UNIVERSITY OF CALIFORNIA SAN DIEGO

Toward Efficient and Fragmentation-Aware Communication in Reconfigurable Networks

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Dilini Rukshani Athapathu

Committee in charge:

Professor George Porter, Chair
Professor Taylor Berg-Kirkpatrick
Professor George Papen
Professor Geoffrey Voelker

2026

The Dissertation of Dilini Rukshani Athapathu is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2026

DEDICATION

To my parents, for their unconditional love and support.

EPIGRAPH

We shape our tools and thereafter our tools shape us.

Marshall McLuhan

Nothing is done, yet nothing is left undone.

Tao Te Ching, *Chapter 48*

I do like to think there is a difference between being resigned to a situation and reconciled to it.

Amor Towles, *A Gentleman in Moscow*

TABLE OF CONTENTS

Dissertation Approval Page iii

Dedication iv

Epigraph v

Table of Contents vi

List of Figures ix

List of Tables xii

Acknowledgements xiii

Vita xiv

Abstract of the Dissertation xv

Introduction 1

Chapter 1 Background 4
1.1 Reconfigurable Networks 4
1.2 Reconfigurable Network Designs 5
1.2.1 Demand-Aware Networks 6
1.2.2 Demand-Oblivious Networks 7
1.3 Collective Communication 8
1.3.1 Communication Overhead in Distributed Training 10

Chapter 2 Reconfigurability within Collective Communication Algorithms 12
2.1 Background 13
2.1.1 Parallelization Strategies 13
2.1.2 Collective Communication Patterns 14
2.2 State of the Art Algorithms 15
2.2.1 Cost Model and Notation 15
2.2.2 Algorithms for Allreduce 15
2.2.3 Algorithms for All-to-All 17
2.3 Case for Reconfiguration 19
2.3.1 AllReduce with Reconfiguration 20
2.3.2 All-to-All with Reconfiguration 25
2.3.3 Optical Switching Technology 26
2.4 Conclusion and Future Work 27

Chapter 3 Flexible Torus Clusters: Less Fragmentation, Lower Communication Overhead 30

3.1 Problem Statement . . . 31
3.1.1 Fragmentation Metrics . . . 33
3.2 Related Work . . . 34
3.3 System Architecture . . . 35
3.3.1 Simulator Design . . . 35
3.3.2 Node Allocation Rules . . . 36
3.4 Experimental Design . . . 39
3.5 Evaluation . . . 40
3.5.1 Internal Fragmentation . . . 40
3.5.2 External Fragmentation . . . 41
3.5.3 Failed Rate Allocation . . . 41
3.5.4 Measuring True Internal Fragmentation . . . 43
3.5.5 Real World Traces . . . 46
3.5.6 Node-wise Reconfigurable Cluster . . . 48
3.5.7 Node-wise vs Block-wise Reconfigurable Cluster . . . 49
3.5.8 Minimizing Node Allocation to Satisfy Job Requests . . . 52
3.6 Multi-tenant Collectives . . . 55
3.6.1 DNN Model Simulation . . . 56
3.7 Conclusion . . . 61

Chapter 4 Rotor-XDP: Emulating a Circuit-Switched Network Stack with eBPF . . . 64
4.1 System Design . . . 66
4.1.1 OCS Design . . . 66
4.1.2 OCS Emulator Requirements . . . 66
4.1.3 Why XDP/AF_XDP? . . . 67
4.1.4 Emulator and the Actual System . . . 67
4.2 Implementation . . . 68
4.2.1 Emulator . . . 68
4.2.2 Memory Pool . . . 69
4.2.3 Data Path . . . 69
4.3 Evaluation . . . 70
4.3.1 Node-to-Node Throughput . . . 71
4.3.2 Node-to-Node Latency . . . 74
4.3.3 Time Synchronization . . . 74
4.3.4 Embedding OCS Behaviour . . . 75
4.4 Challenges and Limitations of XDP/AF_XDP . . . 77
4.5 Related Work . . . 78
4.6 Conclusion . . . 79

Chapter 5 Mitigating Spurious Retransmissions In Opera . . . 80
5.1 TCP Loss Recovery Algorithms . . . 80
5.2 Evaluation . . . 81
5.2.1 Testbed . . . 82
5.2.2 Topology and Routing . . . 83

5.2.3 TCP Stack Analysis 85
5.3 Conclusion 89

Chapter 6 Conclusion 91

Bibliography 92

LIST OF FIGURES

Figure 1.1. Example Connection Matrices Implemented by a Circuit Switch 5

Figure 2.1. Performance Comparison of State-of-the-Art Allreduce Algorithms 18

Figure 2.2. Goodput gain achievable with different reconfiguration delays on a 4x4 Torus against the best-performing algorithm (Swing) 21

Figure 2.3. Goodput gain achievable with different reconfiguration delays on a 8x8 Torus against the best-performing algorithm 22

Figure 2.4. Goodput gain achievable with different reconfiguration delays on a 16x16 Torus against the best-performing algorithm 23

Figure 2.5. Goodput gain achievable with different reconfiguration delays on a 32x32 Torus against the best-performing algorithm 24

Figure 2.6. Goodput gain achievable with different reconfiguration delays on a 64x64 Torus against the best-performing algorithm 25

Figure 2.7. Throughput gain achievable with different reconfiguration delays with all-to-all 26

Figure 3.1. Internal fragmentation: a node (4 compute units) is allocated to a job, leaving most of its capacity unused. 32

Figure 3.2. External fragmentation: a 4×2×1 request cannot be allocated due to lack of contiguous free resources. 32

Figure 3.3. 3D visualization of a 4×4×4 block. A single TPU tray (a node) has four TPU chips organized as a 2×2×1 mesh. 36

Figure 3.4. 2D visualization of a 4×4×4 block that is depicted in Figure 1. Each layer in 3D figure is lined up horizontally so that all the TPUs in each layer are clearly visible. 37

Figure 3.5. Requested shape: (2×1×1); Single Best Fit Strategy (Due to node-wise allocation, two more nodes are allocated to Job 1 than it needs, causing internal fragmentation.) 38

Figure 3.6. Requested shape: (4×6×4); Face Stitching: Stitches face along y-direction 38

Figure 3.7. Requested shape: (6×6×4); General Multiblock Strategy 38

Figure 3.8. Snapshot Process: Tune arrival rate to maintain target utilization. The snapshots of the cluster state are collected every 20 steps when the utilization is within ±2% of target. 40

Figure 3.9. Internal fragmentation across different utilization levels for the block-wise reconfigurable cluster (64 blocks in the cluster) 41

Figure 3.10. External fragmentation across different utilization levels for the block-wise reconfigurable cluster (64 blocks in the cluster) 42

Figure 3.11. Failed allocation rate across different utilization levels for the block-wise reconfigurable cluster (64 blocks in the cluster) 43

Figure 3.12. CDF of slice distribution: The synthetic distribution approximates TPUv4 closely, though it includes irregular slice requests absent in the original. . . 45

Figure 3.13. True internal fragmentation of a block-wise reconfigurable cluster at 90% Utilization (64 blocks in the cluster) 46

Figure 3.14. CDF of the requested GPU count for each cluster 48

Figure 3.15. Internal fragmentation of each cluster in Helios traces in a 3D reconfigurable torus with node-wise allocation 49

Figure 3.16. Each cluster contains one block, and the requested shape is 5×1×1 50

Figure 3.17. Each cluster contains two blocks, and the requested shape is 6×4×2 51

Figure 3.18. CDF of the admitted jobs 52

Figure 3.19. External fragmentation at 90% utilization (64 blocks in the cluster) 53

Figure 3.20. Failed allocation rate at 90% utilization (64 blocks in the cluster) 53

Figure 3.21. External fragmentation at 75% utilization (64 blocks in cluster) 55

Figure 3.22. Failed allocation rate at 75% utilization (64 blocks in cluster) 56

Figure 3.23. DLRM with a small data size 60

Figure 3.24. DLRM with a large data size 61

Figure 3.25. CosmoFlow model under different configurations 62

Figure 4.1. Use of XDP/AF_XDP in the Emulator 67

Figure 4.2. Use of XDP/AF_XDP in the Physical System (This figure is reproduced from one that appears in [42]) 68

Figure 4.3. Emulation: Data Path 70

Figure 4.4. Node-to-Node (Version 1) 72

Figure 4.5. One-to-One Mapping between Namespaces and HW Queues (Version 2) . 73

Figure 4.6. AF_XDP Packet Generator Results for FPGA-NIC SKB_MODE 75

Figure 4.7. AF_XDP to AF_XDP One Way Delay 76

Figure 4.8. Node-to-Node and Opera RTT 78

Figure 4.9. Number of hops a packet takes when running with Opera 79

Figure 5.1. Ping RTT Distribution when Opera is moving through all 32 topologies between node-1 and node-2 84

Figure 5.2. Distribution of hop counts between node-1 and node-2 for ping experiments conducted in Fig. 5.1 85

Figure 5.3. Number of spurious retransmission observed in each second for five trials of `iperf3` during a 60s period with Opera when loss recovery algorithm is RACK. 86

Figure 5.4. Number of spurious retransmissions observed for different reordering windows. The reordering window for default RACK is dynamically adjusted between MIN_RTT/4 and 1-SRTT based on the window step value. 87

Figure 5.5. Distribution of hop counts for intended paths between all 32 nodes (32x32) 89

LIST OF TABLES

Table 2.1. Tradeoff between Port Count and Switching Times 27

Table 3.1. Service Time Distribution 39

Table 3.2. Average fragmentation index across 500 snapshots for a block-wise reconfigurable cluster (64 blocks in the cluster) 44

Table 3.3. Average internal fragmentation across 500 snapshots for a block-wise reconfigurable cluster (64 blocks in the cluster 45

Table 3.4. Number of Deep Learning jobs handled by each cluster in Helios [24] 47

Table 3.5. Examples of Shape Selection for Helios Traces: Convert the GPU count to a tori shape (x·y·z) and find the tori shape from the existing TPUv4 distribution 47

Table 3.6. Block-wise vs Node-wise Reconfig. Cluster Comparison 54

Table 3.7. Synthetic vs Exact Shape Comparison 57

Table 3.8. Average Internal Fragmentation (64 blocks in cluster) 57

Table 3.9. DLRM Test Scenarios 58

Table 3.10. Distributions of TPUv4 slices employed in the simulations 63

Table 5.1. RACK Window Step Distribution 90

Table 5.2. Histogram for Reorder Distance (SACK) 90

ACKNOWLEDGEMENTS

I would like to acknowledge my advisor, Professor George Porter, for his unwavering support and guidance throughout the past five years. This dissertation would not have been possible without his mentorship and encouragement.

I am also grateful to Professor Geoffrey Voelker for his kindness and support during my time at UC San Diego.

I thank my committee members for their valuable feedback and guidance. I am also grateful to Max Mellette, Alex Forencich, and Yibo Guo for their insightful discussions and support, as well as to the members of the Systems and Networking group at UC San Diego for fostering a collaborative and stimulating research environment.

I would also like to express my sincere gratitude to Professor Justine Sherry and Professor Srinivasan Seshan for providing me with the opportunity to engage in research that ultimately inspired me to pursue a Ph.D. I am also grateful to Adithya Abraham Philip and Ranysha Ware for their friendship and support during my time at Carnegie Mellon University.

Finally, I would like to thank Sanjiva Weerawarana, Shafreen Anfar, and the entire Ballerina language and the standard library team at WSO2. Working on RFC-related efforts sparked my curiosity and planted the seeds for future research. Without this experience, this journey would not have been possible.

Chapter 2, in full, is a reprint of the paper ”Reconfigurability within collective communication algorithms” published in the 2nd Workshop on Networks for AI Computing, NAIC ’25, page 43–49, New York, NY, USA, 2025. Rukshani Athapathu and George Porter. Association for Computing Machinery. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in full, is a reprint of the work submitted to 24th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’27) under the title ”Flexible Torus Clusters: Less Fragmentation, Lower Communication Overhead”. Rukshani Athapathu and George Porter. The dissertation author was the primary investigator and author of this paper.

VITA

2026 Doctor of Philosophy, University of California San Diego

PUBLICATIONS

Rukshani Athapathu and George Porter. 2025. Reconfigurability within Collective Communication Algorithms. In Proceedings of the 2nd Workshop on Networks for AI Computing (NAIC '25). Association for Computing Machinery, New York, NY, USA, 43–49. https://doi.org/10.1145/3748273.3749203

William M. Mellette, Alex Forencich, Rukshani Athapathu, Alex C. Snoeren, George Papen, and George Porter. 2024. Realizing RotorNet: Toward Practical Microsecond Scale Optical Networking. In Proceedings of the ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24). Association for Computing Machinery, New York, NY, USA, 392–414. https://doi.org/10.1145/3651890.3672273

Adithya Abraham Philip, Rukshani Athapathu, Ranysha Ware, Fabian Francis Mkocheko, Alexis Schlomer, Mengrou Shou, Zili Meng, Srinivasan Seshan, and Justine Sherry. 2024. Prudentia: Findings of an Internet Fairness Watchdog. In Proceedings of the ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24). Association for Computing Machinery, New York, NY, USA, 506–520. https://doi.org/10.1145/3651890.3672229

Adithya Abraham Philip, Ranysha Ware, Rukshani Athapathu, Justine Sherry, and Vyas Sekar. 2021. Revisiting TCP congestion control throughput models & fairness properties at scale. In Proceedings of the 21st ACM Internet Measurement Conference (IMC '21). Association for Computing Machinery, New York, NY, USA, 96–103. https://doi.org/10.1145/3487552.3487834

ABSTRACT OF THE DISSERTATION

Toward Efficient and Fragmentation-Aware Communication in Reconfigurable Networks

by

Dilini Rukshani Athapathu

Doctor of Philosophy in Computer Science

University of California San Diego, 2026

Professor George Porter, Chair

Reconfigurable networks that use optical circuit switches are slowly paving their way into datacenters due to their ability to offer high bandwidth at low cost and power. However, the majority of these reconfigurable networks are aimed at conventional datacenter clusters where the traffic patterns are unpredictable. While dynamically reconfigurable network topologies have shown promise for general datacenter workloads, in this work, I investigate the extent to which they may benefit machine learning (ML) collectives. Several algorithms have been proposed to implement collectives on static networks (of different topologies), but less well studied has been the interaction between these algorithms and reconfigurable network fabrics. Therefore, I seek to quantify the potential benefit that a reconfigurable network layer can bring to

the large-scale collectives that underpin large DNN training. Following this, I study the extent of fragmentation in ML torus clusters. In multi-tenant environments, resource allocations in torus clusters often lead to fragmentation, reducing system utilization and increasing queue times. My work demonstrates how enhancing the flexibility of a torus network through reconfiguration can significantly mitigate this fragmentation.

Introduction

Datacenter networks have been fundamental to the advancement of modern computing, forming the backbone of today's digital infrastructure. Most datacenters rely on Clos-based topologies [4, 58] built using electrical packet switches for their datacenter networks (DCNs). Clos topologies have become the de facto standard due to their ability to scale efficiently using commodity switching hardware. However, as network scale continues to grow, containing power consumption and infrastructure cost has become increasingly challenging. To address these constraints, datacenter operators often employ oversubscription. While oversubscription improves cost efficiency, it comes at the expense of increased resource fragmentation, higher tail latencies, and reduced effective bandwidth.

These challenges are expected to intensify in the future for two primary reasons. First, the explosive growth of deep neural network (DNN) workloads has fueled massive investment in hardware accelerators such as GPUs and TPUs, while simultaneously increasing demands for higher bandwidth and ultra-low network latency. Second, the slowdown of Moore's law is making it increasingly difficult to improve the power efficiency and cost-effectiveness of electrical packet-switched networks [7, 44].

This motivates the need for alternative technologies. Reconfigurable interconnects are one such option. Technologies such as optical circuit switches (OCSes) and free-space optics (FSO) can provide very high bandwidths at low cost and power making it suitable to overcome the limitations of electrical packet switches. However, despite this enticing property, OCSes have been slow in making their way into data centers mainly due to 1) Slow switching speed (Commercially available OCSes have millisecond scale swithing times [52, 66]) 2) OCSes are

not a drop-in replacement for packet switches since they need assistance from the entire network requiring many components to work in concert. These reasons have hindered the wide-scale adoption of OCSes in datacenters. However, despite these limitations, there are many efforts in the research community as well as in the industry to overcome these shortcomings and use OCSes in a novel way to build next-generation DCNs.

Today, the explosive growth of machine learning (ML) workloads, particularly deep neural network (DNN) training, presents new opportunities for reconfigurable networks. The traffic patterns generated by these workloads are often predictable and can be computed in advance, making circuit switches especially well-suited for handling such communication demands.

At the same time, it is worth reconsidering whether Clos topologies, which have traditionally been effective for general-purpose datacenter clusters, remain the best fit for today's rapidly evolving ML workloads. Large-scale distributed DNN training jobs often generate substantial communication overhead, limiting the number of servers that can efficiently operate within a single rack. As communication extends across multiple switches and racks, as in multi-tier Clos topologies, the potential for network bottlenecks increases significantly.

Although Clos topologies provide strong performance for diverse and unpredictable datacenter traffic, their traffic-oblivious design is less well-suited for the highly structured, predictable, and communication-intensive patterns characteristic of distributed DNN training. Consequently, many major cloud providers have adopted Torus topologies for large-scale ML training workloads, since the communication structure of these applications naturally maps onto torus networks. Furthermore, resource allocation in multi-tenant Torus environments can produce topology slices of varying sizes, with training workloads on these slices frequently exhibiting 1D, 2D, or 3D logical communication patterns. Therefore, it is necessary to first understand the performance of state-of-the-art collective communication algorithms across different sizes of static Torus topology slices. Building on this understanding, we can then investigate whether reconfigurable networks provide opportunities to improve performance in ways that existing

algorithms and static topologies cannot fully exploit.

Another important challenge in large multi-tenant datacenter clusters is system fragmentation. Given the high cost of neural processing units (NPUs), understanding and mitigating fragmentation is critical for improving overall cluster utilization and efficiency. This also creates an opportunity to explore whether increasing the flexibility of Torus networks through reconfigurable networking can reduce fragmentation and enable more efficient resource allocation.

The flexibility provided by reconfiguration can be extended even further by giving tenants direct control over how their allocated topology slices are reconfigured to optimize individual job performance. For example, dynamically changing the shape of an allocated slice or redirecting otherwise unused links toward dominant communication directions can create opportunities that are not possible in static Torus slices. In this dissertation, we examine each of these aspects in detail.

As datacenter traffic continues to grow alongside the explosive rise of AI workloads and their communication-intensive nature, the limitations imposed by the slowdown of Moore's law are becoming increasingly apparent. In this context, reconfigurable networks are evolving from an optional optimization into a practical necessity for next-generation ML datacenter networks.

Chapter 1

Background

This chapter begins with an introduction to reconfigurable networks and their underlying design principles. We then explore how DNN models operate and how collective patterns naturally emerge from different parallelization strategies. Building on this, we examine various collective communication patterns and highlight the key challenges faced by current ML hardware fabrics. Finally, we discuss state-of-the-art solutions that address these challenges, providing a roadmap of the current landscape in efficient collective communication.

1.1 Reconfigurable Networks

Reconfigurable networks enable communication links between nodes to be dynamically rearranged over time. A variety of circuit-switching technologies can be used to implement such systems, including MEMS-based optical circuit switches [75], arrayed waveguide grating routers (AWGRs) with tunable lasers [70], and free-space optics (FSO) [39].

Unlike packet-switched networks, circuit-switched systems require the establishment of an end-to-end circuit between a source and destination before data transmission can begin. While this enables high-throughput, contention-free communication during a configured state, it introduces an important overhead: reconfiguration time. This is the time required to change the network connectivity pattern and is commonly referred to as the switching time or reconfiguration delay of the circuit switch.

