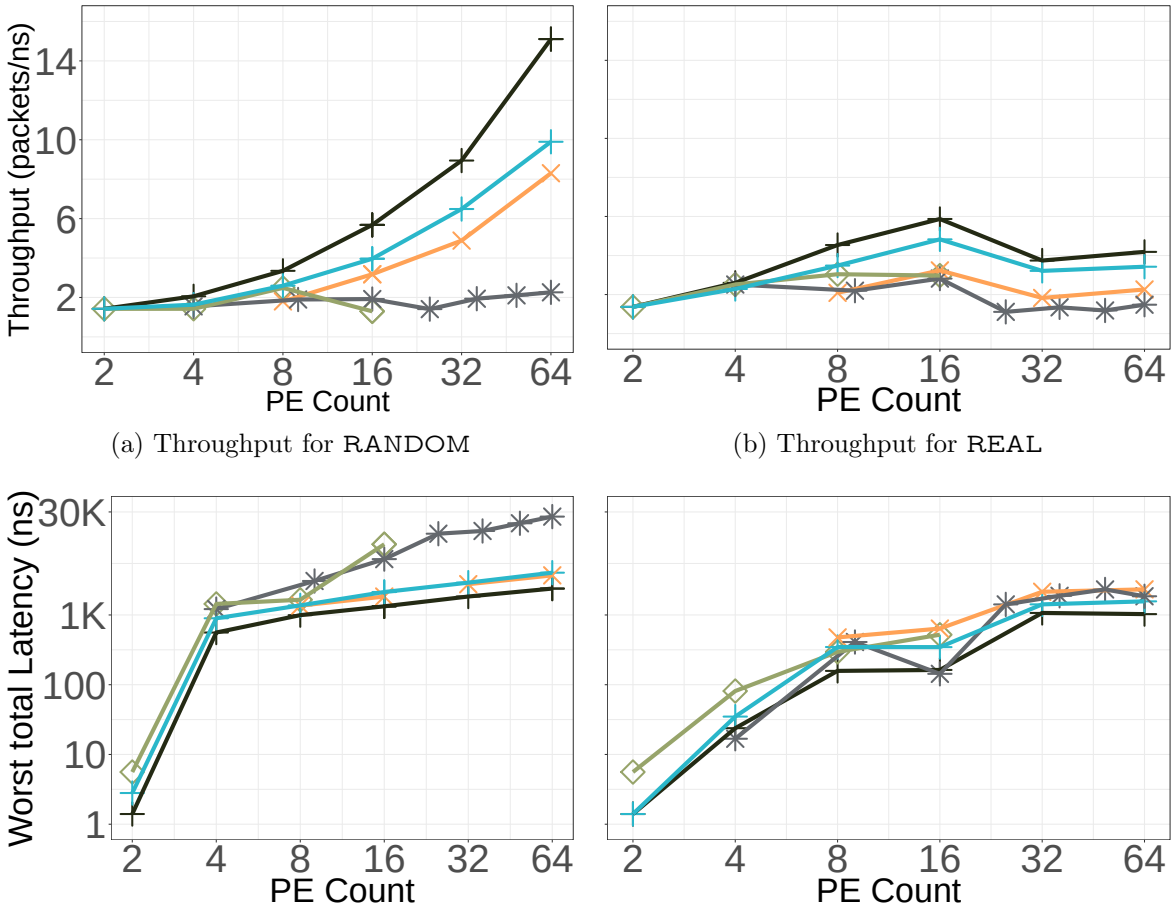
(b) Average Latency

Figure 4.9: NoC packet latency trends for 64 client NoCs using RANDOM traffic pattern.

The sustained throughput of our proposed BFT is able to outperform CONNECT by 1.75× and the deflection variant (XBAR) by 1.4× at 64 clients with RANDOM workload. For real workloads the gap is similar at 1.8× (vs CONNECT) and 1.2× (vs deflection variant) suggesting that the wins hold even for realistic traffic patterns. Deflection overheads

eventually choke Hoplite and yield low sustained throughputs and longer packet latencies. CMU CONNECT is a buffered NoC and should offer higher throughputs but suffers from lower operating frequencies as observed previously in Figure 4.7a. The Xilinx AXI4-Stream Interconnect does not scale beyond 16 clients, and suffers from high contention due to single-stage design and lower frequency.

The worst-case latency wins for the BFT under synthetic `RANDOM` traffic are as large at 6× over deflection-prone Hoplite and Xilinx AXI4-Stream interconnect while as larger as 1.5× when compared to deflection XBAR and CMU CONNECT. We see a similar trend for real workloads (`geomean`), where the latency gap is 1.5- 2.3× smaller than other NoCs.



(a) Throughput for RANDOM

(b) Throughput for REAL

(c) Worst-Case Latency for RANDOM

(d) Worst-Case Latency for REAL

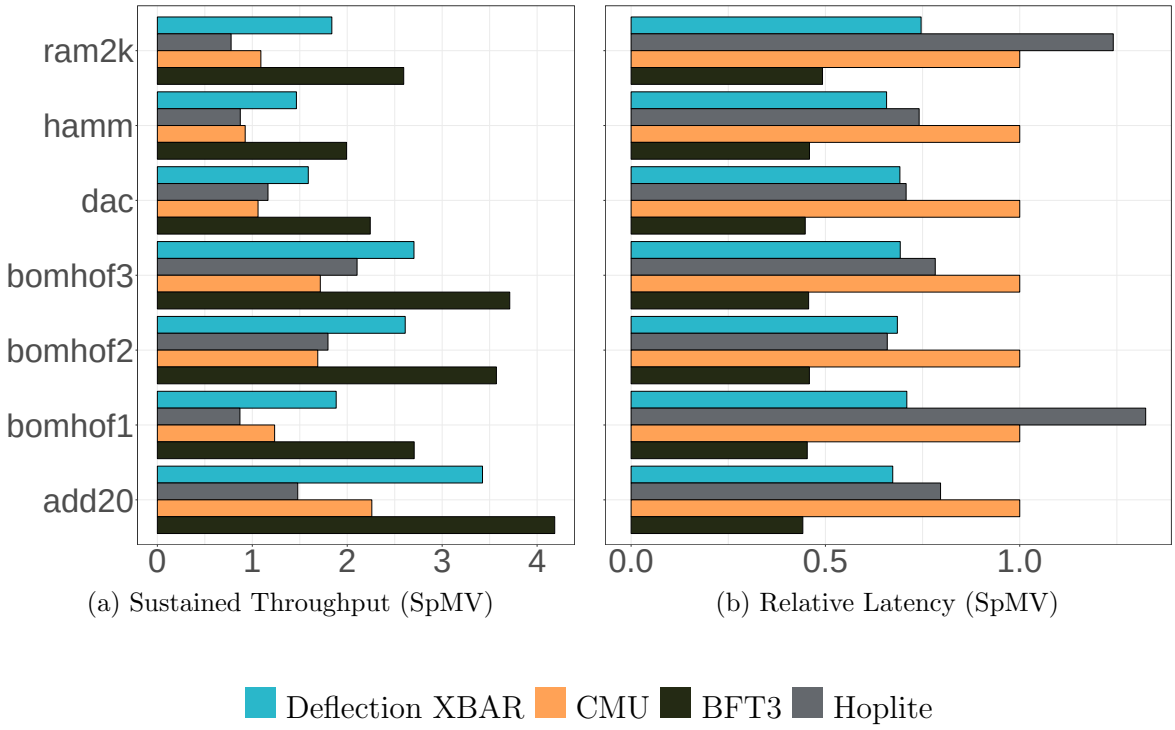
Figure 4.10: PE scaling impact on throughput and worst-case latency for RANDOM and real workloads (geomean).

4.4.5 Routing Realistic Workloads

We evaluate 64 client systems running realistic FPGA accelerator workloads and show sustained throughput and worst-case latency in Figures 4.11 and 4.12. These communication traces for real workloads have 1K–1.5M packets depending on the dataset. We exclude

Xilinx AXI4-Stream Interconnect since it does not scale beyond 16 clients.

For Sparse Matrix-Vector Multiplication (SpMV) workload shown in Figure 4.11a, our BFT NoC is able to outperform CMU CONNECT by 2–2.35×, deflection routed XBAR by nearly 1.2 –1.5× and Hoplite by 1.8–3.2×. Graph Analytics workloads , shown in Figure 4.12a, have a high degree of locality which allows our BFT to outperform CONNECT by 1.25–2 ×, deflection XBAR by 1.65-2.25×, and Hoplite by 1.7×.



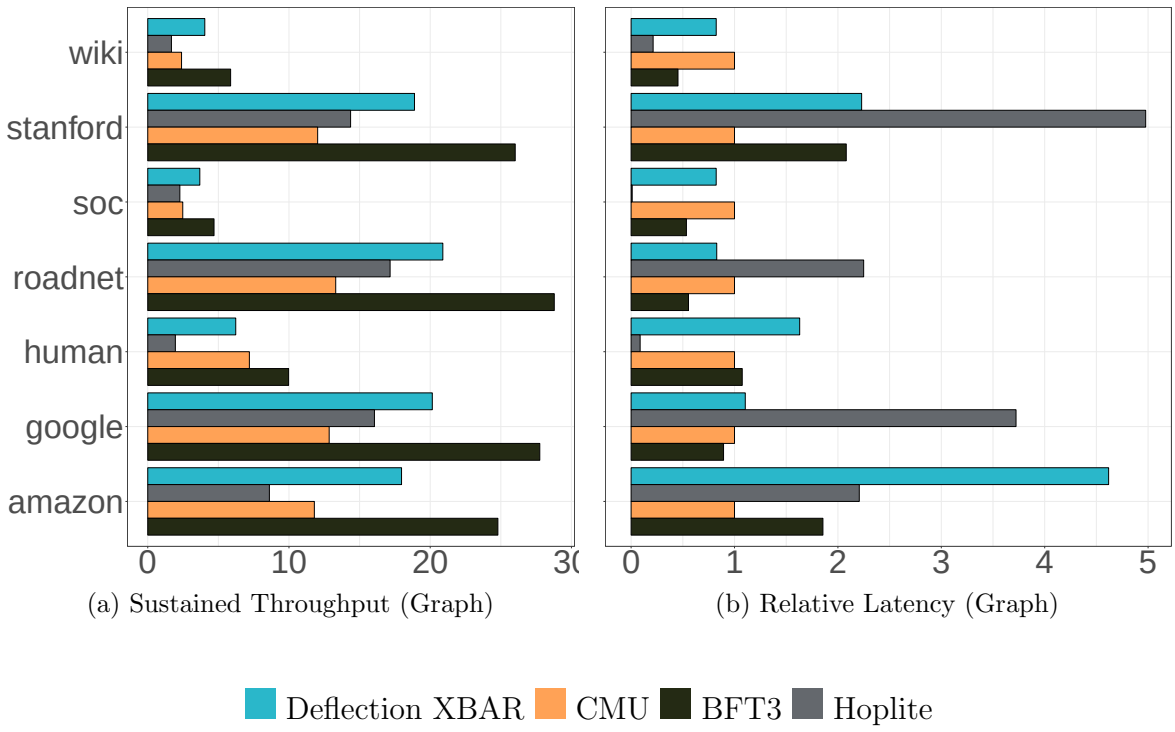
(a) Sustained Throughput (SpMV)

(b) Relative Latency (SpMV)

Figure 4.11: Throughput-Latency trends for `REAL` SpMV traffic.

We measure worst-case end-to-end latency (source queueing + in-flight) for routing realistic workloads. The worst-case latency figure is scaled with the CMU CONNECT latency as baseline. Our BFT is able to reduce worst-case latency for SpMV workloads shown in Figure 4.11b by around 2.3× over CONNECT , 1.5× over Deflection XBAR and 3× over Hoplite. For Graph analytics, shown in Figure 4.12b, the latency wins range between 1.1-2× over CONNECT and 1.1-1.7 × over Deflection BFT. The `stanford` and `amazon` workload graphs have extremely high locality with very little message communication over the NoC (most messages ≈80% stay in the same client). Hence, the routing

latency observed is in the order of a few cycles due to very light NoC injection rate. Hoplite outperforms our BFTs for Graph analytic workloads such as `wiki`, `soc` and `human`, but not on throughput as we saw previously on Figure 4.12a.



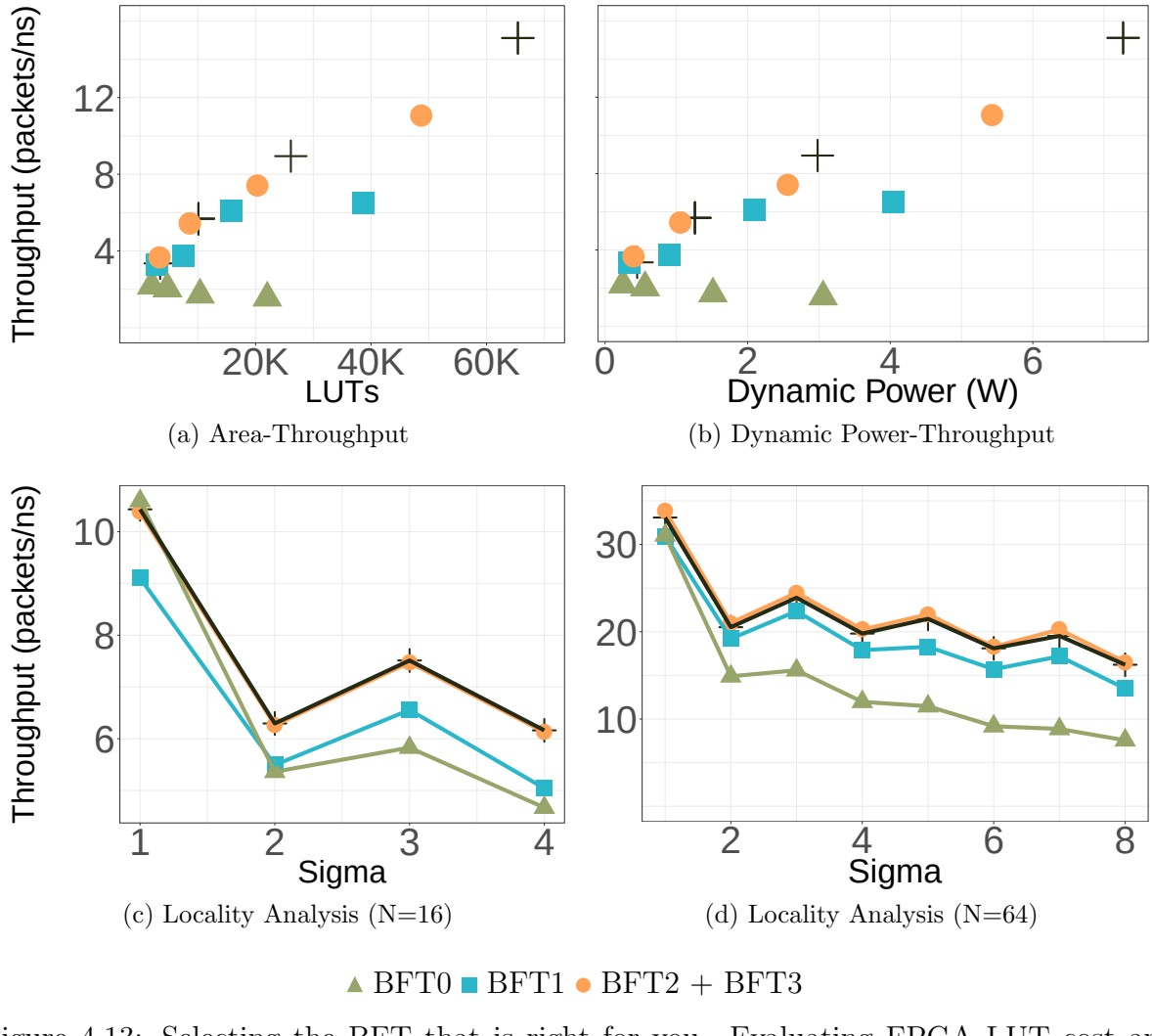
(a) Sustained Throughput (Graph)

(b) Relative Latency (Graph)

Figure 4.12: Throughput-Latency trends for `REAL` Graph traffic.

4.4.6 Choosing a BFT that is best for you

Constrained by cost: Given a particular resource and power budget , we hope to show how to select a particular BFT topology and its associated bisection bandwidth richness. We consider four configurations with richness varying between a binary tree to a multi-stage crossbar. We evaluate the NoCs under `RANDOM` traffic and 100% injection rate.



(a) Area-Throughput

(b) Dynamic Power-Throughput

(c) Locality Analysis (N=16)

(d) Locality Analysis (N=64)

Figure 4.13: Selecting the BFT that is right for you. Evaluating FPGA LUT cost and Dynamic Power tradeoffs for different system sizes, RANDOM traffic pattern.

In both Figure 4.13a (area) and Figure 4.13b (power), we see three regions of operation:

- In the first region below 10K LUTs (1 Watt), there is little to distinguish between the variants in terms of throughput, so the BFT0 topology (binary tree) is adequate.
- In the second region, between 10–20K LUTs (1 to 2.5 Watts), the BFT topology that matches the bisection bandwidth of a mesh (BFT2) offers the best result.

- In the extreme case where resource costs are not a factor >20K LUTs (2.5 Watts), the BFT3 (multi-stage crossbar) is best.

Constrained by application locality: We provide further insight by analyzing the effect of locality on topology selection. For this experiment, we vary the locality parameter SIGMA of our synthetic workloads from 1 (nearest neighbor) to $\sqrt{N}$ (uniform RANDOM), for a topology with system size N. The locality parameter restricts the destination address of a packet to lie within the SIGMA×SIGMA rectangle of the source address. We plot observations for N=16 and N=64 in Figure 4.13c and Figure 4.13d respectively. For nearest neighbor style communication, all topologies perform equally well due to little contention among packets. As packets travel larger distances, the degree of contention in the NoC increases and richer topologies like BFT2 and BFT3 perform better. An important observation here is that BFT3 is matched in performance by BFT2, suggesting that the BFT3 (multi-stage) topology is excessive and not needed for real-world locality.

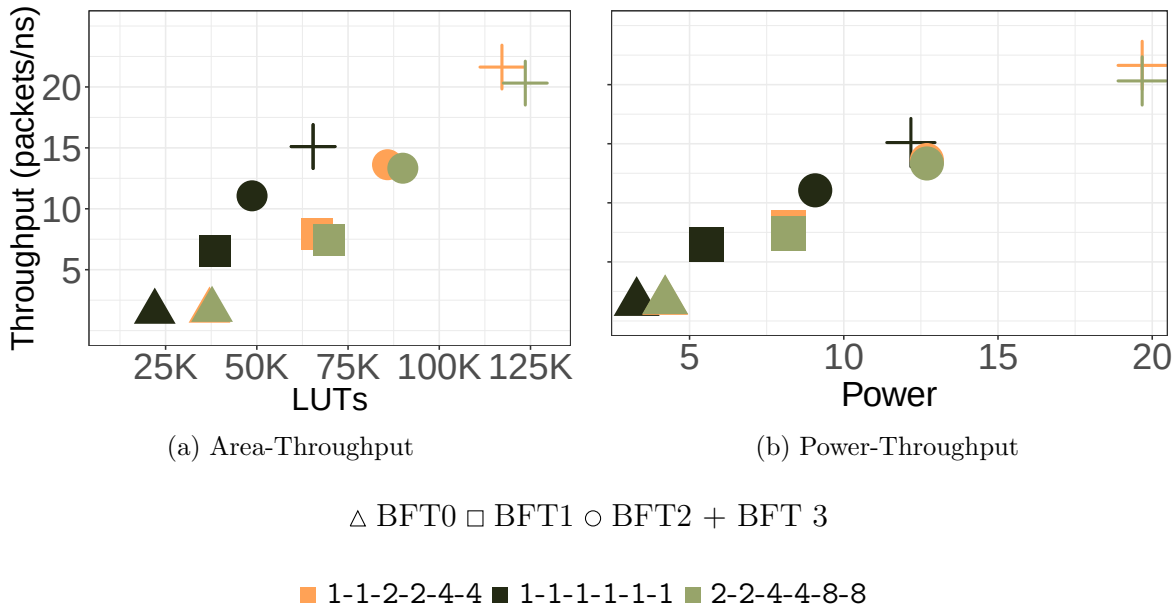
4.4.7 Analyzing BFTs with SRL FIFOs

Conventional BFT NoCs [89, 61] use buffers along the links of the tree to hold contending packets. We wish to evaluate the effectiveness of this approach. We have previously quantified the effect of adding 32-deep SRL FIFOs in Table 4.1 and concluded that they increase cost by 1.5×.

The number of pipeline stages are doubled every two levels to account for longer wires at the top assuming an H-tree layout. We analyze two pipeline configurations with one and two pipeline registers at the leaf nodes used for seeding the doubling every two levels.

Figure 4.14a shows sustained throughput (packets/s) as a function of LUT utilization for two pipeline configurations: 1-1-2-2-4-4 and 2-2-4-4-8-8 for a NoC of system size 64. For 1-1-2-2-4-4, this means a single register at levels 1 and 2 of the BFT, 2 registers at levels 3 and 4 of the BFT, 4 registers at levels 5 and 6 of the BFT. For 2-2-4-4-8-8, the registers are twice as many as 1-1-2-2-4-4 at every level. Refer to Figure 2.2 to refresh on the levels in a BFT. We compare these configuration against our flow-controlled BFT by injecting RANDOM traffic at 100% rate. We observe that the buffered BFTs offer higher throughput but at a disproportionate increase in cost. For instance, we can slightly exceed the throughput of a Buffered BFT2 design with a richer flow-controlled BFT3 design while needing fewer LUT resources. This allows us to tradeoff wiring bandwidth for improved performance. FPGA developers who wish to further boost performance of the BFT3 design can certainly add buffering to do so. Another observation here is that over-pipelining the

links hurts rather than helping the NoC, with the 1-1-2-2-4-4 design consuming fewer LUTs and sustaining higher throughput.



(a) Area-Throughput

(b) Power-Throughput

Figure 4.14: FPGA LUT cost and Dynamic Power tradeoffs for different NoC configurations. 64 clients, RANDOM traffic at 100% injection rate

Figure 4.14b captures the power-throughput tradeoffs for the same NoC configurations. Again, we observe that the buffered NoCs consume more dynamic power due to the increased activity of the FIFO resources and associated link pipelines. The buffered BFT2 is again delivering lower throughput while consuming more power than the flow-controlled BFT3 design.

Thus, wire-rich flow-controlled BFTs offer a better alternative to buffered BFT NoCs when considering resource costs as well as power consumption. It remains to be seen whether these conclusions will apply to the Intel Stratix-10 Hyperflex register fabric. It is a different cost model than the LUT-oriented pipelining implementations explored here, and will be studied as part of future work.

4.5 Conclusions

We show how to construct FPGA-friendly BFT NoCs with lightweight flow control to deliver improved throughput, and latency outcomes while adding in-order delivery and bounded delivery on routing time. In contrast to contemporary FPGA NoCs like Hoplite and CONNECT which either sacrifice NoC features or NoC implementation cost, the BFT is able to deliver both and offers bandwidth configurability to the FPGA developer. We augment the BFT topology with a lightweight latency insensitive interface, deterministic routing policy, and round-robin arbitration to deliver the desired features. Across a range of synthetic and realistic workloads, our topology outperforms others by 1-6×.

Chapter 5

HopliteBuf+HopliteBP: Evolving Application Customizable FPGA NoCs

We can overcome the pessimism in worst-case routing latency analysis of timing-predictable Network-on-Chip (NoC) workloads by single digit factors through the use of a hybrid FPGA-optimized NoC. Timing predictable FPGA-optimized NoCs such as HopliteBuf integrate stall-free FIFOs that are sized using offline, static analysis of a user-supplied flow pattern and rates. For certain bursty traffic and flow configurations, the static analysis delivers very large, sometimes infeasible, FIFO size bounds and large worst-case latency bounds. Alternatively, backpressure-based NoCs such as HopliteBP can operate with lower latencies for certain bursty flows. However, they suffer from severe pessimism in the analysis due to the effect of pipelining of packets and interleaving of flows at switch ports. As we show in this work, a hybrid FPGA NoC that seamlessly composes both design styles on a per-switch basis, delivers the best of both worlds with improved feasibility (bounded operation), and tighter latency bounds. We select the NoC switch configuration through a novel evolutionary algorithm based on Maximum Likelihood Estimation (MLE). For synthetic (`RANDOM`, `LOCAL`) and real world (`SpMV`, `Graph`) workloads, we demonstrate ≈2–3× improvements in feasibility, ≈1–6.8× in worst-case latency while only requiring LUT cost ≈1–1.5× larger than the cheapest HopliteBuf solution. We also deploy and verify our NoC (PL) and MLE framework (PS) on a Pynq-Z1 to adapt and reconfigure NoC switches dynamically.

5.1 Introduction

The Hoplite [35] FPGA NoC, discussed in detail in section 2.5, is a LUT-optimized network-on-chip architecture targeting fracturable Xilinx FPGAs for high-speed, low-cost operation. Several variants of Hoplite targeting efficient implementations with different cost-feature tradeoffs such as HopliteRT [39], and HopliteBuf [60] have been published. These designs eliminate the livelock limitations of the original Hoplite design and provide provable upper bounds on packet latency. This is crucial for safety-critical real-time systems, where timing properties of the underlying hardware are used to ensure that applications meet their scheduling deadlines, and performance isolation between different communicating components is required [94]. Of particular interest is the HopliteBuf variant that adds stall-free SRL32 FIFOs to the switch and uses static analysis tools to prove upper bounds on FIFO sizes and worst-case routing latency (`wclatency`). However, under certain scenarios, the static analysis exaggerates FIFO and latency bounds making them impractical for real designs. Under these circumstances, a different variant of the NoC with lightweight backpressure, HopliteBP [41] may be preferred. If the entire NoC adopts a backpressure-based routing style, the blocking effects of backpressure due to pipelining and interleaving of flows will severely limit provable NoC performance [41]. Furthermore, the FPGA implementation requirements of flow control, however lightweight they may be, were a primary motivation for the deflection-oriented design of Hoplite. Instead of a homogeneous NoC design, a carefully selected hybrid NoC architecture that combines both HopliteBuf and HopliteBP styles in a fine-grained fashion yields a superior solution than either alternatives alone.

An application flowset is composed of a multitude of unique traffic traces (or flows), each carrying packets between a source and destination PE. Each flow is characterised by its regulated injection rate (the rate at which packets can be injected into the NoC) and data block size (number of consecutive packets injected). We explain the composition of a flow in greater detail in section 5.4.1. We show a set of three network flows on a 4×4 NoC interacting in prescribed ways in Figure 5.1. HopliteBuf provides better worst-case latency for the horizontal traffic in Figure 5.1a, as HopliteBP suffers from backpressure propagated on the horizontal connections. However, for vertical traffic in Figure 5.1b, HopliteBP delivers 1.5× better worst-case latency. The cyclic loop of dependencies between flows shown in Figure 5.1c further affects HopliteBuf, reducing its range of statically analyzable rates, while HopliteBP is less affected. We analyze this example in further detail in Section 5.2. Based on these observations, it is clear that one-style-fits-all approach will not work and we need to configure each switch in the NoC carefully by learning the effect of interactions between the conflicting traffic flows of the application. As the interference pattern of the

network flows can be quite complex, and a brute-force approach not feasible for large NoC sizes, we develop a evolutionary strategy to discover high-quality solutions for a given QoR (quality of result) function. We formulate a Maximum Likelihood Estimation solution where we adjust the probabilities of each switch configuration based on iterative analysis trials.

The key contributions of this work include:

- Design of a hybrid FPGA NoC architecture that combines HopliteBuf and HopliteBP switches to exploit the best of both worlds.
- Development of an evolutionary strategy to learn switch configurations for a defined QoR function using Maximum Likelihood Estimation (MLE), scalable to large NoC sizes.
- Formalization of the Hoplite NoC analysis suite and explanation of its flow model for hybrid NoCs; used for estimating feasibility and upper bounds on routing latencies, including worst case and injection latency.
- Quantification of worst-case latency, feasibility, cost and optimization runtime across real and synthetic applications. We demonstrate ≈1–6.8× reduction in worst-case latency and ≈2-3× improved feasibility by learning switch level configuration.
- We implement and verify our entire switch learning framework on a Pynq-Z1. We implement the MLE optimization tool on the Cortex A9 processor (PS) to configure the FPGA NoC switches dynamically, while realizing the NoC itself on the FPGA.
- Quantification of performance and area overheads of an ASIC implementation of our Hybrid HopliteBuf+BP NoC. We demonstrate 3–3.7× smaller area overheads while reducing average latencies by 1.3–12×.
- Adapting the MLE framework to learn switch parameters for a NoC with a completely different topology; we learn uphill routing connections for the crossbar topology of a Butterfly Fat Tree (BFT) NoC from chapter 4 on a *per-switch basis*. This allows us to further customize the NoC's bandwidth for the application at hand. We observe ≈1–1.7× reduction in latencies with the MLE optimised BFT NoC while preserving the desired properties of the original NoC (in-order packet deliveries, livelock freedom and bounded latencies).