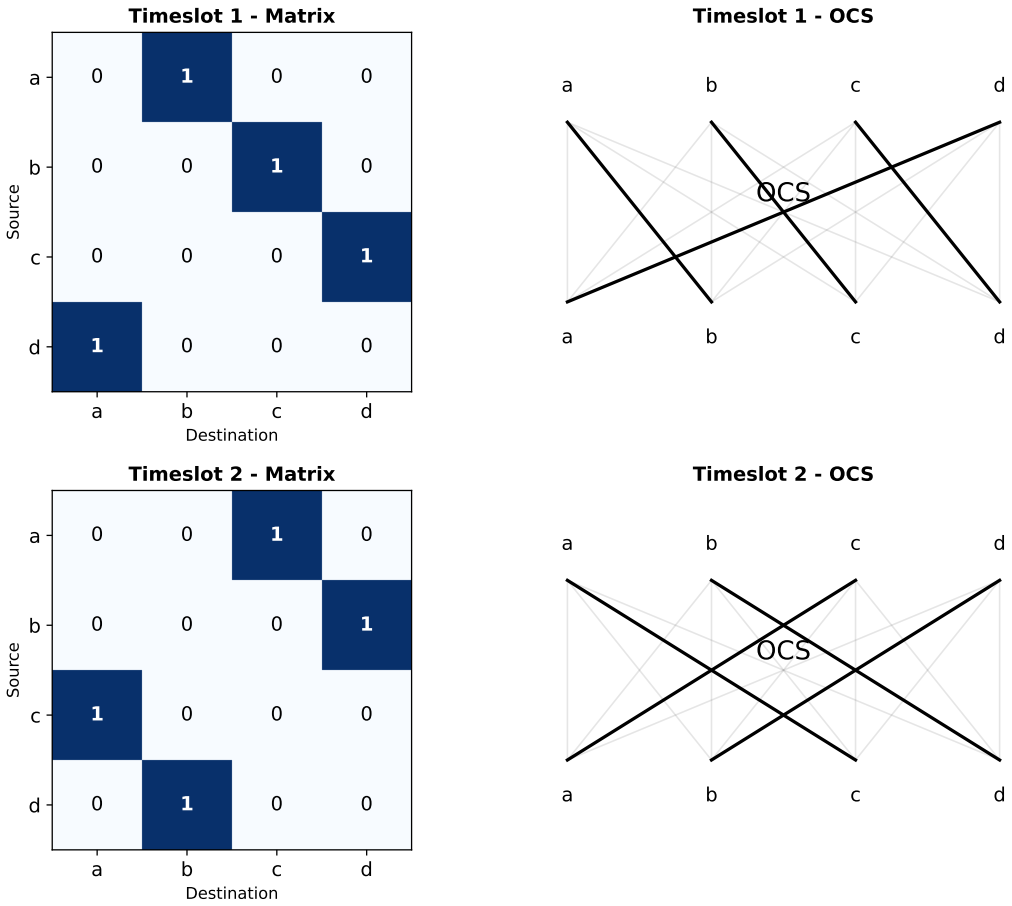


Figure 1.1. Example Connection Matrices Implemented by a Circuit Switch

Assume that we have a simple reconfigurable network consisting of four nodes a, b, c and d. Figure 1.1 shows example connection matrices implemented using circuit switching across two time slots. In time slot 1, node a is connected to b, b is connected to c, and so on. In the subsequent time slot, the network is reconfigured such that a is connected to c, b to d, and so forth, demonstrating the ability of the system to adapt its connectivity over time.

1.2 Reconfigurable Network Designs

Reconfigurable networks are typically designed following one of two paradigms: demand-aware or demand-oblivious. Reconfigurable networks that uses demand-aware approach takes the traffic demand into account when setting up OCS circuits and demand-oblivious approach

operates independently of the demand. Slow-switching networks are suitable for demand-ware networks as they give time for demand collection and topology calculation. Fast-switching networks are candidates for demand-oblivious networks as they need not collect demand matrices or do any topology calculations.

1.2.1 Demand-Aware Networks

The strategy of demand-aware networks' is to set up OCS circuits based on the traffic demand. To realize this design, the following mandatory steps need to be in place.

- Gather traffic statistics
- Estimate traffic demand
- Calculate optimal topology
- Reconfigure circuits

Algorithms and the strategies used for each of these steps vary based on the placement of the OCSes and the goal of the reconfigurable network design. For example, both Helios [24] and Jupiter [50] use OCSes to connect pod-level switches but aim to achieve different goals. Helios provides low power, low cost, and low complex datacenter architecture and has the same performance as a fully-provisioned packet-switched network. However, it needs a separate electrical packet switching network to fulfill the pod-level bursty traffic demands. Jupiter on the other hand wants to solve a real-world problem that Google datacenters face with incremental evolution of their network fabrics. Two-thirds of their fabrics contain at least two generations of network infrastructure and the spine layer has become the bottleneck when higher link speed blocks are added to the fabric. Therefore, to solve the link speed derating issue they got rid of the spine layer switches in their Clos fabrics and replaced them with OCSes. They didn't need a separate packet-switched network due to the relatively predictable traffic demands in their fabrics. This removed the need for them to support worst-case permutation traffic. For both these

designs high switching reconfiguration time of OCSes was not a problem. For Helios [24], this is due to the stability of the pod-level aggregated traffic demand and similarly Jupiter found that reconfiguring block-level topology more frequently does not yield any benefits [50].

As faster switching OCSes are becoming available, it is worth exploring whether the demand-aware approach is still feasible. Mordia's [49] aim was just that. Mordia interconnects top-of-rack (ToR) switches instead of pod-level switches and proposed a microsecond latency OCS control plane. However, their design still needs a separate packet switch for small flows that are latency-sensitive. Another limitation of Mordia was that its fan-out was limited to only 24 ToRs. ProjecToR [15] on the other hand makes use of FSO to propose a high fan-out interconnect based on just a single technology. They argue that maintaining a parallel EPS network leads to inefficiency and higher cost and hence proposed a two-topology approach that makes use of just one technology.

All these proposals we have looked into so far are based on rack-based topology meaning servers are organized into racks and the network core (either OCS or EPS-based network core) connects racks. RDC [71] deviates from this approach by logically grouping servers by placing OCSes in between servers and ToRs. In their design servers are not bound to a particular ToR but can move logically across ToRs. Their motivation for doing this is to efficiently handle the cross-rack traffic as more and more traffic is escaping rack boundaries.

1.2.2 Demand-Oblivious Networks

Demand-oblivious networks follow a static schedule regardless of the traffic demand making way for a decentralized control plane. RotorNet [43], Opera [41], and Sirius [6] are three such designs. All these three designs rotate through a fixed set of configurations over time eliminating the need for network-wide demand collection and making the overall design less complex. However, time synchronization and routing is an important part of oblivious networks. Time synchronization is necessary since switches follow a pre-determined static schedule and nodes (servers or ToRs) need to be aware of the switch state to decide when to

transmit which traffic. Routing is important to handle arbitrary traffic matrices including skewed traffic. Demand-oblivious networks use some form of Valiant Load Balancing (VLB) [67] routing scheme for that. VLB does not need centralized coordination and can split the traffic uniformly across all network paths.

The reconfigurable network designs discussed above are primarily tailored for traditional datacenter clusters, where workload traffic is often unpredictable. However, modern workloads are increasingly dominated by DNNs, which exhibit structured and predictable communication patterns. Unlike conventional applications, these workloads follow regular communication structures that can be exploited for more efficient network design.

1.3 Collective Communication

We begin this section by presenting an overview of Deep Neural Networks (DNNs) to establish the basis for understanding when and how communication patterns emerge during training. A DNN is composed of a sequence of layers, each consisting of interconnected nodes known as neurons. Each neuron is associated with learnable parameters, namely weights and biases, which collectively define the behavior of the model.

Training a DNN involves iteratively performing forward and backward passes over the network. This process is repeated for multiple iterations until the model converges to the desired level of accuracy. To accelerate training, it is common practice to distribute both data and model computation across multiple Neural Processing Units (NPUs). The primary parallelization strategies used today are outlined below:

- **Data Parallelism:** In data parallelism, each NPU maintains a complete replica of the model while the training dataset is partitioned across devices. Each NPU independently performs forward and backward passes on its assigned data subset, producing local gradients. These gradients are then aggregated across all NPUs, synchronized, and redistributed so that each device can update its model parameters before the next training iteration begins.

- **Pipeline Parallelism:** Pipeline parallelism partitions the model vertically at layer boundaries, assigning consecutive layers to different NPUs. During training, intermediate activations are passed between devices in the forward pass, while gradients flow in the reverse direction during backpropagation.

- **Operator Parallelism:** Operator parallelism divides individual tensor operations horizontally across NPUs, enabling each device to compute a portion of the operation concurrently.

- **Hybrid Parallelism:** Hybrid parallelism combines multiple of the above strategies to better balance computation and communication across devices.

References for the above parallelization strategies: [25, 56, 55]

The parallelization strategies described above give rise to a variety of collective communication patterns. These patterns define how data is exchanged, aggregated, and distributed across multiple devices during distributed training. An overview of the key communication primitives is provided below.

- **Allreduce:** Aggregates data from all devices and distributes the result back to all participants. Commonly used for gradient synchronization in data parallel training.

- **All-to-all:** Each device sends distinct data to every other device and receives data from all others.

- **Allgather:** Collects data from all devices and concatenates it so that every device has the full combined dataset.

- **Reduce-scatter:** Combines a reduction and a scatter operation, where data is first reduced across devices and then partitioned and distributed, so each device receives only a segment of the result.

- **Broadcast:** A single device sends identical data to all other devices. This is commonly used for distributing model parameters or configuration values.

- **Reduce:** Aggregates data from all devices into a single result on a designated root device.
- **Scatter:** Splits data on a root device and distributes different portions to each participating device.
- **Gather:** Collects data from all devices and sends it to a single root device without performing any reduction.

References for the above collective communication patterns: [62, 48]

1.3.1 Communication Overhead in Distributed Training

As we increase the number of workers or NPUs, the communication overhead of distributed DNN training increases significantly. During communication phases, NPUs remain idle, waiting for data transfers to complete. This leads to substantial underutilization of compute resources. This problem is expected to worsen in the future, as improvements in GPU performance continue to outpace improvements in network bandwidth, further exposing communication bottlenecks.

To address this issue, several state-of-the-art approaches have been proposed to reduce communication overhead. From a hardware perspective, one solution is to design faster interconnects. NVIDIA's NVLink and NVSwitch [26, 59] are prominent examples of high-bandwidth, low-latency interconnect technologies. However, these systems are typically limited to a single server or node containing 8 to 16 GPUs. While they may provide terabytes-per-second bandwidth between GPUs within a node, large-scale training workloads often span across many nodes. In such cases, communication must traverse the conventional network fabric, which can become a significant bottleneck.

Another approach is to design more efficient collective communication algorithms. However, this remains a challenging problem, as only a limited number of well-studied algorithms exist, and they are not universally optimal across different network topologies. Designing such

algorithms requires carefully balancing latency and bandwidth efficiency, which is inherently difficult.

One might consider automating collective algorithm design using optimization techniques [9, 54, 73, 38]. However, these approaches face scalability limitations and often fail to generalize to larger or more complex topologies, which are increasingly important in modern distributed training systems.

An alternative direction is to optimize the network topology itself to better match DNN communication patterns. This is where reconfigurable network architectures come into play. Systems such as TopoOpt [72] and TPUv4 [27] leverage reconfigurable network designs to better align the communication fabric with machine learning workloads. However, both approaches do not support dynamic reconfiguration during training. TopoOpt, for instance, generates customized shifted-ring topologies to optimize collective operations, but it may still suffer from the inherent latency limitations of ring-based communication patterns. Similarly, Google's TPUv4 employs a 3D torus-based ML fabric composed of 4×4×4 supercomputing cubes. Within each cube, the interconnect links are fixed and non-reconfigurable. Only the inter-cube links, corresponding to the six faces of each cube, are reconfigurable, limiting the flexibility of the overall network during training.

In our work, we investigate the potential of dynamically reconfiguring the network during DNN training and demonstrate how enhancing the flexibility of a torus topology through reconfiguration can improve overall system efficiency.

Chapter 2

Reconfigurability within Collective Communication Algorithms

The unprecedented growth of AI workloads has resulted in massive investments in high-performance Neural Processing Units (NPUs), such as GPUs and TPUs, as well as their supporting interconnects [27, 46]. Despite the ever-growing diversity of DNN models today, their underlying structure largely remains the same. DNN models consist of many layers of neurons and have multiple weights and biases associated with them. A model training will go through multiple iterations until it reaches its desired accuracy. Now, to improve the efficiency of model training and inference as well as to deal with billions of parameters that are associated with them, the data and the model are parallelized across multiple NPUs. This leads to various communication patterns during model training.

Recent work has investigated ways of dynamically changing the physical topology of the network at runtime, using electronic [57] or optical circuit switches [43, 15, 6, 71, 41]. While dynamically reconfigurable network topologies have shown promise for general datacenter workloads, in this paper, we investigate the extent to which they may benefit machine learning (ML) collectives. Several algorithms have been proposed to implement collectives on static networks (of different topologies), but less well studied has been the interaction between these algorithms and reconfigurable network fabrics. Existing reconfigurable network design proposals that optimize ML workloads, such as TopoOpt [72] and TPUv4 [27], use a one-shot

reconfiguration, making the network static during model training. High reconfiguration delays of commercially available optical circuit switches are the main reason for not allowing the topology to reconfigure during training iterations. Other works, such as MixNet [36], leverage the strong locality inherent in expert-parallel traffic to partition the cluster into regionally reconfigurable high-bandwidth domains. MixNet [36] assumes a 25 ms OCS reconfiguration latency and relies on overlapping topology reconfiguration with expert computation.

However, there are many advances in circuit switch technologies to bring down the switching delays to the nanosecond scale [6, 12]. Therefore, it is important to explore the benefits that such future technology advances can bring to these ever-growing DNN models. In this paper, we seek to quantify the potential benefits that such a reconfigurable network layer can bring to the large-scale collectives that underpin DNN training.

First, we briefly describe the parallelization strategies and the various communication patterns that can occur during model training due to parallelism. Next, we discuss the state-of-the-art algorithms that are optimized for Torus-like topologies since they are one of the popular interconnects used for running ML workloads. Algorithms such as Recursive Doubling, Swing, and pair-wise exchange of All-to-all all result in congested links[1] that lead to suboptimal performance. Hence, we analyze whether the reconfigurability within a topology slice can eliminate the congestion to achieve each of these algorithms' full potential.

2.1 Background

2.1.1 Parallelization Strategies

Different parallelization strategies exist to improve the training time of DNNs. This requires the synchronization of activations, weights, and gradients among NPUs during the forward and backward passes.

[1]Multiple messages from different flows competing for bandwidth on a single link is referred to as congestion in this paper.

Data Parallelism:

For Data Parallelism, training samples are distributed across multiple NPUs. Each NPU holds a copy of the DNN model and locally executes the forward and backward passes. However, during each iteration, all the NPUs need to synchronize their model weights.

Model Parallelism:

Model Parallelism comes into play when large DNN models cannot fit into a single NPU or a single server with multiple NPUs. With model parallelism tensor operations and their weights are distributed across multiple NPUs and each NPU computes a different part of the DNN model simultaneously.

Pipeline Parallelism:

With Pipeline Parallelism, consecutive layers of the model are assigned to different NPUs and each layer is processed sequentially.

2.1.2 Collective Communication Patterns

These parallelization strategies result in different communication patterns during the model training. The following is a list of different communication patterns that can occur during DNN training: Allreduce, All-to-all, Broadcast, Reduce, Scatter, Gather, Allgather, and Reduce-scatter. Allreduce and all-to-all are the most common collective communication patterns that occur in DNN training. Hence, we limit our study to just those two.

Allreduce

Allreduce does a reduction operation on the data across all the NPUs, and the result are stored locally on every NPU.

All-to-all Personalized

With an all-to-all pattern, each NPU holds a unique chunk of data that needs to be sent to each of the other NPUs.

2.2 State of the Art Algorithms

Collective communication patterns can be realized through various algorithms. In this section, we review the algorithms that have been developed for static network topologies. Next, in section 2.3, we will examine how they might work on a reconfigurable network.

2.2.1 Cost Model and Notation

We use the $\alpha - \beta$ cost model [63] to estimate the cost of communication of each algorithm. We disregard the computation time to solely focus on the network. α is the latency cost associated with every message (startup time), independent of its size. β is the transfer time per byte. Therefore, the total cost to transfer n bytes between any two nodes can be modeled as $\alpha + n\beta$ in the absence of congestion.

We focus on running collectives on a torus topology and assume that each node is equipped with multiple ports. If D denotes the number of dimensions, then a single node would have $2D$ ports. The links are bidirectional, and p represents the number of nodes. We assume each NPU to have a single process.

2.2.2 Algorithms for Allreduce

Ring Allreduce

Ring allreduce requires the NPUs to be organized as a ring. Each NPU sends a block to its right neighbor and receives a block from its left neighbor. Ring allreduce first does a reduce-scatter and follows it up with an allgather. In the reduce-scatter phase, each NPU splits its data into p equally sized blocks and is responsible for doing the reduction for a single block. Each NPU will then have a reduced portion of the vector at the end of the reduce-scatter phase. Then, during allgather, each NPU would distribute the reduced portion of the vector to all other NPUs. The reduce-scatter and the allgather each perform *(p-1)* steps. Therefore, the total number of steps required by Ring allreduce is *2(p-1)*. In each step, a single port would transmit *n/2Dp* bytes.

Then the total cost to run ring allreduce can be modeled as $T(n) = 2(p-1)\alpha + ((n/D)((p-1)/p)\beta$.

On a 1D Torus, two ring algorithms can be run simultaneously in each direction, each with a data size of *n/2*. On a *2D* Torus, two edge-disjoint Hamiltonian cycles can be found if a certain constraint is met [53]. Two edge-disjoint Hamiltonian cycles gives us four rings to run four simultaneous reductions. However, this algorithm does not work on $D > 2$ [53].

Bucket

On a square ($axa = p$) $2D$ torus, bucket algorithm first runs a row-wise reduce-scatter. This requires *(a - 1)* steps. Then a column-wise reduce-scatter is run on the data that has already been reduced in the previous step, and finally a column-wise allgather is done followed by a row-wise allgather [8]. On a multiport *D*-dimensional torus, the algorithm splits the data into *2D* parts and concurrently runs *2D* bucket algorithms [53]. There is no congestion since each bucket algorithm starts with a different dimension and direction. Total cost of bucket algorithm on a *D*-dimensional torus is $T(n) = 2D\sqrt[D]{p}\alpha + (n/D)\beta$.

Recursive Doubling

Recursive Doubling algorithm has two versions. 1) Latency-Optimal Recursive Doubling 2) Bandwidth-Optimal Recursive Doubling [63]. We use the multi-port version proposed by Daniele et al. [53] in this paper. In recursive doubling each node communicates with *D* nodes at distance 2^i (one per dimension) where *i* is the current step. In the latency optimal version, each node sends its entire vector to the destination node and receives the destination node's vector. Before moving on to the next step, each node aggregates its vector with the receiver's. The latency-optimal recursive doubling algorithm executes $\log_2(p)$ steps and the total cost of latency assuming that it utilizes all the ports is $T(n) = \log_2(p)\alpha + (n/2D)\log_2(p)\beta$.

In the bandwidth optimal version, like the previous two algorithms, it does a reduce-scatter followed by an allgather. The reduce-scatter and the allgather each perform $\log_2(p)$ steps. During the reduce scatter period, the algorithm halves the size of the transmitted data at

each step and doubles the distance between communicating nodes. The allgather reverses the communication pattern, doubling the size of the transmitted data at each step and halving the distance between communicating nodes [53]. Assuming all ports are utilized $T(n) = 2\log_2(p)\alpha + (n/D)((p-1)/p)\beta$.

However, unlike the ring allreduce and bucket algorithms, the Recursive Doubling algorithm results in congested links when the distance between the communication nodes grows.

Swing

Swing is a new allreduce algorithm proposed by Daniele et al. [53] that aims to lower the congestion prevalent in recursive doubling by reducing the distance between communicating nodes. It also has two versions: 1) Latency Optimal and 2) Bandwidth Optimal. The bandwidth optimal version does a reduce-scatter followed by an allgather. Just like the recursive doubling, reduce-scatter and allgather each execute $\log_2(p)$ steps. Reduce-scatter phase halves the size of transmitted data at each step, and allgather doubles the size of transmitted data at each step. Swing differs from recursive doubling by communicating to a node at distance $\sum_{i=0}^{s}(-2)^i$ where s is the number of steps [53]. Unlike recursive doubling, the communicating peer of each node swings from left to right and vice versa. With Swing, congestion deficiency decreases with the number of dimensions since the nodes closer together have more communications done before moving on to distance nodes [53].

2.2.3 Algorithms for All-to-All

All-to-all communication pattern can be implemented by a variety of algorithms, and the decision to use what algorithm depends on the message size and the number of processes. Modified bruck algorithm [65] is typically used for very small size messages, and for large messages, the pair-wise exchange method is used.

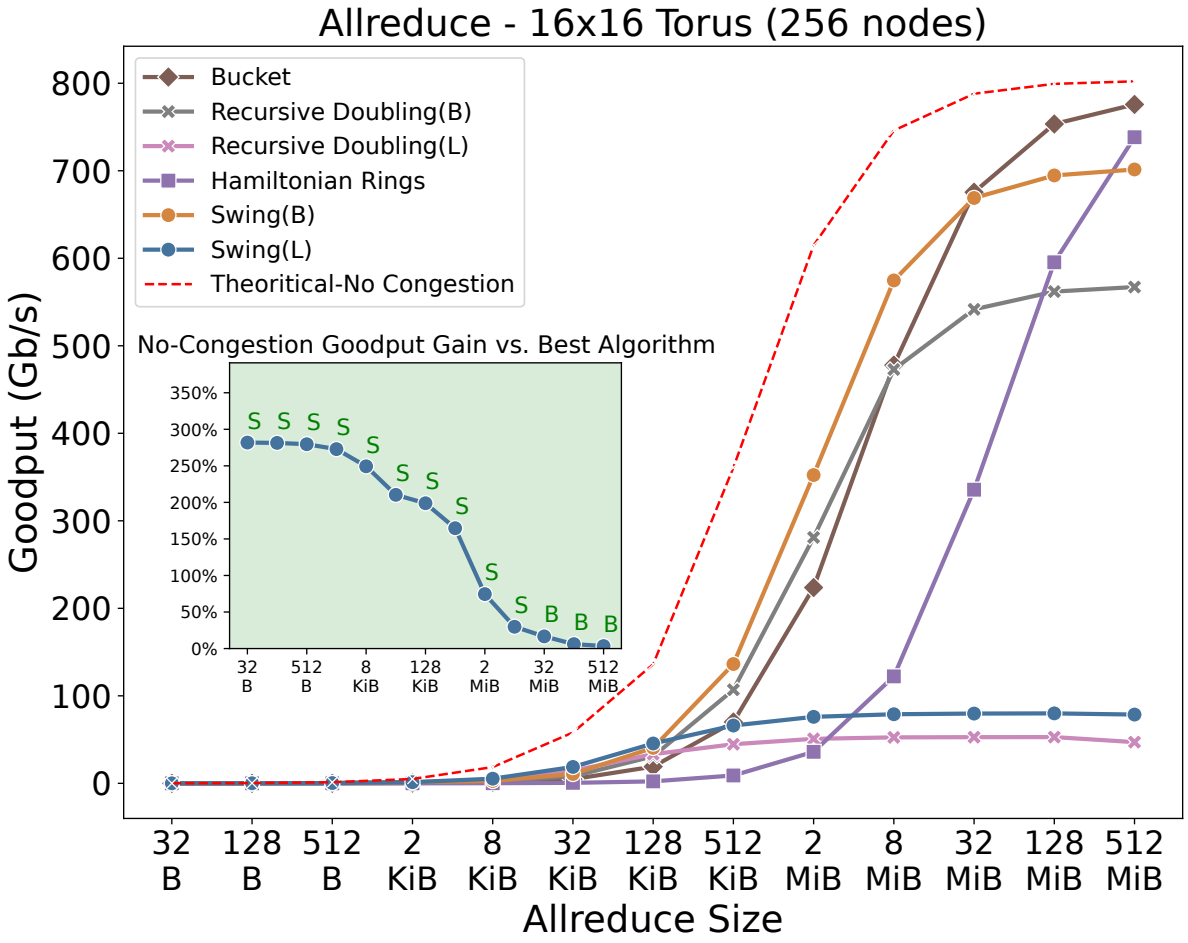


Figure 2.1. Performance Comparison of State-of-the-Art Allreduce Algorithms

Pair-wise Exchange

In the Pair-wise Exchange method, each process exchanges data with every other destination process directly, and this method takes p - 1 steps to finish the collective [63].

Bruck Algorithm

Bruck algorithm on the other hand can finish with just $\log_2(p)$ number of steps but at the cost of sending more data. Therefore, Bruck is used for short-sized messages.

The basic Bruck algorithm has three phases. First each process i needs to shift its data up by i elements. Second phase is executed in $\log_2(p)$ rounds. In the second phase, each process i send to process $j = (i+2^k)$ all the elements whose k^{th} bit of elements that are destined for j

are equal to one. k represents the current step. By the end of the $\log_2(p)$ steps all the data gets sent to the right destination process. However, the data is not in the correct order so in the third phase each process needs to do a local inverse shift of the blocks. To remove the cost of local reordering in the third step, a modified bruck algorithm has been proposed [65].

2.3 Case for Reconfiguration

We first analyze the performance of the above-mentioned algorithms on a static torus topology (Figure 2.1). We consider a 16x16 2D-Torus with allreduce sizes ranging from 32 bytes to 512MiB. We use the metric *goodput = n/T(n)* to measure the performance of each algorithm. Goodput denotes how many bytes are reduced per time unit. We use the Structural Simulation Toolkit (SST) [61] and the algorithms implemented by Daniele et al. [21] to analyze the algorithm performance. We set the link bandwidth to 400Gb/s with 100ns latency. Per hop packet processing latency is set to 300ns.

Latency optimal Swing and Recursive Doubling perform well up to 128KiB. Bandwidth optimal Swing has the best performance between data sizes 512KiB and 32MiB. However, when the allreduce sizes increase beyond 32MiB, the bucket and ring algorithm start to outperform all other algorithms. We also plot the *goodput gain* that is achievable with Swing/Recursive Doubling if we are to avoid the congestion that is present in the aforementioned algorithms against the best-performing algorithm. 100% gain means no-congestion version performs 2x faster than the best performing algorithm, which is denoted in the first letter of the algorithm. Figure 2.1 shows that there is still a considerable goodput gain to be achieved if the congestion can be avoided.

On a Torus topology, Ring Allreduce and Bucket algorithm only need each node to communicate with its neighbors, hence, they see no congestion. Reconfiguration offers no added advantage for these algorithms directly. Recursive Doubling and Swing, on the other hand, need to talk to nodes multiple hops away, so some messages have to hop through multiple

nodes to get to the destination. If a direct link is provided between these distance nodes through reconfiguration or some other mechanism (adding extra links), link congestion can be avoided, improving the communication performance. However, there is a latency cost associated with reconfiguration. In this section, we investigate the maximum delay that these algorithms can spare if we are to reconfigure the topology slice during every step of the Recursive Doubling/Swing algorithm. We plot the theoretical goodput gain(ideal) achievable with different reconfiguration delays. However, we still included the per-hop packet processing latency of 300ns for the theoretical goodput gain of reconfiguration (e.g., to account for routing table updates, packet draining, and other network reconfiguration overheads), even though Optical Circuit Switches (OCS) do not incur per-packet processing delays for a fair comparison.

2.3.1 AllReduce with Reconfiguration

We experimented with various topology sizes and a range of *fully exposed* switching times to determine how much reconfiguration overhead these AllReduce algorithms can tolerate while still delivering performance benefits.

4x4 Torus (16 Nodes)

On the 16-node 2D torus we are running allreduce sizes ranging from 32 bytes to 512MiB.

Figure 2.2 shows the goodput gain if the links between nodes are to be reconfigured to provide a direct connection among nodes during each step of the Recursive Doubling/Swing algorithm with different reconfiguration delays. Up to 512KiB data size, reconfigurability can provide more than 50% goodput gain if the reconfiguration delay can be kept under 500ns. When the data size increases beyond 2MiB with a small number of nodes, reconfigurability offers no advantage since bandwidth cost starts to dominate.

Observation 1: On a small number of nodes, allreduce sizes up to 2MiB can see the benefit of reconfigurability only if the reconfiguration delay can be kept under 500ns.

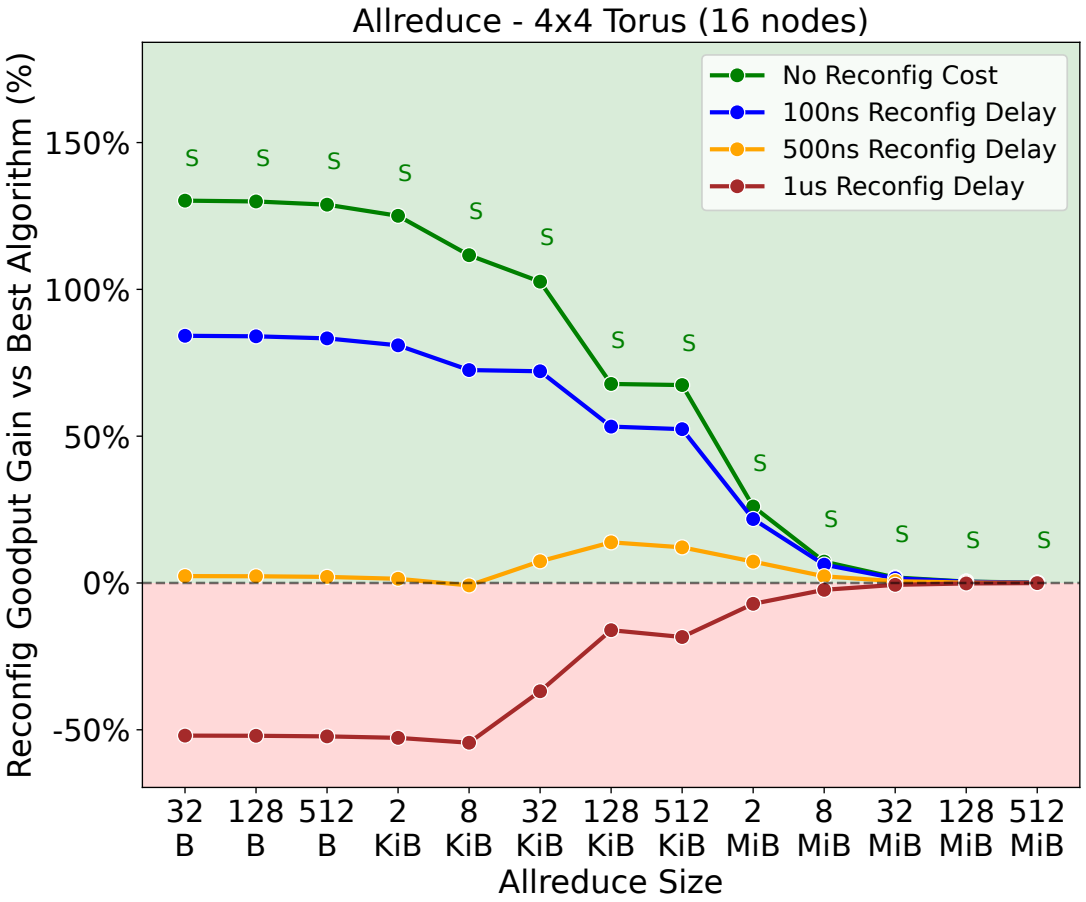


Figure 2.2. Goodput gain achievable with different reconfiguration delays on a 4x4 Torus against the best-performing algorithm (Swing)

8x8 Torus (64 Nodes)

On a 64-node Torus, up to 8MiB, Swing(L/B) has the best performance. However, for data sizes larger than 8MiB, the Bucket and the Hamiltonian Ring algorithms outperform all other algorithms. More than 100% performance gain can be achieved with reconfigurability up to data sizes 128KiB with a reconfig delay of 100ns. It should also be noted that Swing and Recursive Doubling, which perform worse than Bucket and Ring with large data sizes, can be improved with reconfigurability to get the same performance as the best-performing algorithm (Figure 2.3).

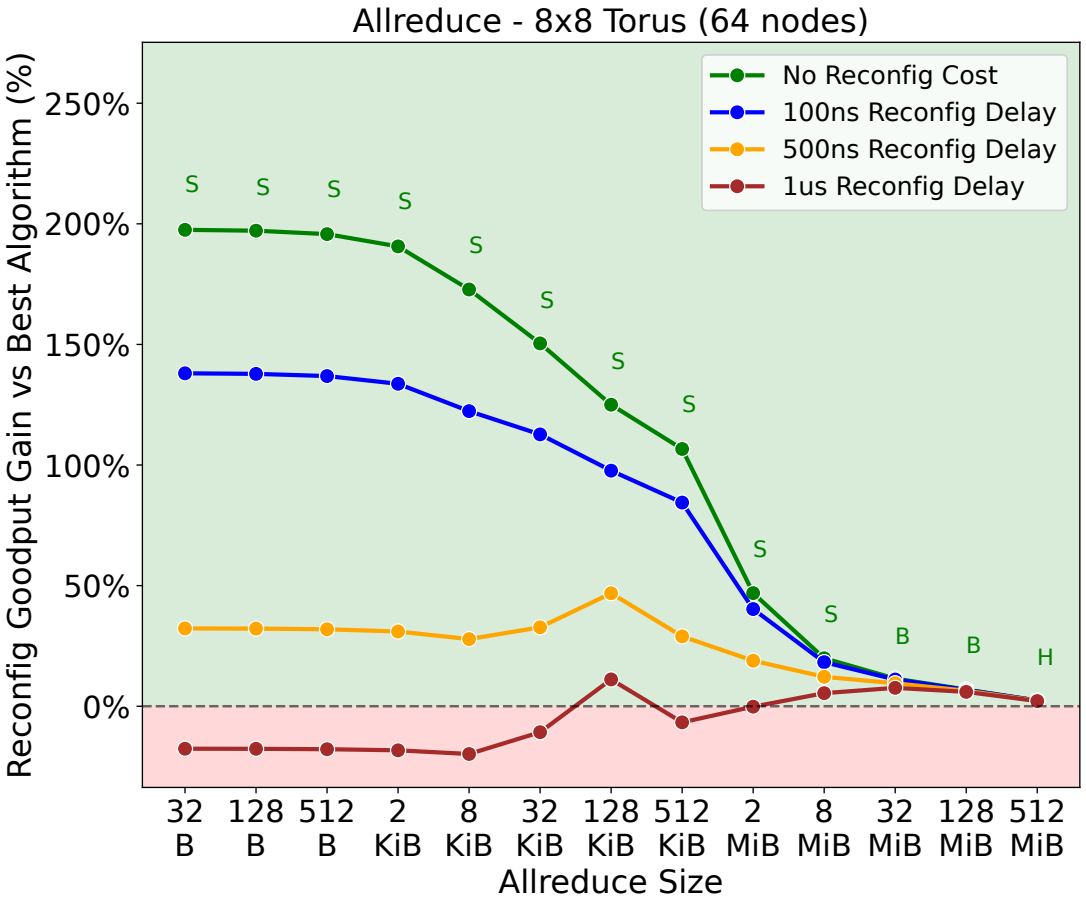


Figure 2.3. Goodput gain achievable with different reconfiguration delays on a 8x8 Torus against the best-performing algorithm

16x16 Torus (256 Nodes)

When the node count increased from 64 nodes to 256, we can see reconfig delay only needs to be less than $1\mu s$ to see any goodput gain over the best-performing algorithm (Figure 2.4).

> **Observation 2:** The flexibility of the reconfiguration delay increases with the increasing number of nodes.

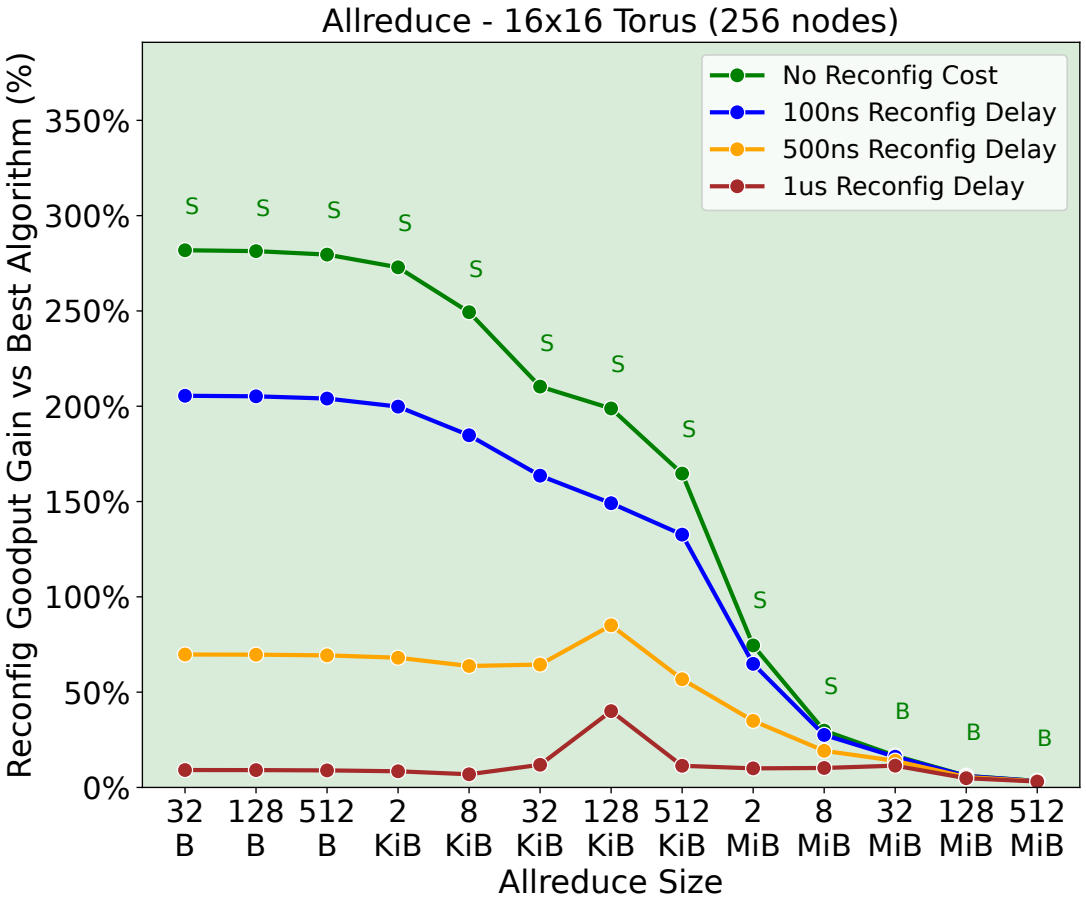


Figure 2.4. Goodput gain achievable with different reconfiguration delays on a 16x16 Torus against the best-performing algorithm

32x32 Torus (1024 Nodes) and 64x64 Torus (4096 Nodes)

For 1K number of nodes, even 1μs reconfig delay shows a 50% goodput gain up to sizes 512KiB compared to the best-performing algorithm. With 4k nodes, 1μs reconfiguration delay can improve the performance by more than 2x (Figures 2.5 & 2.6).

Observation 3: The Higher the number of nodes, the larger the reconfiguration gain.