5.2 Motivating Example

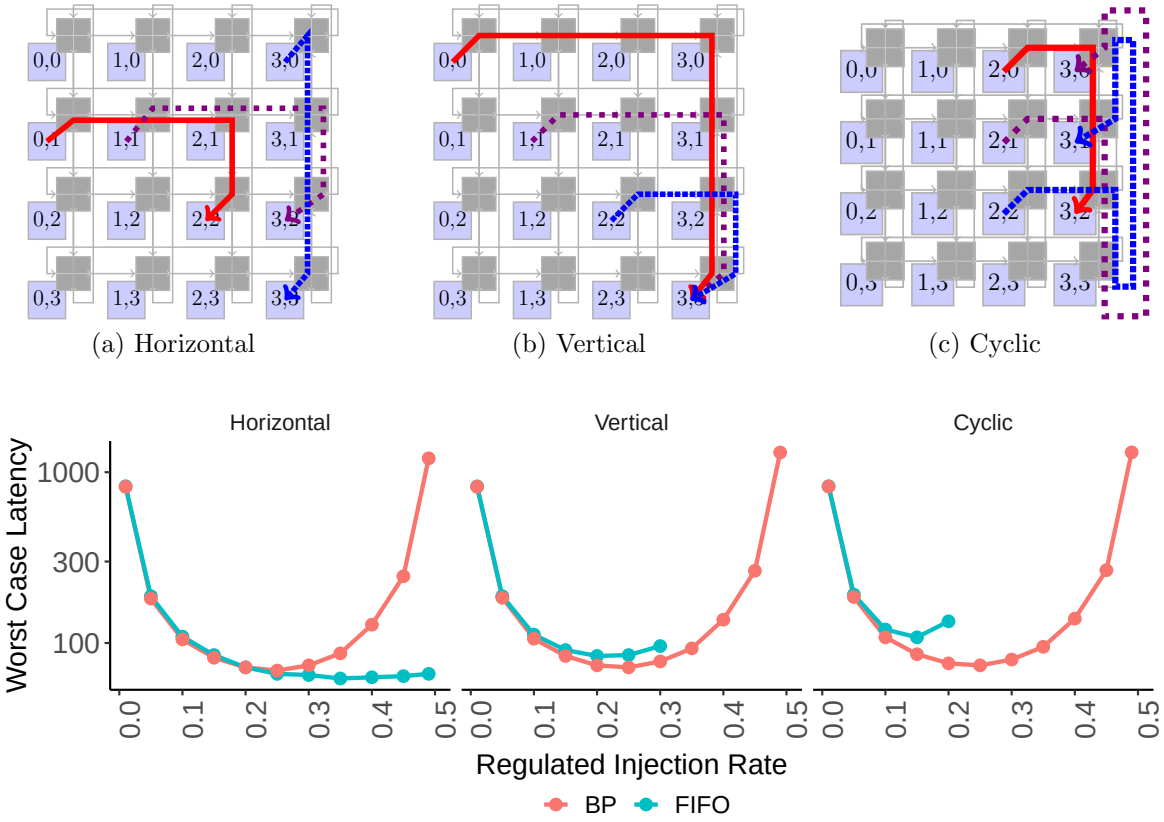
In this section, we discuss the motivation for learning NoC parameters for a given application flowset. Specifically, in order to maximise a user requested QoR (quality of result) for a specific application flowset, we build a case for building a hybrid NoC design by learning mode of operation (between HopliteBuf and HopliteBP), for every switch in the NoC. We refer the reader to section 2.5 for more details on HopliteBuf and HopliteBP NoCs.

To discuss how switch configuration affects the results of the analysis for different flow patterns, we perform a set of analyses for the three flow sets: (a) Horizontal, (b) Vertical and (c) Cyclic shown in Figure 5.1. We assume that all flows have the same block size $f_i.b = 8$ and we use the same value of regulated injection rate $f_i.\rho$ (x axis of Figure 5.1d) across all flows. In Figure 5.1d, we then plot the maximum analytical latency of the NoC while varying $f_i.\rho$. We consider two network configurations, one where all switches use stall-free FIFOs (HopliteBuf), and one where all switches use backpressure (HopliteBP). We stop plotting the latency whenever the network declares infeasibility due to unbounded latencies or the size of any buffer under HopliteBuf exceeds 32 (as this enables one-LUT per bit FIFO implementation).

- **Effect of regulation rate**: As we increase the regulated injection rate (equivalent to decreasing the amount of regulation), latency initially decreases for all cases in Figure 5.1d. This is because increasing the regulation rate ($f_i.\rho$) affords each PE increased opportunity to inject its packets into the network. However, past a certain regulated injection rate rate, the latency starts increasing due to increased contention caused by conflicting flows, which dominates any benefits of a reduced regulation rate.

- **Horizontal**: For the set of flows shown in Figure 5.1a, we note that HopliteBuf (fifo) delivers lower worst-case latency. This is because under HopliteBP, f_1 (▬) suffers interference not only from f_2(▪▪) but also f_3 (▪▪▪) indirectly due to horizontal backpressure stalls originating from switch $(3, 1)$; thus increasing latency compared to HopliteBuf.

- **Vertical**: Here, HopliteBP achieves lower latency. This is due to the effects of buffering: since f_2 (▪▪) suffers interference from f_1 (▬), HopliteBuf will accumulate its packets at switch $(3, 1)$. Hence, the maximum number of consecutive packets that f_2 (▪▪) can inject south at $(3, 1)$ becomes larger than its data block size $f_2.b$. In turn, this means that the queueing delay of f_3 (▪▪▪) at switch $(3, 2)$'s FIFO can exceed injection delay at the source of f_3 (▪▪▪) due to backpressure induced stalls in HopliteBP.



(a) Horizontal (b) Vertical (c) Cyclic

(d) Worst-Case Latency Scaling Trends

Figure 5.1: Interacting network flows on a 4×4 NoC arranged to show the benefits of mixing FIFO and Backpressure switches.

- **Cyclic**: The negative effects of vertical buffering are magnified in a cyclic flow pattern and the HopliteBuf designs show significantly worse latency than HopliteBP, as shown in Figure 5.1d.

A user-supplied set of network flows can contain a combination of these three patterns, among others. The key idea explored in this work is to learn the effect of these flowset interactions on routability and hardware costs to make a determination between HopliteBuf and HopliteBP on a *per-switch* basis. The use of such hybrid NoC switch configurations allows to optimize both analytical flow latency as well as hardware cost. This in-turn allows us to tailor our NoC offerings to the application at-hand and QoR requested, by learning the effects of the provided flows on a *per-switch* basis.

5.3 Design of the HopliteBuf+BP Switch

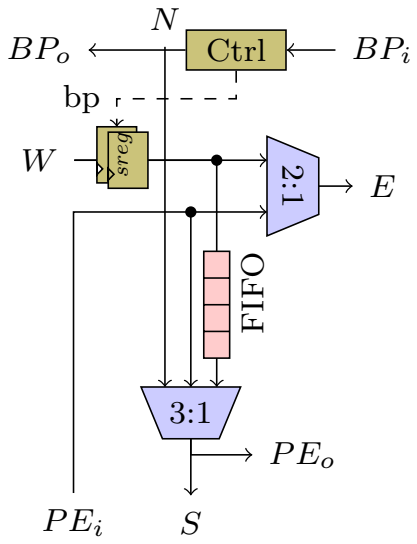


Figure 5.2: Block diagrams of HopliteBuf+BP design.

HopliteBuf+BP (FIFO + Backpressure in Figure 5.2) is a unified switch that includes both switch components and supports runtime configuration of the operating mode. The HopliteBuf design presented in [60] cannot coexist with HopliteBP switches in the same horizontal ring, as it will not respond to a backpressure signal (BP_0), and thus, despite assertion of this signal, may end up sending a packet to the W input of a backpressured HopliteBP switch. In order to have switches with a $W \rightarrow S$ FIFO in the same horizontal ring as HopliteBP switches, we define this third switch, that extends HopliteBuf by incorporating backpressure and shadow registers in the design. When implementing a soft NoC, a HopliteBuf+BP switch is used in place of HopliteBuf whenever the analysis determines that the switch could be reached by a backpressure signal generated by a HopliteBP switch on the same horizontal ring. We can also realize a hard, configurable NoC architecture based on the same principle. Specifically, the HopliteBuf+BP structure can be used to support both stall-free FIFO and backpressure designs, by simply implementing a programmable register that can either be configured during compilation or in response to evolving traffic conditions when routing a workload (without requiring full FPGA bitstream reconfiguration). As summarised in Table 5.1, the unified switch is ≈1.5–1.6× more expensive than vanilla HopliteBuf/BP switches.

	Hoplite	HopliteBuf	HopliteBP	HopliteBuf+BP
LUTs	59	161	189	247
FFs	86	91	167	175

Table 5.1: Resource utilization of Hoplite based switches on Xilinx Virtex-7 for 32b datawidth and 32-deep SRL32 FIFOs.

All switches are complemented with a static analysis and traffic regulation component [41]. We leverage this analysis to prove feasibility (bounded latencies and FIFO sizes), compute worst-case FIFO size (for HopliteBuf), as well as worst-case bounds on latency (injection+routing) of packets. The analysis computes *composable* latency bounds, that is, the bounds do not depend on detailed information about activity of unrelated PEs. To this end, it limits the maximum number of packets that a PE can inject in the network in any interval of time using a network *regulator*. In this way, to compute the latency for flow f_i, it only needs to know the regulation parameters and source/destination PEs for every other flow in its set of interfering flows. The analysis employs a 32 bit leaky bucket regulator [95], which uses two parameters, regulation rate $\mathbf{f_i}.\boldsymbol{\rho}$ and data burst size $\mathbf{f_i}.\mathbf{b}$: data burst is the maximum number of consecutive packets that the flow can send through the regulator, while the rate is the maximum long-term throughput of the regulator in packets per cycle. We review the static analysis used in these designs in greater detail in Section 5.4.

5.4 Latency and Backlog Analysis

We now discuss how to derive latency bounds, as well as backlog (the maximum FIFO occupancy) bounds, for a NoC design incorporating a mix of HopliteBuf and HopliteBP switches. This section is attributed to work done originally in [41] (See Statement of Contributions). The analysis derives four sets of parameters:

- The worst-case injection latency for any flow f_i, that is, the maximum time that node $(f_i.xs, f_i.ys)$ can be stalled while trying to inject one or more packets of f_i.
- The worst-case queuing delay for any flow f_i turning $W \rightarrow S$ on a FIFO enabled (HopliteBuf/Unified) switch, that is, the maximum time that a packet of f_i can be queued in the FIFO.

- The per-hop delay for any flow f_i, from (xs, ys) to (xd, yd). The total latency for f_i is then the sum of its injection latency, queuing delay (if applicable), and per-hop delay.
- Finally, to prove that stall-free FIFOs can be used without dropping packets, we determine the worst-case backlog (buffer occupation) for any FIFO enabled switch.

We begin by introducing the flow and traffic regulation model. We then summarize the existing analysis for HopliteBuf [60] and HopliteBP [41], and finally show how to extend it to a hybrid architecture.

5.4.1 Flow Model

We assume that each NoC node sends packets as part of one or more flows; a flow f_i represents a sequence of data blocks, each comprising $f_i.k$ consecutive packet flits, sent between the same source node $(f_i.xs, f_i.ys)$ and destination node $(f_i.xd, f_i.yd)$. The data block size $f_i.k$ is related to the architecture of the PEs and the width of the NoC; as an example, if blocks represent 32 bytes cache lines and the NoC width is 8 bytes, we obtain $f_i.k = 4$. We make no assumption on the exact time at which successive data blocks are produced by the source node, as such information might be too difficult to analyze [1]. Instead, our goal is to determine a worst-case analytical bound to the latency suffered by any data block of flow f_i. The bound includes both the injection latency $Injection(f_i)$, that is, the maximum time that node $(f_i.xs, f_i.ys)$ takes to inject all packets of the block into the network, as well as the queuing delay, that is, the maximum time that any packet of f_i can be queued in a $W \rightarrow S$ FIFO. As part of the queuing analysis, we also derive a backlog bound for each FIFO. We say that a network is stable if all flows have bounded total latency and the backlog for all FIFOs is smaller or equal to the FIFO size.

5.4.2 Traffic regulation

We are interested in computing a *composable* latency bound, that is, the bound for a flow should not depend on detailed information on the activity of unrelated PEs. To this end, we limit the maximum number of packets that a flow can inject in the network in any interval of time using a network *regulator*. In this way, to compute the latency for f_i we

[1] For example, in the case of requests produced by a cache, the specific timing of fetches/write-backs depends on both the initial cache state, and the path through the program.

only need to know the regulation parameters and source/destination for every other flow f_j; as long as those remain constant, any change to the way f_j generates data blocks has no effect on f_i. As in [60, 41], we employ a token bucket regulator [95], which uses two parameters, regulation rate $f_i.\rho$ and burst size $f_i.b$: burst size is the maximum number of consecutive packets that the flow can send through the regulator, while the regulation rate is the maximum long-term throughput of the regulator in packets per cycle. The implementation of the regulator uses two counters. The rate counter is incremented every cycle and adds a token to the token counter every $1/f_i.\rho$ cycles (the regulation period). The token counter has a maximum size of $f_i.b$. At any clock cycle, the node is allowed to inject a packet only if the NoC is ready (meaning that no other packet is being sent to the same output port, given that the PE port has the lowest priority) and the token counter is not empty; sending a packet removes a token from the counter. For this work and Chapter 6 ([16]), we set the injection rate of a flow f_i to be equal to its regulation rate (also known as regulated injection rate). This regulation rate (equivalent to "injection rate" or "regulated injection rate") can be set by the system designer or learnt as part of maximising an application's QoR (chapter 6). There is no buffer associated with the regulator; hence, a node that wishes to inject a packet but is blocked by the regulator is simply stalled. We make no assumption on the arbitration policy among flows sourced from the same node: if during a clock cycle a node is ready to inject a packet from both a flow f_i and a flow f_j, then it can select either of the two flows.

5.4.3 Injection Latency

To determine the injection latency for flow f_i, we need to compute the set of conflicting flows Γ_i^C that can prevent a packet of f_i from being injected in the NoC. This set comprises all flows sourced from the same node $(f_i.xs, f_i.ys)$ as f_i, plus all higher priority flows that traverse the switch at $(f_i.xs, f_i.ys)$ in the same direction as f_i: the $W \rightarrow E$ flows if f_i is injected to the E port, or the $N \rightarrow S$ and $W \rightarrow S$ flows if f_i is injected to the S port. After obtaining the conflict set Γ_i^C, the injection latency can be computed as follows, where $b(\Gamma_i^C)$ is the sum of data block sizes of all flows in Γ_i^C and $\rho(\Gamma_i^C)$ is the sum of their regulation rates:

Theorem 1 (Theorem 1 in [41] and 2 in [40]) *Assume $\rho(\Gamma_i^C) < 1$ and the client wishes to inject a sequence (data block) of $f_i.k \leq f_i.b$ packets for flow f_i. Then the delay to inject all packets in the block is upper bounded by:*

$$Injection(f_i) = \lceil 1/f_i.\rho \rceil - 1 + \left\lceil \frac{b(\Gamma_i^C)}{1-\rho(\Gamma_i^C)} \right\rceil + \left\lceil (f_i.k-1) \cdot \max\left(\frac{1}{f_i.\rho}, \frac{1}{1-\rho(\Gamma_i^C)}\right) \right\rceil. \tag{5.1}$$

Note that $1-\rho(\Gamma_i^C)$ is the available regulation rate for f_i after removing the rate of flows in Γ_i^C. In essence, the first packet in the data block is delayed first by the regulation period (inverse of the rate) $1/f_i.\rho$; then, by the data block size of interfering flows, which is captured by term $b(\Gamma_i^C)/\big(1-\rho(\Gamma_i^C)\big)$; and finally, the remaining $f_i.k-1$ packets are injected at the minimum rate (maximum of the periods) between $f_i.\rho$ and the available regulation rate $1-\rho(\Gamma_i^C)$. We make three important observations:

- If $\rho(\Gamma_i^C) \geq 1$, then in the worst case f_i can never inject, hence its latency is unbounded and the network is unstable.
- The bound only holds for data blocks of size at most $f_i.b$: larger blocks can be delayed by $b(\Gamma_i^C)$ multiple times, leading to significantly worse latency. Furthermore, increasing the burst size of the regulator beyond the flow's block size does not provide any advantage to f_i. While it could increase $b(\Gamma_j^C)$ for some other flow f_j, in the rest of the work, we simply set $f_i.b = f_i.k$ for all flows.
- Increasing the regulation rate $f_i.\rho$ improves the latency of f_i, because it reduces the regulation period $1/f_i.\rho$; it also increases both the guaranteed $(1/Injection(f_i))$ as well as average-case regulation rate for f_i. However, it can also decrease the available rate $1-\rho(\Gamma_j^C)$ for some other flow f_j, leading to a higher latency for f_j.

Therefore, the regulation latency of a flow represents a measure of how latency-critical the flow is: the higher the rate, the better the latency for that flow, at the expense of other flows. This competitive relationship between traces becomes important when learning unique rates to address inequity in per-trace regulation.

5.4.4 Analysis for FIFO Mode

For a flow f_i turning $W \rightarrow S$ on a FIFO enabled switch (vanilla HopliteBuf or unified HopliteBuf+BP), let f_i' denote the flow after it leaves the S port: the effect of queuing packets in the buffer can cause the data block size $f_i'.b$ to be higher than $f_i.b$. In turn,

this will affect the data block size and latency of flows either injected into S or turning $W \rightarrow S$ on downstream switches in the same vertical ring. Hence, this effect must be bounded to obtain safe worst-case estimates. To this end, the analysis in [60] relies on the theory of network calculus [95], which is commonly employed to derive deterministic bounds on network latency and backlog. The analysis derives a set of linear equations that relate the data block size $f_i'.b$ of f_i' to the data block size of other flows leaving the S port on other FIFO switches on the same vertical torus. We then solve the system of equations; if resulting data block size values for all flows are finite and positive, then they must indeed represent valid upper bounds. Otherwise, we deem the network unstable. Assuming stability, the queuing delay for f_i and backlog for the switch can then be derived based on the regulation rate and data block size of all flows that traverse the switch in the $N \rightarrow S$ and $W \rightarrow S$ direction.

It is important to note that due to the torus topology, the system of equations can contain cyclic dependencies between data block size variables. An example is provided in Figure 5.1c: here, each flow turning $W \rightarrow S$ is delayed by the other two flows that traverse the switch $N \rightarrow S$. Hence, increasing the data block size of a flow causes an increase in the data block size of the other two flows, which in turn affect the first one. This feedback loop introduces pessimism in the analysis, causing a reduction in the sustainable per-link utilization: for this example, the network becomes unstable at a per-link utilization of 75% (a regulation rate of 1/4 for each flow).

5.4.5 Analysis for Backpressure Mode

Since they do not contain a FIFO, no buffer analysis is required for backpressure enabled switches (vanilla HopliteBP or Unified). Instead, the analysis in [41] accounts for the effects of backpressure by adding all backpressured flows to the set of conflicting flows Γ_i^C. Specifically, if f_i turns $W \rightarrow S$ at a switch, we add to Γ_i^C all other flows that traverse that switch in the $N \rightarrow S$ or $W \rightarrow S$ directions, as they can backpressure f_i. Furthermore, if another flow f_j shares a W port on any switch with f_i, then all flows that backpressure f_i can also indirectly cause backpressure on f_j; therefore, they must be added to Γ_j^C.

5.4.6 Hybrid NoC Analysis

To analyze our hybrid NoC, composed of individual switches operating in either FIFO or backpressure mode, we combine the analysis for FIFO and backpressure modes; we first run the buffer analysis for each vertical ring that contains at least one FIFO enabled

switch. Note that for any flow f_i that turns $W \rightarrow S$ on that vertical ring at a backpressure switch, we consider the original data block size $f_i.b$ rather than the modified data block size $f_i'.b$, as the flow does not traverse a buffer. We then run the backpressure analysis for each horizontal ring that contains at least one backpressure switch. Note that a flow f_i can suffer backpressure from $N \rightarrow S$ and $W \rightarrow S$ flows only if it turns $W \rightarrow S$ at a backpressure enabled switch; however, a flow f_j can suffer indirect backpressure from f_i if it shared a W port on any switch (HopliteBP/Unified) or the horizontal ring. Finally, it is important to note that based on the analysis, adding a $W \rightarrow S$ FIFO buffer to a switch is a trade-off: on the negative side, the buffer increases the data block size of turning flows, and thus the interference on the vertical ring; on the positive, if we can prove that the FIFO is stall-free, we avoid generating backpressure on the horizontal ring. However, if the FIFO can be stalled, then it is not advantageous. For this reason, the HopliteBP design does not include a FIFO.

5.5 Evolutionary Learning of Switch Configuration

We now present the key ideas explored in this work. We learn (for a given application flowset, data block size and defined QoR), the configuration mode (between HopliteBP and HopliteBuf) for every individual switch in the NoC. This allows us to tailor the NoC towards the application at hand by optimizing for a user defined QoR. We cast the learning of switch level configurations as a Maximum Likelihood Estimation problem.

The key idea explored in this work is the use of hybrid NoC switch configurations that mix HopliteBuf and HopliteBP styles in a single NoC. Depending on the interference pattern of network flows, each switch configuration may be tailored in an application-specific manner. In Figure 5.3, we show the best switch configuration for a simple 3×3 NoC with a fixed set of nine flows (shown in red in the leftmost NoC subfigure). Recall, as discussed in section 5.3, for a horizontal row to support a mix of HopliteBP and HopliteBuf modes of operation, each switch in that horizontal row must be upgraded to the unified HopliteBuf+BP switch. Subsequently, this row of HopliteBuf+BP switches can be independently configured to operate either as HopliteBP or HopliteBuf. We vary the block size b and regulation rate ρ for each flow (identically). At very low rates and data block sizes, HopliteBuf offers the cheapest and best-performing design, while at larger rates and blocks, the design starts to migrate to HopliteBP-dominated solutions. Given that an $N \times N$ NoC will have N^2 switches, each with a boolean decision to make, the design space grows exponentially with problem size.

Now, we discuss the optimization technique we use to learn the switch configurations

for a particular set of flows. Specifically, given the regulation rate (provided by user or a learning algorithm[2]) and data block size of each flow, we determine whether each switch in the network implements a stall-free FIFO or backpressure logic to optimise for the user defined QoR (`wclatency*cost`, `wclatency`, etc).

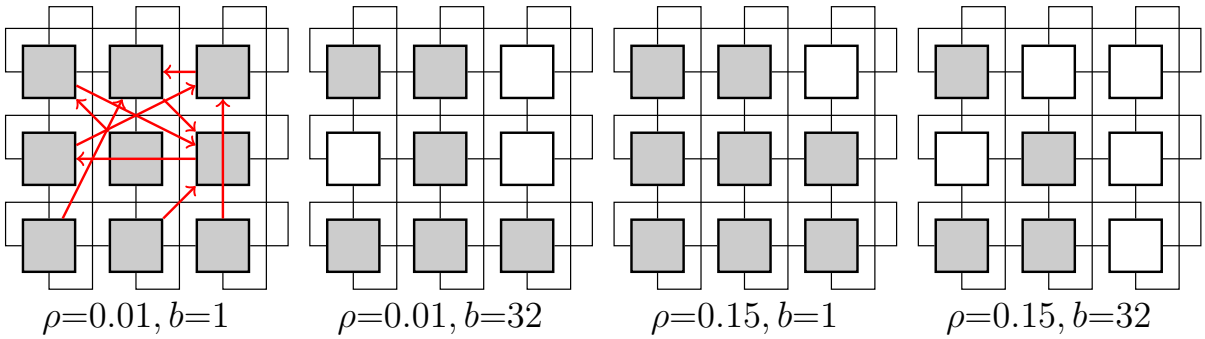


Figure 5.3: FPGA NoC Switch Configuration for a 3×3 NoC for a set of nine flows (shown in leftmost subfigure) with identical but four combinations of rate ρ and block size b characteristics. At low rates and block sizes, most switches tend to be cheap HopliteBuf ◘ mode of operation, while a growing number convert to expensive HopliteBP □ mode of operation with increasing rate+size.

Note that an $N \times N$ NoC has N^2 switches, each with a boolean decision to make. This $O(2^{N^2})$ solution space necessitates the use of a scalable approach rather than brute-force exploration (a 7×7 NoC can have ≈562 trillion solutions). We choose to model this as a Maximum Likelihood Estimation algorithm. We model each switch of the NoC as an independent random variable from a *2-point Bernoulli distribution* B_i. This means that there are two outcomes possible for each switch: 0 represents a HopliteBuf and a 1 represents HopliteBP. For each switch i, we represent the probability of choosing a Backpressure configuration as p_i while the FIFO choice becomes $1 - p_i$. We seed each switch with zero bias by ensuring that the skew of each switch's distribution B_i, $\frac{1-2p_i}{\sqrt{(p_i(1-p_i)}}$, is 0 by starting with $p_i = 0.5 \; \forall$ i.

We aim to *evolve* the optimal NoC configuration by producing multiple candidates solutions for the g^{th} generational step for each switch s_i^g of the $N \times N$ NoC. Unlike conventional neural networks with a known training set, we generate our training set on-the-fly based on the results of the generational search. Unlike Naïve Bayesian inference strategies explored

[2]rate learning is explored in chapter 6

by InTime [96], our approach directly aims to minimize an objective function rather than train with a binary classifier. For our setup, we evolve by sampling each switch's Bernoulli distribution $B_i^g(p_i^g)$ to produce either a FIFO (0) or a Backpressure configured (1) switch. For each generation step g, we produce C potential candidate NoC configurations $H_C^g \in (0,1)^{N^2 \times C}$, as shown below in Equation 5.2. For our experiments we chose C=100.

$$H_C^g = \underbrace{\begin{bmatrix} s_{1,1}^g & s_{2,1}^g & \cdots & s_{C,1}^g \\ \vdots & \vdots & \ddots & \vdots \\ s_{1,N^2}^g & s_{2,N^2}^g & \cdots & s_{C,N^2}^g \end{bmatrix}}_{\text{Each } \textbf{column} \text{ is a flattened NoC configuration } (h_n)} \tag{5.2}$$

We test each candidate configuration h_n for fitness on a user defined function and filter out the top λ=25% performing candidates $H_{\lambda:C}^g \in (0,1)^{N^2 \times \lambda}$. This is a greedy step but the memory of previous iterations is reflected in the existing probabilities of the switch configurations. We then adapt each switch's Bernoulli distribution in the general direction of the chosen candidates. We aim to increase the likelihood that the top performing candidates $H_{\lambda:C}^g$ of this generation were sampled from it. For each switch s_i^g at generation g, we define a likelihood function as follows:

$$\begin{aligned} f\left(s_{1:C,i}^g, \ldots s_{\lambda:C,i}^g | p_i^{g+1}\right) &= P\left(s_{1:C,i}^g, s_{2:C,i}^g \ldots, s_{\lambda:C,i}^g | p_i^{g+1}\right) \\ &= \underbrace{P(x_1, x_2 \ldots, x_\lambda | \hat{p})}_{\text{substitute}} \\ &= \hat{p}^{x_1} \cdot (1-\hat{p})^{1-x_1} \ldots \hat{p}^{x_\lambda} \cdot (1-\hat{p})^{1-x_\lambda} \\ &= \hat{p}^{\sum^{\lambda} x} \cdot (1-\hat{p})^{\lambda - \sum^{\lambda} x} \end{aligned} \tag{5.3}$$

Equation 5.3 is the *likelihood* of a Bernoulli distribution for switch i generating the best performing λ samples from a distribution with probability parameter p_i^{g+1}. Remember, $s_{\lambda:C,i}^g$ refers to the top λ (out of total C) performing switch configurations for switch i at generation g. The final expression of Equation 5.3 can be differentiated to find the value of $\hat{p} = p_i^{g+1}$ that maximizes this likelihood. We first apply a logarithmic transformation on this equation before differentiation, as shown below:

$$\ln(f) = \ln(\hat{p}) \cdot \sum^{\lambda} x + \ln(1-\hat{p}) \cdot (\lambda - \sum^{\lambda} x)$$

$$\frac{d\big(\ln(f)\big)}{d\hat{p}} = \frac{\sum^{\lambda} x}{\hat{p}} - \frac{\lambda - \sum^{\lambda} x}{1-\hat{p}} = 0$$

$$\sum^{\lambda} x - \hat{p} \cdot \sum^{\lambda} x = \lambda \cdot \hat{p} - \hat{p} \cdot \sum^{\lambda} x$$

$$\hat{p} = \frac{\sum^{\lambda} x}{\lambda} \tag{5.4}$$

Hence, the updated probability parameter $\hat{p} = p_i^{g+1}$ for each switch s_i's Bernoulli distribution B_i can simply be written as an average over the λ best performing candidates in generation g. We formalize this in Equation 5.5 below:

$$\hat{p} = p_i^{g+1} = \frac{\sum_{k=1}^{\lambda} s_{k:C,i}^{g}}{\lambda}. \tag{5.5}$$

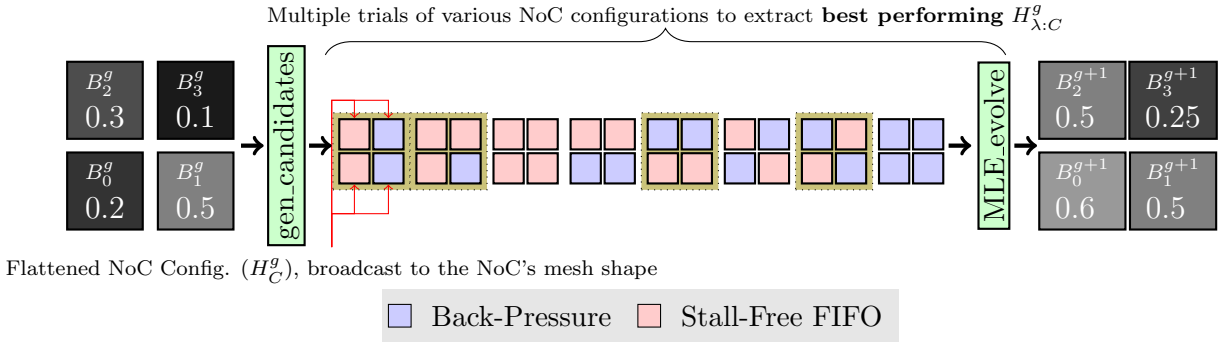


Figure 5.4: Evolution of switch type probabilities for a 2×2 NoC using Maximizing Likelihood Estimation (MLE). One MLE iteration is shown which has a candidate size C=8 (each of the 8 shown 2x2 Hoplite NoC configurations is a candidate) and elite size λ=4. Only ■ marked NoC configurations are used to generate the new Bernoulli distribution B^{g+1}.