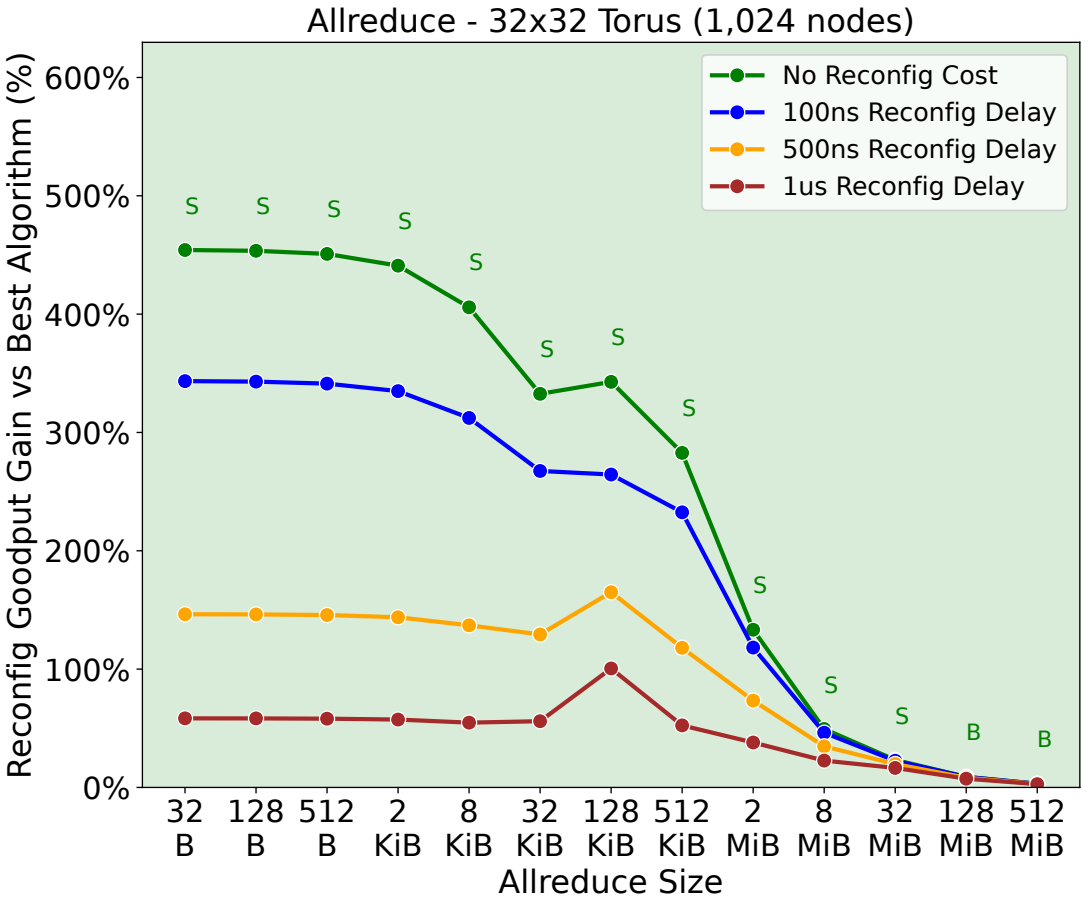


Figure 2.5. Goodput gain achievable with different reconfiguration delays on a 32x32 Torus against the best-performing algorithm

Observation 4: Regardless of the number of nodes (16-4k), allreduce sizes ranging from 32bytes - 2MiB can benefit from reconfigurability.

All the allreduce experiments that we conducted ignored the computation involved in the reduction operation. If the reconfiguration delay can hide behind the cost of computation at least partially, it will improve the performance even further, and will greatly reduce the strict low reconfiguration delay that is required by these algorithms.

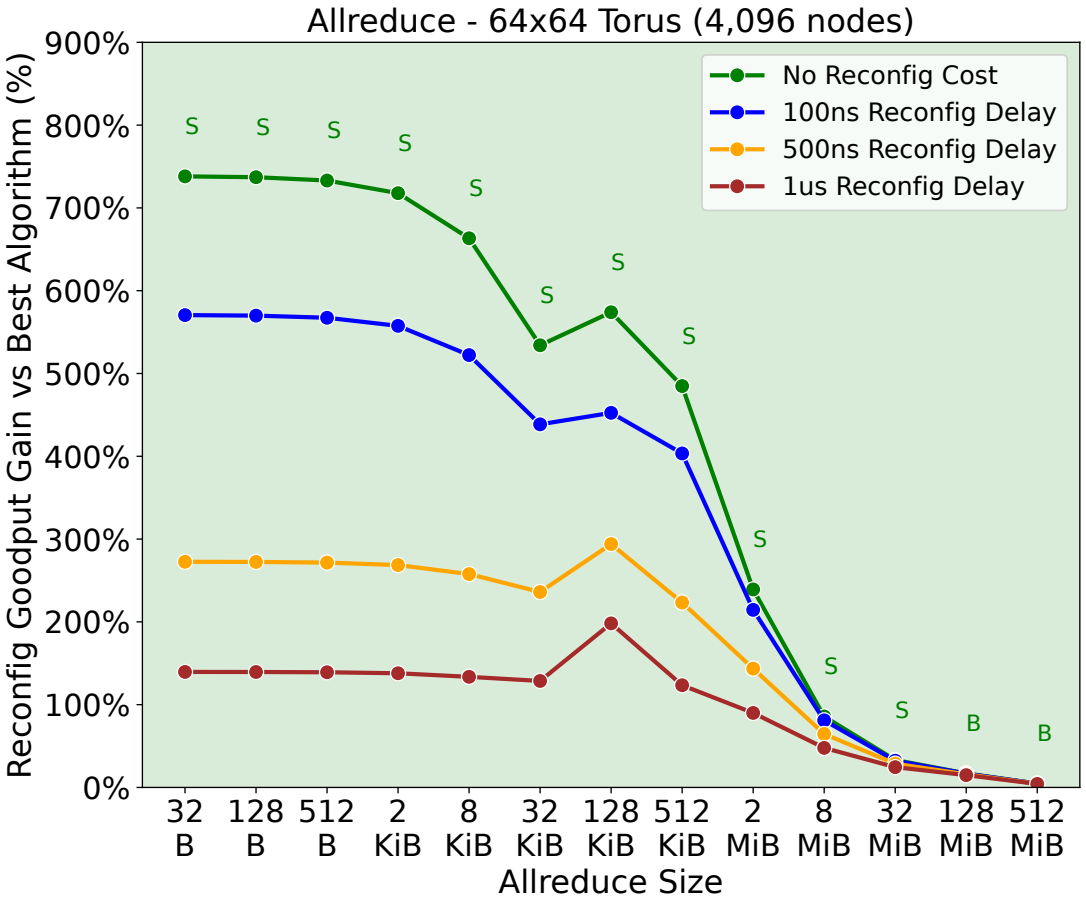


Figure 2.6. Goodput gain achievable with different reconfiguration delays on a 64x64 Torus against the best-performing algorithm

2.3.2 All-to-All with Reconfiguration

We use the all-to-all implementation of `emberMpiLib` library to gather runtime statistics for the all-to-all pattern using SST simulator [61]. Exchanged block sizes are uniform. We use throughput(number of bytes transmitted by a single NPU/T(n)) as our performance metric and run all-to-all experiment on an 8x8 torus. For reconfigurable results, we use the pair-wise exchange method in our calculation.

Figure 2.7 shows the reconfigurable goodput gain for all-to-all communication pattern. When the block size increases, congestion starts to hamper the performance of the static torus topology, and the reconfiguration can help improve the performance by ∼4x for data sizes ¿

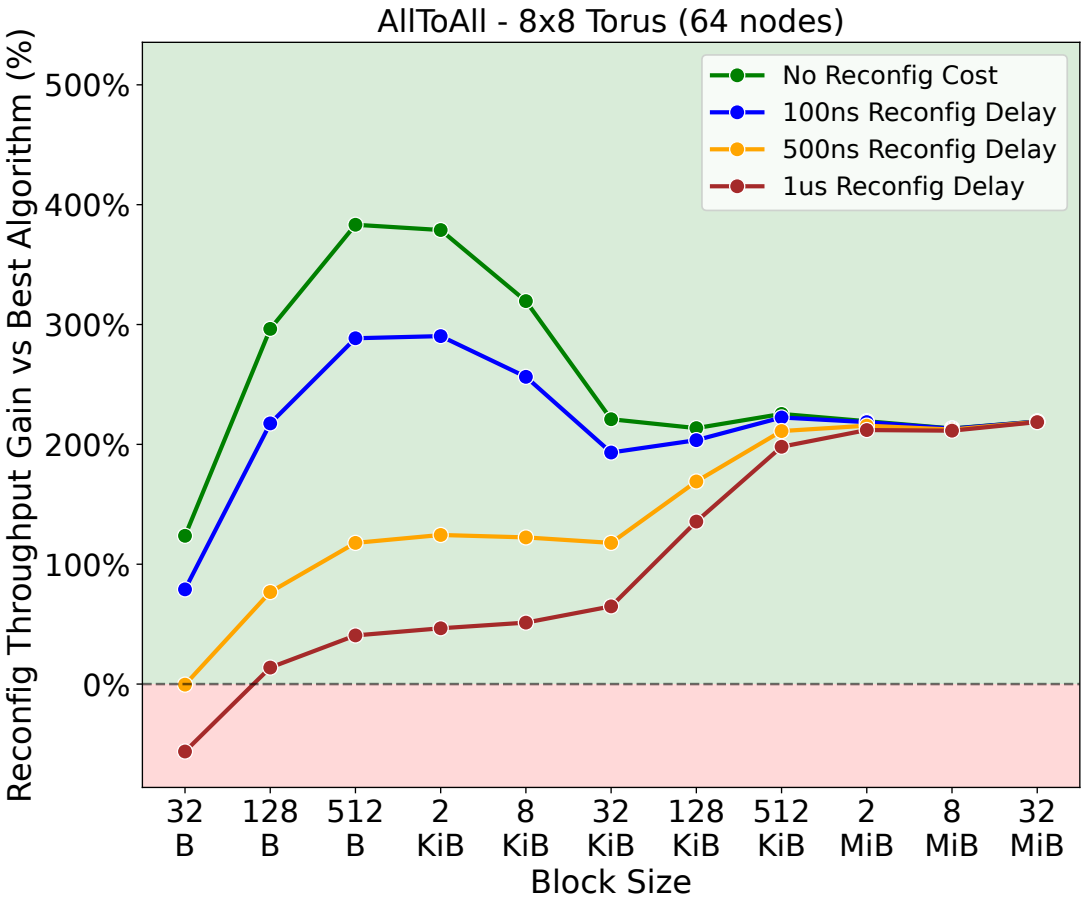


Figure 2.7. Throughput gain achievable with different reconfiguration delays with all-to-all

2MiB, even with a $1\mu s$ delay.

However, commercially available optical circuit switches today have millisecond-scale switching times, making it impossible to use OCS during a training job. However, there have been many advances in optical switching technology over the recent years to get the switching times of optics to nanosecond scale. We explore the available options in the next section.

2.3.3 Optical Switching Technology

We give an overview of the recent advances made in optical circuit switching technology in terms of reconfiguration delay and port count since there is a fundamental tradeoff between switching times and port count in commodity Optical Circuit Switches (OCS).

Table 2.1 shows that with a small port count of 16x16, 10ns reconfiguration is possible giving us 10x more gain than the 100ns delay that we use in speeding up the existing collective algorithms. However, note that the values shown represent only the switch reconfiguration time and do not include transceiver locking time, which can add additional latency before data transmission resumes.

On a 3D-Torus network, a single node at most would only need to communicate with six other nodes. If an OCS technology with a small port count but with low reconfiguration delay can be associated with each node on a torus, giving the flexibility of each port link to be dynamically changed, then the goal of reducing the runtime of the existing state-of-the-art collective algorithms for allreduce can be made a reality. A topology design work that can realize this goal in a multi-tenant machine learning fabric is still ongoing.

Table 2.1. Tradeoff between Port Count and Switching Times

Technology	Port Count	Switching Times
MEMS [52]	320x320	10-15 ms
RotorNet [43]	128x128	10 us
Silicon Photonics [3]	32x32	7 us
PLZT (EpiPhotonics) [12]	16x16	10 ns
Guided Wave [60]	16x16	nanoseconds

2.4 Conclusion and Future Work

Unlike the conventional datacenter workloads, communication patterns of ML workloads are predictable and periodic [72]. Once the parallelization strategy determines the communication pattern, the same pattern repeats at every iteration during the entire training job [51]. If the reconfiguration delay can be kept within nanosecond range, the communication pattern can be mapped to a circuit switch matching at every step of the algorithm. How the OCS technology will be integrated into the topology is an interesting question that we currently explore.

There is another advantage of reconfigurability that needs no consideration of switching times. When a DNN model is run on a 3D torus, typically one dimension is used for data

parallelism, and the other two dimensions are used for model parallelism and vice versa [37]. Communications that result in data and model parallelism occur at different times, resulting in 1D, 2D, and sometimes 3D logical communications during training. These 1D and 2D logical communications typically cannot fully utilize all the torus links. Therefore, if we allocate the unused links to the dimension (direction) where the current communication is happening through reconfiguration, a considerable speedup of the runtime can be achieved. Since the reconfiguration delay can be hidden during computation time, even the commodity OCS switches with high switching times can be used for this purpose, depending on the computation time.

Another advantage of reconfiguration is its ability to change the shape of the torus slices. Resource allocations and scheduling decisions on large-scale datacenters can result in having to run training and inference jobs on imbalanced topology slices [27], and it can lead to poor performance in collective operations. Having different dimension sizes on a torus increases the Swing and Recursive Doubling algorithms' congestion compared to a square torus [53]. Imbalanced toruses can also hurt the latency of the bucket algorithm if the dimension sizes are different [53]. If, through reconfiguration, the shape of the topology slice can be changed prior to running the training or inference, the runtime can be saved considerably. Providing the ability to reconfigure a topology slice that is allocated to users opens up many opportunities for optimizing the training and inference jobs, increasing the utilization of expensive NPUs.

In this work, we considered reconfiguring links at every step of the algorithm to gain maximum performance. However, if we select the n number of steps that cause the most congestion for reconfiguration, then we still might gain some performance benefits but with a flexible reconfigurable delay. We will analyze the tradeoff between the performance gain and the reconfigurable flexibility in future work.

Another well-known advantage of reconfigurability is the ability to route around failures, increasing the availability. Performance degradation that happens due to fragmentation can also be eliminated if the congested links can be isolated using reconfiguration. We plan to explore all these options in our future work.

Acknowledgments

Chapter 2, in full, is a reprint of the paper ”Reconfigurability within collective communication algorithms” published in the 2nd Workshop on Networks for AI Computing, NAIC ’25, page 43–49, New York, NY, USA, 2025. Rukshani Athapathu and George Porter. Association for Computing Machinery. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Flexible Torus Clusters: Less Fragmentation, Lower Communication Overhead

With the explosive growth of Deep Neural Network (DNN) workloads, we now have DNN models with billions and trillions of parameters [14]. As a result, there have been massive investments and rapid developments in ML hardware accelerators such as GPUs and TPUs and their interconnects to efficiently run these models [47, 16]. On the other hand, conventional datacenter clusters such as clos topologies are becoming a bottleneck for distributed DNN training jobs [10, 69]. Hence, large scale cloud providers have started to use different fabrics such as rings and toruses to better support these DNN workloads [29, 72, 16]. Reconfigurable networks are also on the rise due to its low power consumption and flexibility. Reconfigurable networks allow us to dynamically rearrange links over time using optical circuit switches (OCS) making the topologies much more flexible than traditional fixed link networks. Existing reconfigurable network designs such as TPUv4 [16] and TopoOpt [72] uses one shot reconfiguration to optimize the topology for ML workloads. Athapathu et al. [5] also analyzes the state of the art collective communication algorithms that are optimized for torus-like topologies, and further discusses how a reconfigurable layer can improve the performance of these large-scale collectives that underpin DNN training.

Many large cloud providers use various torus variants in their machine learning clusters (Google TPU, AWS Trainium, etc.). This is mainly due to the fact that overall composition of

communication patterns in ML workloads forms a torus. Therefore, in this paper, we limit our analysis to 3D torus networks. The aim of this paper is twofold. 1) Understand the extent of fragmentation in a reconfigurable torus network 2) Lower the communication overhead of a torus slice through reconfiguration. We explain these two objectives as follows.

Resource allocations in torus variants in a multi-tenant environment often result in fragmentation, causing lower system utilization and long queue times. There are two types of fragmentation that can occur in a torus: internal and external. Internal fragmentation is the result of an allocating strategy where more nodes are allocated to a job than it needs (Figure 3.1). External fragmentation, on the other hand, is caused by a contiguous job allocation strategy that makes free system resources to be scattered across the cluster in small blocks. This result in a situation where the available free blocks cannot be contiguously allocated for a job, lowering system utilization (Figure 3.2). To make use of the scattered nodes, if a non-contiguous allocation strategy is used, then the communication interference can degrade the job performance. Therefore, in this work, we investigate how feasible it is to achieve the best of both worlds (lower communication overhead and reduced internal/external fragmentation) by using reconfigurability in torus networks. Our measurement study will help understand the extent of fragmentation in torus clusters so that better allocation strategies/new topology designs can be formulated in future work.

3.1 Problem Statement

Fragmentation is a common phenomenon that occurs when multiple jobs contend for resources in a cluster. In a TPUv4-like cluster, tenants cannot request resources of any topology size or shape due to constraints in the allocation rules [17]. For example, if a tenant only needs 100 TPU chips, they need to allocate 128 chips (4x4x8 slice), wasting 28 TPUs (internal fragmentation). However, it doesn't show up as fragmentation since the tenants have no option but to allocate more TPUs than they need, since the shapes that can be allocated have specific

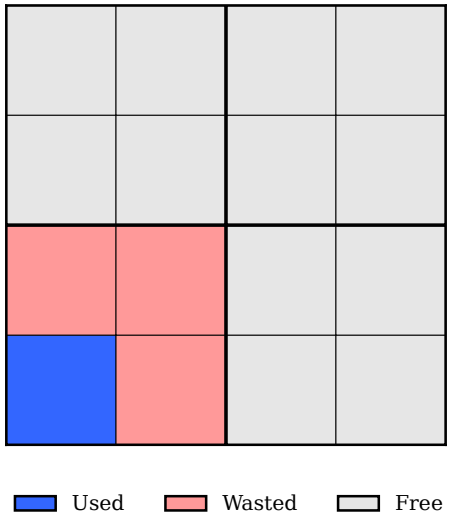


Figure 3.1. Internal fragmentation: a node (4 compute units) is allocated to a job, leaving most of its capacity unused.

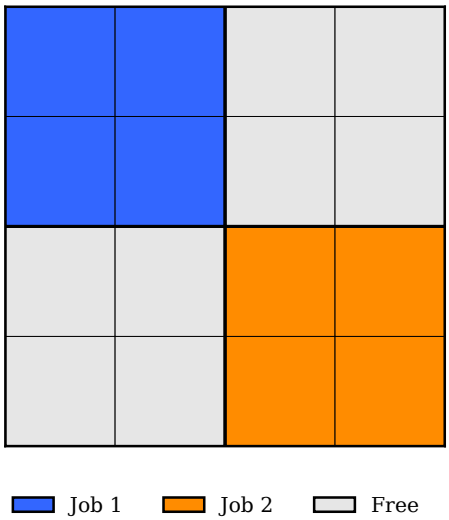


Figure 3.2. External fragmentation: a 4×2×1 request cannot be allocated due to lack of contiguous free resources.

rules: mainly the nodes are allocated strictly on multiples of four on all three dimensions of a torus [17]. Another problem is the external fragmentation that is caused by a contiguous resource allocation strategy. Even though this is crucial to reduce congestion and communication overhead, it degrades the system utilization.

We now describe the fragmentation metrics that we use throughout the paper.

3.1.1 Fragmentation Metrics

Fragmentation metrics tell us how inefficiently the cluster is being used. We measure both the internal and external fragmentation, as well as the failed allocation rate. While internal/external fragmentation is important to understand the cluster inefficiency, failed rate allocation is important to figure out how badly that cluster inefficiency can affect the job allocation. We calculate these metrics as follows.

Internal Fragmentation

Internal fragmentation reflects the average wasted capacity of the cluster. This is a weighted average of waste, where weights are proportional to job size. For example, if job A requires only five nodes but ten nodes are allocated, then 5 nodes are wasted (50%). If job B needs 95 nodes but 100 nodes are given, then the waste percentage of that job is 5%. Therefore, we calculate the global system-wide waste fraction as $sum(Allocated - Requested)/sum(Allocated)$: $10/110 = 9.1\%$. We use this as our internal fragmentation metric, which answers the question *"What fraction of the system's capacity is wasted?"*

External Fragmentation

External fragmentation indicates the fragmented fraction of free space. In other words, it answers the question *"How much of the free space is wasted because it is scattered as small, non-contiguous blocks?"* It indicates the severity of fragmentation among the remaining free resources. We calculate external fragmentation as $1 - (MaximumAllocatableCuboid/TotalFreeTPUs)$. If the total number of free TPUs can form a single contiguous cuboid then there is no external fragmentation $(ExternalFragmentation = 0)$. If all the free TPUs are scattered across the cluster then we say the cluster is experiencing high external fragmentation $(TotalFreeTPUs - 0)/TotalFreeTPUs = 1$.

Failed Allocation Rate

The failed allocation rate measures the percentage of jobs that are being denied. It tells us how often our allocator cannot satisfy job requests compared to the total number of requests $(FailedAllocations/TotalRequests)$.

First part of the paper is focused on measuring the internal and external fragmentation as well as the failed rate allocation of a simulated block-wise reconfigurable cluster that is described in section 3.3. We then developed a synthetic TPU distribution with a mix of irregular values in the requested TPU slices to measure the true internal fragmentation. We also compute the fragmentation based on real-world traces on our simulated cluster. Finally we show that a node-wise reconfigurable cluster can reduce fragmentation even with a TPU slice distribution where the nodes are allocated strictly on multiples of four on all three dimensions. In the second part of the paper, we look at how the individual job performance of a torus slice can be improved through reconfiguration.

3.2 Related Work

Recently, there has been an increase in interest in using dynamically reconfigurable physical topologies in datacenters [72, 27, 29, 6]. Morphlux [31] is one such proposal where the accelerators within a server are connected using photonic links instead of traditional electrical links. They also focused on fragmentation unlike the other papers but only consider slice sizes smaller than a rack (4, 8, 16 and 32). It is understandable because resources are fragmented by slices when they are smaller than a rack. But constraints on compute allocation sizes (be it large or small) hides the true indication of internal or external fragmentation which is the focus of our paper. There is a separate body of research [33, 11, 34] dedicated to scheduling and job packing/placement algorithms to reduce internal and external fragmentation in torus networks. Li et al [33], propose a packing-based job scheduling strategy and a job placement reconfiguration algorithm to reduce the external fragmentation. Bin packing [28, 68] is also an efficient way to

manage resources in torus clusters. However, bin packing solutions are NP hard. Our goal in this paper is to investigate how the flexibility of a dynamically reconfigurable topology can help reduce fragmentation and improve individual job performance in torus clusters.

3.3 System Architecture

The simulator we have developed for this paper, is based on the TPUv4 machine learning supercomputer, which features a 3D torus interconnect [76]. TPUv4 resources are organized as 4×4×4 cubes (A cube is referred as a block in this paper), and each block is equipped with 16 TPU trays. A single TPU tray (a node) has four TPU chips organized as a 2×2×1 mesh. All links are fixed within the block. However, each block face exposes 16 reconfigurable links, making it possible to merge multiple blocks to form larger topology shapes (Figure 3.3).

TPUv4 supercomputer supports various TPU slices [27]. TPU slices smaller than 64 chips (a block) can only support 2D meshes, and the smallest number of chips that can be allocated is four, according to the cloud TPU slices reported in TPUv4 architecture [17]. However, allocating a single TPU chip within a node seem possible according to TPUv4 paper [27]. In this paper, we assume a cloud TPUv4 architecture hence the scheduler does node-wise allocations in our experiments. When more than 64 chips are needed, topology shape will end up being multiples of four on all three dimensions.

3.3.1 Simulator Design

In this section, we describe our simulator that mimics a TPUv4-like machine learning supercomputer. A 2D visualization of a block is depicted in Figure 3.4. This is the same block that is shown in figure 3.3. Each layer in 3D figure is lined up horizontally so that all the TPUs in each layer are visible. A cluster is made up of a requested number of blocks during cluster initialization. The simulator supports both large and small slices described in TPUv4 paper [27]. In this paper, we assume that our cluster nodes are homogeneous and bandwidth is uniform across both reconfigurable and fixed links. Our experiments also only focus on TPU chips as our

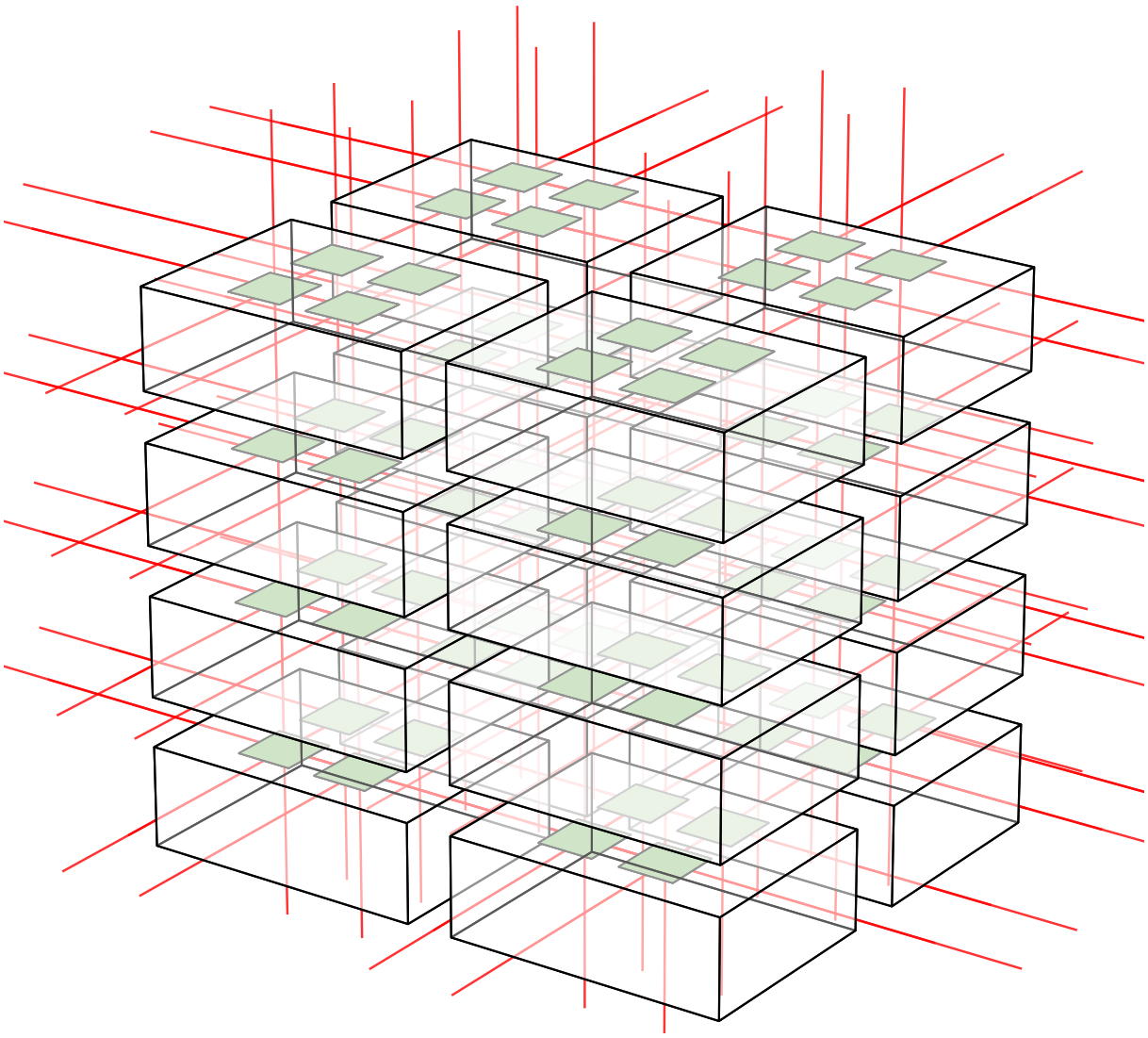


Figure 3.3. 3D visualization of a 4×4×4 block. A single TPU tray (a node) has four TPU chips organized as a 2×2×1 mesh.

primary resource for allocation.

3.3.2 Node Allocation Rules

We draw job samples from the TPU slice distribution provided in the TPUv4 paper [27]. Allocation requests are sent to the cluster in (x·y·z) form. For example, if a tenant wants 8

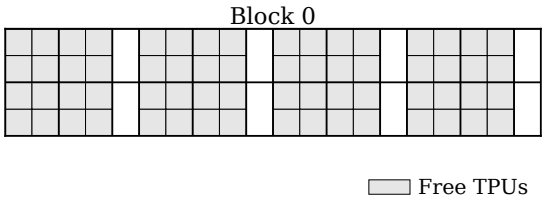


Figure 3.4. 2D visualization of a 4×4×4 block that is depicted in Figure 1. Each layer in 3D figure is lined up horizontally so that all the TPUs in each layer are clearly visible.

TPU chips, the allocation request can be made as (4×2×1). The slice shape requested by the tenant is preserved by the node-wise allocator. We use a contiguous node allocation strategy, and assume that the faces of the blocks are reconfigurable. We refer to this cluster as a *block-wise reconfigurable cluster*.

Node allocator for the *block-wise reconfigurable cluster* follows two rules: 1) When a request is made, allocator always tries the single block fit first: trying to find a node cuboid within a block. 2) If the requested shape cannot be preserved using a single block, then the allocator stitches reconfigurable faces along the x, y, and z directions. Cross-unit anchor alignment is not enforced with face stitching. However, if the face-stitching fails, the allocator falls back to general multi-block contiguous merge strategy where we first build a virtual Bx·By·Bz layouts of blocks and searches for any contiguous cuboid in that virtual coordinate space. Allocator also searches for multiple block-grid layouts (Bx·By·Bz) allowing rectangular (kx·my·nz) block arrays and not just near-cubic arrangements. Figures 3.5, 3.6 and 3.7 depict the above mentioned rules.

However, it is important to note that even though the simulator supports allocation of any shape with both even/odd values in all three dimensions according to node-wise allocation rules, for our experiments in this paper, job shapes are drawn from the TPU slice distribution [27] (where the shape of a slice has multiples of four on all three dimensions when the requested number of nodes exceed 64 chips) unless otherwise noted.

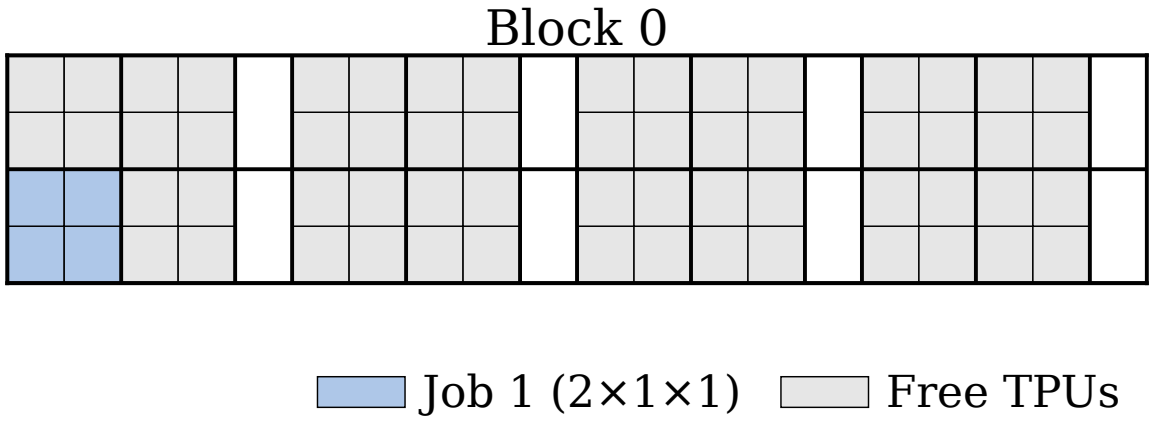


Figure 3.5. Requested shape: (2×1×1); Single Best Fit Strategy (Due to node-wise allocation, two more nodes are allocated to Job 1 than it needs, causing internal fragmentation.)

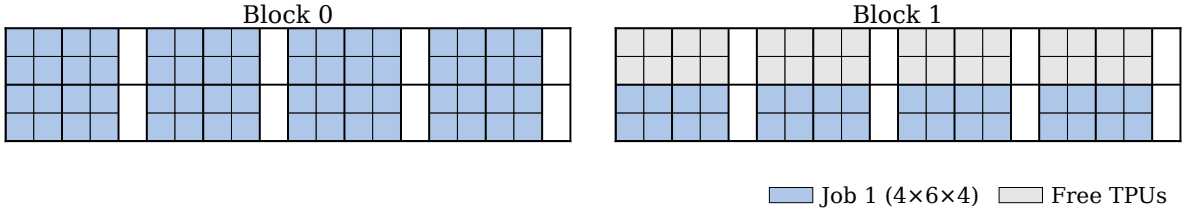


Figure 3.6. Requested shape: (4×6×4); Face Stitching: Stitches face along y-direction

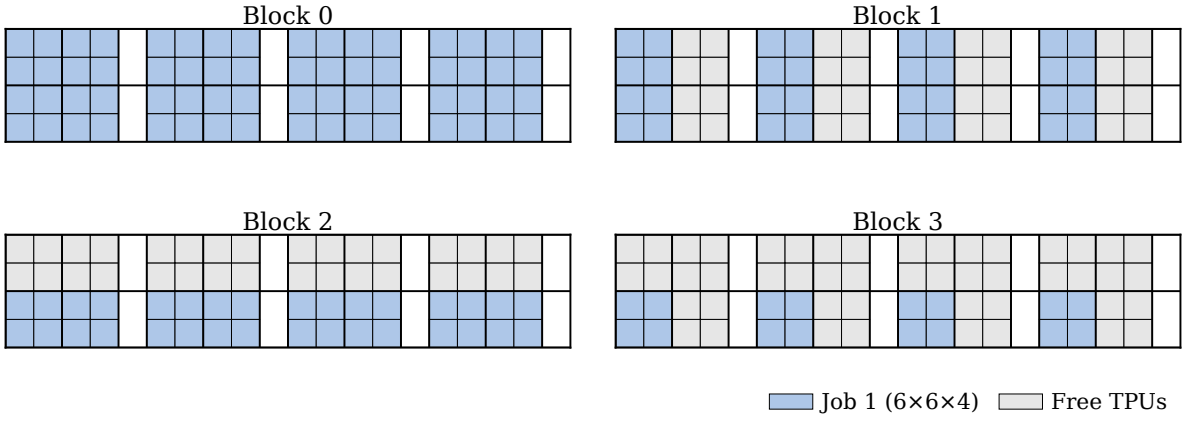


Figure 3.7. Requested shape: (6×6×4); General Multiblock Strategy

Table 3.1. Service Time Distribution

Job Size	Service Mix	Explanation
Short	(0.6, 50)	60% of jobs last 50 arrivals
Medium	(0.3, 200)	30% of jobs last 200 arrivals
Large	(0.1, 1000)	10% of jobs last 1000 arrivals

3.4 Experimental Design

To produce realistic allocation and deallocation of jobs, we drive the simulator with job arrivals and service times. Deallocation of jobs is determined by the service time distribution. It indicates how long a job runs before it leaves the system. In the simulator, time is measured in the number of job arrivals. For example, if the service time is 200, then that job will get deallocated 200 arrivals after it was created. We use a simple but a realistic service time distribution (Table 3.1) where we have many short tasks and few long running jobs. This captures the well-known skew in deep learning clusters where many short experiments coexist with fewer long-running training jobs [74]. This discrete mixture approximates the heavy-tailed nature of job durations and is commonly used in system modeling [20]. Therefore in our experments, each job gets a random service time drawn from this mix and when simulation time reaches that expiration, the job gets deallocated.

We use an adaptive job arrivals to achieve and maintain a target utilization level (eg, 10%, 70%, etc) since the fragmentation changes based on the utilization. Arrivals follow a Poisson process, and to control the utilization at a specific level, we tune the arrival rate (λ). For example, every 50 steps, the simulator checks the current utilization and adjusts the λ up if the utilization is below the target level or down if it is above the target utilization. When the simulation starts, it first enters the warm-up stage. Warm-up ends adaptively when utilization stays within ±5% of target for ten consecutive checks. The snapshots of the cluster state are then collected every 20 steps when the utilization is within ±2% of target. Figure 3.8 shows this process in detail. For each experiment, we collect 500 snapshots for analysis.

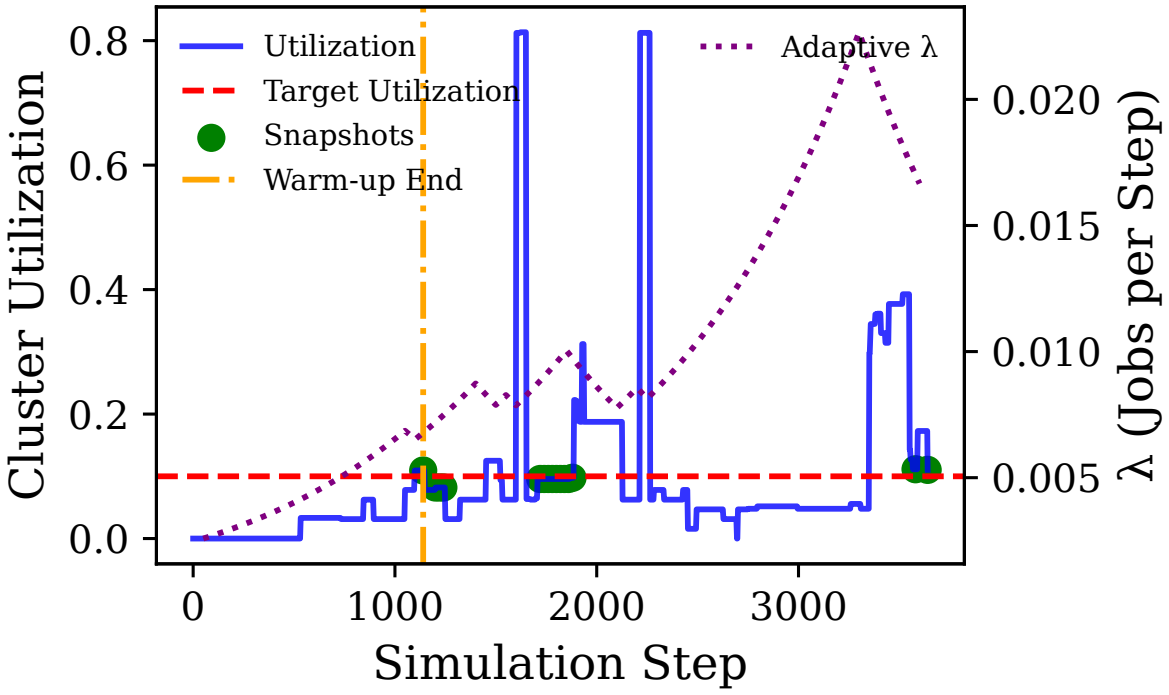


Figure 3.8. Snapshot Process: Tune arrival rate to maintain target utilization. The snapshots of the cluster state are collected every 20 steps when the utilization is within ±2% of target.