Finally, in Figure 5.4, we first show the high-level representation of a single iteration of the MLE algorithm. Given a set of probabilities associated with the distributions B^g,

we generate H_C^g samples of possible NoC configurations. We then select the top-K best-performing solutions $H_{\lambda:C}^g$ to revise the distributions as B^{g+1} for the next iteration. We also represent an actual trace of probability evolution across a complete MLE evolution for a 2×2 NoC in Figure 5.5. We note that the first 20 iterations show the results of a sampling when the $p_i \approx 0.5$ for the switches. With subsequent iterations learning from the combinations that worked well in the first phase to converge towards the best-performing switch configurations for each round. While MLE can also be used for Bayesian inference [97] with respect to how samples are generated and probabilities updated, for our scenario, we directly minimize an objective function rather than performing a classification.

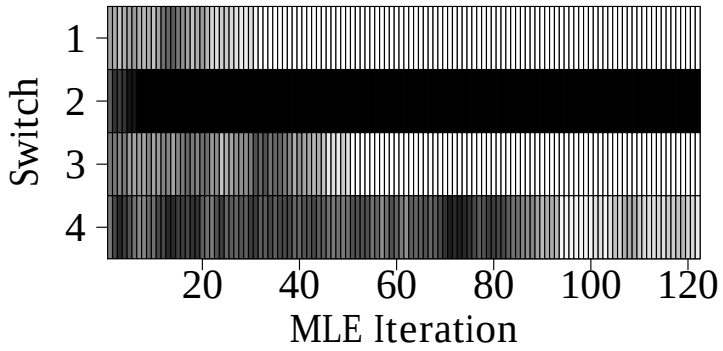


Figure 5.5: The complete MLE algorithm is shown for a regulation rate of 0.5 and a block size of 16. Switch colors that settle for white are Backpressure switches while those that are Black are FIFO-based.

5.6 Evaluation

5.6.1 Methodology

For switch learning, our MLE optimizer is written in Python3. We implement the latency and buffer analysis in Matlab, and convert it to C code using Real Time Workshop; the optimizers communicate with the analysis tool using direct data transfer based on the Python `ctypes` APIs. We run all experiments on a 16-core Intel Xeon E5-2697A CPU and parallelize our search across 32 threads. We measure our MLE switch learning implementation to be 50-500× faster when compared against open-source Python implementations of black-box optimizers `CMA-ES` [82] and `RBFOPT` [98], while exploring the solution space just as effectively.

We evaluate our entire framework across a range of 100 synthetically-generated communication workloads with `RANDOM` and `LOCAL` communication patterns. For `RANDOM` patterns, each PE chooses destinations via uniform sampling of other PEs. For `LOCAL` patterns, sampling is restricted to a +/- 2 radius of neighbouring PEs. We run our analysis across various data block sizes to mimic the diversity of communication interfaces and endpoints like DRAM, PCIe, and Ethernet. Furthermore, we also test for optimizing real-world applications designs for FPGA accelerators such as Sparse matrix vector multiplication (`SpMV` [47]) used by many deep learning kernels [99] and `Graph` [48] analytics. The extracted traces exhibit a rich diversity of co-relation and bandwidth requirements; with load factors of 37-92% between the benchmarks. Furthermore, the traces in these real benchmarks are a snapshot of real edge-point data, allowing us to evaluate our designs on ability to route data that closely mimics interactions of end-points in a production setting.

In addition to the static analysis above, we also implement and verify the entire MLE generated NoC on a Pynq-Z1 board, with unified HopliteBuf+BP switches in the FPGA fabric (PL), and configuration tools for the NoC switches and regulators running on the ARM A9 (PS). For the unified NoC, the software layer on the ARM configures the switches *dynamically* over an AXI bus, depending on the application requirements, without the need for complete bitstream reconfiguration. For a 6x6 network, the runtime of the MLE optimization on the PS is 9-60s, and the reconfiguration of the switches instantiated in the PL takes 850ms.

5.6.2 Results

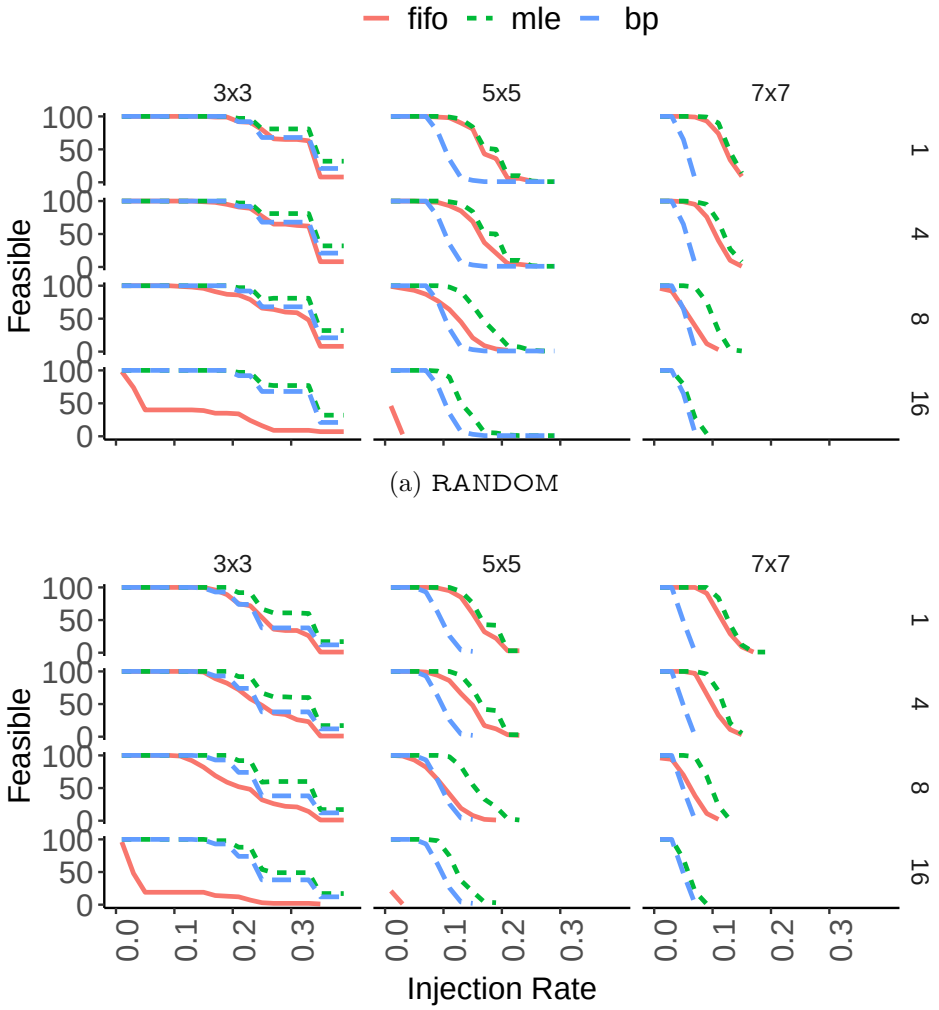
In this section, we present the results of the static analysis of our hybrid FPGA NoC under various system sizes, data block sizes, and optimization methods. We study the effects of learning switch level configurations with regulation rate as a user controlled parameter. We are primarily interested in determining network feasibility (bounded latency), FIFO size needed, worst-case latency, fitness of the solution, and time required to optimize the NoC.

For the following set of experiments, we focus on only learning switch level configuration to generate hybrid NoCs. We treat the regulated injection rate rate as a user set universal parameter and quantify the resulting NoCs on their ability to optimised for the requested QoR over a linear range (between 0 and 1) of these rates.

5.6.2.1 Learning for Feasibility

In Figure 5.6, we determine the subset of the 100 synthetic RANDOM and LOCAL flowsets that can be routed feasibly on the NoC, that is, with bounded latencies and fit within the 32-deep FIFO capacity limits of an SRL32 structure. We configure MLE to produce feasible NoC configurations for provided flowset. As we vary system size, regulation rate, and data block size, we note several interesting trends:

- First, we observe that feasibility rate drops from 100% to 0% as we increase regulation rate (x-axis), with steeper losses observed for larger system sizes. This is understood as there are more flows competing for bandwidth that does not scale linearly with system size. For a torus, doubling of system size increases bisection bandwidth by only $\sqrt{2}$.
- As we increase data block lengths from 1–16 (figures along a column), we note a counter-intuitive effect. Now the HopliteBP and MLE designs show the highest feasibility envelopes while HopliteBuf loses severely. HopliteBuf is unable to maintain feasibility as the SRL32 FIFOs run out of capacity for larger data blocks. It is possible to increase feasibility by increasing FIFO sizes, but that impacts LUT cost and causes HopliteBuf to exceed the footprint of the HopliteBP switches.
- As we increase system size, we note that HopliteBP designs start to lose feasibility quickly due to pessimism in the analysis for backpressure-based designs. For pipelined backpressure-based switches, the analysis must account for the cascading effect of network flows, which significantly depresses sustainable rates. LOCAL pattern suffers a greater loss in feasibility due to the short distances traversed by the flows and resulting larger conflict set of flows.



(a) RANDOM

(b) LOCAL

Figure 5.6: Feasible set of RANDOM and LOCAL flowsets on system sizes 3×3–7×7 (*across columns*) and data block sizes 1–16 (*across rows*). Note that x and y axes are shared vertically and horizontally respectively.

- Finally, we observe that MLE-optimized NoC smoothly navigates the entire design space to deliver the highest feasibility rates across all combinations. This confirms that neither extreme solution works best in all cases, and a mix-and-match approach is necessary to get the best outcomes. Importantly, by mixing and matching switch configurations, MLE NoCs exceed the potential of either NoCs in isolation, achieving

≈2–3× higher feasibility over vanilla designs.

5.6.2.2 Learning for Worst-Case Latency

Next, in Figure 5.7, we show the effect of varying regulation rate of a 5×5 NoC when routing feasible LOCAL traffic traces. We configure the MLE to learn switch configurations to minimise `wclatency`. We summarise our observations with varying regulation rates and block sizes:

- First, we note that as the rate increases and reaches an inflection point, we see an increased spread in latencies across all configurations. This is due to the traffic regulation model dominating cycle counts below this inflection rate, forcing the network congestion effects to take a backseat. Second, as we increase data block sizes, the latencies increase and the spread shift upwards. We also note that there is noise around the calculation of mean curves due to the shifting feasibility combinations of the flowsets.
- HopliteBuf NoCs are competitive at lower regulation rates. At higher regulation rates, the worst-case latencies can be quite large for certain flowsets. This is a direct result of deeper FIFOs and associated head-of-line blocking effects [100] for those scenarios. Also, HopliteBuf NoCs do not scale to large block sizes (=16) because they run out of stall-free FIFO capacity.
- The HopliteBP NoCs initially start with higher latency at the lower rates, but become competitive at larger regulation rates. At higher rates, HopliteBP NoCs suffer from loss of feasibility and cease to scale. At higher data block sizes, HopliteBP NoCs continue to scale beyond HopliteBuf for higher rates but saturate below MLE.
- MLE-optimized NoCs deliver competitive latencies across all rates and block sizes. At lower rates, the MLE-optimizer prefers reducing FIFO size and mimics HopliteBuf solutions as the regulation dominated latency is a fixed effect. At increased rates, MLE can strategically replace congestion hotspots with HopliteBP designs and avoid the increased buffering effects that cause high latencies for HopliteBuf designs. At large data block sizes and rates, MLE-optimized NoCs are the only feasible combinations outperforming **both** HopliteBuf and HopliteBP NoCs.

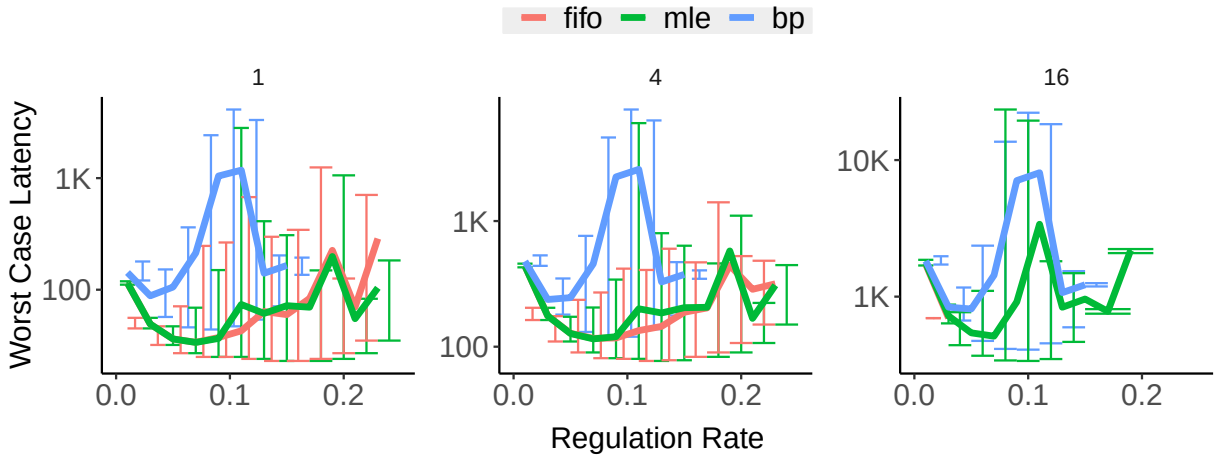
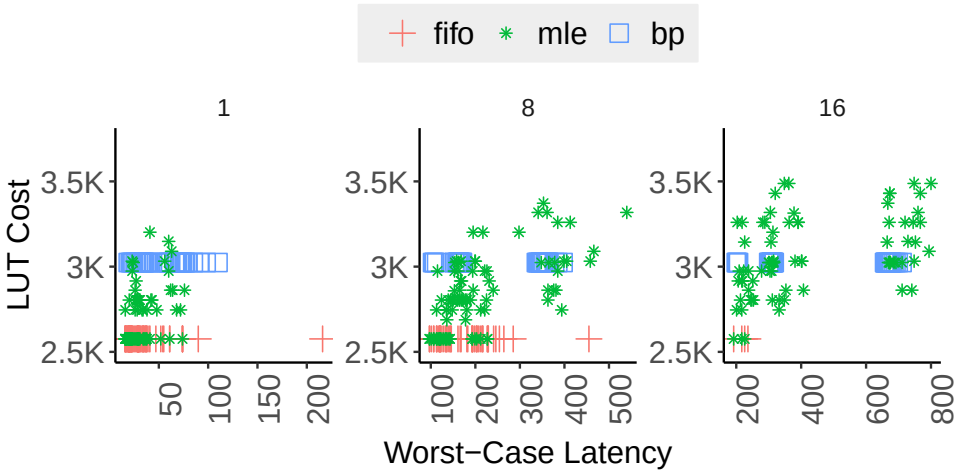


Figure 5.7: Worst-case latency as a function of regulation rate for 100 synthetic LOCAL flowsets for various data block sizes.

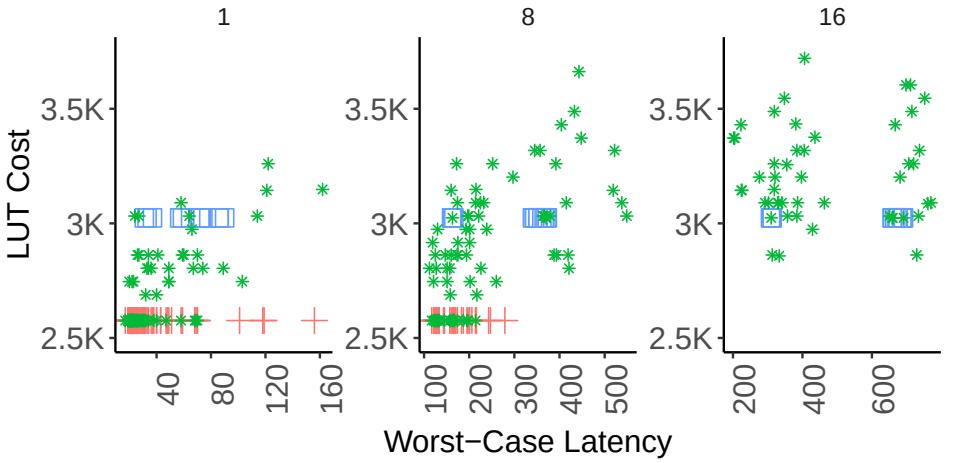
5.6.2.3 Learning for Cost Constrained Worst-Case Latency

For the next set of experiments, we configure the MLE to optimise for the cost constrained latency function of `wclatency*cost`, with an aim to analyse its ability to balance the growth of the two negatively correlated terms while simultaneously minimising their product. In Figure 5.8, we look at cost-latency tradeoffs when considering feasible RANDOM and LOCAL flowsets that are routed at a system size of 4×4 and a regulation rate of 0.17. Regulation rate of 0.17 allows us to stress-test the design; as many applications start to declare infeasibility beyond this rate when block size increases.

At small data block sizes, MLE-optimized NoCs end up occupying the cost range closer to the lower-cost HopliteBuf designs. As block sizes increase, vanilla FIFO designs run out of capacity and declare infeasibility and MLE-optimized NoCs consume increasingly more LUTs. This suggests that MLE will replace HopliteBuf switches with the more expensive HopliteBP/HopliteBP+Buf versions in exchange for feasibility or proportionate latency gains. We also note the narrower spread of HopliteBP latencies which is a direct result of having fewer feasible combinations at that rate and block size.



(a) RANDOM

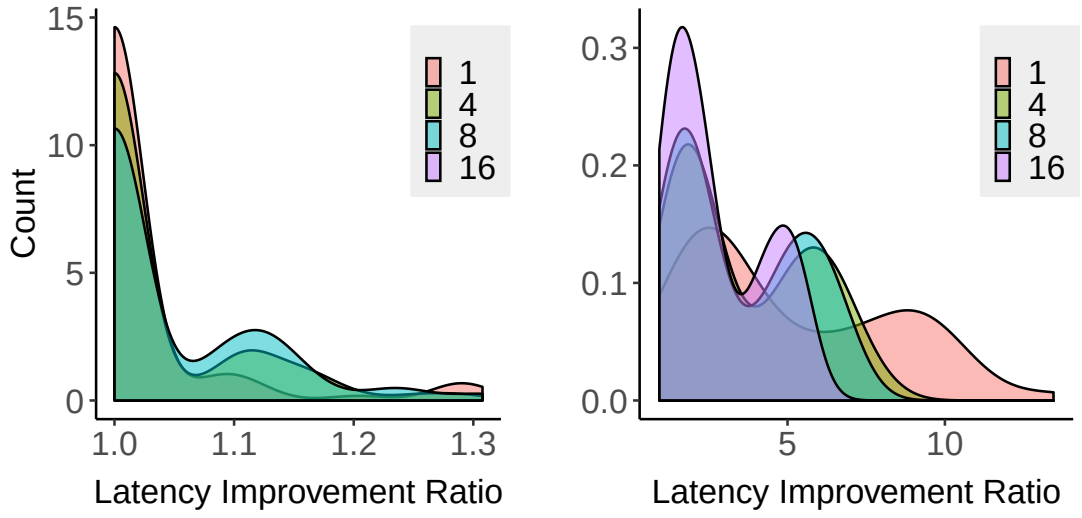


(b) LOCAL

Figure 5.8: LUT Cost-Worst Case Latency tradeoffs for a 4×4 NoC with 0.17 regulation rate for 100 synthetic RANDOM and LOCAL flowsets across various block sizes.

We also quantify the extent of latency improvement distribution over HopliteBuf and HopliteBP switches in Figure 5.9 for 100 LOCAL flowsets at 0.13 regulation rate across different data block sizes on a 4×4 NoC. When compared to HopliteBuf designs, we note latency reduction by as much as 2× with little sensitivity to block sizes. When compared against HopliteBP, the latency wins can be as much as 15×. This larger win is attributed to pipelining effects in backpressure-based networks that can cause the analysis to include

a large number of flows in the conflict set to produce safe upper latency bounds.



(a) vs. HopliteBuf (b) vs. HopliteBP

Figure 5.9: Cumulative distribution of improvements of worst-case latency MLE NoC over HopliteBuf and HopliteBP

5.6.2.4 Routing Real World FPGA Applications

We now show the effect or regulation on worst-case latency, across all flows, of realistic FPGA workloads running on a 16-client NoC. In Figure 5.10, we study the effect of regulation rate on worst-case latency of flows with a data block size of 4. With low congestion at low rates, all NoC designs exhibit similar characteristics. As we increase injection rates, we notice increased worst-case latencies across most workloads. In particular, we note greater feasibility and lower worst-case latencies for MLE-optimized NoCs. For smaller data block sizes (See Table 5.2), MLE prefers HopliteBuf NoCs as most conflicts can be absorbed in the shallow SRL FIFOs. However, smaller blocks sizes require extremely wide NoCs with hundreds of bits of payload sent as a single block. We conclude that MLE-optimized hybrid NoCs are at par or better than either designs by sustaining 1-9× higher rates while achieving 1-6.8× lower latency across all real application benchmarks and block sizes.

		Rate×10^3			*Latency*					*Rate*×10^3			*Latency*		
		F	B	M	F	B	M			F	B	M	F	B	M
wiki	1	17	8	17	743	6.7K	743	google	1	17	8	17	789	6.7K	789
	4	–	8	8	–	16.2K	16.2K		4	–	8	8	–	16.1K	16.1K
stnfd	1	23	11	23	1310	6K	1310	soc	1	17	8	17	789	8.5K	789
	4	–	11	11	–	14.5K	3.2K		4	–	8	8	–	20.2K	20.2K
dac	1	19	10	19	1238	10.6K	1216	rdnet	1	37	19	37	1661	1609	1635
	4	–	10	10	–	25.6K	25.6K		4	3	19	27	1.4K	3.9K	3.4K
ram2k	1	28	14	28	998	2K	998	mmpls	1	27	14	27	2K	3.2K	2K
	4	2	14	15	2.1K	4.8K	3.5K		4	–	14	16	–	7.6K	6.1K
gene2	1	17	8	17	789	11.3K	789	bmhf3	1	25	14	25	1.5K	1.9K	1.5K
	4	–	8	8	–	27K	27K		4	–	14	14	–	4.7K	4.6K
bmhf2	1	25	12	25	1.6K	10.5K	1.5K	bmhf1	1	20	9	20	720	5.2K	720
	4	–	12	12	–	26.2K	3.8K		4	–	9	9	–	12.4K	12.4K
amzon	1	17	8	17	789	11.3K	789	add20	1	19	9	19	898	5.2K	898
	4	–	8	8	–	27K	27K		4	–	9	9	–	12.4K	12.4K

Table 5.2: Application Performance over block size of 1,4 for `Graph`/`SpMV` traces. (`F`: FIFO, `B`: Back-Pressure, `M`: MLE)

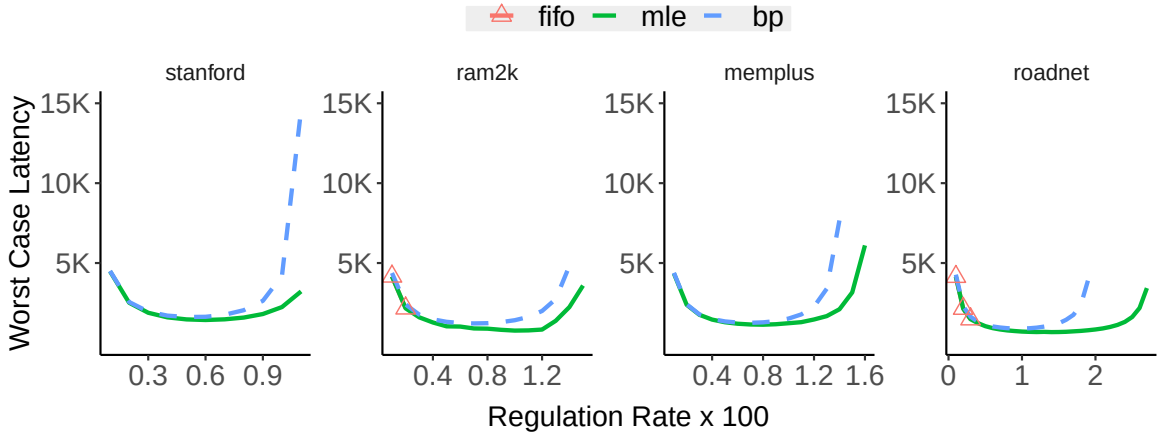


Figure 5.10: Latency Scaling of a 4x4 NoC for `Graph`/ `SpMV` traces

5.6.2.5 Analysing Efficiency of MLE

Finally, we turn our attention to the MLE optimization flow and try to understand how the solver discovers good solutions. In Figure 5.11, we plot the solution quality (LUT

cost × worst-case latency) and time taken to discover the solutions across a range of 100 synthetic RANDOM workloads targeting a 5×5 NoC at a regulation rate of 0.1. The HopliteBuf and HopliteBP NoCs are one-shot solutions that do not need any search and are hence observed in the left corner of the space. MLE's highly composable binary decision making proceeds in an iterative fashion before it stabilizes. We observe a narrower spread of objective function values after 1–10 s. CMA-ES [82] explores the space just as effectively but takes ≈5–10× longer due to the necessary but obtuse integer quantization of real-valued distributions used by the optimizer. We also use RBFOPT [98], that generates marginally inferior solutions and is 50–500× slower than our MLE approach. While RBFOPT claims to require fewer explorations, the number crunching after each sample is compute intensive and dominates the fast analysis process for our problem.

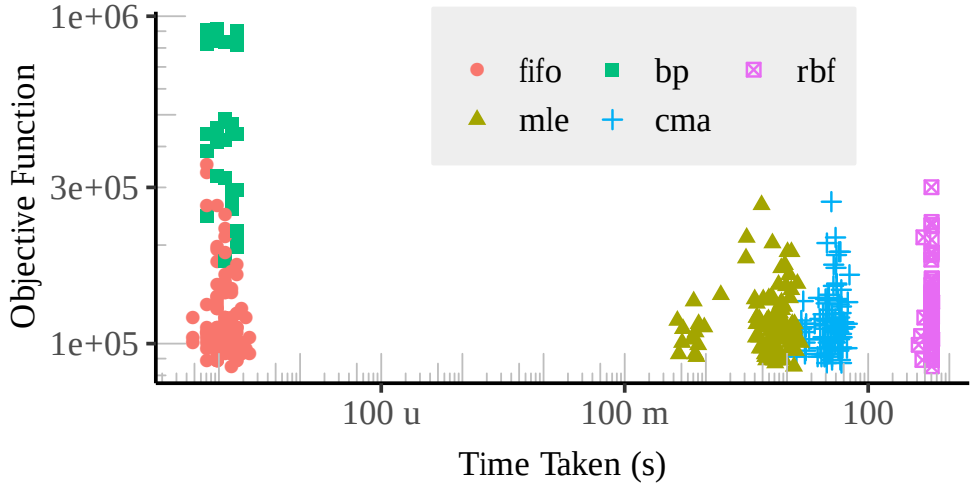
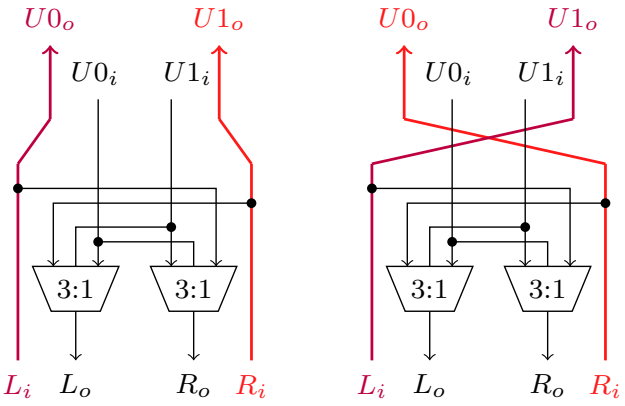


Figure 5.11: Solution quality and time taken by MLE optimizer across flowsets mapped to a 5×5 NoC with 0.1 regulation rate

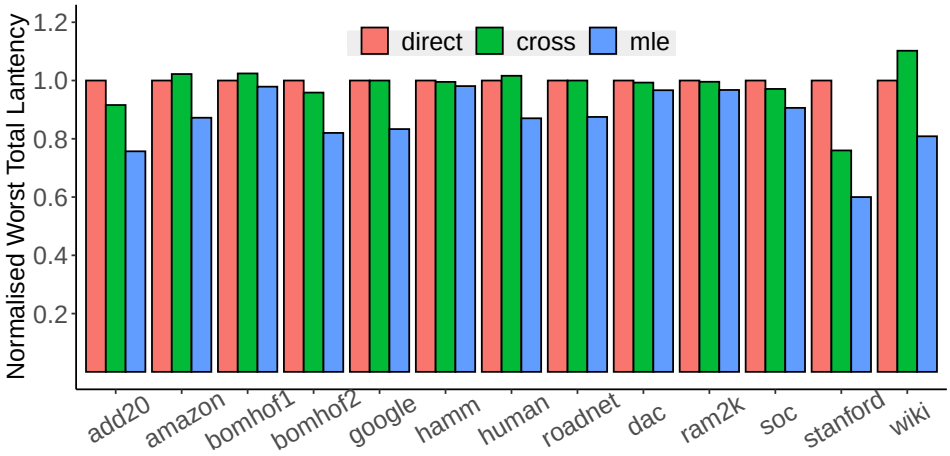
5.6.2.6 Adapting MLE for NoCs with different topologies and parameters

In this section, we take a look at MLE's ability to optimise diverse NoC design parameters for NoCs with markedly different topologies. For this case study, we focus on the Butterfly Fat Tree crossbar (BFT3) topology in [3] (chapter 4). A BFT3 NoC with N endpoints is made up of $log_2(N) * \frac{N}{2}$ unique pi switches. We refer the reader to [3] (chapter 4) for further information on the BFT3 NoC and pi switches. A packet climbing a pi switch is configured to have a `direct routing` in [3] (chapter 4); packets entering a pi switch's left port are routed to the left port of U (**L** → **U0**) with a similar connection for packets

climbing a switch from the right (**R** → **U1**), as shown in Figure 5.12a. Alternatively, `cross` routing connects **L** → **U1** and **R** → **U0**, shown in Figure 5.12b.