We slightly adjusted the original TPU slice distribution reported in the TPUv4 paper[27] to ensure that the total percentage sums to 100%. Table 3.10 (first column) shows the slice distribution we used.

3.5 Evaluation

We report the internal and external fragmentation as well as the failed allocation rate for various utilization levels (20%, 50%, 75%, and 90%). A total and 500 snapshots were collected for each experiment and the y-axis shows the fragmentation index for each snapshot.

3.5.1 Internal Fragmentation

Figure 3.9 shows the internal fragmentation for a 64 block cluster. Internal fragmentation is insignificant across all utilization levels because internal waste can only come from (1×1×1) and (1×1×2) slices. Moreover, (1×1×1) and (1×1×2) slices only make up about 2.5% of the

workload distribution contributing to this low level of internal fragmentation (Table 3.10: TPUv4 Distribution; First Column). Therefore, this result is expected. If we do not have (1×1×1) or (1×1×2) in the slice distribution, we might not see internal fragmentation at all.

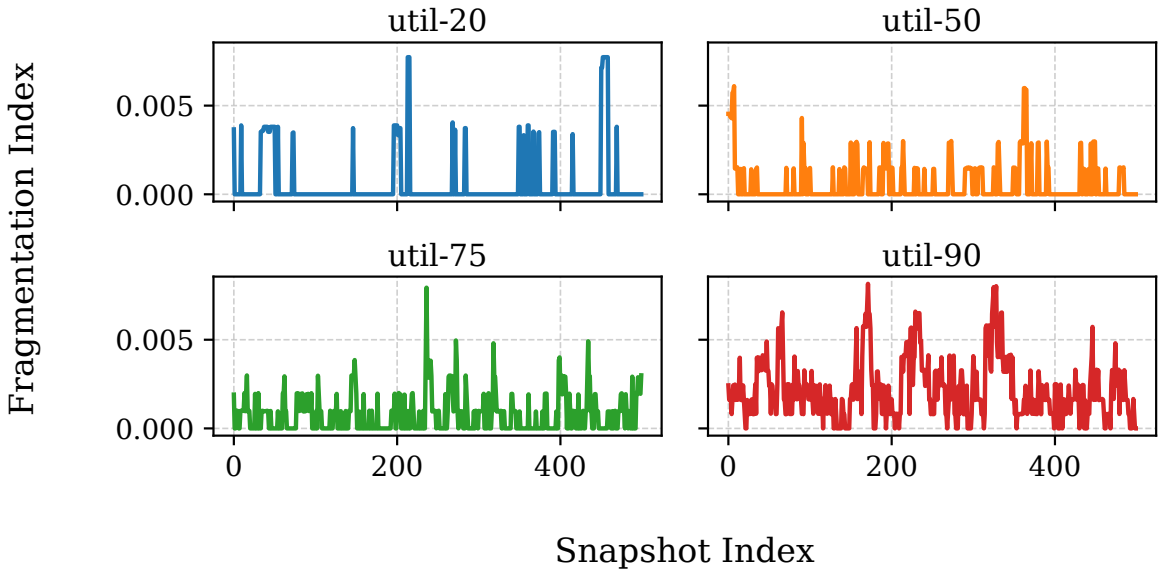


Figure 3.9. Internal fragmentation across different utilization levels for the block-wise reconfigurable cluster (64 blocks in the cluster)

3.5.2 External Fragmentation

Figure 3.10 shows the external fragmentation. As expected, fragmentation remains low (˜2%) at low levels of utilization. However, when utilization reaches 90%, fragmentation can occasionally rise to nearly 100%. Table 3.2 reports the average fragmentation across 500 snapshots. We observe that at 90% utilization, on average 59% of the free space is wasted because it is scattered into small blocks. As a result, the allocator is unable to satisfy a considerable number of job requests as discussed in the following failed allocation rate metric.

3.5.3 Failed Rate Allocation

As explained above, the allocation failure rate (Figure 3.11) follows the same trend as external fragmentation across different utilization levels. As external fragmentation increases,

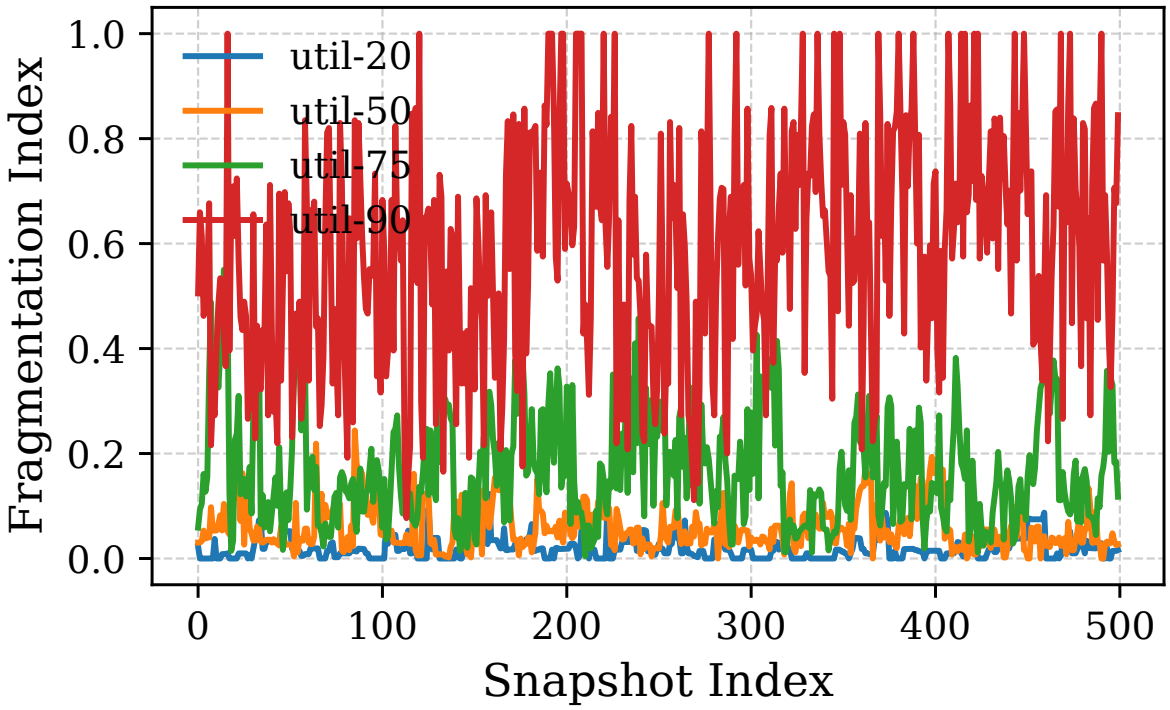


Figure 3.10. External fragmentation across different utilization levels for the block-wise reconfigurable cluster (64 blocks in the cluster)

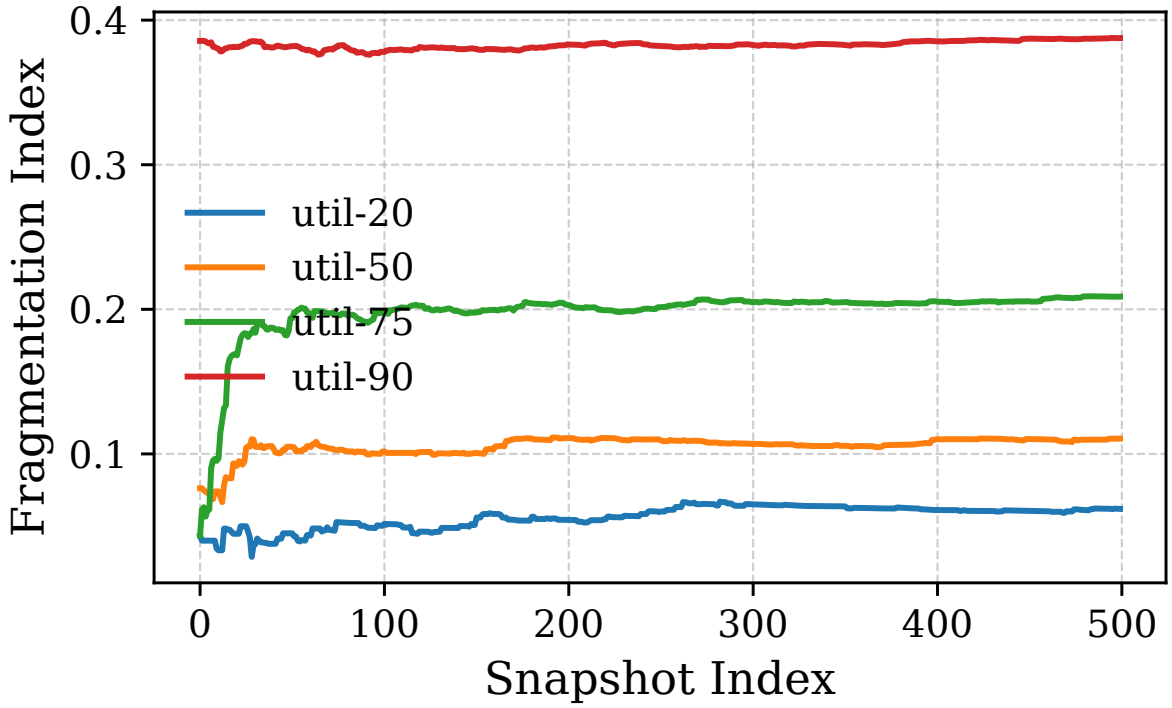


Figure 3.11. Failed allocation rate across different utilization levels for the block-wise reconfigurable cluster (64 blocks in the cluster)

the job scheduling failure rate also rises. At 90% utilization, on average ˜38% of job requests cannot be satisfied (Table 3.2).

Observation 1. As the external fragmentation increases, the job scheduling failure rate increases correspondingly. At 90% utilization, a 64-block block-wise reconfigurable cluster wastes, on average, 59% of its free space due to fragmentation, causing approximately 38% of job requests to go unsatisfied.

3.5.4 Measuring True Internal Fragmentation

Now that we have a baseline for fragmentation matrics, we attempt to measure true internal fragmentation. Internal fragmentation we measured previously does not reflect the true value because in the original TPU distribution [27] most of the TPUv4 slices are available only in increments of 64 chips, with shapes that are multiples of four on all three dimensions. Tenants

Table 3.2. Average fragmentation index across 500 snapshots for a block-wise reconfigurable cluster (64 blocks in the cluster)

Util. (%)	Avg. Internal Frag. Index	Avg. External Frag. Index	Avg. Failed Rate
20	0.000553	0.020079	0.056318
50	0.000574	0.059695	0.105441
75	0.000791	0.171978	0.197930
90	0.002165	0.589660	0.382770

have no option but to pick slices according to this rule. However, when the slices are smaller than a block, TPU slices are available in [1, 2, 4, 8, 16, 32] sizes. As mentioned previously, if a tenant wants 100 TPUs, the tenant has to pick 128 TPUs (2 blocks) for the job, and waste 28 TPUs. But that is not visible to the cluster manager since the tenant picks slices according to the block rules. Therefore, to investigate the true internal fragmentation, we developed a synthetic distribution with a blend of odd/irregular values in the requested TPU slices. Maximum TPU size still follows the original distribution and the synthetic distribution still has a mix of small, medium, and large workloads (Table 3.10, second column). CDF comparison of original and synthetic slice distribution is shown in Figure 3.12.

With synthetic experiments, once a sample is taken from the synthetic distribution, we map the synthetic shape to the closest original slice. If an exact match is not found in the original slice distribution, we pick the slice with $product(xyz) >= RequestedCount$ that minimizes $(product - RequestedCount)$. If multiple candidates have that product, we pick the most cube-like shape. This way, the synthetic distribution drives the sampling, but all shapes used in the simulation will always be from the original slice space. With the original TPU distribution, we only have information about the picked cuboid and the actually allocated shape. So the internal fragmentation is based on $(Allocated - PickedCuboid)$. With synthetic distribution, we have information about the actual number of TPUs a tenant needs $(RequestedTPUCount <= PickedCuboid <= Allocated)$. So now the internal fragmentation can be calculated as $(RequestedTPUCount - Allocated)$.

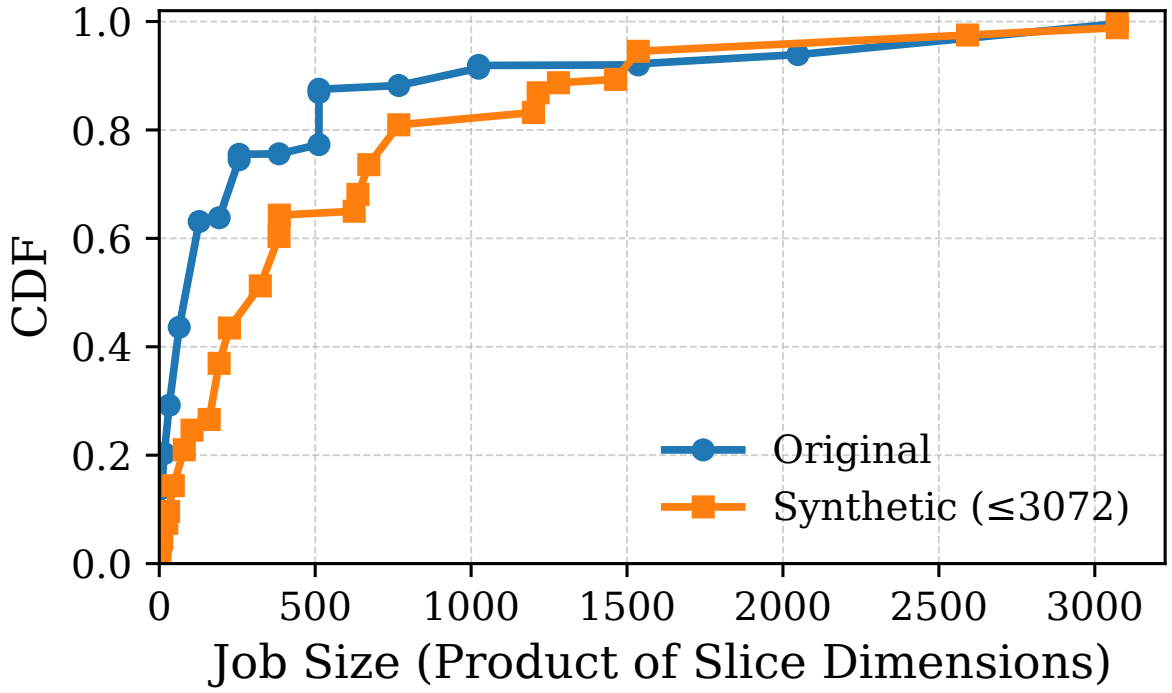


Figure 3.12. CDF of slice distribution: The synthetic distribution approximates TPUv4 closely, though it includes irregular slice requests absent in the original.

Figure 3.13 compares the original internal fragmentation vs true internal fragmentation at 90% utilization. We can clearly see that the true internal fragmentation is much higher than the originally reported index. Table 3.3 shows the average true internal fragmentation across all utilization levels. According to the results we can clearly see that at least 10% of the allocated nodes are wasted regardless of the utilization level.

Table 3.3. Average internal fragmentation across 500 snapshots for a block-wise reconfigurable cluster (64 blocks in the cluster

Utilization (%)	Original Frag.	True Frag.
20	0.000417	0.099271
50	0.000240	0.111309
75	0.000667	0.113766
90	0.002334	0.151100

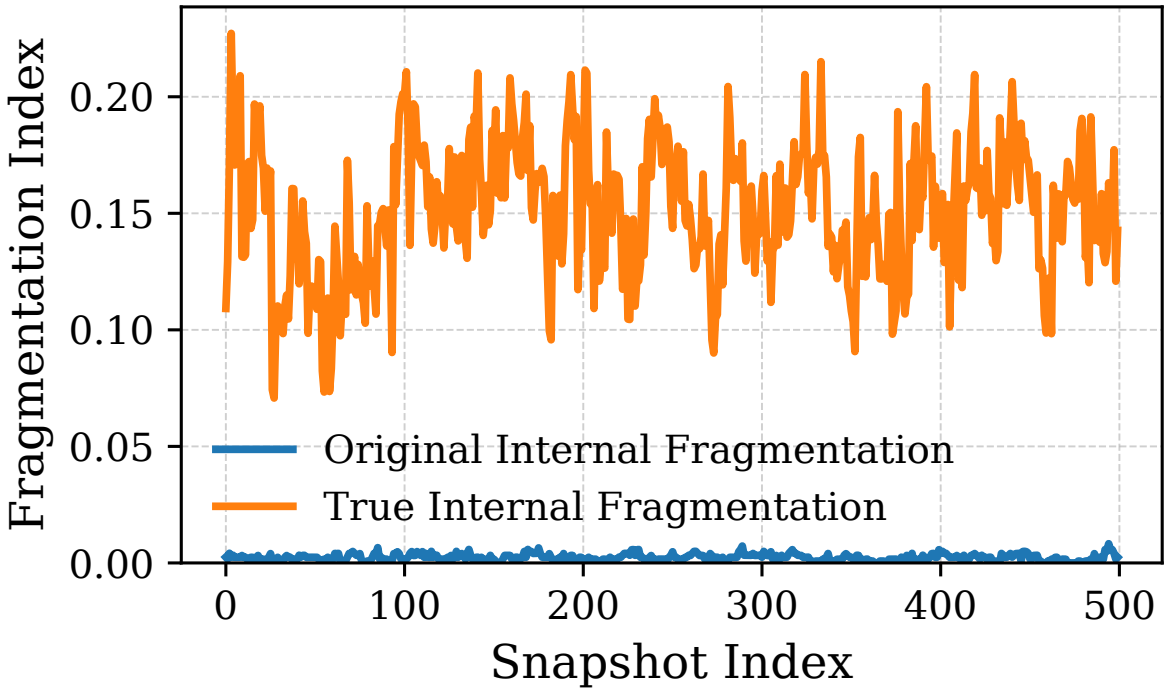


Figure 3.13. True internal fragmentation of a block-wise reconfigurable cluster at 90% Utilization (64 blocks in the cluster)

Observation 2. Internal fragmentation persists across all utilization levels: at least 10% of the allocated nodes remain unused, regardless of the cluster's utilization.

3.5.5 Real World Traces

Next, we compute the fragmentation based on real-world traces taken from the Helios paper [24]. Helios [24] has traces for four independent GPU clusters called Earth, Saturn, Uranus, and Venus. These clusters are composed of NVIDIA Pascal/Volta GPUs and the GPUs are interconnected via a hierarchical network. Despite the difference in the topology, we used the traces with our simulation since it provides rich characteristics about resources requirements for deep learning jobs. Traces include a total of 1.58M jobs and their characteristics (Table 3.4).

From the trace data, we first build a probability mass function (PMF) for each cluster. Then we build a per-cluster GPU job size samplers and use it to generate synthetic workloads

Table 3.4. Number of Deep Learning jobs handled by each cluster in Helios [24]

Cluster	Number of Jobs
Earth	427148
Saturn	698896
Uranus	329117
Venus	125303

Table 3.5. Examples of Shape Selection for Helios Traces: Convert the GPU count to a tori shape (x·y·z) and find the tori shape from the existing TPUv4 distribution

GPU Count	Tori Shape	Allocated
63	4×4×4	64
65	4×4×8	128
128	4×4×8	128
129	4×4×12	192

that match the empirical distribution of GPU counts across the four clusters. Since we only get a GPU count from the sampler, we first need to convert the GPU count to a tori shape (x·y·z). We try to find the exact tori shape from the exisiting TPUv4 distribution that matches the GPU count. If that is not feasible, we pick the slice with the $product >= gpu\_count$ that minimizes $(product - gpu\_count)$. If multiple candidates have the same product, we pick the most cube-like shape. Few examples are given in table 3.5.