(a) `Direct` Routing (b) `Cross` Routing



(c) MLE configured BFT3 crossbar against `direct` and `cross` vanilla crossbars

Figure 5.12: Analyzing MLE's ability in optimising for the parameter design space of a Butterfly Fat Tree crossbar NoC.

In this work instead, we use MLE to configure the routing of the crossbar, choosing between `cross` or `direct`, on a *per-switch* basis for each of the $log_2(N) * \frac{N}{2}$ switches, tuned to optimise a specified QoR. Contrasting Figure 5.12a and 5.12b, MLE introduces no overheads since only direct connectivity is changed. Furthermore, properties of the original

work: livelock freedom and in-order packet delivery are also preserved. We optimise a BFT3 NoC with 16 endpoints to minimise the total worst latency while routing `real` benchmarks at 100% injection rate in Figure 5.12c. We observe that an application optimised BFT3 achieves latencies that are ≈1.1-1.7× lower than the vanilla `direct` and `cross` BFT3 versions, strongly suggesting that the MLE framework of this work can be leveraged for other NoC topologies and tuning different parameter spaces.

5.6.2.7 Comparing an ASIC implementation with Timing Predictable NoCs

NoC	Area
MLE-1	**0.009**
MLE-2	**0.019**
MLE-3	**0.028**
PhaseNoC [63]	0.027
SurfNoC [62]	0.033

Table 5.3: Area in mm^2. MLE NoCs' area calculated using Intel 45 nm process (3.3x10^6 transistors/mm^2)

In this section, we compare and analyze a potential ASIC implementation of an 8x8 Hoplitebuf+BP NoC where the NoC switches can be fabricated once and be configured to operate in FIFO or backpressure mode, configured on a per-switch basis for the application at hand. We compare the result of MLE optimised analysis of the hoplite NoC and compare it against the state-of-the-art in timing predictable NoCs: mainly SurfNoC [62] and PhaseNoC [63]. We evaluate the NoCs on 2 fronts: 1. The Area overhead for each router (Table 5.3) and 2. Average routing latency (Figure 5.13). For estimating the area of the HopliteBuf+BP switch, we model each 6-LUT of the switch using transistors [101]. We estimate area consumption of the resulting design using Intel 45 nm technology with a transistor density of 3.3x10^6. We observe that the HopliteBuf+BP (MLE-1 in Table 5.3) switch is 3-3.7× smaller than its counterparts, thus allowing us to replicate the NoC up to 3 times (MLE-3) and tripling the available routing bandwidth, while still maintaining area parity with Phase and Surf NoCs. This is especially useful for increasing routing feasibility when routing becomes infeasible with increasing injection rates, owing to the lightweight flow controls of the switches (compared to Surf and Phase).

For latency comparison in Figure 5.13, we compare the average latencies of each trace in the application flowset with random-uniform communication patterns with a data block size of 6, where each node in the NoC randomly chooses another destination node in the NoC. We generate 100 such random application flowsets to eliminate any bias in the distributions and average the results over these application flowsets. The data for Phase and Surf NoCs is extracted from [63]. We add additional channels (MLE-2, MLE-3) whenever the MLE generated NoC becomes infeasible with increasing injection rate. Note that even with replicated channels, the MLE optimised HopliteBuf+BP ASIC NoC is still cheaper than

the Surf and Phase NoCs (Table 5.3). We observe in Figure 5.13 that MLE generated NoC analysis results in average latencies that are 1.3-12× lower than Phase and Surf NoC variants, owing to the application aware tuning of the NoC and always available routing bandwidth flow control of buffering and backpressure, without any TDM scheduled routing breaks.

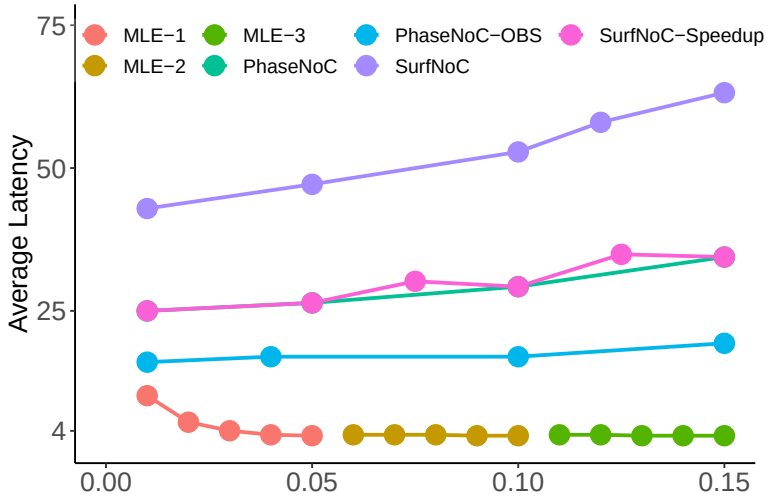


Figure 5.13: Average Latency vs Rate. Phase and SurfNoC from [1]

5.7 Conclusions

In this work, we show how to evolve hybrid FPGA NoC switch parameters to deliver a combination of feasibility, worst-case latency, and cost improvements over homogeneous FPGA NoCs. We demonstrate switch learning by combining HopliteBuf, a stall-free FPGA NoC, with HopliteBP, a lightweight backpressure-based FPGA NoC using a fine-grained per-switch static configuration model. We use Maximum Likelihood Estimation technique to evolve NoC configurations that offer ≈2-3× improvements in feasibility, ≈1–6.8× in worst-case latency over synthetic and real world applications.

Chapter 6

Learning Injection Rates and Switch Configuration

We can learn a NoC configuration on a per-switch basis to optimise QoR of workload, as seen in Chapter 5. Switch learning for the Hoplite NoC improves routing feasibility by ≈2–3× and reduces worst-case latency by ≈1–6.8×. We can further improve routability by learning to tune regulation rates for each traffic trace to maximise available routing bandwidth. We capture critical dependency between traces by modelling the regulation space as a multivariate gaussian distribution and learn the distribution's parameters using Covariance Matrix Adaptation Evolution Strategy (CMA-ES). We also propose *nested* learning that learns switch configurations and regulation rates in-tandem. Compared to standalone switch learning, nested learning helps achieve ≈1.5× lower cost constrained latency, ≈3.1× faster individual rates and ≈1.4 × faster mean rates. We also evaluate improvements to vanilla HopliteBuf/BP NoCs' routing using only standalone rate learning (no switch learning); with ≈1.6× lower latency across synthetic and real world benchmarks.

6.1 Introduction

We have previously introduced the design of a resource-shared, high performance and application customizable Hoplite NoC that tunes switch parameters per workload. We use a Maximum Likelihood Estimator (MLE) that is designed to learn switch configurations that maximise the FPGA workload's Quality of Result (QoR) like worst case latency, cost-constrained latency, sustained throughput etc. The MLE algorithm chooses between the

HopliteBuf and HopliteBP variants of the switch architecture. HopliteBuf switches add a stall-free SRL32 FIFO to absorb lower priority $W \rightarrow S$ packets that lose out against the higher priority $N \rightarrow S$ packets. While useful in lowering contention at the switch, these FIFOs can quickly run out of capacity with increasingly bursty traffic, as we saw previously in sections 5.6.2.1 and 5.6.2.2. HopliteBP switches instead introduce a lightweight backpressure interface for the same losing $W \rightarrow S$ packets. This latency insensitive backpressure signal then propagates along the horizontal rings and stalls all horizontal traffic across the ring *in place*. While this addresses the inability of HopliteBuf to handle increasing bursty traffic, the backpressure signal spreads from the contention switch and also stalls packets not involved in the contention, as we say previously in sections 5.6.2.1 and 5.6.2.4. We refer the reader to section 2.5 for more details about the HopliteBuf and HopliteBP switches. A user supplied set of network flows can contain of combination of network traffic flows that can emphasise the strengths and weaknesses of both switches. We make use of the Maximum Likelihood Estimation algorithm (introduced in section 5.5) to understand these flowsets interactions and maximise workload QoR by learning, for each switch in the NoC, the switch configuration (between HopliteBuf and HopliteBP).

NoCs make use of regulators to guarantee application bandwidth Service Level Agreement (SLA) by mitigating a variety of packet starvation, deadlocks and denial of service scenarios. For a given application flowset consisting of variety of traffic traces, HopliteBP and HopliteBuf rely on token bucket regulators [95] for every trace to schedule the injection of its packets into the NoC. For a given flow, its token bucket regulator limits the injection rate and data block size of that flow to its prescribed regulation rate (ρ) and burst size (b) respectively. The token bucket regulator is explained in greater detail in section 5.4.2. Regulation is an important variable in the Hoplite suits of analysis tools to guarantee tight bounds on routing latency and NoC costs. Previous versions of Hoplite, [41] and Chapter 5 ([2]), leave regulation rate out of the decision space as a user parameter; evaluating NoC designs by measuring performance over a range between [0,1] with a fixed increment linear search. Furthermore, only a single regulation rate is used to regulate all traces in an application flowset, which can create unintended bottlenecks in NoC routing and distort search spaces of other NoC parameters.

This work extends the scope of regulation by *learning* regulation rates for *each trace*, with the aim of maximising the requested QoR, specific to the given application flowset. This allows us to surgically address localised contention bottlenecks within the NoC by individually regulating traces involved in the bottlenecked hotspot. We encode the rates for a flowset as a multivariate gaussian distribution to capture the dependence of a trace's routability on other traces within the flowset and learn the parameters of this distribution using Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) [82]. Furthermore,

we introduce nested learning where we learn the switch configurations of the NoC (from chapter 5) and the regulation rates (this chapter) of traces *in tandem*. This coupling of two search spaces allows one to modulate the other in a synergistic manner, thus allowing for a more effective NoC design that outperforms designs born out of learning only either one of the search spaces.

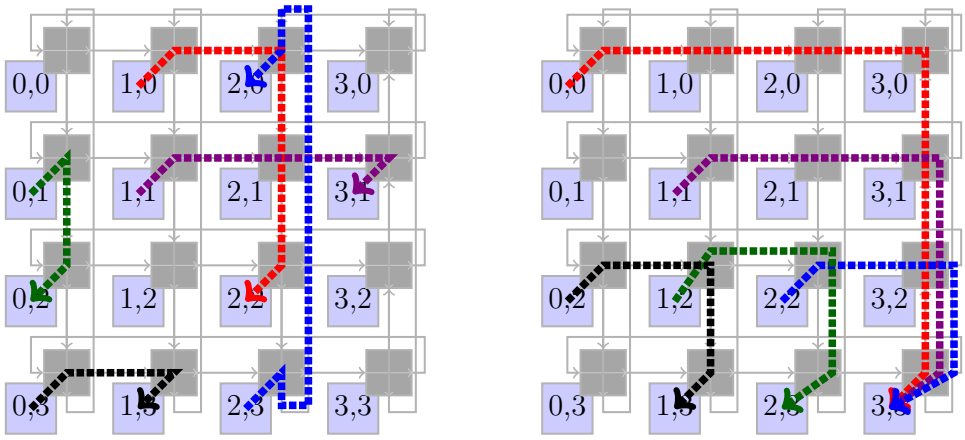
The key contributions of this work include:

- Development of a multivariate gaussian distribution based evolutionary learning strategy to learn regulation rates, specific to the application flowset and QoR requested.
- Development of a nested learning strategy to learn switch level configurations and regulation rates for an application flowset in tandem.
- Quantification of worst-case latency, feasibility, cost and optimization runtime across real and synthetic applications; when learning rates and switch configurations in-tandem (*nested learning*), a further ≈1.5× lowered cost constrained latency, ≈3.1× faster individual rates and ≈1.4× faster mean rates.

6.2 Motivating Example

In this section, we discuss the motivation for learning NoC parameters for a given application flowset. Specifically, in order to maximise a user requested QoR (quality of result) for a specific application flowset, we build a case for: learning the regulation rates for every traffic trace present in the application flowset. This work is based on learning regulation rates for NoC designs integrating the HopliteBuf, HopliteBP and HopliteBuf+BP switches into a unidirectional torus topology with DOR routing. We refer the reader to sections 2.5 and 5.3 for information about design of these 3 switches and related work. We also refer the reader to section 5.5 for a refresher on learning switch configuration in an application aware manner.

We now make a case for learning regulation rates for every trace in an application flowset by presenting a set of arguments using Figure 6.1a and 6.1b. Specifically, each traffic trace within an application flowset interacts with other traces in a complex non-deterministic manner, thus affecting NoC routing. Flowset-specific learning allows us to maximise NoC routability by encoding the latent effects of these traces' interactions into the learnt rates. We now discuss two specific advantages of learning unique regulation rates



(a) Localized Contention (b) Turning Traffic in HopliteBuf

Figure 6.1: Interacting network flows on a 4×4 NoC arranged to show the benefits of learning regulation rates for each traffic trace

for every trace in the flowset, specific to the applications and QoR at hand. We contrast this proposed learning approach against the practice of using a single rate to regulate all traces within a flowset in [41] and Chapter 5 ([2]).

- **Local contention zones**: In Figure 6.1a, the NoC comprises of a localised contention hotspot around switch (2,1), comprising of 3 traces (▪▪▪, ▪▪▪, ▪▪▪). Regardless of mode of operation of the switches, these 3 traces generate more contention than the other two traces (▪▪▪, ▪▪▪) if we regulate all of them by the same amount. As a result, this localised hotspot (▪▪▪, ▪▪▪, ▪▪▪) forms a bottleneck in the feasibility and latency analysis, thus leaving available NoC bandwidth untapped by (▪▪▪, ▪▪▪). For example, the non conflicting traces' (▪▪▪, ▪▪▪) regulated injection rates can reach levels that are much higher than the hotspot's, without having any adverse affect on the routability of the application flowset. An application flowset can generate a multitude of these disparate contention zones within a NoC. Learning *unique trace level* regulation rates, specific to the application flowset, allows us to extract information about such trace interactions and recover lost NoC routing bandwidth. In turn, this allows for a more efficient optimization of the application QoR.

- **Learning switch configuration**: In Figure 6.1b, we investigate learning switch level configurations for the traces shown. We start off with all of the NoC's switches operating in HopliteBuf mode. The application flowset consists of predominantly

turning traffic traces. Note that switch (3,2) south link has 3 traffic traces exiting through it (▪▪▪ , ▪▪▪, ▪▪▪) which forms a potential *choke point* that can quickly become infeasible as regulated injection rates of these three traces increase (FIFO runs out of capacity). The lost feasibility can be recovered by replacing this HopliteBuf switch (3,2) by a HopliteBP NoC. However, this can lead to frequent and incessant stall of its entire row due to backpressure signals induced from its west port. This unfairly penalizes innocent traces (▪▪▪, ▪▪▪) which were not even interacting with the infeasibility causing traces but now are made to sacrifice their available bandwidth to make routing feasible. Furthermore, the entire row must now be upgraded to a HopliteBP+Buf configuration to support backpressure and FIFO signals, resulting in a more costly NoC design. Instead, an easier alternative is to address the infeasibility at its source. We can tune the regulation rates of the traces most responsible for eating up FIFO capacity at switch (3,2); trace (▪▪▪) since it is not only involved in the contention at switch (3,2) but also is responsible for increasing the data block size of the trace exiting at south port of switch (3,1).

Such subtle and non-deterministic interactions between traffic traces of a complex flowsets can be challenging for a NoC to address. The key idea explored in this work is to learn the regulation rates of each traffic trace in an application flowset to:

1. Tune localised hotspots by encoding this contention information into regulation related decision making for affected traces.
2. Increase efficiency of switch learning with nested learning where rate learning can help expand the domain of feasible switch solutions as well as directly improve NoC routing conditions.
3. For a NoC of size NxN, efficiently navigate for solutions in a multi-modal search space that asymptotically grows as $(0, 1]^{N^4}$ and makes simple search based techniques impractical.

6.3 Evolutionary Learning of Regulation Rates

We now present the key ideas explored in this work. We learn (for a given application flowset, data block size and defined QoR), unique regulation rates for every trace in the application flowset. This allows us to tailor the NoC towards the application at hand by optimizing for a user defined QoR. We learn regulated injection rates of every trace in the

flowset using Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES). We also present *nested* learning, where we learn both regulation rates and switch configurations in-tandem.

6.3.1 Learning Regulated Injection Rates

We now explain how we learn the degree of regulation introduced in each trace for a given application flowset. An application flowset can be composed of a diverse types of traces and much like the *butterfly effect*, each trace generates varying degrees of contention within the NoC via second order interactions with other traces. We wish to improve the overall routing conditions of these complex traffic interactions by regulating the amount of traffic generated by each trace within a time frame. Specifically, for a given application flowset, we learn these regulated injection rates, *for every trace in the flowset*, to maximise a user requested QoR function. We explore two methods of rate learning: 1. `Univariate` and 2. `Multivariate`.

6.3.1.1 Univariate Rate Learning

One method is to linearly search over a uniform distribution $\in$ (0,1] with the increment precision truncated to 32 bits to mimic the regulation dynamics of the token bucket regulators. Each search sample is then universally applied to all regulators and the resulting QoR determined. The sample that best optimises for the requested QoR can be selected as the desired regulation rate. This is equivalent to the linear search of chapter 5 (learning only switch configurations) $\in$ (0,1], with NoC routability measured as a function of the regulation rate. The precision of the linear search increments can be fine-grained to support 32 bit regulators instead of the chunkier increments of Chapter 5. Furthermore, the brute force linear search can be replaced with an intelligent binary search or a `univariate` gaussian modelling, with no effect to the final quality of solution. While information about the application flowset is encoded in this *light and fast* methodology, it fails to take into account any unique dependencies between traces and their latent effects on other regions of the NoC since all traces are equally regulated. Furthermore, it prevents the NoC from reaching its peak available bandwidth since regulation is bottlenecked by the worst trace in the flowset. We compare this `univariate` learning against `multivariate learning` in more detail in section 6.4.2.

6.3.1.2 Multivariate Rate Learning

We now present the main idea of learning regulation rates: modelling the regulation rate of each trace as a `multivariate` gaussian distribution and learning this distribution's parameters using Covariance Matrix Adaptation Evolutionary Strategies (CMA-ES) [82]. Formally, as shown in equation 6.1, for a NoC of size N×N, given a flowset F composed of set of traces $(f_1, f_2, ..., f_i, ..., f_n)$ and data block size ρ, we wish to learn the corresponding regulation rates R=$(r^{f_1}, r^{f_2}, ..., r^{f_i}, ..., r^{f_n})$ such that the user defined objective QoR function Q, calculated as a function of the Hoplite analysis suite H, can be maximised (or minimised).

$$F = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_i \\ \vdots \\ f_n \end{bmatrix}$$

$$R = \begin{bmatrix} r^{f_1}(\text{rate for flow} f_1) \\ r^{f_2}(\text{rate for flow} f_2) \\ \vdots \\ r^{f_i}(\text{rate for flow} f_i) \\ \vdots \\ r^{f_n}(\text{rate for flow} f_n) \end{bmatrix} : r^{f_i} \in (0, 1] \forall i$$

$$R = \text{argmax}(Q(H(F, \rho, R))) \tag{6.1}$$

Recall that each trace in a flowset can impact the others via second order interactions and we encode this dependence by modelling R as a multivariate gaussian distribution $R \sim \mathcal{N}(\mu, \sigma^2)$, with $\mu \in [0, 1]^n$ and $\sigma \in [0, 1]^{n \times n}$. This is a key differentiation from `univariate` of section 6.3.1.1 and [2] as it allows us to extract crucial information about these interactions. Furthermore, increasing the dimension of the regulation search space allows us to directly address bottlenecks in NoC routing by: 1. learning the regulation rates of the most difficult traces in an application as well as 2. encode this information in rates of traces that might be involved in indirect interactions. `Multivariate` learning, when paired with algorithms that learn other NoC parameters (switch configuration of

chapter 5 in section 6.3.2), also provides access to a much richer search space in contrast to `univariate` learning. This is explained by `univariate` unilaterally regulating all traces with the same amount, thus introducing a strong bias in other algorithm's search space.

We now explain the modelling of `multivariate` learning as a black box optimization problem using CMA-ES. Just like switch learning of MLE in chapter 5, the core of `multivariate` learning involves sampling C candidate rates $R_C^g \in (0,1]^{n^2 \times C}$ from a multivariate distribution $\mathcal{N}(\mu_g, \sigma_g^2)$ at every generation g as shown in Equation 6.2. For our experiments, we choose $C = 100$. Note that the population size is same as the MLE based switch learning of Chapter 5 ([2]), despite an increase in complexity from Bernoulli to Multivariate Gaussian distribution. This is because we also proportionally increase the model complexity from Maximum Likelihood Estimation to Covariance Matrix Adaptation. Each sample in R_C^g is then tested for fitness on the QoR function. As shown in Equation 6.3, the top λ performing regulation rate samples $R_{\lambda:C}^g \in (0,1]^{n^2 \times \lambda}$ are then selected for CMA-ES to evolve the multivariate distribution in the general direction of top performing samples by updating the mean μ and covariance σ according to the evolutionary steps detailed in [82]. Observe in Equation 6.3 that CMA-ES maintains an evolutionary path to keep a historical track of past generation's multivariate parameters $(\mathcal{N}(\mu_1, \sigma_1^2), \mathcal{N}(\mu_2, \sigma_2^2), ..., \mathcal{N}(\mu_{g-1}, \sigma_{g-1}^2)$. This encoding of trajectory of past decisions plays an important role to help balance the *exploration* vs *exploitation* tradeoff and prevents CMA-ES from getting stuck in local optimas.

$$R_C^g = \begin{bmatrix} \text{Candidate 1's rate for } f_1 \text{ at generation g} & \text{Candidate 2's rate for } f_1 \text{ at generation g} & \cdots & \text{Candidate C's rate for } f_1 \text{ at generation g} \\ \vdots & \vdots & \ddots & \vdots \\ \text{Candidate 1's rate for } f_i \text{ at generation g} & \text{Candidate 2's rate for } f_i \text{ at generation g} & \cdots & \text{Candidate C's rate for } f_i \text{ at generation g} \\ \vdots & \vdots & \ddots & \vdots \\ \text{Candidate 1's rate for } f_n \text{ at generation g} & \text{Candidate 2's rate for } f_n \text{ at generation g} & \cdots & \text{Candidate C's rate for } f_n \text{ at generation g} \end{bmatrix}$$

$$R_C^g = \underbrace{\begin{bmatrix} r_g^{f_1,1} & r_g^{f_1,2} & \cdots & r_g^{f_1,C} \\ \vdots & \vdots & \ddots & \vdots \\ r_g^{f_n,1} & r_g^{f_n,2} & \cdots & r_g^{f_n,C} \end{bmatrix}}_{\text{Each } \textbf{column} \text{ is a flattened regulated injection rate } (h_n)} \sim \mathcal{N}(\mu_g, \sigma_g^2) \tag{6.2}$$

$$\mathcal{N}(\mu_{g+1}, \sigma_{g+1}^2) = CMA(\{\mathcal{N}(\mu_1, \sigma_1^2), \mathcal{N}(\mu_2, \sigma_2^2), ..., \mathcal{N}(\mu_g, \sigma_g^2)\}, R_C^g, R_{\lambda:C}^g) \tag{6.3}$$

Representative updates to a 2D Gaussian distribution for `multivariate` learning of regulation for a 2×2 HopliteBP NoC are shown in Fig 6.2. As a black box optimization, CMA-ES enables us to learn best R in a large search space of $(0,1]^{N^4}$ without requiring

any explicit knowledge of the QoR function or the myriad functions in the Hoplite analysis suite. This foregoes the need for any calculation of gradients and allows for a well composed interaction with other black box learners (like MLE for switch configuration); thus allowing for symbiotic nested learning techniques over multiple NoC parameters.

6.3.2 Nested Learning

In this section, we explain the formulation of nested learning, where multiple parameters of the NoC can be learnt to optimise for a specific application flowset. We focus on learning regulation rates as well as switch level configuration of the NoC *in-tandem*; with each influencing the other's search space and having a direct impact on the learnt parameters. This mutual dependence arises from the direct coupling of regulation and switch configuration on the performance of the NoC. When learning for a specific application flowset, changing one set of parameters (example: regulation rates) will influence potential application routing, thus shaping the search space for the other parameter set (example: switch configuration).

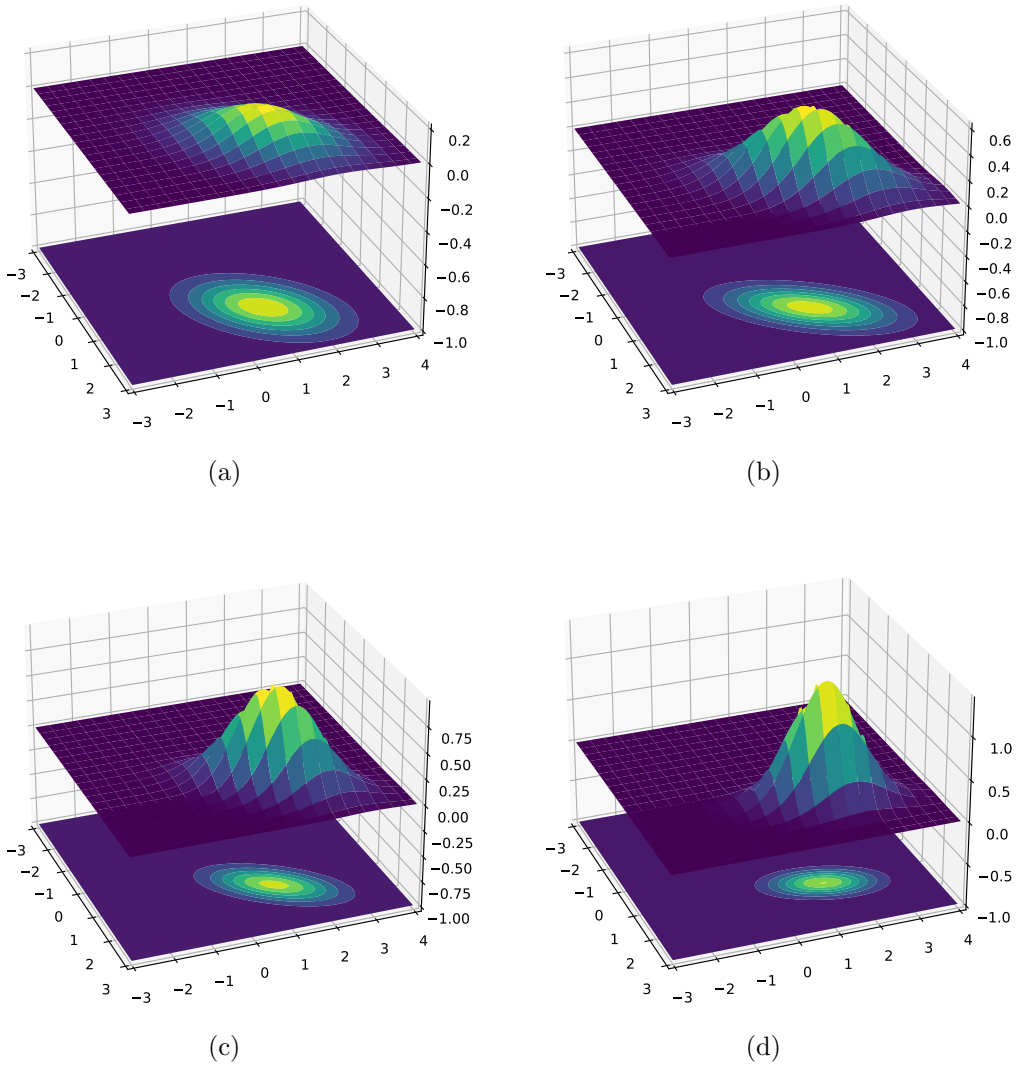


Figure 6.2: Evolution of a 2D gaussian distribution when learning regulation rates for a 2x2 HopliteBP NoC with 2 traces of data block size 1 in a RANDOM flowset. Note the convergence of the distribution over epochs as CMA-ES progressively learns better solutions. This is a toy example for illustrative purposes. We evaluate our design on larger NoC sizes in section 6.4

Algorithm 1: Nested Learning

```
Result: R reg., S switch conf.
CMA.init;
while not CMA.converge do
    CMA.gen_samples;
    for s_CMA in CMA.samples do
        MLE.init;
        while not MLE.converge do
            MLE.gen_samples;
            for s_MLE in MLE.samples do
                qor=calc(s_CMA, s_MLE);
                MLE.tell(qor);
            MLE.fit();
        qor=calc(s_CMA, MLE.best);
        CMA.tell(qor);
    CMA.fit();
```

Algorithm 1 explains the algorithm for rate learning along with learning switch level configuration for Hoplite NoCs. The objective of this algorithm is to find rate R, switch configurations S such that $(R,\ S) = \text{argmax}(Q(H(F, \rho, R, S)))$. Note that `multivariate learning` has a much larger search space $\in (0, 1]^{N^4}$ compared to switch learning which has a search space of 2^{N^2}. Also, a gaussian distribution for rates is computationally more challenging than a Bernoulli distribution for switches. Thus, we nest the switch configuration learning algorithm (MLE) inside rate learning (CMA). This decision helps cut down learning times many folds since the much larger search space of rate learning will only need to converge once; with the simpler switch learning converging to generate best switch configuration for each rate candidate generated by CMA. The `fit` functions used by respective algorithms perform the action of selection of fittest candidates out of CMA.samples and MLE.samples respectively, followed by evolution of their respective distributions.