With real-world traces we have information about the actual number of GPUs a tenant need. The CDF of the requested GPU count for each cluster is shown in Figure 3.14. So now the internal fragmentation and wasted capacity can be calculated $(OriginallyRequested - ActuallyAllocated)$. Calculation for external fragmentation remains the same.

Figure 3.15 shows the internal fragmentation result for each cluster. Earth cluster has the highest internal fragmentation due to 90% of its jobs requiring only just one GPU wasting a significant amount of resources. This is because our allocator assigns atleast four TPUs to each job. This shows the importance of resource sharing within a single node among many workloads. Just as in the previous experiments real cluster experiment also shows that higher utilization lead

to higher fragmentation since most blocks are fragmented.

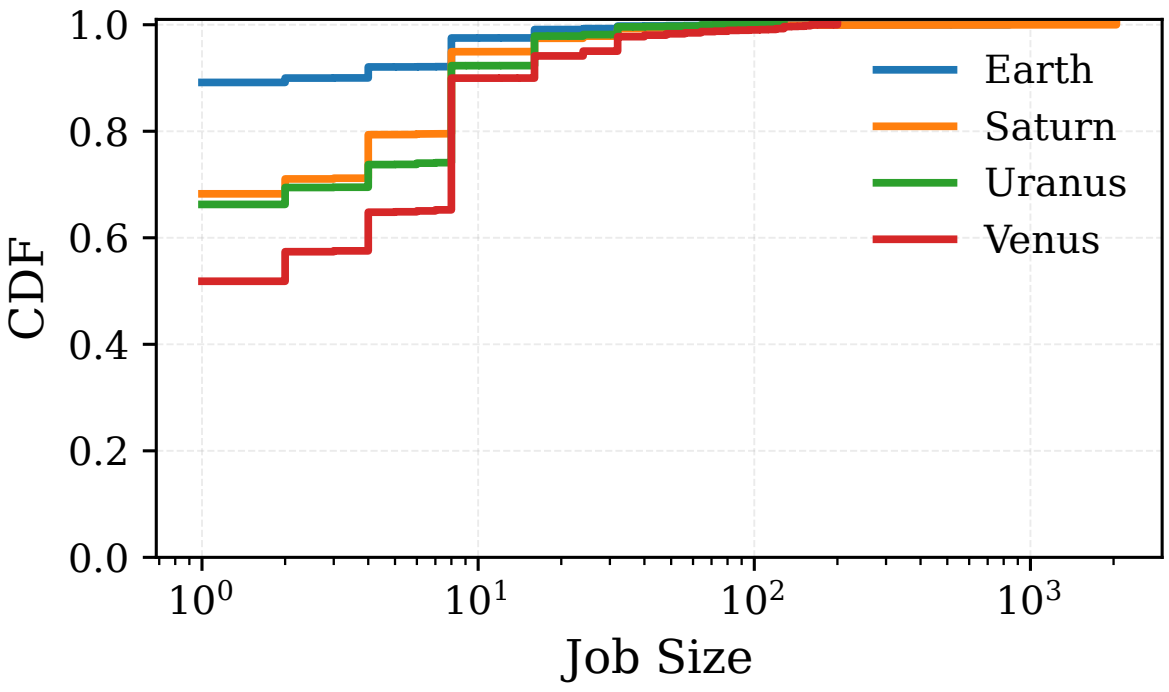


Figure 3.14. CDF of the requested GPU count for each cluster

Observation 3. Real-world traces highlight the importance of enabling resource sharing within a single TPUv4 node across multiple workloads to reduce internal fragmentation, as well as adopting allocation strategies that facilitate such sharing.

3.5.6 Node-wise Reconfigurable Cluster

Next, we explore how increasing the flexibility of a block-wise reconfigurable cluster can improve system efficinecy. In this section, we show that instead of only allowing block faces to be reconfigurable if we make all node faces to be reconfigurable allocation can be done much more efficiently reducing fragmentation. However, it is important to note that the TPU chips within a node still has fixed links. We refer to this cluster as a *node-wise reconfigurable cluster*.

With a node-wise reconfigurable cluster, we first try contiguous multi-axis recursive stitching which we call the ”best-fit”. That is, the allocator tries to fit shapes contiguously across

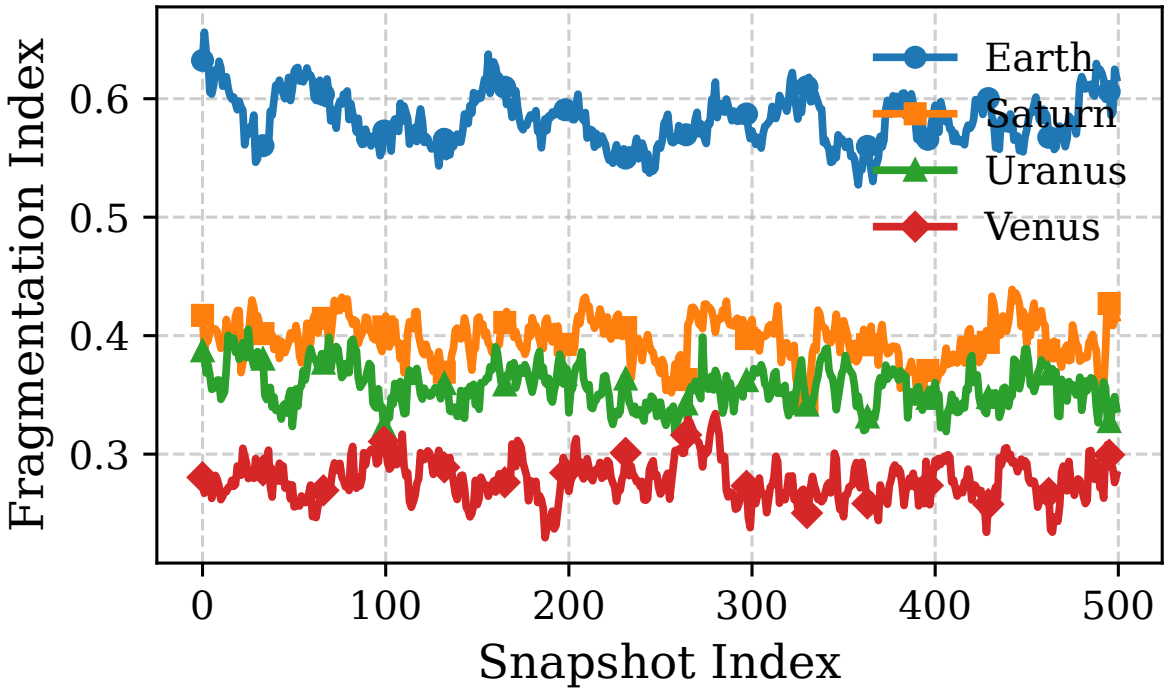
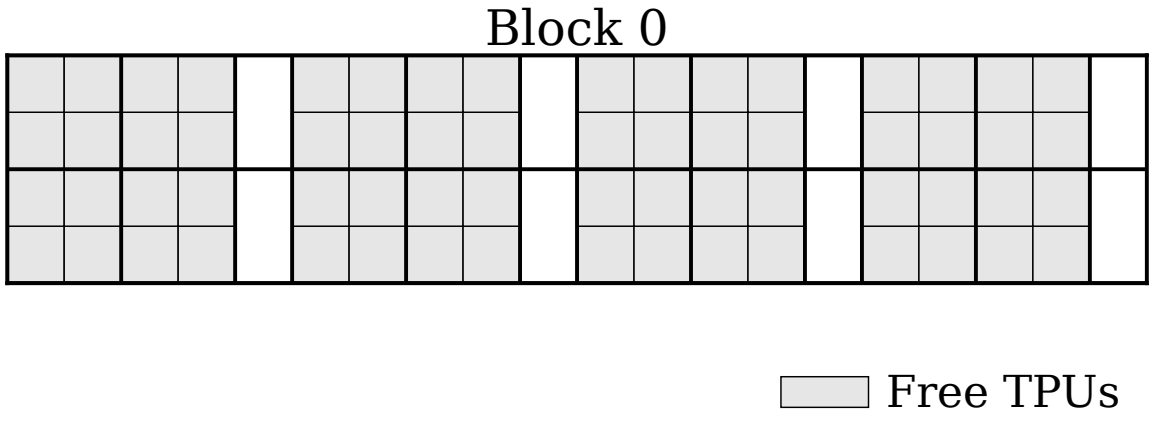


Figure 3.15. Internal fragmentation of each cluster in Helios traces in a 3D reconfigurable torus with node-wise allocation

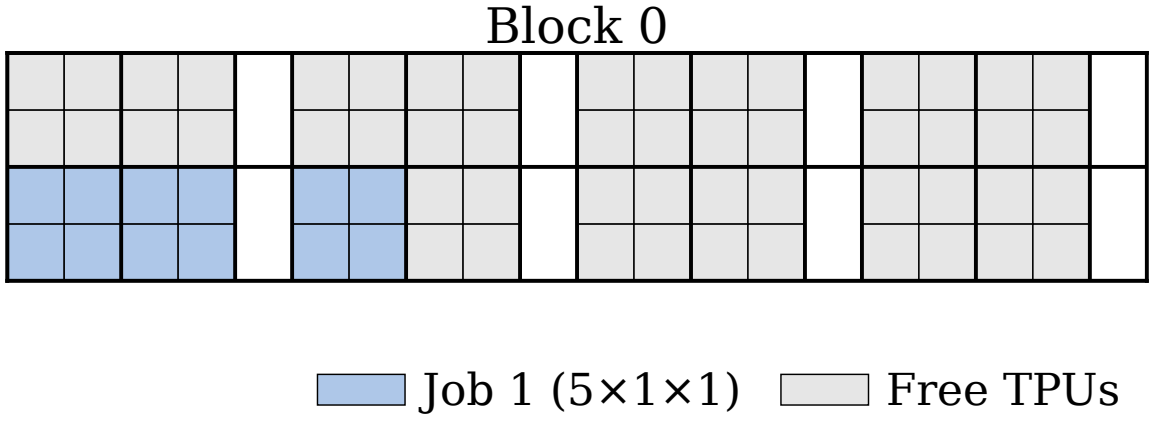
multiple blocks using best-fit heuristics. It also tries to minimize the fragmentation by choosing the block with the fewest remaining free nodes. If this recursive plan fails, then the allocator tries the non-contiguous stitching (scatter-fit), preserving the requested virtual shape. Node-wise reconfigurable cluster is much more flexible than block-wise cluster because block-wise cluster enforces face-based stitching rules, requiring units to sit on exposed boundaries since the links within a block are fixed. Next we give few examples to illustrate the difference between node-wise and block-wise reconfigurable cluster.

3.5.7 Node-wise vs Block-wise Reconfigurable Cluster

We first show the allocation flexibility of node-wise reconfigurable cluster against the block-wise cluster with two examples.



(a) top: Block-wise reconfigurable cluster; fails to allocate nodes since the requested shape cannot be preserved

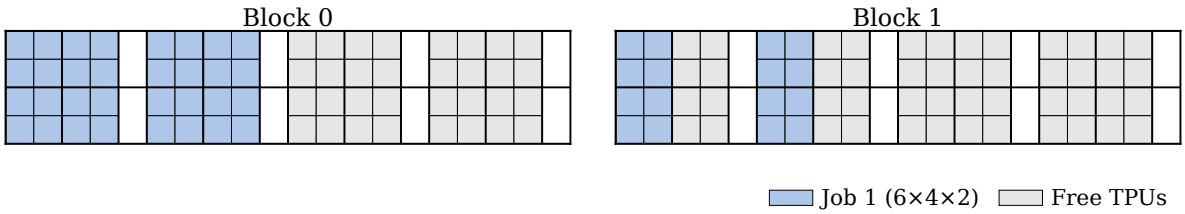


(b) bottom: Node-wise reconfigurable cluster

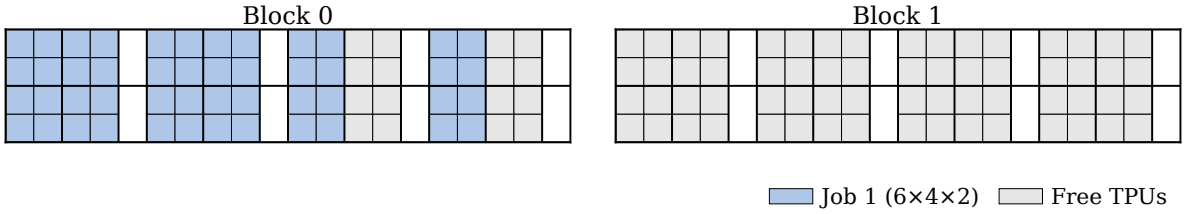
Figure 3.16. Each cluster contains one block, and the requested shape is 5×1×1

Example 1

For this example, let us first assume that both clusters are only composed of one block. Figure 3.16 shows that for requested shape (5×1×1), block-wise reconfigurable cluster requires atleast two blocks to preserve the shape hence it fails to allocate nodes to the job. But node-wise reconfigurable cluster is flexible enough to preserve the shape and allocate nodes within the same block.



(a) top: Block-wise reconfigurable cluster



(b) bottom: Node-wise reconfigurable cluster

Figure 3.17. Each cluster contains two blocks, and the requested shape is 6×4×2

Example 2

In example 2 (Figure 3.17), we can see that while the block-wise cluster requires nodes from two blocks to satisfy the required shape (6×4×2), node-wise cluster allocate the required node within the same block reducing the number of fragmented blocks.

Cluster Comparison

Internal fragmentation will have no impact from the reconfiguration setting since we are drawing job samples from the TPUv4 distribution and still follows node-wise allocations. But we can improve the external fragmentation as well as failed rate allocation with flexibility garnered from the node-wise reconfigurable cluster. Figures 3.19 and 3.20 shows the comparison of block-wise vs node-wise cluster for external fragmentation and failed rate allocation at 90% utilization. This shows that even if the allocated shape is still selected from the TPUv4 distribution which allocate slices in multiple of four, external fragmentation and failed rate allocation improved because node-wise reconfigurable cluster can pack jobs using fewer blocks due to its flexible reconfiguration settings. We also show the CDF of the admitted jobs (Figure 3.18) for one of

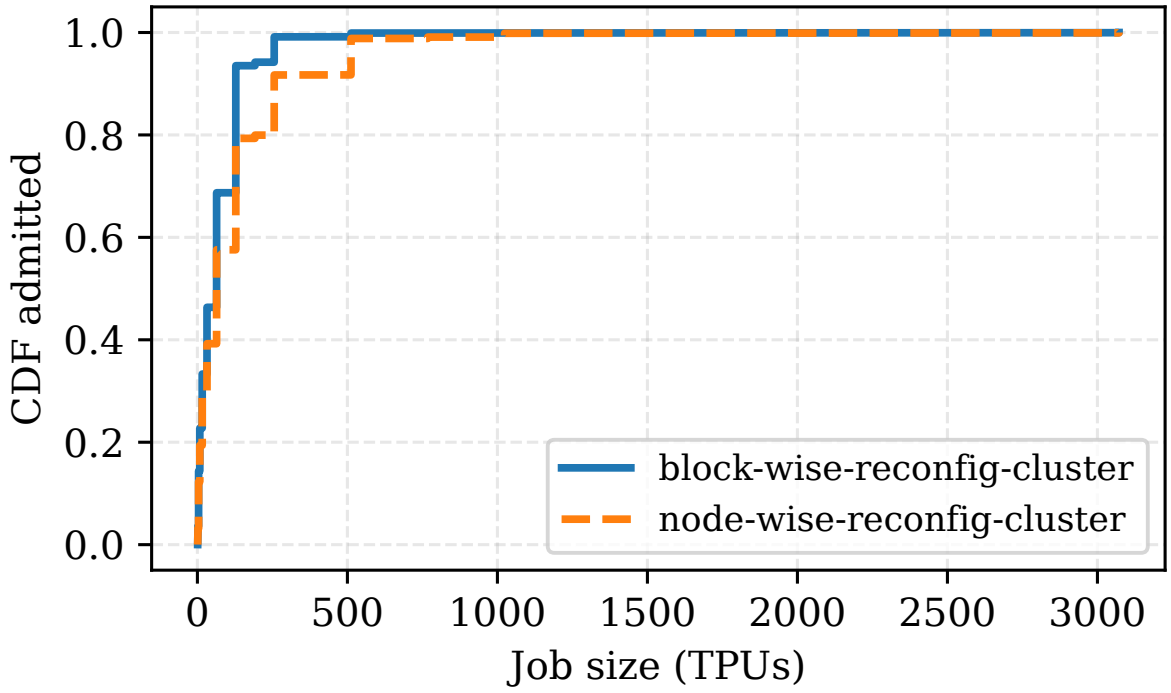


Figure 3.18. CDF of the admitted jobs

the experiments (at 90% utilization) for reference. Table 3.6 compares the average external fragmentation as well as failed rate allocation for these two cluster types. According to the results, the node-wise reconfigurable cluster reduces external fragmentation by approximately half on average compared to a block-wise cluster. Successful job allocation rate is also higher in node-wise cluster compared to the block-wise cluster.

Observation 4. A node-wise reconfigurable cluster reduces external fragmentation by roughly 50–60% across moderate utilizations, with gains tapering to 44% at high load. Correspondingly, the job allocation rate improves by approximately 15% on average.