Since the nested search space can rapidly explode in size with increasing NoC size for challenging flowsets, we prepend Algorithm 1 with steps to find the appropriate starting point. We do this for two reasons: 1. To remove starting points with obviously infeasible solutions (for example, little to no regulation for all traces) and 2. Provide a better seed/starting position to `multivariate learning` for a tighter initial spread of gaussian distribution. To this end, we first introduce a universal regulated injection rate of 1 (implying no regulation) and apply it to all traces in the application flowset. We keep *halving* this

rate until we achieve feasibility in the NoC. This simple step helps us prune the search space of regulation rates to only include feasible solutions. Since we keep halving the rate until feasibility is achieved, the time complexity is logarithmic in nature; taking only $|\log_2(K)|$ steps to find the first feasible rate K. A simple search, as used in [2], exhibits linear asymptotic time complexity in contrast; taking $(\frac{1-K}{G})$ steps, with G $\in$ (0,1) being the resolution of the search increment. We use this first *feasible* rate to seed a `univariate` learning of regulation rates. We learn this `univariate` rate until convergence and use this to seed our final phase of `multivariate` learning. Although `univariate` learning offers much less fidelity and does not produce an optimal solution, it seeds `multivariate` learning's starting parameters such that expected coordinate-wise distance to the true optimum is minimised. Without this seeding, the CMA optimizer can exhibit numerical instability; with the underlying probability density function lacking any coherent structure pertinent to the problem and hence generating solutions on pure chance. Asymptotically, the optimizer can take infinite steps before converging to a valid solution. Hence, seeding results in a tighter (implying smaller standard deviation) multivariate gaussian distribution, taking fewer steps for `multivariate` learning to converge. We evaluate this in greater details in section 6.4.2.2 and Figure 6.9.

6.4 Evaluation

6.4.1 Methodology

For regulation rate learning, we implement the learning suite using the python based `cma` library. We parallelize computation of QoR for each sample candidate generated in an epoch by spawning individual `Process` objects using the `multiprocessing` library. Each spawned `Process` can either calculate the QoR directly for vanilla Hoplite configurations (no switch learning; section 6.4.2.2) or learn switch configurations as part of the internal loop of Algorithm 1 (nested learning; 6.4.2.1). For learning switch configurations as part of nested learning, we use the same setup that is used in section 5.6.1.

Similar to section 5.6.1, we evaluate our entire framework across a range of 100 synthetically generated communication workloads with `RANDOM` (uniform sampling of destination PE) and `LOCAL` (+/- 2 radius of neighboring PEs) communication patterns. We evaluate our design across various data block sizes to mimic the diversity of communication interfaces and endpoints like DRAM, PCIe, and Ethernet. Furthermore, similar to section 5.6.1, we also test for optimizing real-world applications designs for FPGA accelerators such as Sparse matrix vector multiplication (`SpMV` [47]) used by many deep learning

kernels [99] and `Graph` [48] analytics. This allows us to evaluate our designs on ability to route data that closely mimics interactions of end-points in a production setting.

6.4.2 Results

In this section, we present the results of the static analysis of our hybrid FPGA NoC under various system sizes, data block sizes, and optimization methods. We study the effects of:

- In-tandem learning of regulation rates and switch configurations (Section 6.4.2.1)
- Standalone learning of regulation rates with the switch configuration preemptively set to all HopliteBuf or HopliteBP (Section 6.4.2.2).

We are primarily interested in determining network feasibility (bounded latency), FIFO size needed, worst-case latency, fitness of the solution, and time required to optimize the NoC.

6.4.2.1 In-Tandem Learning Regulation Rates and Switch Configurations

In this section, we investigate the effectiveness of learning regulation rates while also learning switch configuration on a per-switch basis. We compare this *nested* learning against standalone switch learning of Chapter 5 ([2]). We contrast `multivariate` rate learning with `univariate` rate learning against a variety of RANDOM, LOCAL and REAL benchmarks. For both learning types, we use MLE as our algorithm of choice to learn switch level configurations.

Note that the previous Chapter 5 ([2]) separates out the regulation rate as a single value determined by the user. `Univariate` rate learning is an extension of linear searching of the previous Chapter 5 ([2]); in that the rate, while still being a single value that regulates all traffic, is now learnt as part of the optimization problem instead of being linearly searched over a range with some fixed increment. If this increment was made infinitesimally small and the rate point with the best QoR chosen, then linear search of Chapter 5 ([2]) and `univariate` would generate equivalent solutions.

It is also helpful to recall that since both rates, `univariate` and of Chapter 5 ([2]), are a single value that are universally applied to all regulators in the NoC, they do not offer the level of fidelity required to individually regulate NoC bandwidth to each traffic trace. `Multivariate` learning instead learns unique rates for each *src-dst* traffic trace pair. Over

a range of benchmarks, we aim to investigate if `multivariate` rate learning can implicitly identify potential congestion hotspots within the NoC and tweak regulation of offending individual traffic to outperform `univariate` learning by achieving lower routing latency while sustaining higher rates across a range of benchmark flowsets.

6.4.2.1.1 Routing RANDOM, LOCAL and REAL Benchmarks: We first compare `multivariate` and `univariate` rates by evaluating their performance on numerous RANDOM, LOCAL and REAL traffic benchmarks of varying combinations of data block and NoC sizes. We now list a few important differences between `multivariate` and `univariate` learning, along with their impact on performance of the final NoC design:

- As NoC sizes increase, `multivariate` is afforded an increased capacity to tune regulation rates due to increase in parameter space N^4. This larger parameter space allows it to work with MLE at a more granular level to optimise for the requested QoR. In contrast, `univariate's` parameter space remains the same. This implies that the effectiveness of `univariate` decreases with increase in NoC size; it is unable to individually respond to dramatically different routing conditions between numerous local regions of a bigger NoC. Thus, `multivariate's` increased tuning capability allows it to implicitly tag locally congested regions and throttle down contention within these local NoC zones by surgically regulating only the offending traces.
- Increasing data block sizes can quickly overwhelm the cheaper HopliteBuf (FIFO) switches throughout the NoC. That pressure can be relieved by MLE's switch discovery by selectively replacing these pinched HopliteBuf switches with HopliteBP switches. However, `univariate` unilaterally lower rates for all traces which in turn increases HopliteBuf's feasibility, thus not exposing these switch level decisions to the MLE algorithm. In contrast, `multivariate` and MLE are able to work in tandem to lower rates as a *last resort*; only where pressure cannot be relieved by $Buf \rightarrow BP$ swaps. Thus, the search space of MLE's switch level learning and `multivariate's` rate learning are well composed when they interact; with each being in lock-step with other, thus allowing for a more efficient optimization for the requested QoR.

We now compare the quality of solutions generated by `multivariate` and `univariate` learning for different RANDOM, LOCAL and REAL benchmarks. Rate learning and switch learning algorithms are configured to optimise the combined QoR of $wclatency{*}cost$:

- In Figure 6.3, we plot the efficacy of routing 10 different RANDOM traffic benchmarks each across NoC sizes of 4x4 and 5x5 while varying data block sizes between 4 and 8. We observe that `multivariate` outperform `univariate` by up to $\approx 1.5\times$.

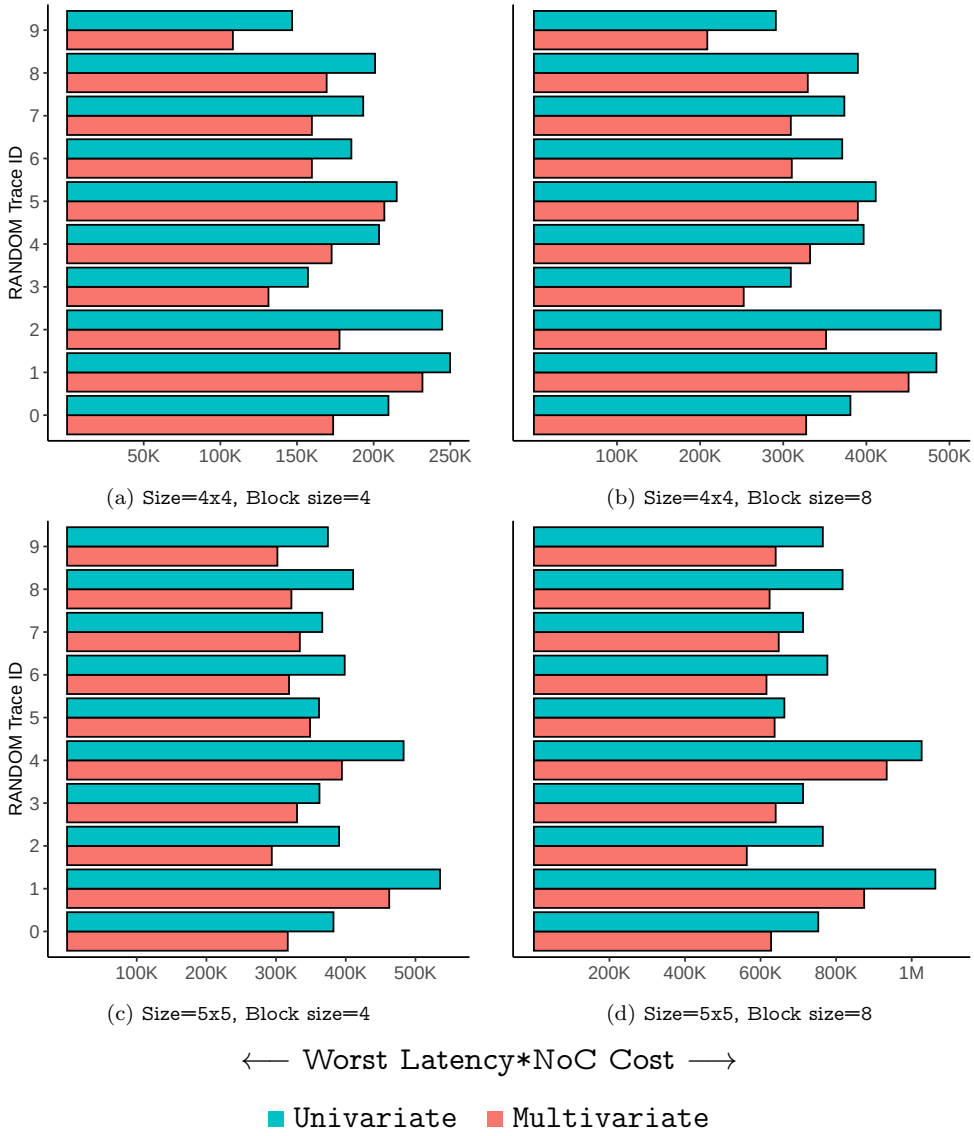


Figure 6.3: QoR achieved by univariate and multivariate rate learning strategies, compared for data block sizes of 4, 8 and NoC sizes of 4x4, 5x5 for 10 different RANDOM traces.

- In Figure 6.4, we plot the efficacy of routing 10 different LOCAL traffic benchmarks each across NoC sizes of 4x4 and 5x5 while varying data block sizes between 4 and 8. We observe that `multivariate` outperform `univariate` by up to ≈1.3×.

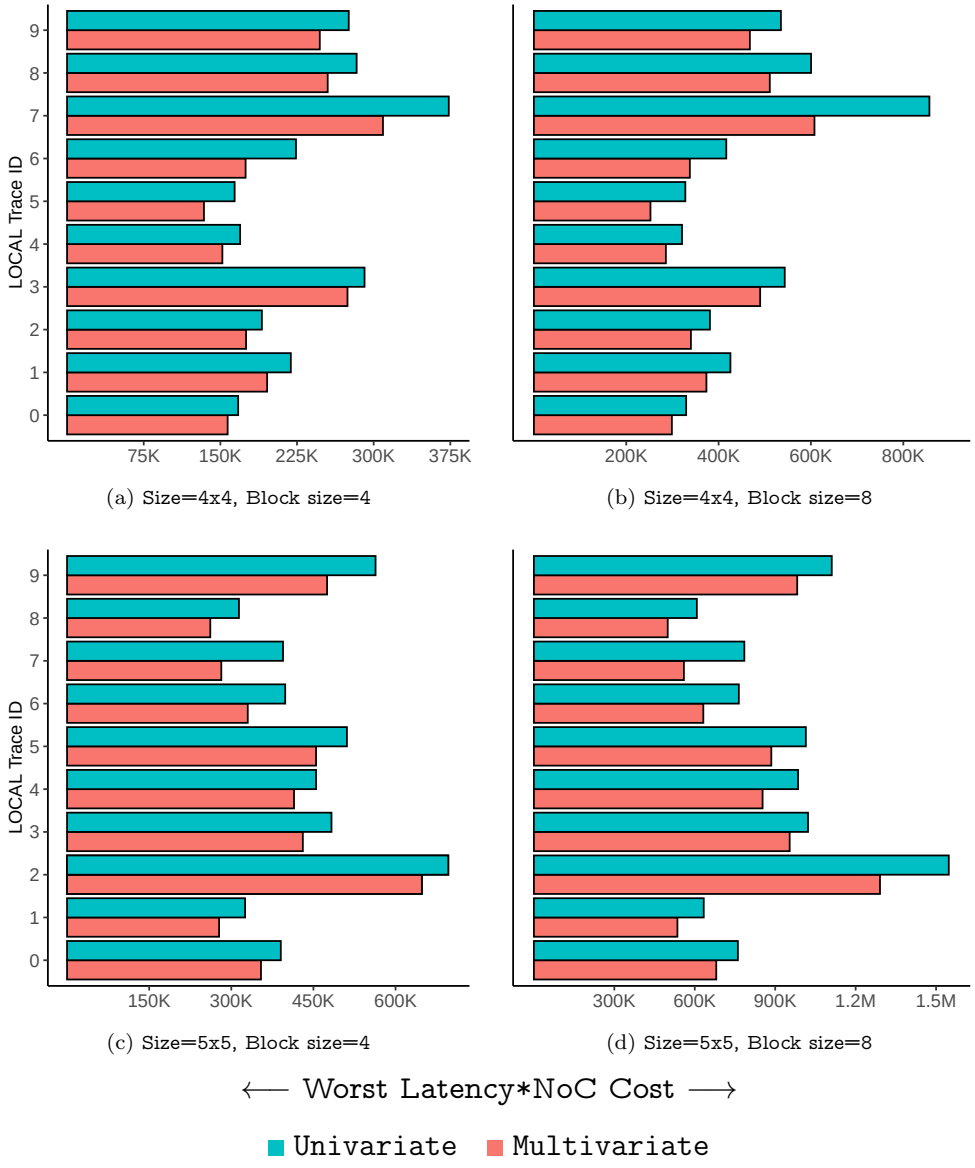
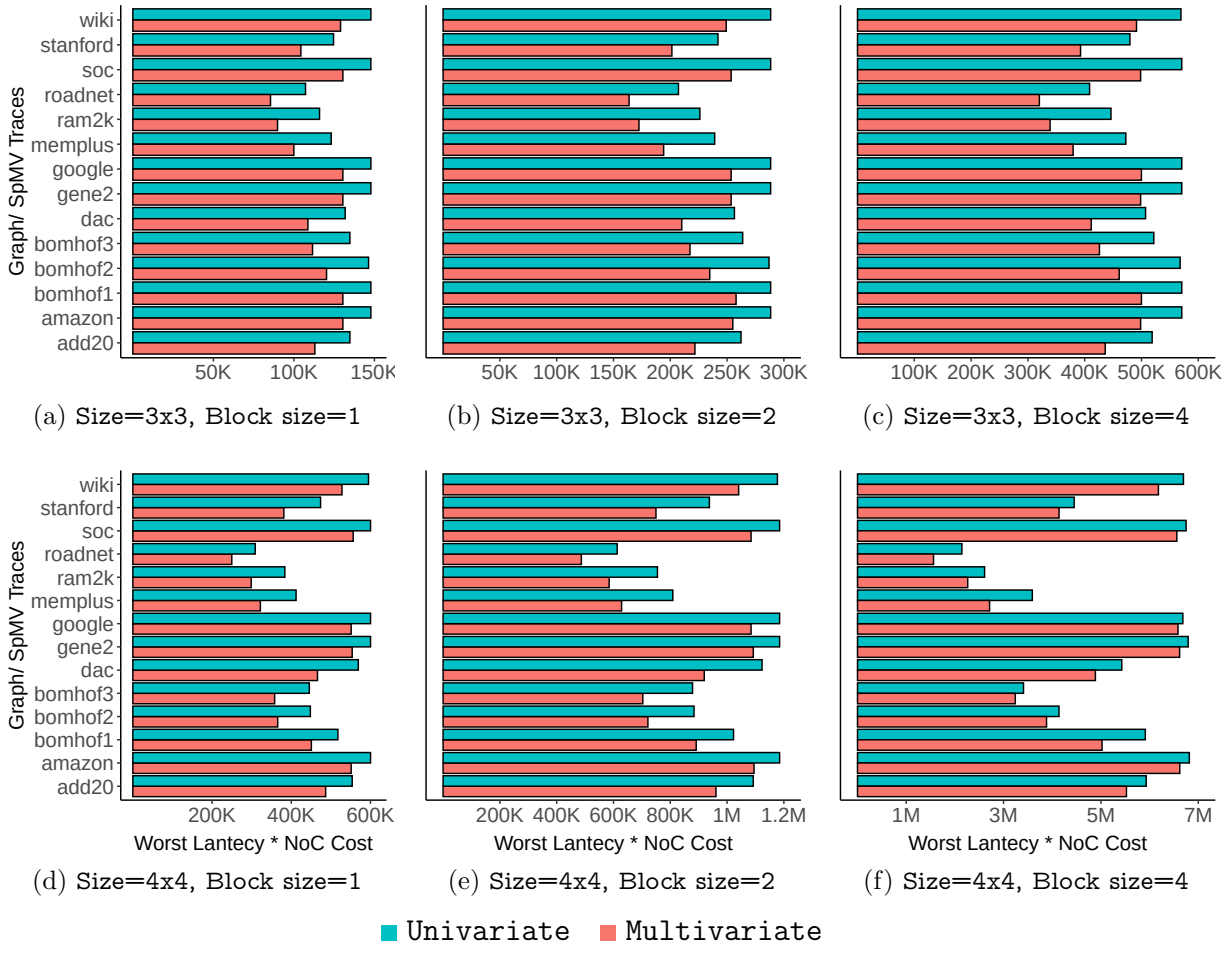


Figure 6.4: QoR achieved by univariate and multivariate rate learning strategies, compared for data block sizes of 4, 8 and NoC sizes of 4x4, 5x5 for 10 different LOCAL traces.

- In Figure 6.5, we plot the efficacy of routing 14 different REAL traffic benchmarks from Graph and SpMV suites in progressively difficult routing conditions; increasing NoC sizes to 4x4 and increasing data block sizes from 1 to 4. These REAL bench-

marks are characterised by very high load factors (% of active traces in benchmark out of N^4 maximum). As noted in section 5.6.2.4, this leads to traffic traces within the NoC to interact in a *spaghetti-like* effect. While linearly searching over regulation rates in sec 5.6.2.4, this affect was evident in low feasibility and exploding worst case latencies. Both `univariate` and `multivariate` are able to learn regulation rates to best mitigate this effect, achieving feasible combination of rates and switch configurations across all 14 benchmarks. Continuing the trend observed in RANDOM and LOCAL, `multivariate` outperforms `univariate` across the board by as much as ≈1.2×, owing to its better ability to detangle (by regulating) conflicting traces within the NoC.



(a) Size=3x3, Block size=1 (b) Size=3x3, Block size=2 (c) Size=3x3, Block size=4

(d) Size=4x4, Block size=1 (e) Size=4x4, Block size=2 (f) Size=4x4, Block size=4

Figure 6.5: QoR achieved by univariate and multivariate, compared for data block sizes of 1,2 and 4 (columns) and NoC sizes of 3 and 4 (rows) for REAL traces.

6.4.2.1.2 Analyzing Learnt Rates: With this experiment, we compare and contrast the breakdown of distribution of rates, specifically 1. if `multivariate` rates are able to able to achieve a wider spread of rates, implying traffic aware regulation and 2. if `multivariate` is able to achieve, on average, higher rates than `univariate`. We focus on `REAL` benchmarks for this experiment due to their particularly high susceptibility to regulation; owing to their high load factor induced *spaghetti-like* contention. We also choose real benchmarks for this experiment due to the high diversity of traffic behaviour within a benchmark. This allows us to evaluate the degree of acute homogeneity introduced by `univariate` learning and its worsening impact by achieving much lower rates and contrasting this with `multivariate's` ability to instead exploit this diversity for faster rates overall. For this, we plot, for a NoC of size 4x4, the range of learnt `multivariate` rates for all 14 `REAL` benchmarks while increasing the data block sizes from 1, 2 and finally to 4 in Figure 6.6. We also plot the average `multivariate` rates and compare it with `univariate`.

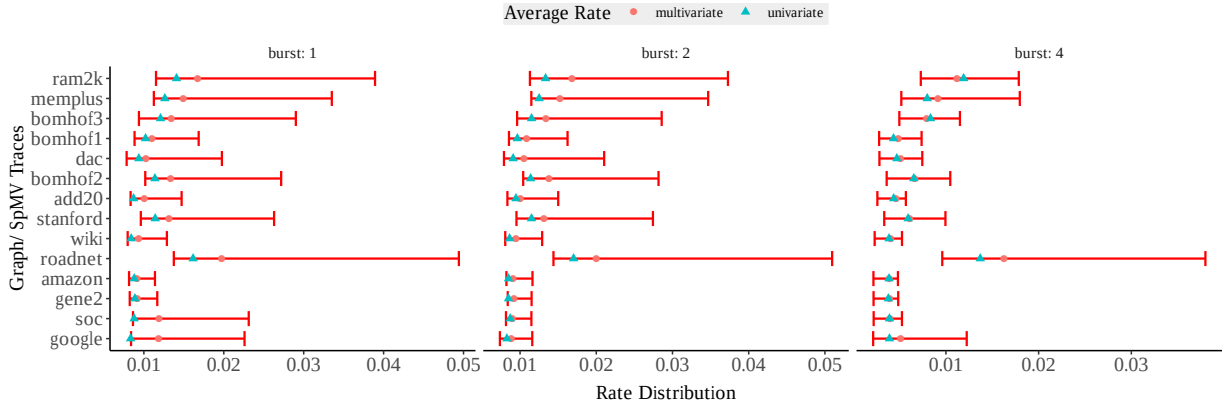


Figure 6.6: Range (as whisker plot) and average rates of learnt `multivariate` rates on a *per trace basis* along with `univariate` rates for `REAL` benchmarks for a NoC of size 4x4, over a range of data block sizes (`burst`).

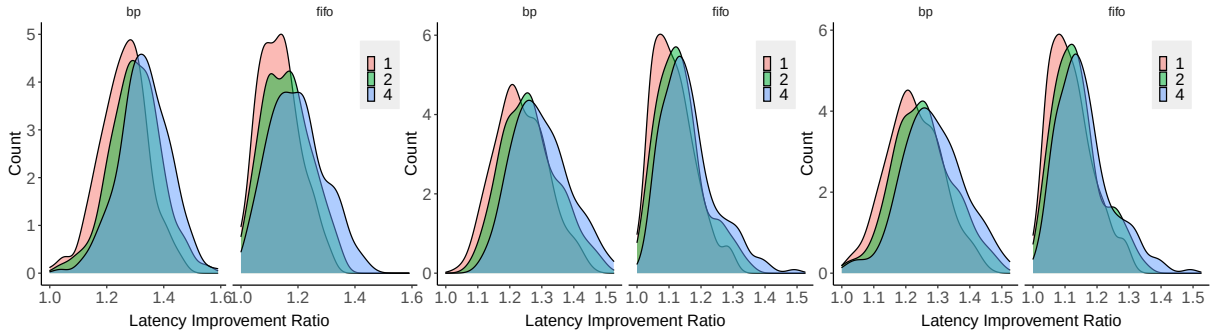
We observe that the mean `multivariate` rate is higher than the univariate rate for nearly every benchmark across every data block size; up to ≈1.4× higher. This suggests that `multivariate` learns to first lower the rates of most contentious traces in the benchmarks, thus allowing other traces to achieve ≈3.1× higher regulated injection rates, further evidence that turning down the "heat" in certain parts of the network helps other traces immensely. This is also evidenced, in benchmarks like `dac`, `roadnet` etc, by the observation that the *minimum* of the `multivariate` rates is actually ≈1.8× lower than its `univariate` counterpart, despite its average being higher than `univariate`. Another implication of this is that

the "one size fits" all strategy of having each trace be regulated by the same amount leaves quite a bit of NoC bandwidth on the table, explained by the `univariate` NoC being pushed more easily into infeasibility because any "superhot" contention zone within the NoC affects the entire interconnect, thus forming a bottleneck for regulation. `Multivariate`, on the other hand, will simply lower regulation rate for traces involved in such contentions.

6.4.2.2 Learning Regulation Rates for Vanilla Hoplite NoCs

In this section, we analyze the effectiveness of `multivariate` over `univariate` learning when learning regulation rates for traces that need to be routed on a vanilla Hoplite configuration, that is, all switches are either HopliteBuf (fifo) or HopliteBP (bp). Vanilla HopliteBuf NoCs' FIFOs can quickly fill up as a result of increased contention induced buffering. HopliteBP NoCs suffer from pessimistic analysis, born out of long chain of backpressure signals that can throttle all links across a row. We investigate if `multivariate` and `univariate` learning can help alleviate some of these inherent weaknesses of vanilla NoCs, while also comparing both rate learning techniques against each other.

6.4.2.2.1 Analyzing Distribution of Latency Improvement: In Figure 6.7, we plot the latency improvement distribution of `multivariate` over `univariate` while routing 100 different `RANDOM` benchmarks (Figure 6.7a), 100 different `LOCAL` benchmarks (Figure 6.7b) and all 14 `REAL` benchmarks (Figure 6.7c) on 4x4 HopliteBuf (fifo) and HopliteBP (bp) NoCs while varying data block sizes between 1, 2 and 4.



(a) `RANDOM` traces (b) `LOCAL` traces (c) `REAL` traces

Figure 6.7: Latency improvements of `multivariate` learning over `univariate` learning, for vanilla HopliteBuf (`fifo`) and HopliteBP (`bp`) NoCs of size 4x4, routing `RANDOM`, `LOCAL` and `REAL` traces.

We observe that `multivariate` achieves better latency by as much as ≈1.6×. As data block size increases, we see bigger latency improvements (the distribution shifts to the right) across all 3 benchmarks and for both Hoplite designs. Recall that in the case of HopliteBuf, FIFOs are able to absorb packets over a larger time horizon for a low data block size of 1. Hence, in this case, we see `univariate` being quite competitive across all benchmarks, with improvement limited to ≈1.3×. But as data block size increases, FIFOs quickly run out of capacity to absorb packets, with `multivariate` continuing to improve over `univariate`. While not being as sensitive to data block size owing to lack of FIFOs in their switch design, HopliteBP also has `Multivariate` outperform `univariate` by ≈1.5×, helped by `multivariate's` more granular regulation resulting in fewer row spanning backpressure stalls.

6.4.2.2.2 Routing `REAL` Benchmarks: In this section, we dive deeper into routing of `REAL` benchmarks by vanilla HopliteBuf (fifo) and HopliteBP (bp), with regulation rates being learnt by `univariate` and `multivariate` learning. In Figure 6.8, we plot the worst latency for routing 14 different `REAL` benchmarks on vanilla HopliteBuf and HopliteBP NoCs on size 4x4, while varying data block sizes between 1, 2 and 4.

We observe that latency progressively increases for HopliteBP with increasing data block size, with `multivariate` tapping down latency by as much as ≈1.2× compared to `univariate`. We observe a similar scaling behaviour in the case of HopliteBuf with data block size of 1 and 2, with `multivariate` resulting in ≈1.3× lower latency. However, we observe that both `multivariate` and `univariate` perform very poorly when the data block size is increased to 4 for HopliteBuf; with only `ram2k` and `roadnet` resulting in feasible designs. All other benchmarks are not even feasible on a vanilla HopliteBuf, despite best efforts by regulation learning algorithms. We have previously discussed this artifact of FIFOs filling up quickly in the face of contention and increasing data block size in section 5.2. This affect is particularly amplified in the case of `REAL` benchmarks owing to their high load factor. This highlights the importance of combining regulation rate learning with switch level discovery. While `multivariate` is able to better manage contention at a traffic trace level by tuning regulation of offending traces involved in the contention, it is not as effective at mitigating contention induced in the NoC as an artifact of the underlying switch architecture. Thus learning switch configurations in tandem with regulation rates works best; as similar experiments of Figure 6.5 were able to discover feasible NoC designs and regulation rates for all 14 benchmarks in the `REAL` suite.

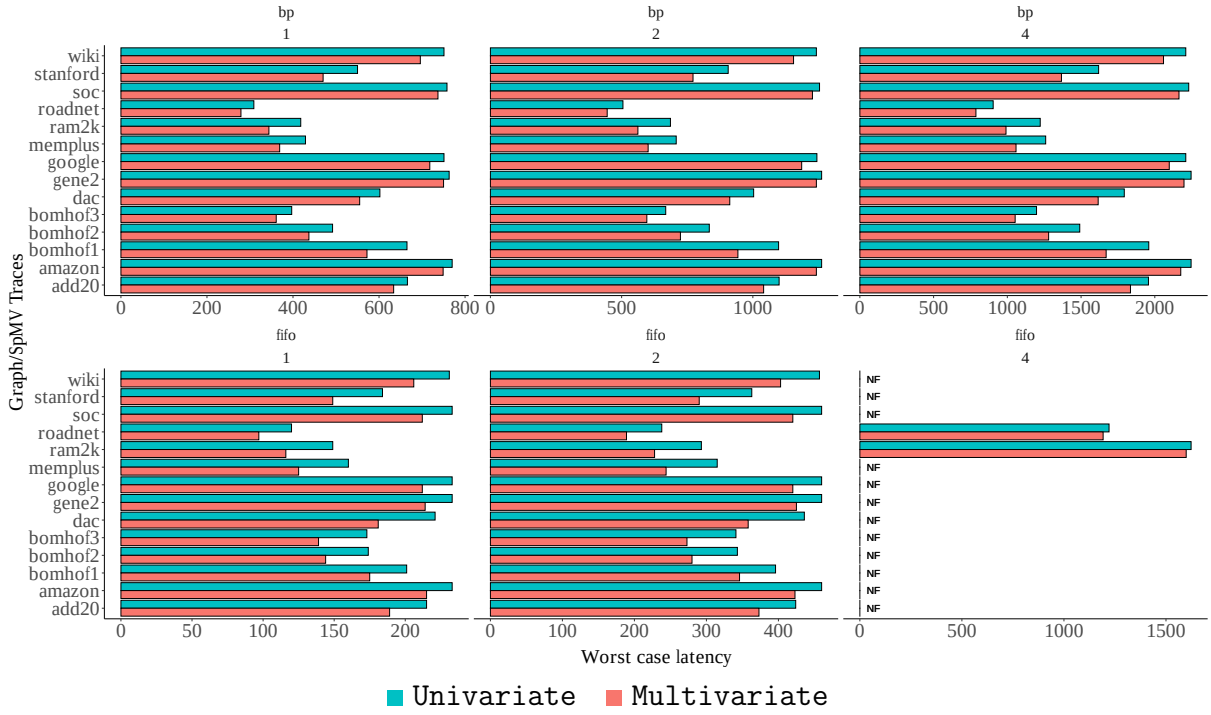


Figure 6.8: Univariate and multivariate rate learning compared for vanilla HopliteBuf and HopliteBP NoCs of size 4x4 while routing `REAL` traces of data block size of 1,2 and 4. "NF" tagged flowsets could not be routed feasibly for both learning types.

6.4.2.2.3 Understanding `Multivariate` Rate Learning: In Figure 6.9, we present how `multivariate` rates are learnt using Covariance Matrix Adaptation for a 4x4 HopliteBP NoC routing `roadnet` benchmark with a data block size of 1. As discussed in section 6.3.2, `multivariate` learning is split into three phases: 1. Rate Halving, 2. Univariate rate learning and finally 3. Multivariate rate learning.

The phases of Rate Halving and Univariate learning are motivated by reducing the search space for the `multivariate` phase by first finding a universal regulation rate (applied to regulate all traffic traces) that can achieve the best QoR. Rate halving comprises of starting from a universal rate of 1 (no regulation) and halving it every next iteration until the NoC can sustainably route the benchmark at hand. Figure 6.9a shows the rate being halved at every iteration until sustainability can be confirmed for all traces in the benchmark. For `roadnet`, this rate is $\frac{1}{2^6}$. At this stage, the rate is modelled as a univariate gaussian distribution with a mean μ (Figure 6.9b) and standard deviation σ (Figure 6.9e) which is then learnt over multiple iterations to optimise for the requested QoR. The

termination condition of the univariate phase is determined by the trajectory of best QoR over past iterations and the convergence of standard deviation (notice it reduce after every iteration to rest at ≈0.005).

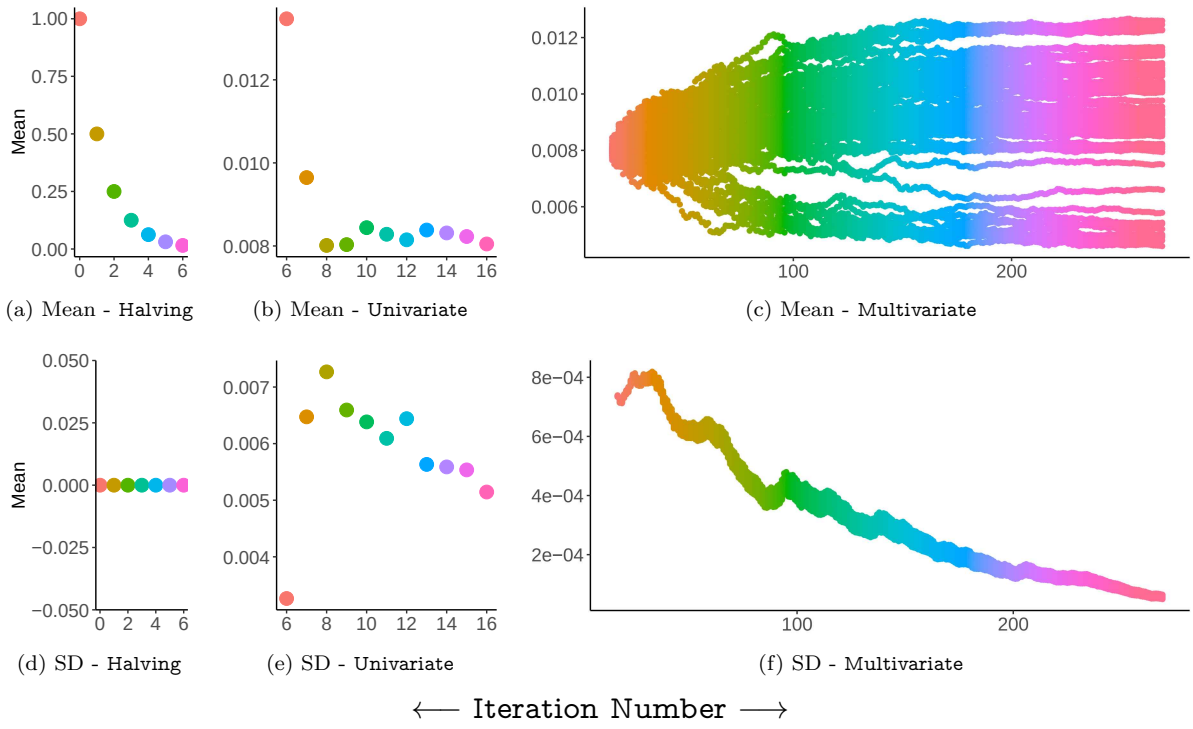


Figure 6.9: Chronological updates to mean and standard deviation (SD) of a `multivariate` gaussian distribution for a 4x4 HopliteBP NoC routing the `roadnet` benchmark with a data block size of 1. Note the 3 stages of learning: 1. Rate Halving, 2. `Univariate` and finally the dominant 3. `Multivariate`. Also note the 0 SD for rate halving.

Finally, a multivariate gaussian distribution is modelled to represent the rate of each trace individually, starting off with the same mean for very dimension, set equal to the best rate found at the termination of the univariate phase, as shown in Figure 6.9c. The standard deviation, in Figure 6.9f, for every dimension is initialised to the 10% of `multivariate's` initial rate . This gaussian distribution's parameters: mean and standard deviation for each dimension, are then learnt over the course of multiple iterations. Notice that the mean rates for every trace diverge to settle at values that best optimise the QoR, spread over the entire range, with some of the final rates being almost ≈3× faster than others.

`Multivariate` learning is terminated by analyzing the trajectory of best QoR over the iteration history as well as convergence of standard deviation of each rate, with standard deviations converging from $\approx 7 * 10^{-4}$ to $\approx 5 * 10^{-5}$.

6.5 Conclusions

In this work, we show how to learn regulation rates of each trace for an application that is being routed on hybrid or homogeneous FPGA NoCs. The regulation rates are learnt to deliver a combination of feasibility, worst-case latency, and cost improvements over statically configured univariate rates. We also present *nested learning* to learn NoC switch parameters and regulation rates together to maximise NoC routing QoRs. We model rate learning by casting each traffic trace's regulation rate as a random variable from a multivariate gaussian distribution. We using Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) to learn rates in-tandem with switch configuration and achieve achieve $\approx$1.5$\times$ lower cost constrained latency, $\approx$3.1$\times$ faster individual rates and $\approx$1.4$\times$ faster mean rates.

Chapter 7

GraphNoC: Graph Neural Networks for Accelerating NoC Evaluation

We can simplify design of FPGA NoCs by replacing slow and expensive conventional NoC benchmarking tools with accurate and fast Graph Neural Networks based models. FPGA configurability allows for tuning and designing of NoCs specific to the application being implemented on the FPGA. However, such application-specific NoC designs can require thousands of incremental updates and customization to the NoC design, with each NoC configuration needing performance evaluation to guide the design process. Each of these benchmark runs can take up to minutes with conventional tools like RTL simulation for modest packet trace lengths. As a result, tuning and design of a NoC even for a single FPGA application can last up to days, presenting a critical bottleneck to developer efficiency and iteration speed. We address this by presenting the GraphNoC framework to encode any FPGA NoC and any application traffic into graphs. We create a dataset of these graphs, comprising of different FPGA NoCs and applications. We use this dataset to train GNNs, including foundation models, to predict NoC routing latencies that can accelerate benchmarking run-times by up to 148× (506× using GPU) with prediction top-20 accuracies up to 97.2%. We also show these GNNs can accelerate end-to-end FPGA application-specific NoC design by up to 4.3× (37× using GPU) while regressing final NoC latency by only 30 cycles.

7.1 Introduction

FPGA NoCs provide a shared interconnect for communication between endpoints of the application being implemented on the programmable logic of the FPGA. Unlike ASIC NoCs, FPGA configurability can be leveraged to design and tune low-cost and highly performant NoCs specific to the application being implemented. However, design of these application-specific FPGA NoCs will be significantly more time consuming than traditional ASIC NoCs as we must choose various NoC parameters [16, 2] such as switches, buffer depths, regulator rates, to best match them for the FPGA application being implemented. Design and performance optimization of these NoC parameters can take days of simulation and exploration, with each exploration step requiring computationally expensive FPGA RTL simulations lasting tens of minutes [102] for even modest packet traces. In a quest to remove this bottleneck in the design of application-specific FPGA NoCs and super-charge the harnessing of FPGA reconfigurability, *quick* and *accurate* performance benchmarking of NoCs can play a key role by significantly reducing these expensive time and computational investments.

Current work around alternatives to RTL simulations for NoC performance prediction can be broadly grouped into 2 categories: 1) Analytical models and 2) Machine Learning models. Analytical models are tightly coupled to the specific FPGA NoC's architecture and cannot be easily applied to other NoCs. For example, the HopliteBuf analysis tool [41, 2, 16] only works for FIFO and backpressure based Hoplite routers. NoC designs like the Backpressure based Butterfly Fat Trees (BFTs) [3] and HopliteRT [39] provide a provable upper bound on worst case FPGA routing latencies. But these bounds are exclusively a function of the NoC architecture alone and not the application being routed. On the other hand, machine learning based models, enabled by the capacity of deep neural networks to learn non-linear vector mappings, can generalise better to a wide variety of NoC architecture and applications. However most of these models are based on conventional neural network layers like MLPs and Convolutions. Thus, they can only ingest highly structured euclidean data [103, 104], in turn limiting their application to only highly regular NoC topologies like meshes and tori.

To conclude, RTL simulations are accurate and generalisable but slow and expensive. Analytical models are faster but tied to a specific NoC design and do not usually take FPGA applications into account. Conventional machine learning based techniques are fastest but not generalisable to NoC architectures with irregular topologies. In this paper, we aim to democratize and accelerate design of application-specific FPGA NoCs by introducing a Graph Neural Network (GNN) based NoC benchmarking framework that combines the best of RTL simulations, analytical tools and conventional machine learning based models into

a single design: 1) Highly accurate NoC performance estimates, 2) Support wide variety of NoC architectures and topologies, and 3) Fast run-time. Our framework can encode any FPGA NoC (regardless of its topology or architecture) and any FPGA application's traffic into graphs and leverage GNN [105] based models to *accurately* and *quickly* predict the NoC's routing latencies for that particular FPGA application. The key contributions of our work include:

- Framework to encode any FPGA NoC (regardless of topology/architecture) and any FPGA application into graphs, compatible with both soft [3, 41] and hard NoCs [4].
- Creating a dataset to train GNN models, including multi-modal foundation models, to predict performance of FPGA NoCs for a given FPGA application. It consists of over 1.5 million samples of FPGA NoCs (HopliteBuf/BP, BFT) and their ground-truth performance on over 200 synthetic and real-world FPGA applications.
- Quantification of trained GNNs on FPGA NoCs: Up to **97.2% top-20 accuracy** for HopliteBuf/BP NoCs while being up to **≈148× (≈506× on GPU)** faster than RTL simulations for BFT NoCs. As a result of this quick and accurate NoC benchmarking, application-specific FPGA NoC design times are reduced by up to **≈4.3× (≈37× on GPU)**.
- Open-sourcing the entire work, including dataset and trained GNN models, enabling the FPGA community to leverage and build on top of our research for their own FPGA NoCs.

7.2 Background

Unlike ASICs, FPGA reconfigurability facilitates design and tuning of NoCs specific to the application. However, as discussed above, iterating on the NoC design comes at a cost; RTL simulations for performance analysis can be a bottleneck. In this section, we first review existing alternatives to RTL simulations. Based on this review (summarized in Table 7.1), we motivate the need of Graph Neural Networks for fast and accurate performance benchmarking of FPGA NoCs.

7.2.1 Alternatives to RTL Simulation

1. **Analytical Modeling**: They estimate NoC performance by building a mathematical representation of a specific NoC's architecture into its formulation. FPGA NoCs like Ho-

pliteRT [39] and Flow Control BFTs of [3] can guarantee an upper bound on the worst case routing latency as a function of the NoC architecture itself. However, these performance numbers are a function of the NoC architecture only and do not account for the application being routed. Analytical models of the HopliteBuf/BP suite of [2, 41] make use of network calculus to build application aware models to predict NoC performance, routability and size of statically allocated FIFO buffers. However, while the HopliteBuf/BP suite is application aware, it can only be applied to estimate performance of overlays built using the HopliteBuf/BP switches connected in a directed torus topology. Such analytical models are often hyper-sensitive to even minor perturbations to the NoC architecture, leading to poor generalization capabilities and preventing them from being used for any other FPGA NoC.

2. **Conventional Neural Networks**: Designed by stacking a large number of linear and non linear layers, they are trained using supervised learning. The training data is a tuple of: 1) Input FPGA NoC and FPGA application traffic, and 2) Ground truth performance numbers extracted from actual simulation. This bestows said models with powerful capacity to represent increasingly abstract concepts, foregoing the need for low-level NoC details to be encoded into the input; these details can be learnt as part of the training loop. We see commercial NoC IP vendors offering machine learning based models to predict performance of their NoC products [106]. [104] leverages Artificial Neural Networks (ANNs) to achieve a 1500–2000× speedup over cycle-accurate Booksim NoC simulator [107] with a predictor error of 5–8% for mesh and torus shaped NoCs. However, conventional neural networks, regardless of prediction capacity, exhibit high degree of rigidity in their structure. They can only ingest inputs with a specific shape and thus are incompatible with the rest. This limits only *highly regular* and *symmetric* NoC topologies (mesh, tori etc) to be compatible with an ANN model in a straightforward implementation. Other non-euclidean topologies, while technically possible, will require extensive and obtuse tensor manipulation for seamless composition into inputs. This introduces a strong inductive bias, which leads to poor generalization to different NoC topologies. We present evidence for this claim shortly in Section 7.2.3.

	NoC Archs.	Irregular Topologies	Time to Predict
RTL Simulation	**Adaptable**	**Compatible**	Tens of minutes
Analytical Simulation	Fixed	**Compatible**	Seconds to Minutes
Conventional NNs	**Adaptable**	Incompatible	**Sub-Seconds**
GNNs	**Adaptable**	**Compatible**	**Sub-Seconds**

Table 7.1: Comparing alternatives to RTL based simulations.

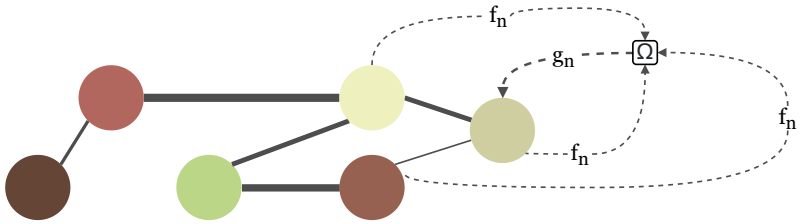


Figure 7.1: Foundation of Message Passing GNNs.

7.2.2 Motivating Graph Neural Networks for NoCs

To summarize our discussions in Table 7.1, RTL simulations are accurate but slow. Analytical tools are faster but tightly coupled to a specific NoC. Conventional NNs are quick and generalisable, but only for NoCs with regular topologies.

Graph Neural Networks (GNNs) work directly on graphs. A graph consists of node attributes, edge attributes and an adjacency matrix specifying graph connectivity. A graph neural network is composed of a series of differentiable functions and a message passing scheme [108]. While different types of graph neural networks [109, 75] apply different combinations of differentiable functions and message aggregation schemes, the general mechanics of a layer of graph neural networks (shown in Figure 7.1) are similar and can be described as:

$$h_i = g_n(\Omega_{j \in N_i}(f_n(h_i), f_n(h_j))) \tag{7.1}$$

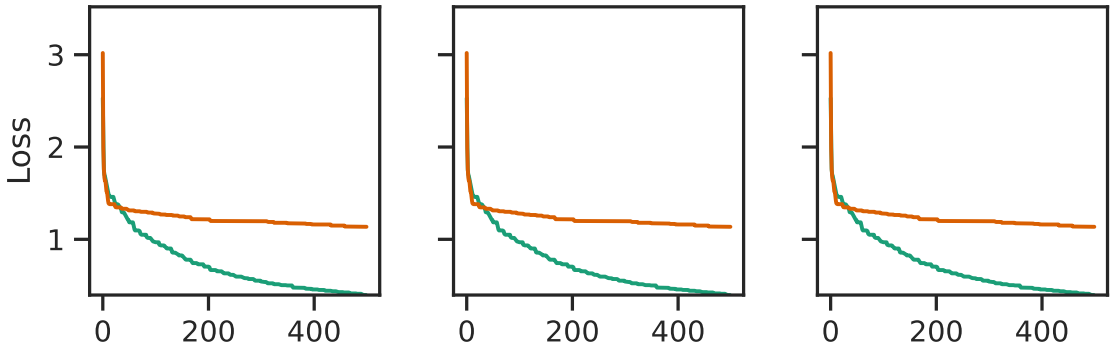
where h_i are the attributes of the node i. h_j are the attributes of the neighbor node $j \in N_i$. f_n and g_n are neural network operations. Ω is the message aggregation operator (like sum, max, min etc).

NoCs can be represented with high fidelity as graphs; by encoding NoC topologies into the adjacency matrix while NoC architecture (switches, flow control etc) can be encoded into the graph's nodes and edges. Since graphs are heterogeneous (different types of nodes and edges within same graph), the FPGA application can also be richly represented using a different set of node and edge attributes within the same graph. Finally, a Graph Neural Network based model (GNN) can ingest these graphs as inputs, irrespective of graph connectivity and size. These qualities make GNNs an ideal candidate to build and train models for NoC performance benchmarking.

7.2.3 Motivating Example: GNNs vs Conventional NN Models

We now motivate the use of GNNs over conventional neural networks by comparing their composability with non-euclidean NoC topologies. We setup an experiment to compare their ability to predict worst case routing latency for 2 FPGA NoCs with extremely different topologies: Torus shaped 4×4 HoptliteBuf/BP [2] and Tree shaped BFT0 [3] with 8 PEs. We construct 2 models with the same number of parameters: 1) MLP based conventional ANN with 9 layers and 1024 embeddings, and 2) 10 layer GraphSage [110] GNN (explained later in detail in section 7.3.2) with 256 embeddings. We train both networks to predict NoC performance for the same set of FPGA applications.

While encoding these NoCs as inputs, we encounter ANNs' incompatibility (discussed above in section 7.2.1) with irregular NoC topologies: an MLP based network can only ingest a vectorized tensor. As a result, we are forced to unroll and flatten both NoCs' switches and traffic traces into a 2D tensor of shape ($\text{num}_{\text{switches}} + \text{num}_{\text{traces}}$, inp_{size}). This results in significant loss of information; the NoC's specific topology is homogenised and hence not captured in the inputs. In contrast, the GNN model can ingest the NoC topology, encoded as graphs, without any loss of connectivity related information. Hence, GNNs can use message passing [108] (Figure 7.1, Equation 7.1) to leverage graph connectivity in extracting higher order features more effectively.



(a) Train (b) Validation (c) Test

Figure 7.2: Loss (**Y axis**) over epochs (**X axis**) for **ANN** vs **GNN**

We observe these effects reflected in the quality of training in Figure 7.2. Despite training both the models with the same training recipe (500,000 steps, 128 batch size, 0.1 dropout), we see that **ANN's** loss plateaus at 1.13 whereas **GNN's** loss continues to improve down to 0.39, a ≈2.9× improvement. Based on these observations, we conclude that

GNN based models are an attractive proposition to accurately predict routing performance in sub-second run-times for a wide variety of FPGA NoC architectures and topologies.

GNNs have not been leveraged to benchmark FPGA NoCs, with previous related works [102] very limited in scope: 1) Focusing on a single ASIC NoC only which, unlike FPGA NoCs, has fixed switch architecture, routing and flow-control, 2) Single-channel only application support, and 3) Not open-sourced. We now summarize a list of key features (and our objectives) of a GNN based NoC performance predictor:

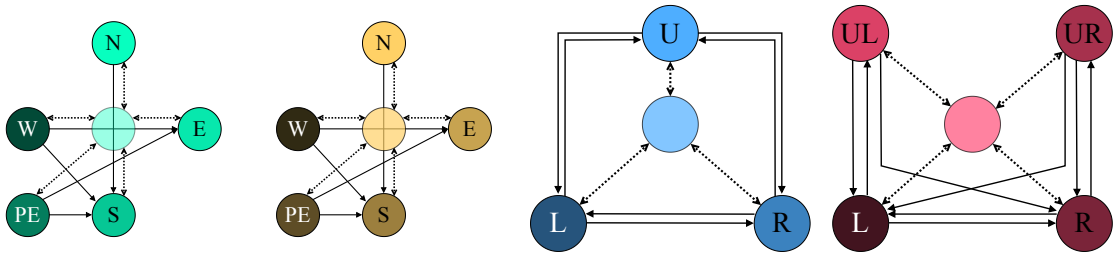
1. **NoC Diversity**: The graph encoding framework is compatible with any FPGA NoC architecture/topology, by encoding diverse NoC parameters (switches, flow-control, topology etc) into richly detailed graphs.
2. **Application Traffic Expressibility**: In addition to encoding NoCs into input graphs, any FPGA application can be encoded into a graph with rich detail.
3. **Fast and Accurate GNN Models**: The trained GNN models accurately predict NoC performance while significantly accelerating benchmarking run-time compared to existing techniques (RTL simulation, analytical models etc).
4. **Open to FPGA Community**: The framework, dataset and models are open-sourced for the community to leverage for their own use-case and make improving contributions.

7.3 The GNN based GraphNoC framework

In this section, for a given FPGA NoC and the traffic generated by the FPGA application's endpoints, we present the 2 key building blocks to using GNNs to predict NoC performance for that application: 1) Encoding the given NoC and application into graphs, and 2) Designing a GNN that ingests this graph and predicts NoC performance.

7.3.1 Encoding NoC and Application into Graphs

To use GNNs, we must first encode the given FPGA NoC and FPGA application into graphs. Our encoding framework is designed from the ground-up to support any NoC and application, independent of architecture and topology, and offers precise control over graph contents. We do this by leveraging the principles of heterogeneous graphs [111] in a *layered*, *hierarchical* fashion:



(a) HopliteBP (b) HopliteBuf (c) BFT-T (d) BFT-Pi

Figure 7.3: Hoplite [2] and BFT [3] switches encoded as graphs.

- *Layer 1*: Encoding NoC switches into graphs (Figure 7.3).
- *Layer 2*: Encoding NoC topology into a graph by using layer 1 as building blocks (Figure 7.4).
- *Layer 3*: Encoding application's traffic into graphs, and overlaying them onto layer 2's topology graph (Figure 7.4).

7.3.1.1 Layer 1: Encoding FPGA NoC Switches into Graphs

Note that an FPGA NoC can be composed of a mixture of switch types. For example, the BFT-1, BFT-2 topologies of [3] are composed of a mix of `pi` and `t` switches. Further, the mixing of different switches is sometimes central to designing highly performant NoCs specifically for the FPGA application [2]. Hence, for every switch (even of different types) of the NoC, capturing its fundamental properties (physical dimensions, arbitration, flow control etc) is essential.

A graph can be composed of nodes and edges of different types. To encode a NoC switch as a graph, we use 2 types of nodes. The `port` nodes capture the IO ports of the switch. Figure 7.3 shows a switch's ports (with text overlays) encoded as `port` nodes, with different shades of the same color capturing each port's different properties (numerically represented as tensors). The `central` node of a switch sub-graph is used to represent the architectural properties of the switch (flow-control etc), as shown in the translucent nodes at the center of each graph in Figure 7.3. The `port` nodes are connected to each other via the directed `route` edge (⟶) to capture the available routing data-paths between ports. Finally, the bidirectional `switch` edge (⟷) connects the `central` and `port` nodes.

7.3.1.2 Layer 2: Encoding NoC Topology into Graphs

We now present the rules around connecting the NoC switches (layer 1 graphs) to encode the NoC topology into a graph. We introduce a new directed edge type called `topology` that connects together the ports of different switches to reflect the NoC's topology. For example, in Figure 7.4, we connect the different `port` nodes of HopliteBP (Figure 7.3a) and HopliteBuf (Figure 7.3b) graphs with the `topology` edge (⟶) to encode the hybrid HopliteBuf/BP NoC topology. Additionally, the directed nature of this edge type implicitly captures the direction of data-flow between switches of the NoC. Finally, note that the graph's recursive (*i.e* topology graph made of switch sub-graphs) and heterogeneous (*i.e* graph made of different node and edge types) properties allows us to encode any NoC topology into a graph. This is key to GraphNoC supporting any FPGA NoCs, even hard FPGA NoCs like Versal [4], irrespective of topology and architecture.

7.3.1.3 Layer 3: Encoding Application Traffic into Graphs

We now present the methodology to encode an FPGA application's traffic into a graph, which is then overlayed onto the encoded NoC topology graph (layer 2, in turn built using layer 1). This final graph is then ingested as input by our GNN model.

An FPGA application's traffic is a collection of individual traffic traces between 2 NoC endpoints. We aim to encode any number of traffic traces between 2 endpoints. This allows for an accurate expression of application behaviour and better scalability; any new traffic traces can simply be added to the application graph without modification. Each traffic trace is represented as a `traffic` type node, with its numerical value capturing attributes like injection rate etc. A new directed edge type `application_traffic` connects the source NoC port (entry point of traffic) to the `traffic` node. Then, another `application_traffic` edge connects this `traffic` node to the destination NoC port (exit point of traffic). For example, in Figure 7.4, the PE ports of switch (1,0) send traffic to the PE ports of switch (1,1) using the `application` node and `application_traffic` edge (●,⟶). Having encoded the application traffic traces into graphs, they can be overlayed on the layer 2 NoC topology graph to complete a (NoC, application) pair encoding into a heterogeneous graph, as shown in Figure 7.4.

7.3.2 Message Passing Graph Neural Networks

Having encoded the FPGA NoC and the FPGA application's traffic into a single graph, we now present the GNN based model used to predict the performance of that NoC for that

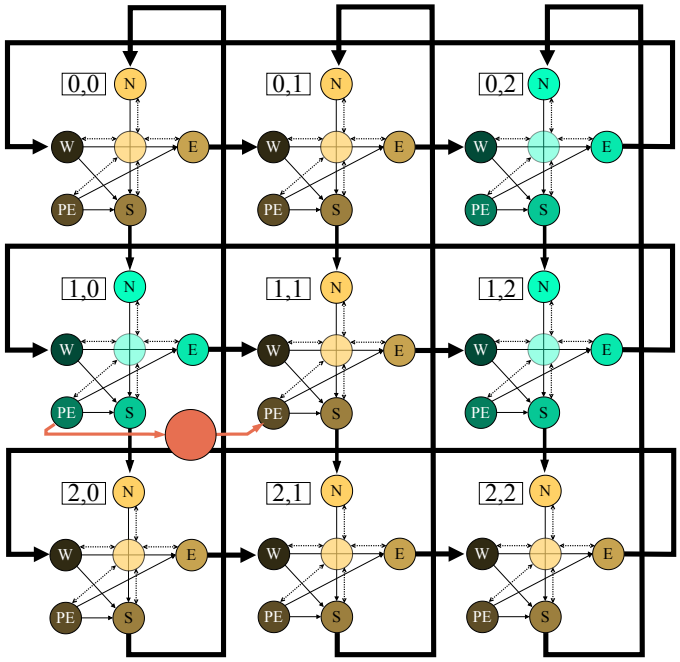


Figure 7.4: Encoding a 3×3 Hoplite NoC and application being routed. Notice the different edge types: port routes (——), switch (‹·····›), topology (⟶) and application (⟶).

application. As shown in Figure 7.5, our GNN based model is composed of 3 sequential stages of transformations:

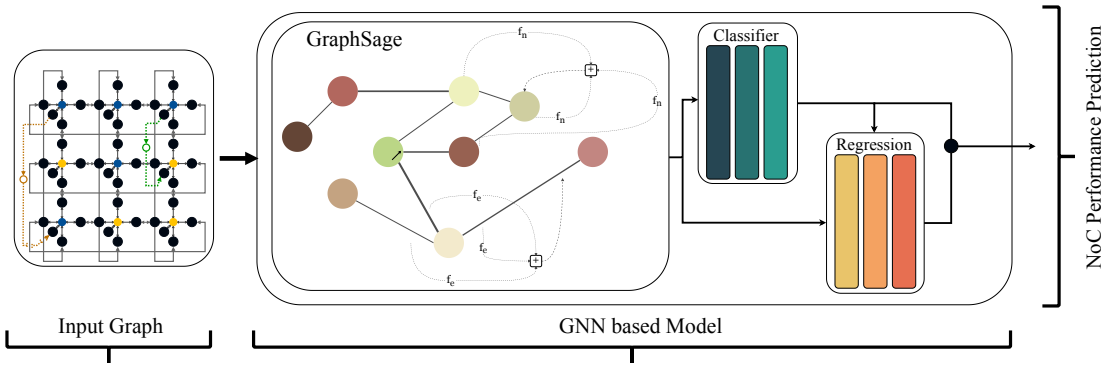


Figure 7.5: Design of GNN models: GraphSage, followed by residual application of classification and regression MLPs.

1. *GraphSage GNN Module* [110]: As discussed, our input graph is heterogeneous and is composed of 3 different node types (`port`, `central`, `traffic`) and edge types (`route`, `switch`, `application_traffic`). For each node x_i and its neighbor nodes x_j over an edge type, the GraphSage layer implements the in-place operation of Equation 7.2, where W_1, W_2 are the learnt weights. The GraphSage module is made up of a sequence of such layers, with the exact recipe detailed later in section 7.5.

$$x_i^{'} = W_1.x_i + W_2.mean.x_j, x_j \in \xi_{x_i}, x_i \in X \tag{7.2}$$

2. *Classification Module*: Our GNN model predicts performance as a function of classifier and regression (see Figure 7.5) modules as shown in Equation 7.3. The classifier predicts class c, whose cardinality C in Equation 7.4 is determined by the resolution of the upper limit on performance predictions. The classifier is composed of 3 `Linear` layers, with the first 2 gated with the `ReLU` non-linear activation and layer normalization. The classifier is trained with the `CrossEntropy` loss function.

3. *Regression Module*: The regression module predicts a value between [0, res) to complete the performance prediction of Equation 7.3. To increase it's predictive capacity, we add a residual connection [112] to ingest outputs from both GraphSage and classification modules. The regression module is composed of 4 `Linear` layers, with the first 3 gated with the `ReLU` non-linear activation and layer normalization. The regression module is trained with the `Huber` loss function.

$$perf_{app}^{NoC} = res * c + reg, c \in C \tag{7.3}$$

$$|C| = \frac{limit}{res} \tag{7.4}$$

7.4 Constructing the GraphNoC Dataset

FPGA NoCs and applications are encoded as graphs in section 7.3.1, which are then ingested as inputs by GNN models of section 7.3.2 to predict any NoC performance metric (latency, throughput, feasibility etc) for that application. We now lay the foundations for a community-driven dataset, consisting of a diverse set of encoded graphs of FPGA NoCs and applications, that can be used to train GNN models to quickly and accurately predict NoC routing performance.

Algorithm 2: Framework to add data to GraphNoC

```
Result: train set, val set, test set
NoCTool = RTL/Analytical.Sim;
for var_NoC in NoC.variations do
    for app in Application.collections do
        data = [];
        for rate in Rate.range do
            perf = NoCTool.calculate(var_NoC, app, rate);
            graph = encode(var_NoC, app, rate, perf);
            dataset.add(graph);
        shuffle(data);
        train.add(data[:train_split]);
        val.add(data[train_split:train_split+val_split]);
        test.add(data[train_split+val_split:]);
```

We now present the steps for adding encoded graphs to the dataset in a NoC and application agnostic manner in Algorithm 2. We first initialize the NoC tool to calculate performance. The tool can be RTL simulation or analytical models (NoC specific availability; section 7.2.1), depending on use-case. We loop over all the different variations of the baseline NoC (eg. mix of Buf/BP switches in HopliteBuf/BP [2]). For a particular NoC variation var_{NoC}, we consider a range of synthetic and real-world [47, 48] FPGA applications. For all combinations of NoC (var_{NoC}) and application (*app*) pairs, we evaluate ground truth performance using the NoC tool over a range of injection rates and encode each pair as a graph. Finally, this set of graphs is split between train, validation and test sets.

We position the GraphNoC dataset as an open-sourced initiative that the FPGA community can leverage and contribute to. This will allow the FPGA community to train increasingly sophisticated GNN models over time, which will in-turn enable them to explore design space of their own FPGA NoCs even faster. To prove GraphNoC's flexibility with any FPGA NoC and application, we seed the dataset with 3 different FPGA NoCs, over 200 different FPGA applications and both RTL and analytical models as ground truth sources for latency prediction. See Table 7.2 for details.

	HopliteBuf/BP	BFT-TREE	BFT-XBAR
Applications	14 real-world, 2 synthetic	100 synthetic	100 synthetic
NoC Variations	500	BFT0	BFT3
Rate (incr=1)	[0.1, 25]	[1, 100]	[1, 100]
NoC Sizes	3x3, 4x4, 5x5, 6x6, 7x7	4, 8, 16, 32	4, 8, 16, 32
Ground Truth	Analytical Sim. [2]	RTL Sim.	RTL Sim.

Table 7.2: Seeding GraphNoC dataset with 3 different NoCs.

7.5 Experimental Setup and Training Recipes

We author 3 GNN models of varying complexity using the `PyTorch Geometric` [113] library: small (GNN_s; 1 million), medium (GNN_m; 2.9 million) and large (GNN_l; 7.7 million). We detail authoring and training recipes in Table 7.3. Each model is trained and run on consumer grade hardware with Nvidia 4060 TI GPU, Intel i5-13500 CPU, 32GB RAM. We use `EarlyStopping` [114] on GraphNoC's dataset to predict worst case routing latencies under 1000 cycles. GNN's tensor-based computation is broadcast across multiple dimensions, called batch size, allowing multiple NoC samples to be evaluated at once. To ensure fair speedup calculations, we also parallelize conventional RTL simulation and analytical modeling tools by wrapping each call in a `Process` module of python's `Multiprocessing` library.

Model	≈Num. Params.	Num. Layers	Emb. Size	Batch Size	Rate	Schedule	Dropout
GNN_s	1 Million	20	64	128	$5 * 10^{-7}$	Hold	0.2
GNN_m	2.9 Million	15	128	64	$5 * 10^{-7}$	Hold	0.2
GNN_l	7.7 Million	10	256	64	$5 * 10^{-7}$	Hold	0.2

Table 7.3: Authoring and training recipes of GNN models.

We design experiments to evaluate the GraphNoC encoding framework, dataset and trained GNN models for speed, accuracy and impact on accelerating the designing and tuning of application-specific FPGA NoCs. We also train a foundation GNN model to test

multi-modal abilities of GrapNoC to predict performance for completely different NoCs. See Figure 7.5 to refresh on the main learning task for a GNN based model, along with its inputs and outputs.

7.6 Evaluation

7.6.1 Measuring Accuracy of GraphNoC GNN Models

Worst case latency is critical in defining service level agreements and guarantying reliable operations of NoCs under all operating conditions. In this experiment, we measure GraphNoC GNNs' accuracy in predicting worst case routing latency for a given FPGA NoC and FPGA application. We setup this experiment by training GNNs to predict worst case latencies for different sizes (3×3 to 7×7) of hybrid HopliteBuf/BP [2] NoCs across a range of real and synthetic FPGA applications (Table 7.2). Hoplite's analytical models generate stricter bounds on latency than RTL simulations [41]. Hence, we measure the accuracy of GNN predictions against ground truth of the analytical models for the test subset of GraphNoC dataset. We study the accuracy through 2 different lenses:

	Train Set								Validation Set								Test Set							
	Top				Delta				Top				Delta				Top				Delta			
Type	20	15	10	5	20	15	10	5	20	15	10	5	20	15	10	5	20	15	10	5	20	15	10	5
GNN_s	90.2	86.6	79.6	61.9	80.7	77.7	72.5	60.2	90.8	87.6	81.2	63.6	85.2	82.7	77.9	66.0	89.6	85.6	78.3	60.5	76.8	73.6	68.2	55.7
GNN_m	95.5	93.1	88.2	73.6	86.4	83.9	79.5	68.4	96.2	94.6	91.2	80.6	90.6	89.0	86.0	77.0	95.6	93.4	89.0	77.2	84.0	81.4	77.3	67.0
GNN_l	97.0	95.4	91.9	81.3	88.8	86.6	83.0	73.9	97.8	96.5	94.0	86.5	92.5	91.1	88.6	82.2	97.2	95.4	92.0	83.0	86.5	84.2	80.4	72.2

Table 7.4: Measuring accuracy of differently sized GNNs against GraphNoC dataset for HopliteBuf/BP hybrid NoCs.

- *Precision* (`Delta-K`): Precision is a measure of *absolute* difference between the ground truth and GNN predictions, important for measuring accuracy for routing latencies with small magnitudes. We measure precision with the `Delta-k` metric, defined as the number of predictions within K absolute cycles of the ground truth. For example, for a ground truth latency of 80 cycles, the GNN prediction would have to lie in [75, 85] to be counted for `Delta-5` precision. We observe that GraphNoC's GNNs can achieve `Delta-5,10,15,20` as high as 72.2%, 80.4%, 84.2%, 86.5% for the test set in Table 7.4 (right most column). Additionally, as we increase GNN size from small to large, the increase in `Delta-20` is only 12.6% whereas the increase is much higher for `Delta-5` at 29.6%, suggesting that larger GNNs can better address high precision use-cases.

- *Scale* (`Top-K`): Scale is a measure of *relative* difference between ground truth and GNN predictions. Unlike precision, scale takes into account the order of magnitude of ground truth latencies. We measure scale with the `Top-K` metric, defined as the number of predictions within K% points of the ground truth. For example, for a ground truth latency of 300 cycles, the GNN predictions would have to lie in [285,315] to be counted for `Top-5`. As shown in Table 7.4, we note that trained GNN models can achieve `Top-5,10,15,20` as high as 83%, 92%, 95.4%, 97.2%. Like precision, as we move from small to large GNNs, scaling benefits from big increases for `Top-5` (37.1%) but not for `Top-20` (8%).

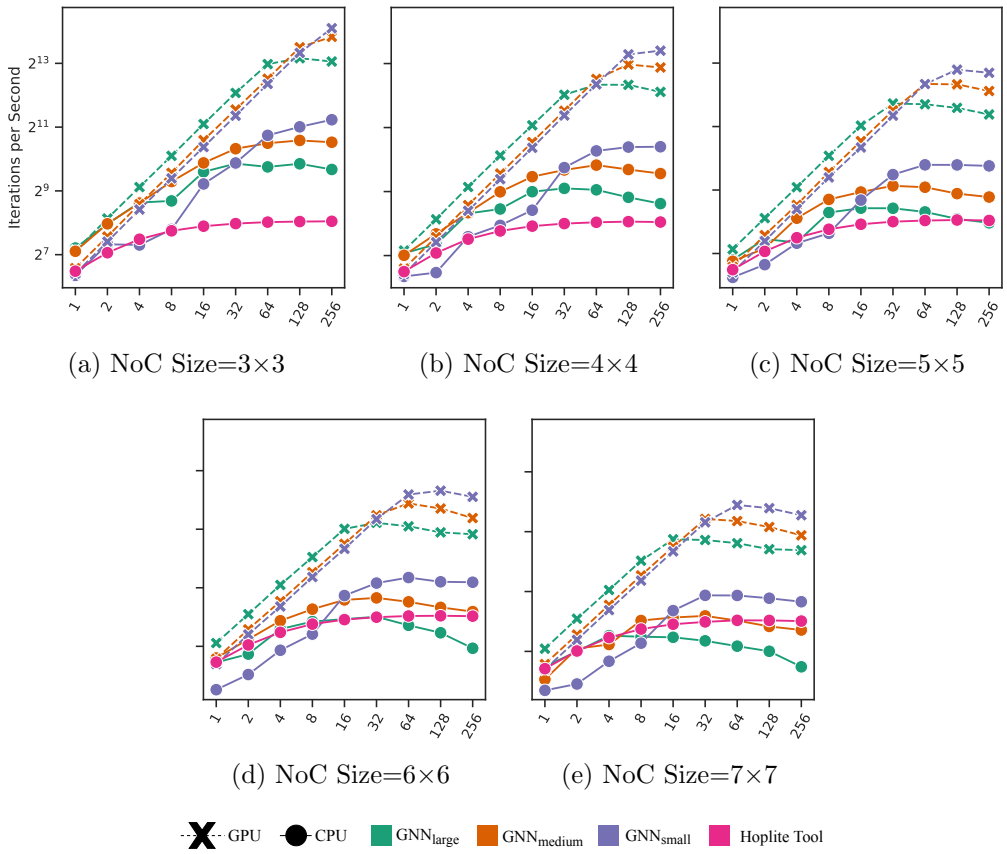
To summarize this experiment, GraphNoC GNN models can achieve highly accurate predictions for NoC performance. We observe high accuracies across different NoC architectures, NoC sizes and FPGA applications. These accurate predictions are maintained across different latency scales with high precision, achieving up to 97.2% `Top-20` and 86.5% `Delta-20` metrics. Thus, despite analytical models generating the strictest of upper bounds on latency, GNN can be used in their place for designing/testing FPGA NoCs. We further analyze distribution of GNN predictions and measure outlier deviations vs ground truth in experiment 7.6.4.

7.6.2 Benchmarking Throughput of GraphNoC GNN Models

In the previous experiment, we established that GNNs can accurately predict NoC routing latencies. In this experiment, we investigate GNNs' prediction speed; to establish if the minimal accuracy loss can be tolerated in exchange for significant speedups over using conventional RTL simulation or analytical models. Since Hoplite's analytical models are significantly faster than RTL simulations (Table 7.1), we stress test the potential of GraphNoC's GNNs to outperform the already fast analytical model instead. In this experiment, we measure maximum throughput (input samples per second) achieved by GNN models (on both CPU and GPU) and compare it to the analytical model. We vary batch size from 1 to 256 (refer to section 7.5 implementation details), for different HopliteBuf/BP variations and sizes in GraphNoC's dataset. We measure throughput for 2 different computation scenarios:

- *Constrained Compute*: One of the benefits of using GNNs is to accelerate NoC benchmarking and application-specific design for compute constrained organizations. Given a maximum batch size (a function of end-user's compute capabilities), we investigate the speedups GNNs offer over the analytical model in Figure 7.6. We observe that, given a batch size, GNNs can help increase throughput by up to ≈66× using

GPUs and ≈9× using CPUs. We also note that GNNs benefit from increased utilization of compute resources with larger batch sizes (x axis). For example, speedups over analytical model increase from ≈1× (batch size=1) to ≈14.5× (batch size=64) in Figure 7.6e. However, unilaterally increasing batch size hurts GNNs' throughput, with the affect more pronounced for larger GNNs or NoC sizes; a clear sign of saturation of available compute. For example, as we increase NoC size (Figures 7.6a → 7.6e), we observe that GNNs see a drop in their throughput beyond a certain batch size, with this affect being observed at increasingly smaller batch sizes for larger GNNs.



(a) NoC Size=3×3 (b) NoC Size=4×4 (c) NoC Size=5×5

(d) NoC Size=6×6 (e) NoC Size=7×7

Figure 7.6: Throughput (**Y axis**, *log*$_2$ scale) as a function of batch size (**X axis**) for GNNs and conventional tools for HopliteBuf/BP NoCs.

- *Unconstrained Compute*: We now analyze if GNNs also maintain their competitiveness when compute is not a bottleneck. To answer, we consolidate the results of

Figure 7.6 by selecting highest achieved throughput per NoC size, and present them in Table 7.5. We observe that GNNs of all sizes: small, medium and large outperform Hoplite's analytical modeling tool by ≈9×, ≈5.8× and ≈3.5× respectively when running on CPUs. When the GNNs make use of the GPU, this gap further increases to ≈66×, ≈55× and ≈34× respectively. We also note that the competitive edge of GNNs increases as we move from larger NoC sizes (7×7; ≈1.8× CPU, ≈14.5× GPU) to smaller NoC sizes (3×3; ≈9× CPU, ≈66× GPU), primarily explained by the analytical model's static setup time and flow-dependency bottlenecks overshadowing any size related throughput gains (hovering around ≈260).

To summarize this experiment, GraphNoC's GNNs can significantly accelerate FPGA NoCs' performance benchmarking for a given FPGA application. This speedup over the already fast analytical models is maintained for all GNNs over all NoC sizes. These speedups further increase by an order of magnitude when GNNs leverage the GPU. This is evidenced by GNNs outperforming conventional tools by up to ≈9× on CPUs and ≈66× on GPUs. The high accuracy (section 7.6.1), combined with these significant speedups, presents a potential opportunity for GraphNoC to significantly accelerate application-specific NoC design/tuning. We investigate this possibility in the next experiment.

	Iterations per Second				
Type	N=3x3	N=4x4	N=5x5	N=6x6	N=7x7
Hoplite Analytical Tool	264	266	266	263	262
$GNN_{s,CPU}$	2404	1351	887	652	468
$GNN_{m,CPU}$	1533	907	561	402	291
$GNN_{l,CPU}$	926	548	344	257	184
$GNN_{s,GPU}$	17585	10830	7093	5100	3801
$GNN_{m,GPU}$	14575	8026	5192	3773	2751
$GNN_{l,GPU}$	9157	5181	3388	2382	1717

Table 7.5: Best achieved throughput, across all batch sizes.

7.6.3 Application-Specific FPGA NoC's Design with GraphNoC

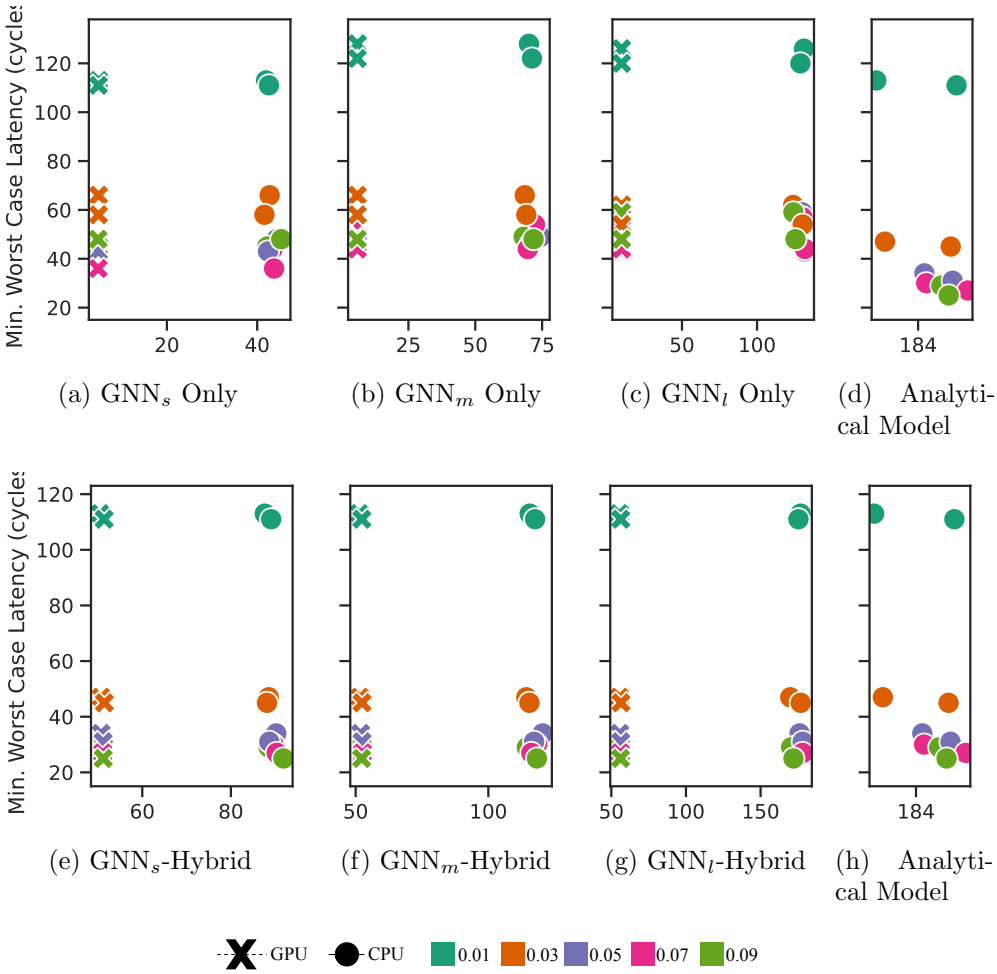
[2, 16]'s application-specific design of the FPGA HopliteBuf/BP NoC exposes the NoC's switches as a categorical choice between HopliteBuf or HopliteBP. For a given FPGA application, this application-specific design results in an optimized NoC with the minimum possible routing latency for this application. The design runs for 100s of epochs. At each epoch, 100s of HopliteBuf/BP's variations are generated and all of them are benchmarked for routing performance on the given FPGA application. At the end of each epoch, the best performing variants are used to seed the base configuration for the next epoch's variants. In the original work, Hoplite's conventional analytical models are used to benchmark all NoC variants. In our experiment, we swap analytical modeling with GraphNoC's trained GNN models. We investigate:

1. *Speed*: If GraphNoC's GNN can significantly accelerate the run-time of application-specific NoC design for FPGAs.
2. *NoC Quality*: If the slight inaccuracy of GNNs' latency predictions (compared to the analytical model; section 7.6.1), accumulating over multiple epochs, significantly deteriorates the final NoC's performance.

To setup this experiment, we target learning switch configuration for a 4×4 HopliteBuf/BP NoC for `random` and `local` FPGA applications' traffic. For each FPGA application, we vary the injection rates' from 0.01 to 0.09. We generate 128 NoC variants in each epoch and run the design process for 100 epochs. The batch size for both GNNs and analytical models is set to 128. We run GNNs on both CPU and GPU. Summarised in Figure 7.7, we investigate 2 different use cases and compare the final results to the original work:

- *GNNs Only*: Here, we calculate all the NoC variants' (128 variants per epoch, 100 epochs total) latency using GNNs only. At the end of the design process, we compare the resulting NoC design's performance (Figures 7.7a–7.7c) against the NoC design generated using original work's analytical model (Figure 7.7d). We observe, that GNN driven NoC design regresses latencies by up to 19, 24 and 30 cycles, while accelerating end-to-end runtime by up to ≈4.3×, ≈2.6× and ≈1.4× for small, medium and large GNNs on CPUs respectively. The speedup increases to ≈37×, ≈30× and ≈20× when GNNs use the GPU.
- *GNNs+Analytical Model*: Here, we invest the time saved by GNN driven runs into 25 additional epochs driven by analytical model (100 by GNNs, then 25 by analytical

model). We observe, in Figures 7.7e–7.7h, that this hybrid setting finds the most optimized NoC designs at all times. No latency regressions are observed when compared to NoC designs generated using original work's analytical model. While this hybrid approach is 2.1×, 1.64×, 1.34× slower than just using GNNs (10×, 8.5×, 6.1× on GPU), it is still 2.05×, 1.57×, 1.02× faster than analytical models for small, medium and large GNNs respectively (3.57×, 3.5×, 3.2× on GPU). As a result, even without GPUs, hybrid approach with small GNNs is a good middle ground that moderately accelerates application-specific NoC design without any performance regressions. On the flip side, for use-cases where iteration speed triumphs NoC quality, the GNN only approach might be more attractive.



(a) GNN$_s$ Only (b) GNN$_m$ Only (c) GNN$_l$ Only (d) Analytical Model

(e) GNN$_s$-Hybrid (f) GNN$_m$-Hybrid (g) GNN$_l$-Hybrid (h) Analytical Model

Figure 7.7: Learnt NoC's latency (**Y axis**) vs seconds spent (**X axis**) completing MLE loop for HopliteBuf/BP NoC.

To summarise this experiment, GraphNoC's GNNs can significantly accelerate application-specific NoC designs' runtime without sacrificing performance of the resulting NoCs. We observe that using GNNs accelerates NoC design's runtime by up to ≈4.3× on CPUs (increasing to ≈37× with GPUs) while only regressing up to 30 cycles in routing latency. Latency regressions can be eliminated completely with the hybrid approach while maintaining competitive speedups. Thus, GraphNoC GNNs' can supercharge the leveraging of FPGA reconfigurability by streamlining the design of highly performant NoCs specific to the FPGA application.

7.6.4 Building Foundation GNN Models with GraphNoC

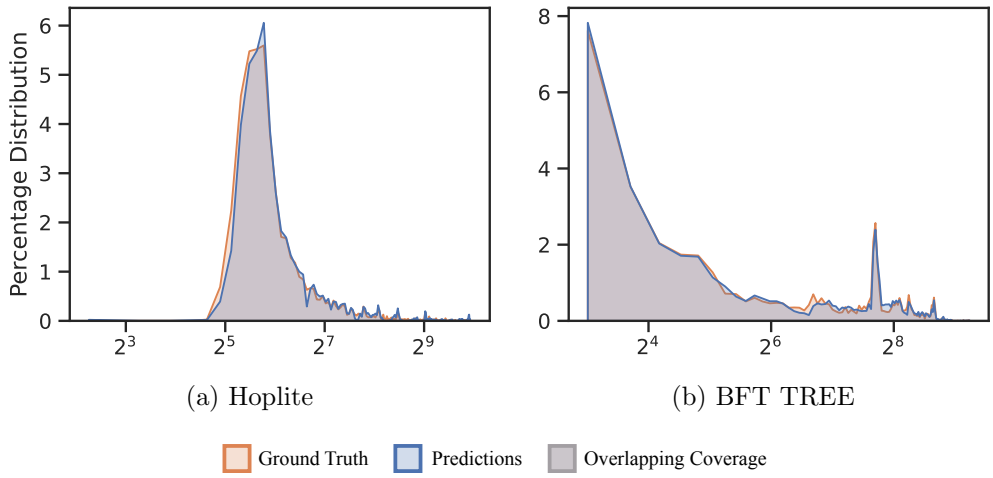
In this final experiment, we investigate a GNN based model's multi-modal capabilities. We do this by training a GNN to predict worst case latency for both HopliteBuf/BP [2] and TREE BFT [3] NoCs. Note that these 2 NoCs have completely different topology, data routing and arbitration rules. We train a 10-layer, 256 dimension GraphSage+Classifier+Regression GNN model. The model is trained on encoded graphs of FPGA NoCs and FPGA applications, distributed equally between TREE BFT (8 PEs) and HopliteBuf/BP (16 PEs; 4×4). We use a batch size of 128 and cyclic learning rate schedule varying between $[10^{-7}, 5 * 10^{-6}]$.

We now share results of this model's performance on the test set. The foundation model is able to achieve a `Delta-5,10,15,20` score of 75.7%, 81.0%, 83.9%, 85.8% and a `Top-5,10,15,20` score of 69.0%, 77.5%, 82.0%, 85.3%. We now plot the distribution of ground truth and predicted latencies, faceted by NoC type, in Figure 7.8. Different topologies, data routing and arbitration rules are evidenced by starkly different distribution of ground truth latencies between Figures 7.8a and 7.8b. Despite this, we note that the model is able to maintain high accuracy across the distribution for both NoCs in Figure 7.8, even robustly covering the abnormal spike of distribution around the 2^8 mark for the BFT in Figure 7.8b. In addition to GNN based models' capabilities to generalise across different NoCs, this is also evidence that GraphNoC's graph encoding framework is able to encode important features of completely different NoCs and applications into richly detailed graphs for maintaining model accuracy.

We now also discuss the speed improvements of this GNN foundation model against the BFT TREE's RTL simulation (Hoplite discussed previously in section 7.6.2). Unlike HopliteBuf/BP, BFT has no analytical modeling tool to calculate latencies, and is thus relegated to using RTL simulations. Despite 20 `Process` wrapped parallel RTL simulations (maximum that can be launched without OOM issues), it is only able to achieve a throughput of ≈4.3 iterations per second. Contrasting this with GNN foundation model's maximum throughput of 640 iterations per second on CPU (increasing to 2176 on GPU) at batch size 128, we conclude that BFT TREE's latency estimation can be accelerated by up to ≈148× on CPU (increasing to ≈506× on GPU).

Based on our evaluation, we can conclude that GraphNoC's foundation GNN models can continue to become more advantageous over time. The foundation model already achieves high accuracy (`Delta-20` 85.8%; `Top-20` 85.3%) and significant speedup (up to ≈506× on GPU, ≈148× on CPU) for 2 completely different FPGA NoCs. As the foundation GNN models are trained on more diverse FPGA NoCs and applications data, the models will build a richer representation of underlying NoC routing fundamentals, simi-

lar to language modelling for LLMs. This can in-turn allow new NoCs to be on-boarded quicker onto GNNs, paving the way for foundation GNN models potentially replacing RTL simulations and analytical models completely in the future.



(a) Hoplite (b) BFT TREE

Figure 7.8: Distribution of ground truth vs latency predictions.

7.7 Conclusion

In this paper, we present GraphNoC, a GNN based framework and dataset, to accelerate NoC benchmarking and application-specific NoC design. GraphNoC can encode any FPGA NoC, including hard NoCs, and any FPGA application into graphs with high fidelity. We encode 3 different FPGA NoCs, over 200 different FPGA applications to create a dataset of 1.5M+ graphs that is used to train GNNs, accelerating NoC benchmarking and application-specific design by up to 148× (506× using GPU) with prediction accuracies up to 97.2%.

Chapter 8

Conclusion

This work presents multiple contributions related to learning design parameters for building highly performant FPGA NoCs customized for the given FPGA application. We also present a Graph Neural Network based framework to accelerate benchmarking of NoCs, a critical bottleneck in the design of application customized NoCs. We close with a summary of our contributions in this field, followed by a discussion around future ideas that can be explored in this area.

8.1 Summary of Contributions

- **Chapter 4: BFT Overlay NoCs on FPGAs**

  First, we showed how to we design a low-cost, FPGA-friendly Butterfly Fat Tree (BFT) NoC with customizable bisection bandwidth and topology to match an application's requirements. We achieve this by choosing a combination of pi and and t switches at various levels of the NoC. Further, we improve the routing mechanism by introducing a backpressure based interface to deliver the application's data packets in-order and presenting a guaranteed upper bound on the application's routing latency.

  They key contributions of this work include:

  – Design of BFT NoC routers with latency-insensitive backpressure interface with deterministic uphill and downhill routing functions and fair round-robin arbitration.

- RTL implementation with LUT-fracturing friendly mapping, and NoC assembly with configurable bisection bandwidth tuning and optional pipelining support.
- Improved sustained throughput, average latency, worst-case latency, power reduction over its deflection based counterpart NoC.

- **Chapter 5: Customizing Switch Types using MLE**

  In this work, we demonstrate customizing the HopliteBuf/BP NoC by learning the effect of interactions between conflicting traffic flows of a given FPGA application. In a quest to maximise NoC routing performance for the given application, we learn the NoC's switch configuration between HopliteBuf and HopliteBP using a fine-grained per-switch static configuration model. We develop a Maximum Likelihood Estimation based strategy to discover high-quality NoC designs for the given FPGA application.

  The key contributions of this chapter include:

  - Design of a hybrid FPGA NoC architecture that combines HopliteBuf and HopliteBP switches to exploit the best of both worlds using Maximum Likelihood Estimation (MLE) based learning.
  - For the FPGA based soft NoC version, we demonstrate ≈1–6.8× reduction in worst-case latency and ≈2-3× improved feasibility by learning switch level configuration.
  - We also perform quantification of performance and area overheads of an ASIC implementation of HopliteBuf+BP NoC whose switches can be individually programmed to operate in either mode. We measure that our ASIC implementation is 3–3.7× smaller while reducing average latencies by 1.3–12×.
  - Showing extensibility of our MLE framework to learn uphill routing connections for the crossbar topology of a Butterfly Fat Tree (BFT) NoC from chapter 4 on a *per-switch basis*. We achieve ≈1–1.7× reduction in latencies with the MLE optimised BFT NoC while preserving the desired properties of the original NoC (in-order packet deliveries, livelock freedom and bounded latencies).

- **Chapter 6: Learning Injection Rates and Switch Configuration**

  In this work, we extend the learnable design space from Chapter 5 to also include regulation rates on a per traffic trace basis. This work can be applied to maximise application performance for a hybrid HopliteBuf/BP NoC from Chapter 5 or a homogeneous Hoplite NoC. We also present *nested learning* to learn NoC switch parameters and regulation rates together to maximise NoC routing QoRs. We cast

each traffic trace's regulation rate as a random variable from a multivariate gaussian distribution and leverage CMA-ES to learn these rates, either in isolation for a given NoC's switch configuration or learn both switch configuration and rates' distribution together.

The key contributions of this chapter include:

- Development of a CMA-ES based learning strategy that cases regulation rates as a random variable, sampled from a continuous multivariate gaussian distribution. The gaussian distribution's parameters (mean and co-variance) are learnt to maximise performance for the given FPGA application.
- Development of a nested learning strategy that allows learning of switch level configuration of HopliteBuf/BP NoC from Chapter 5 in-tandem with regulation rates for each traffic trace in the given application.
- We quantify improvements in application performance with rate and *nested learning*. *Nested learning* further lowers cost constrained latency by ≈1.5×, increases individual rates by ≈3.1× and average rates by ≈1.4×.

- **Chapter 7: GraphNoC: Graph Neural Networks for NoC Evaluation**

  Traditional NoC benchmarking methods like RTL or analytical simulation can become a bottleneck in application customized NoC designs since hundreds of NoC designs must be benchmarked for application performance over thousands of iteration rounds, with each benchmark run lasting for up to minutes in runtime. In this work, we leverage a graph based encoding framework and train Graph Neural Network to predict routing performance for the given FPGA NoC and application.

  The key contributions of our work include:

  - A graph-based encoding framework to encode any combination of FPGA NoC and FPGA application into graphs.
  - Curating a dataset of over 1.5 million samples of FPGA NoCs (HopliteBuf/BP, BFT) and their ground-truth performance on over 200 synthetic and real-world FPGA applications
  - Using the above dataset to train powerful and fast GNN models to predict performance of FPGA NoCs for a given FPGA application.
  - The trained GNNs achieve up to **97.2% top-20 accuracy** for HopliteBuf/BP NoCs while being up to **≈148× (≈506× on GPU)** faster than RTL simulations for BFT NoCs.

– Quantification of improvements in application-specific FPGA NoC designs as a result of GNNs' quick and accurate NoC benchmarking: design times are reduced by up to **≈4.3× (≈37× on GPU)**.

8.2 Future Work

- **Switch Level Topology Construction for Butterfly Fat Trees NoCs**

  In Chapter 4 ([3]), we presented a Butterfly Fat Tree topology based NoC that allowed matching an application's routing bandwidth requirements to the best out of 4 different topologies, as shown in Figure 2.2, ranging from minimal bandwidth of a TREE (BFT0, made up entirely of `t` switches) to the maximum bandwidth of a XBAR (BFT3, made up entirely of `pi` switches). The 2 other remaining topologies presented topologies constructed using alternate levels of `pi` and `t` switches in varying cadence.

  However, there is an opportunity to create more fine-grained topologies that maximise application performance while minimising NoC costs. For example, an application might require more bandwidth than what the BFT2 can manage but not enough bandwidth to warrant a full XBAR. The original work of Chapter 4 will force the user to choose XBAR in such cases, resulting in increased FPGA implementation costs, tighter placement and worse timing closure scenarios. Instead, we can leverage the main principals of Chapter 5 ([2]) to instead expose the choice between `t` and `pi` for every switch as a learnable parameter and leverage algorithms similar to Maximum Likelihood Estimation to learn the best switch configuration for the given FPGA application whose traffic the BFT NoC would be routing.

  A successful implementation of this idea would require solving the following challenges:

  – Encoding incompatible topologies into the cost function of our learning algorithm. Specifically, given `t` and `pi` switches have different input and output arity, they cannot be seamlessly connected with each other like the HopliteBuf and HopliteBP switches of Chapter 5 ([2]) and Chapter 6 ([16]).

  – Ensuring data packets always have a path to the destination PE. The uphill and downstream data paths of the original work of Chapter 4 ([3]) have been carefully connected to ensure that any combination of source and destination PE for a data packet has a viable routing path. This constraint might not be automatically satisfied in an ad-hoc mixing of `t` and `pi` switches.

- **Maximising Application Performance on Next Generation of FPGAs with Hard NoCs**

  Our focus on this work has been on designing and customizing *soft* NoCs to maximise performance for a given FPGA application. However, one of the key limitations of soft NoCs is that they are implemented using the FPGA's programmable logic which is also shared to implement the FPGA application itself. In the cases where the FPGA application itself is quite complex, this leaves very little room on the programmable logic to implement the soft NoC. This can result in scenarios where NoC complexity must be significantly toned down to leave enough programmable logic for the NoC's implementation, resulting in severe degradation of application routing performance. This can also have secondary affects like making timing closure on an over-populated programmable logic quite challenging.

  As FPGA applications have grown in complexity, we have seen an emerging trend of FPGA vendors providing a dedicated hardened FPGA NoC [4]. Such FPGA NoCs, much like other hard macro blocks like HBM, are implemented outside of the programmable logic as shown in Figure 8.1. Like ASIC components, their implementation logic cannot be changed but they expose certain configuration parameters that can be programmed to best suite the FPGA application.

  As opposed to user driven or compiler based decisions to program the hard NoCs, we can instead leverage the various learning techniques presented throughout the thesis to maximise routing performance for the given FPGA application. We now list some parameters that can be investigated for their potential to be cast as a learnable variable:

  - Regulation rate. Each PE connection's regulation rate can be learnt in a quest to optimize various performance apsects like queuing latency, worst case routing latency, minimum throughput etc.
  - PE and NoC connections. Depending on complexity and size of application, an application's PE can be connected to multiple ingestion entry points in the NoC. The choice of the ingestion point can influence the timing and amount of congestion on the NoC's interconnect. Learning entry points for each application PE can help maximise application performance.

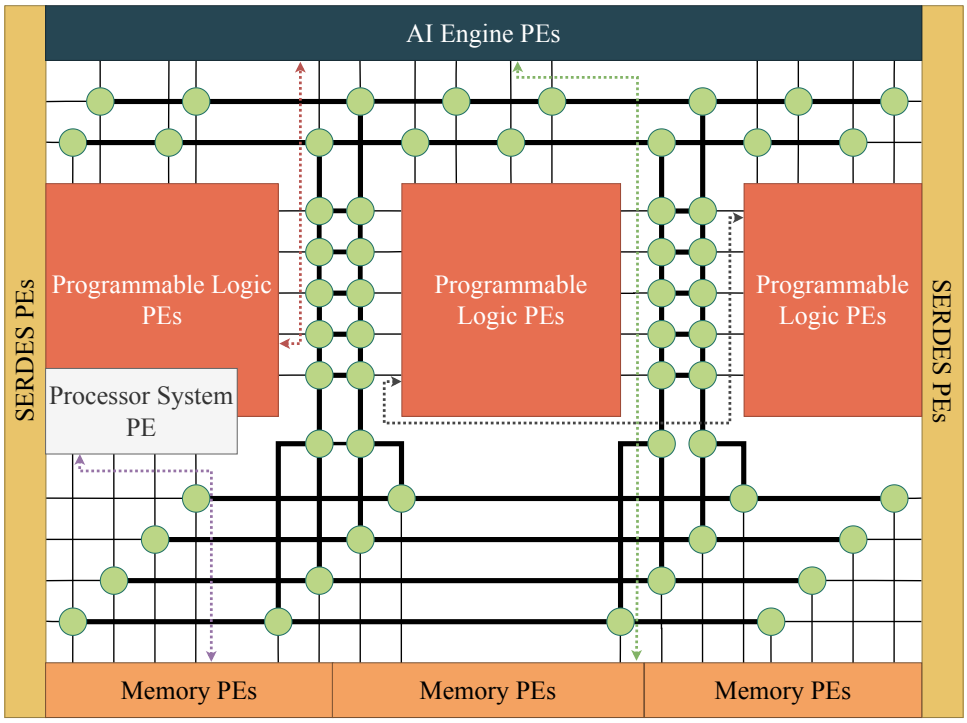


Figure 8.1: A Visual representation of the hard but configurable Versal NoC, re-drawn from [4].

- **Leveraging Graph Neural Networks for Accelerating Timing and Slack Predictions**

  Placement and routing of a physical RTL net-list is a time-consuming process that involves placing the physical net-list blocks onto an FPGA's programmable logic blocks and then kicking off a time consuming process to maximise operable clock frequency by calculating Worst Negative Slack (WNS) and Total Negative Slack (TNS). Taking an application-customized NoC and application through the placement and routing process can still present fresh challenges dependent on placement of net-list blocks and subsequent timing violations. Often this requires manual user intervention to close such timing violations by manually placing various net-list blocks and tweaking certain parameters of timing compilation process.

  While there has been previous work done on using reinforcement learning to place such hard blocks [115], each attempt by the learnt policy will still need lengthy timing analysis and manual tuning of timing compilation parameters. In a quest to

accelerate FPGA bitstream compilation, the process of tuning timing compilation parameters can be optimised by casting such parameters as learnable variables, similar to Chapter 5 ([2]) and Chapter 6 ([16]). Furthermore, subsequent timing analysis can be accelerated by training Graph Neural Networks to predict timing slacks for each individual links. We now list the high level idea of automating timing predictions:

- As a first step, powerful agents borne out of Reinforcement Learning create a draft placement of net-list blocks onto the programmable logic canvas [115].
- As the second step, in similar spirit as Chapter 5 ([2]) and Chapter 6 ([16]), learning of certain timing compilation parameters is cast as learnable parameters.
- As the final step, for each combination draft of the timing parameters (step two) and net-list placement (step 1), powerful GNNs can predict timing slack of each wire link with high accuracy in a fraction of the time it would take for a lengthy traditional compilation process.
- The above three steps are carried out in a loop until convergence towards an optimized placement of net-list blocks with no timing violations.

References

[1] A. Psarras, J. Lee, I. Seitanidis, C. Nicopoulos, and G. Dimitrakopoulos, "Phasenoc: Versatile network traffic isolation through tdm-scheduled virtual channels," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 844–857, 2015.

[2] G. Malik, I. E. Lang, R. Pellizoni, and N. Kapre, "Learn the Switches: Evolving FPGA NoCs with Stall-Free and Backpressure Based Routers," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 18–25, IEEE, 2020.

[3] G. S. Malik and N. Kapre, "Enhancing Butterfly Fat Tree Nocs for FPGAs with Lightweight Flow Control," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 154–162, 2019.

[4] I. Swarbrick, D. Gaitonde, S. Ahmad, B. Gaide, and Y. Arbel, "Network-on-chip programmable platform in versaltm acap architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, (New York, NY, USA), pp. 212–221, ACM, 2019.

[5] J. Shalf, "The future of computing beyond moore's law," *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2166, p. 20190061, 2020.

[6] W. Choi, K. Duraisamy, R. G. Kim, J. R. Doppa, P. P. Pande, D. Marculescu, and R. Marculescu, "On-chip communication network for efficient training of deep convolutional networks on heterogeneous manycore systems," *IEEE Transactions on Computers*, vol. 67, no. 5, pp. 672–686, 2017.

[7] K. Khalil, O. Eldash, A. Kumar, and M. Bayoumi, "N 2 oc: Neural-network-on-chip architecture," in *2019 32nd IEEE International System-on-Chip Conference (SOCC)*, pp. 272–277, IEEE, 2019.

[8] "Solutions for self-driving cars." https://www.nvidia.com/en-us/self-driving-cars/. Accessed: 2023-01-16.

[9] H. Fang, A. Shrestha, D. Ma, and Q. Qiu, "Scalable noc-based neuromorphic hardware learning and inference," in *2018 International joint conference on neural networks (IJCNN)*, pp. 1–8, IEEE, 2018.

[10] N. Akbari and M. Modarressi, "A high-performance network-on-chip topology for neuromorphic architectures," in *2017 IEEE international conference on computational science and engineering (CSE) and IEEE international conference on embedded and ubiquitous computing (EUC)*, vol. 2, pp. 9–16, IEEE, 2017.

[11] "Cost-optimized portfolio product tables and product selection guide." https://www.xilinx.com/content/dam/xilinx/support/documents/selection-guides/cost-optimized-product-selection-guide.pdf. Accessed: 2023-02-15.

[12] "7 series product selection guide." https://www.xilinx.com/content/dam/xilinx/support/documents/selection-guides/7-series-product-selection-guide.pdf. Accessed: 2023-02-15.

[13] "Ultrascale fpga product tables and product selection guide." https://www.xilinx.com/content/dam/xilinx/support/documents/selection-guides/ultrascale-fpga-product-selection-guide.pdf. Accessed: 2023-02-15.

[14] M. Wissolik, D. Zacher, A. Torza, and B. Da, "Virtex ultrascale+ hbm fpga: A revolutionary increase in memory performance," *Xilinx Whitepaper*, 2017.

[15] Y. Fu, E. Wu, A. Sirasao, S. Attia, K. Khan, and R. Wittig, "Deep learning with int8 optimization on xilinx devices," *White Paper*, 2016.

[16] G. Malik, I. E. Lang, R. Pellizzoni, and N. Kapre, "HopliteML: Evolving Application Customized FPGA NoCs with Adaptable Routers and Regulators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 4, pp. 1–33, 2022.

[17] G. S. Malik and N. Kapre, "GraphNoC: Graph Neural Networks for Application-Specific FPGA NoC Performance Prediction," in *2024 International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2024.

[18] L. C. Chan, G. Malik, and N. Kapre, "Partitioning FPGA-Optimized Systolic Arrays for Fun and Profit," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, pp. 144–152, IEEE, 2019.

[19] G. S. Malik, L. Petrica, N. Kapre, and M. Blott, "DarwiNN: Efficient Distributed Neuroevolution under Communication Constraints," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pp. 141–142, 2020.

[20] W. J. Dally and C. L. Seitz, "The torus routing chip," *Distributed computing*, vol. 1, pp. 187–196, 1986.

[21] A. Y. Seydim, "Wormhole routing in parallel computers," *School of Engineering and Applied Sciences Southern Methodist University May*, 1998.

[22] W. J. Dally and B. Towles, "Route packets, not wires: on-chip inteconnection networks," in *Proceedings of the 38th annual design automation conference*, pp. 684–689, 2001.

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

[24] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, *et al.*, "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE micro*, vol. 22, no. 2, pp. 25–35, 2002.

[25] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi, and A. Scandurra, "Spidergon: a novel on-chip communication network," in *2004 International Symposium on System-on-Chip, 2004. Proceedings.*, p. 15, IEEE, 2004.

[26] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, *et al.*, "An 80-tile 1.28 tflops network-on-chip in 65nm cmos," in *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pp. 98–589, IEEE, 2007.

[27] P. Gratz, C. Kim, R. McDonald, S. W. Keckler, and D. Burger, "Implementation and evaluation of on-chip network architectures," in *2006 International Conference on Computer Design*, pp. 477–484, IEEE, 2006.

[28] A. Andriahantenaina and A. Greiner, "Micro-network for soc: Implementation of a 32-port spin network," in *2003 Design, Automation and Test in Europe Conference and Exhibition*, pp. 1128–1129, IEEE, 2003.

[29] J. Choquette and W. Gandhi, "Nvidia a100 gpu: Performance & innovation for gpu computing," in *2020 IEEE Hot Chips 32 Symposium (HCS)*, pp. 1–43, IEEE Computer Society, 2020.

[30] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, and K. Kim, "Hbm (high bandwidth memory) dram technology and architecture," in *2017 IEEE International Memory Workshop (IMW)*, pp. 1–4, IEEE, 2017.

[31] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

[32] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, *et al.*, "Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pp. 1–14, 2023.

[33] D. U. Lee, H. S. Cho, J. Kim, Y. J. Ku, S. Oh, C. D. Kim, H. W. Kim, W. Y. Lee, T. K. Kim, T. S. Yun, *et al.*, "22.3 a 128gb 8-high 512gb/s hbm2e dram with a pseudo quarter bank structure, power dispersion and an instruction-based at-speed pmbist," in *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 334–336, IEEE, 2020.

[34] "Tpu v4." https://cloud.google.com/tpu/docs/v4. Accessed: 2025-01-03.

[35] N. Kapre and J. Gray, "Hoplite: Building austere overlay nocs for fpgas," in *Field Programmable Logic and Applications*, pp. 1–8, Sept 2015.

[36] N. Kapre, "Deflection-routed butterfly fat trees on fpgas," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, Sept 2017.

[37] T. Moscibroda and O. Mutlu, "A case for bufferless routing in on-chip networks," in *Proceedings of the 36th annual international symposium on Computer architecture*, pp. 196–207, 2009.

[38] C. Fallin, C. Craik, and O. Mutlu, "Chipper: A low-complexity bufferless deflection router," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pp. 144–155, IEEE, 2011.

[39] S. Wasly, R. Pellizzoni, and N. Kapre, "HopliteRT: An efficient FPGA NoC for real-time applications," in *F. Program. Technol. (ICFPT), 2017 Int. Conf.*, pp. 64–71, IEEE, 2017.

[40] Wasly, Saud, Pellizzoni, Rodolfo, and Kapre, Nachiket, "Worst case latency analysis for hoplite fpga-based noc," 2017.

[41] T. Garg, S. Wasly, R. Pellizzoni, and N. Kapre, "Hoplitebuf: Network calculus-based design of fpga nocs with provably stall-free fifos," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, Feb. 2020.

[42] R. L. Cruz, "A calculus for network delay. i. network elements in isolation," *IEEE Transactions on information theory*, vol. 37, no. 1, pp. 114–131, 1991.

[43] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[44] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Computing Surveys (CSUR)*, vol. 38, no. 1, pp. 1–es, 2006.

[45] L. Benini and G. De Micheli, "Networks on chips: A new soc paradigm," *computer*, vol. 35, no. 1, pp. 70–78, 2002.

[46] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*, pp. 684–689, 2001.

[47] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, and J. J. Dongarra, "Matrix market: a web resource for test matrix collections," in *Quality of Numerical Software*, pp. 125–137, Springer, 1997.

[48] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," June 2014.

[49] B. S. Landman and R. L. Russo, "On a Pin Versus Block Relationship For Partitions of Logic Graphs," *Computers, IEEE Transactions on*, no. 12, pp. 1469–1479, 1971.

[50] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE transactions on Computers*, vol. 100, no. 10, pp. 892–901, 1985.

[51] B. Lebiednik, A. Mangal, and N. Tiwari, "A survey and evaluation of data center network topologies," *arXiv preprint arXiv:1605.01701*, 2016.

[52] S. Scott, D. Abts, J. Kim, and W. J. Dally, "The blackwidow high-radix clos network," in *33rd International Symposium on Computer Architecture (ISCA'06)*, pp. 16–28, June 2006.

[53] A. Bouhraoua and M. Elrabaa, "An efficient network-on-chip architecture based on the fat-tree (ft) topology," in *Microelectronics, 2006. ICM'06. International Conference on*, pp. 28–31, IEEE, 2006.

[54] A. DeHon, "Compact, multilayer layout for butterfly fat-tree," in *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, (New York, NY, USA), pp. 206–215, ACM, 2000.

[55] Z. Wang, J. Xu, X. Wu, Y. Ye, W. Zhang, M. Nikdast, X. Wang, and Z. Wang, "Floorplan optimization of fat-tree-based networks-on-chip for chip multiprocessors," *IEEE Transactions on Computers*, vol. 63, no. 6, pp. 1446–1459, 2014.

[56] H. Gu, J. Xu, and W. Zhang, "A low-power fat tree-based optical network-on-chip for multiprocessor system-on-chip," in *2009 Design, Automation Test in Europe Conference Exhibition*, pp. 3–8, April 2009.

[57] C. Grecu, P. P. Pande, A. Ivanov, and R. Saleh, "Structured interconnect architecture: A solution for the non-scalability of bus-based socs," in *Proceedings of the 14th ACM Great Lakes Symposium on VLSI*, GLSVLSI '04, (New York, NY, USA), pp. 192–195, ACM, 2004.

[58] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," *SIGCOMM Comput. Commun. Rev.*, vol. 45, pp. 183–197, Aug. 2015.

[59] A. Andreyev, "Introducing data center fabric, the next-generation facebook data center network.." https://code.facebook.com/posts/360346274145943/, 2014. [Online].

[60] T. Garg, S. Wasly, R. Pellizzoni, and N. Kapre, "Hoplitebuf: Fpga nocs with provably stall-free fifos," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, (New York, NY, USA), pp. 222–231, ACM, 2019.

[61] M. K. Papamichael and J. C. Hoe, "Connect: re-examining conventional wisdom for designing nocs in the context of fpgas," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pp. 37–46, ACM, 2012.

[62] H. M. Wassel, Y. Gao, J. K. Oberg, T. Huffmire, R. Kastner, F. T. Chong, and T. Sherwood, "Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 583–594, 2013.

[63] A. Psarras, I. Seitanidis, C. Nicopoulos, and G. Dimitrakopoulos, "Phasenoc: Tdm scheduling at the virtual-channel level for efficient network traffic isolation," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1090–1095, IEEE, 2015.

[64] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[65] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[66] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[67] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[68] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[69] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[70] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[71] D. Jiang, Z. Wu, C.-Y. Hsieh, G. Chen, B. Liao, Z. Wang, C. Shen, D. Cao, J. Wu, and T. Hou, "Could graph neural networks learn better molecular representation for drug discovery? a comparison study of descriptor-based and graph-based models," *Journal of cheminformatics*, vol. 13, no. 1, pp. 1–23, 2021.

[72] J. Shlomi, P. Battaglia, and J.-R. Vlimant, "Graph neural networks in particle physics," *Machine Learning: Science and Technology*, vol. 2, no. 2, p. 021001, 2020.

[73] J. Busch, A. Kocheturov, V. Tresp, and T. Seidl, "Nf-gnn: network flow graph neural networks for malware detection and classification," in *33rd International Conference on Scientific and Statistical Database Management*, pp. 121–132, 2021.

[74] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. B. Wiltschko, "A gentle introduction to graph neural networks," *Distill*, vol. 6, no. 9, p. e33, 2021.

[75] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[76] S. M. J. Jalali, S. Ahmadian, A. Khosravi, S. Mirjalili, M. R. Mahmoudi, and S. Nahavandi, "Neuroevolution-based autonomous robot navigation: a comparative study," *Cognitive Systems Research*, vol. 62, pp. 35–43, 2020.

[77] N. Zhang, X. Chen, and N. Kapre, "Rapidlayout: Fast hard block placement of fpga-optimized systolic arrays using evolutionary algorithm," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 4, pp. 1–23, 2022.

[78] E. Conti, V. Madhavan, F. Petroski Such, J. Lehman, K. Stanley, and J. Clune, "Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents," *Advances in neural information processing systems*, vol. 31, 2018.

[79] K. O. Stanley, J. Clune, J. Lehman, and R. Miikkulainen, "Designing neural networks through neuroevolution," *Nature Machine Intelligence*, vol. 1, no. 1, pp. 24–35, 2019.

[80] T. Bayes, "An essay towards solving a problem in the doctrine of chances," *Biometrika*, vol. 45, no. 3-4, pp. 296–315, 1958.

[81] A. Tversky and D. Kahneman, "Judgment under uncertainty: Heuristics and biases: Biases in judgments reveal some heuristics of thinking under uncertainty.," *science*, vol. 185, no. 4157, pp. 1124–1131, 1974.

[82] N. Hansen, "The cma evolution strategy: A tutorial," *arXiv preprint arXiv:1604.00772*, 2016.

[83] N. Kapre and J. Gray, "Hoplite: Building austere overlay nocs for fpgas," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, Sept 2015.

[84] Y. Huan and A. DeHon, "FPGA optimized packet-switched NoC using split and merge primitives," in *Field-Programmable Technology*, pp. 47–52, Dec. 2012.

[85] M. S. Abdelfattah and V. Betz, "Networks-on-chip for fpgas: Hard, soft or mixed?," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, pp. 20:1–20:22, Sept. 2014.

[86] N. Kapre and T. Krishna, "Fasttrack: leveraging heterogeneous fpga wires to design low-cost high-performance soft nocs," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pp. 739–751, IEEE Press, 2018.

[87] C. Fallin, C. Craik, and O. Mutlu, "Chipper: A low-complexity bufferless deflection router," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pp. 144–155, Feb 2011.

[88] J. Mische, C. Mellwig, A. Stegmeier, M. Frieb, and T. Ungerer, "Minimally buffered deflection routing with in-order delivery in a torus," in *2017 Eleventh IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pp. 1–8, Oct 2017.

[89] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon, "Packet switched vs. time multiplexed fpga overlay networks," in *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 205–216, April 2006.

[90] C. Gomez, F. Gilabert, M. E. Gomez, P. Lopez, and J. Duato, "Deterministic versus adaptive routing in fat-trees," in *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 1–8, March 2007.

[91] E. Caspi, *Design Automation for Streaming Systems*. PhD thesis, University of California, Berkley, 2005.

[92] D. Park, Y. Xiao, N. Magnezi, and A. Dehon, "Case for fast fpga compilation using partial reconfiguration," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, Sept 2018.

[93] "Intel hyperflex architecture high-performance design handbook." https://www.intel.com/content/www/us/en/docs/programmable/683353/21-3/hyper-pipelining-add-pipeline-registers.html. Accessed: 2025-01-11.

[94] S. Jeon, J. Cho, Y. Jung, S. Park, and T. Han, "Automotive hardware development according to iso 26262," in *13th International Conference on Advanced Communication Technology (ICACT2011)*, pp. 588–592, Feb 2011.

[95] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, 2001.

[96] N. Kapre, H. Ng, K. Teo, and J. Naude, "Intime: A machine learning approach for efficient selection of fpga cad tool parameters," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, (New York, NY, USA), pp. 23–26, ACM, 2015.

[97] B. P. Carlin and T. A. Louis, *Bayes and empirical Bayes methods for data analysis*. Chapman and Hall/CRC, 2010.

[98] A. Costa and G. Nannicini, "Rbfopt: an open-source library for black-box optimization with costly function evaluations," *Mathematical Programming Computation*, vol. 10, pp. 597–629, Dec 2018.

[99] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 27–40, 2017.

[100] M. Karol, M. Hluchyj, and S. Morgan, "Input versus output queueing on a space-division packet switch," *IEEE Transactions on Communications*, vol. 35, no. 12, pp. 1347–1356, 1987.

[101] A. Sheikholeslami, R. Yoshimura, and P. G. Gulak, "Look-up tables (luts) for multiple-valued, combinational logic," in *Proceedings. 1998 28th IEEE International Symposium on Multiple-Valued Logic (Cat. No. 98CB36138)*, pp. 264–269, IEEE, 1998.

[102] F. Li, Y. Wang, C. Liu, H. Li, and X. Li, "Noception: a fast ppa prediction framework for network-on-chips using graph neural network," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1035–1040, IEEE, 2022.

[103] J. Silva, M. Kreutz, M. Pereira, and M. D. Costa-Abreu, "An investigation of latency prediction for noc-based communication architectures using machine learning techniques," *The Journal of Supercomputing*, vol. 75, no. 11, pp. 7573–7591, 2019.

[104] A. Kumar and B. Talawar, "Machine learning based framework to predict performance evaluation of on-chip networks," in *2018 Eleventh International Conference on Contemporary Computing (IC3)*, pp. 1–6, IEEE, 2018.

[105] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[106] B. Winefeld, "Using machine learning for characterizations of noc components," March 2019.

[107] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally, "A detailed and flexible cycle-accurate network-on-chip simulator," in *2013 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pp. 86–96, IEEE, 2013.

[108] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International conference on machine learning*, pp. 1263–1272, PMLR, 2017.

[109] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic graph cnn for learning on point clouds.(2018)," *arXiv preprint arXiv:1801.07829*, vol. 222, 2018.

[110] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[111] C. Zhang, D. Song, C. Huang, A. Swami, and N. V. Chawla, "Heterogeneous graph neural network," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 793–803, 2019.

[112] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition. corr abs/1512.03385 (2015)," 2015.

[113] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[114] Y. Yao, L. Rosasco, and A. Caponnetto, "On early stopping in gradient descent learning," *Constructive Approximation*, vol. 26, pp. 289–315, 2007.

[115] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, S. Bae, *et al.*, "Chip placement with deep reinforcement learning," *arXiv preprint arXiv:2004.10746*, 2020.