3.5.8 Minimizing Node Allocation to Satisfy Job Requests

Next question that comes to mind is *"What if we allocate the minimum number of nodes to satisfy the tenant requests instead of always picking shapes from the TPUv4 distribution to lower internal fragmentation?"* That is, so far in all of the experiments we pick the closest node

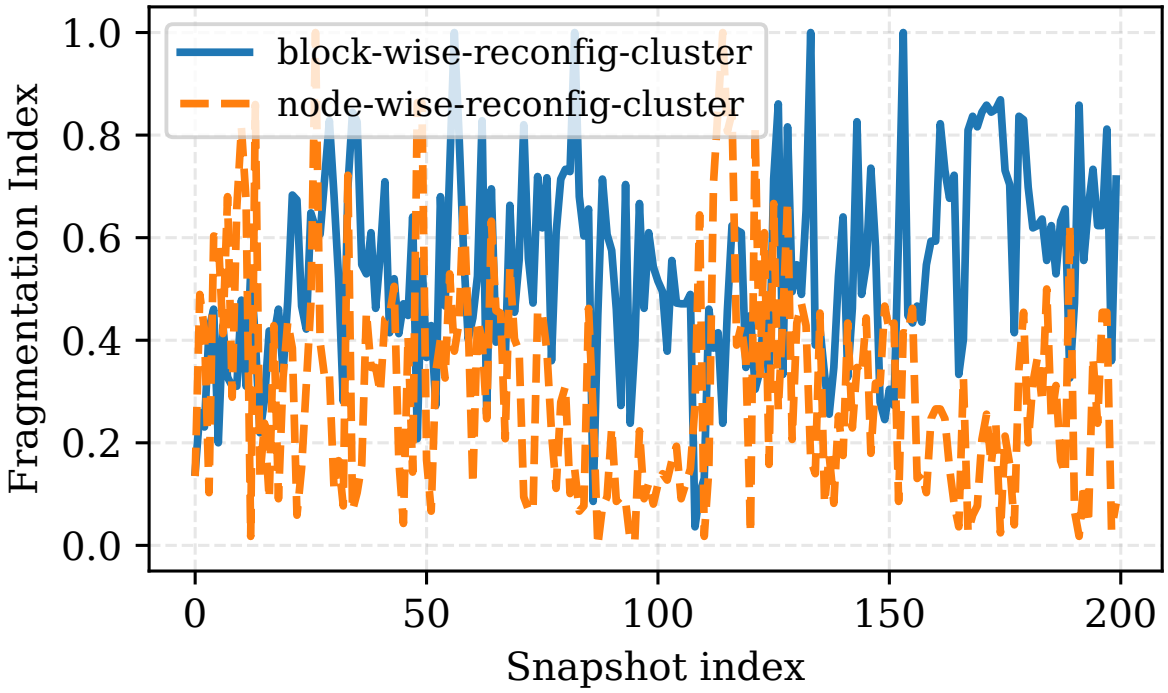


Figure 3.19. External fragmentation at 90% utilization (64 blocks in the cluster)

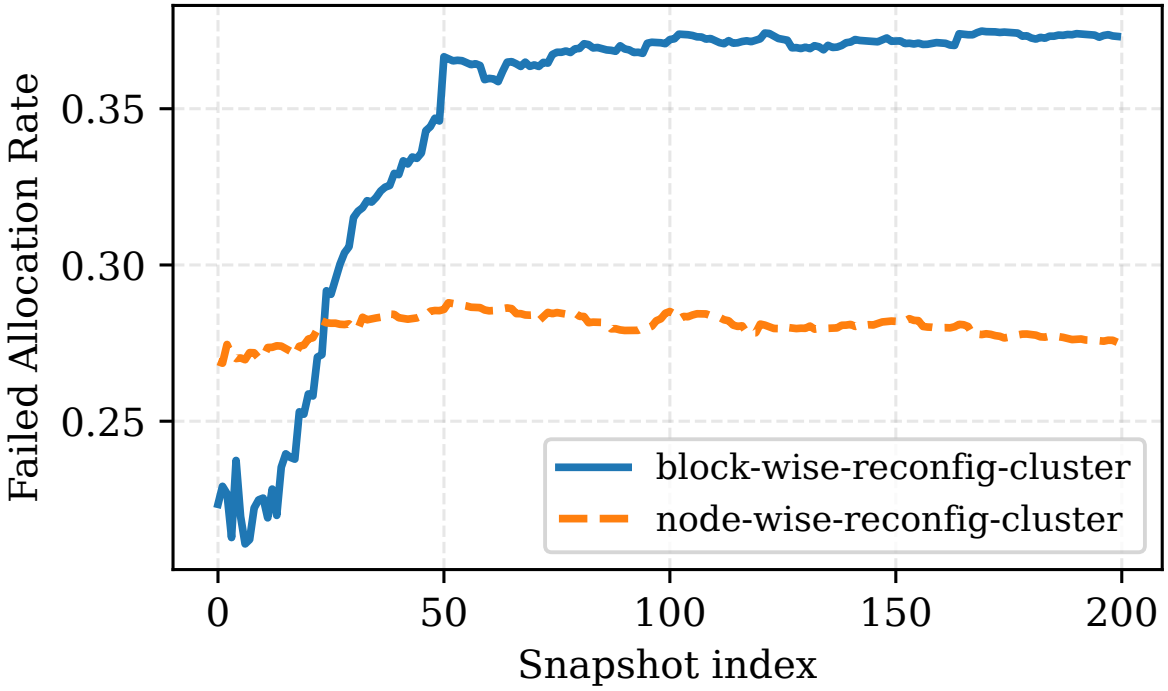


Figure 3.20. Failed allocation rate at 90% utilization (64 blocks in the cluster)

Table 3.6. Block-wise vs Node-wise Reconfig. Cluster Comparison

(a) Average External Fragmentation

Utilization	Block-wise	Node-wise	% Decrease
20%	0.0239	0.0100	58.16%
50%	0.0549	0.0229	58.29%
75%	0.1568	0.0582	62.88%
90%	0.5377	0.3031	43.63%

(b) Average Failed Rate Allocation

Utilization	Block-wise	Node-wise	% Decrease
20%	0.0790	0.0635	19.62%
50%	0.0993	0.0962	3.12%
75%	0.2028	0.1567	22.73%
90%	0.3478	0.2803	19.40%

shape from the TPUv4 distribution. For example, if the request is for (4×3×1) we pick (4×4×4). If we do a node-wise allocation without picking up shapes from TPUv4 distribution then the allocator will choose (4×4×1) as the shape regardless of the cluster type. For this experiment we use the synthetic distribution since it has odd/irregular shapes. Instead of picking tori shape from the existing TPUv4 distribution we send the requested shape directly to the allocator. Then we compare the results with previous synthetic experiment where we pick shapes that are closest to the requested shapes from TPUv4 distribution) . We use the block-wise reconfigurable cluster for this experiment.

Figures 3.21 and 3.22 show the external fragmentation and failed rate allocation rate (at 75% utilization) for synthetic shape (pick the shape that is closest to the requested shape from TPUv4 distribution) vs exact shape (send the requested shape directly to the allocator). According to the results, we can see that sending requested shape to the allocator instead of allocating shapes that are multiple of four in all three directions lead to an increase in external fragmentation and failed allocation rate. Table 3.7 shows that this outcome is same across all utilization levels. Even though allowing exact shapes can reduce internal fragmentation up to

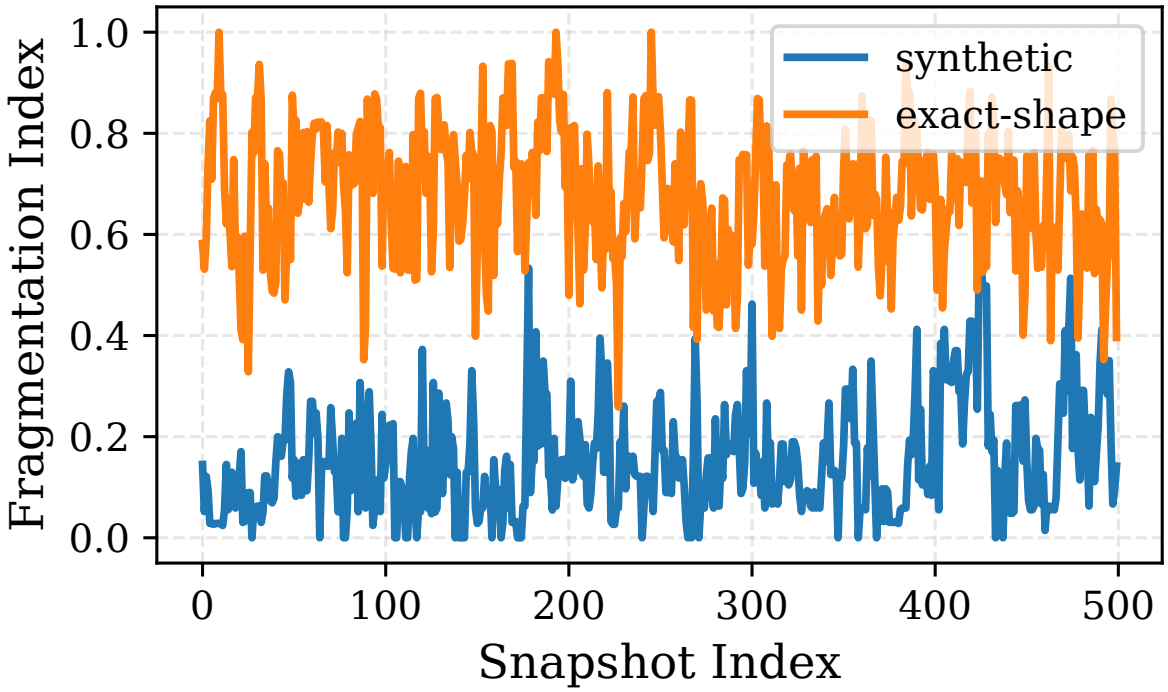


Figure 3.21. External fragmentation at 75% utilization (64 blocks in cluster)

75% utilization (Table 3.8), it tends to increase external fragmentation. This highlights a trade-off between internal and external fragmentation.

> **Observation 5.** While allowing exact requested shapes improves internal fragmentation (up to 75% utilization), it does so at the expense of higher external fragmentation and increased allocation failures, underscoring a clear tradeoff.

3.6 Multi-tenant Collectives

Next important question that we want to tackle is, given a topology slice (x×y×z), can we allow tenants to reconfigure the links within the slice to improve the performance of DNN jobs? So far we only talked about how the flexibility provided by reconfiguration can help reduce fragmentation in ML torus clusters as a whole. However, another important aspect is improving the performance of DNN jobs in a cluster. In this section, we show how a flexible topology

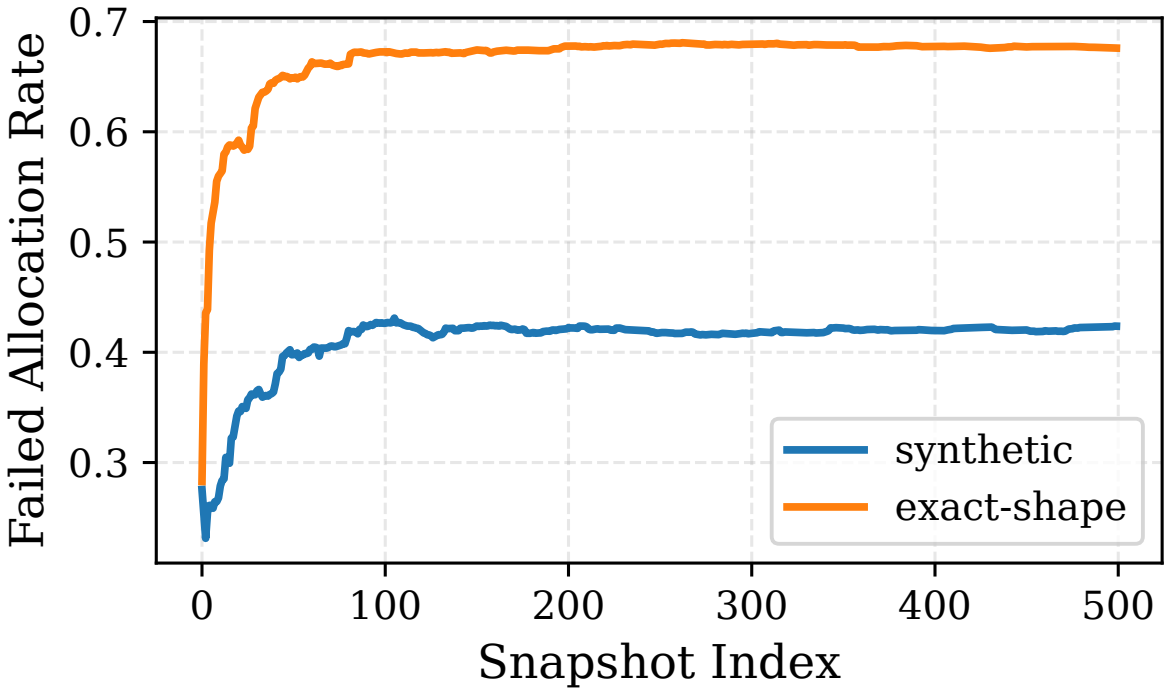


Figure 3.22. Failed allocation rate at 75% utilization (64 blocks in cluster)

slice (x×y×z) allocated to a tenant can improve the performance of DNN jobs. That is, once a topology slice is allocated for a tenant/job, if we give more control to the tenant as to how they want to rearrange the links either to change the shape or increase the link utilization of the topology slice, we can lower the communication overhead for these ML jobs.

We investigate two questions : 1) By changing the torus shape of the allocated slice, can we improve the training time of DNN models? 2) Can we allocate the unused links of the allocated torus slice in the direction of the communication to lower the communication overhead? We aim to answer these two questions with two deep neural network models.

3.6.1 DNN Model Simulation

We use two DNN model simulations. One is Deep Learning Recommendation Model (DLRM) [45, 18] and the other is CosmoFlow model [22, 40]. To improve the efficiency of the model training today we have various parallization strategies such as data, model, operator and

Table 3.7. Synthetic vs Exact Shape Comparison

(a) Average External Fragmentation

Utilization	Synthetic	Exact Shape	% Increase
20%	0.0053	0.0911	1618.87
50%	0.0201	0.4050	1915.42
75%	0.1513	0.6796	349.21
90%	0.6282	0.9835	56.58

(b) Average Failed Rate Allocation

Utilization	Synthetic	Exact Shape	% Increase
20%	0.0811	0.1061	30.82
50%	0.1957	0.3581	83.00
75%	0.4107	0.6665	62.27
90%	0.6600	0.8258	25.12

Table 3.8. Average Internal Fragmentation (64 blocks in cluster)

Job Utilization	Synthetic	Exact Shape	% Decrease
20%	0.0993	0.0902	9.16
50%	0.1113	0.0837	24.80
75%	0.1138	0.1122	1.41
90%	0.1511	0.2229	-47.51

hybrid parallizations. These parallelization strategies causes various collective communication patterns to occur during training. For example, AllReduce and AlltoAll are two of the most common patterns that you will see during DNN training. However, problem with these collective algorithms is the high communication overhead. Therefore, our goal here is to investigate whether increasing flexibility of the topology slice that is allocated to running a particular model can help reduce the communication overhead.

We describe the two DNN models that we selected for our study as follows.

DLRM

Deep Learning Recommendation Model (DLRM) [45] is one of the most popular industrial recommendation models used by various companies such as Meta, Google, Netflix and

Table 3.9. DLRM Test Scenarios

	Small Data Size	Large Data Size
Bottom MLP Allreduce	0.18MiB	128MiB
Top MLP Allreduce	2.77MiB	512MiB
Embedding All-to-all	1MiB	32MiB

Amazon. Common parallelization strategy for the DLRM is to use a combination of of model parallelism and data parallelism for the embedding tables and Multi-Layer Perceptron (MLP) layers. The embedding tables can become extremely large. Therefore, it is usual practice to divide embedding tables across multiple TPUs (Model parallelism). MLPs are relatively small so you can replicate them as many times (Data parallelism). This model and data parallelism causes two collective operations to occur during training: All-to-all and allreduce collectives. Two all-to-all operations occur one in the forward pass and the other in the backward pass. Two allreduce operations are required to synchronize the gradients of the data-parallel MLP layers.

CosmoFlow Model

For the second model, we chose CosmoFlow [40, 22] because it has multiple collectives happening among different subgroups of TPUs within a torus slice. CosmoFlow uses a hybrid of data and operator parallelism. CosmoFlow model has eight layers. Five of them are convolutional and three are dense layers. During the forward pass and backward pass four neighbour communications occur among convolutional layers and and we will have three allgather and reduce-scatters among dense layers. Each node in the operator dimension will take part in six allreduces. This is because instead of having just one giant allreduce collective at the end of the backpropagation we overlap the computation with the communication. And that will result in six allreduces. This is a common optimization technique that most models use.

We use the Structural Simulation Toolkit (SST) [61] as our simulator. It is a packet level network simulator that lets us model and analyze behavior of various topologies. In our case, we use it to answer following two questions.

Question 1: By changing the torus shape of the allocated slice, can we improve the training time of DNN models?

We use DLRM model simulation for this task. Let us assume that the tenant ask for a 64-node topology slice from the TPUv4 supercomputer to train this model. For this experiment we consider three torus shapes: (16×4×1), (8×8×1) and (4×4×4). We are going to test two scenarios with these slices: a small data size and a large data size (Table 3.9). We have kept computation time fixed across both the cases. The other thing to keep in mind is that the links are fixed once a slice is allocated for the job. The goal is to see whether the shape of the slice has an impact on the communication time. We are not reconfiguring any links during the training. Figures 3.23 and 3.24 shows the results for these two cases. In both the cases we can see that (4×4×4) slice has the least communication overhead. With a small datasize, the (4×4×4) torus topology reduces communication overhead by 67% compared to the 16×4 configuration and 40% compared to the (8×8) configuration. For the large data size where the communication dominates the execution time, changing the torus shape through reconfiguration still provides clear benefits. Compared to the 16×4 configuration, which incurs 81.06% communication overhead, reconfiguring the topology to (4×4×4) reduces communication to 68.4%, corresponding to a reduction of approximately 15.6%. Similarly, relative to the (8×8) configuration with 75.63% communication overhead, the (4×4×4) shape lowers communication by about 9.6%. This shows that once a slice is allocated to a tenant, enabling reconfiguration allows the tenant to tailor the slice topology to the application, thereby improving communication efficiency by enabling more communication-optimal topologies for the workload.

Question 2: Can we allocate the unused links of the allocated torus slice in the direction of the communication to lower the communication overhead?

We use CosmoFlow model for this task. With CosmoFlow model we partition the model

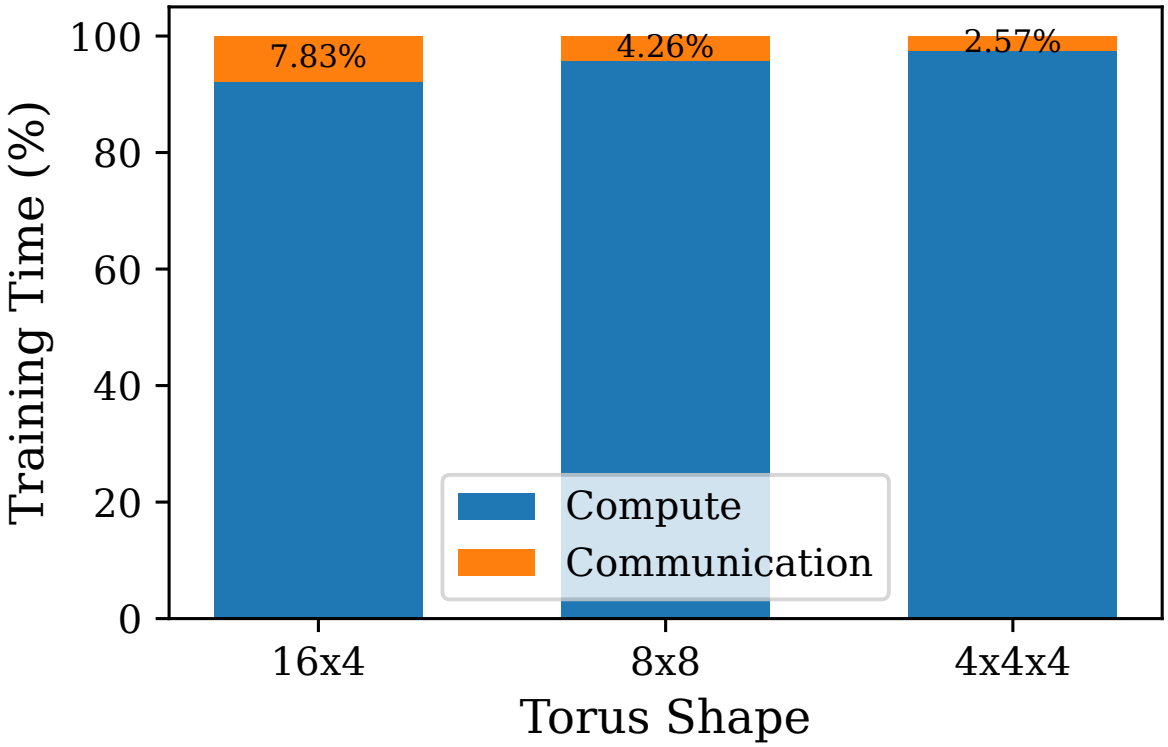


Figure 3.23. DLRM with a small data size

horizontally across four nodes and then we replicate that across another 32 groups of servers. Therefore, the torus slice we need for this job is (1×4×32). We test three scenarios: 1) Fixed torus links: Overlap communication with computation 2) Fixed torus links: No-overlap between computation and communication 3) Reconfigurable torus links: Allocate unused links in the direction of communication (No-overlap between computation and communication). Impact of reconfiguration is shown in figure 3.25. If we first compare the fixed links performance, we can clearly see that overlapping computation with communication can reduce the communication overhead compared the no-overlap case. However in the reconfigurable case, even the no-overlap setting outperforms the overlap case if we can utilize all the links. When compared to the Fixed-Links [No-Overlap] case, where communication accounts for 14.1% of total resources, the Reconfig-Links [No-Overlap] setup reduces this overhead to 5.8%, representing a reduction of approximately 59%. Even relative to the Fixed-Links [Overlap] case, which has a communication

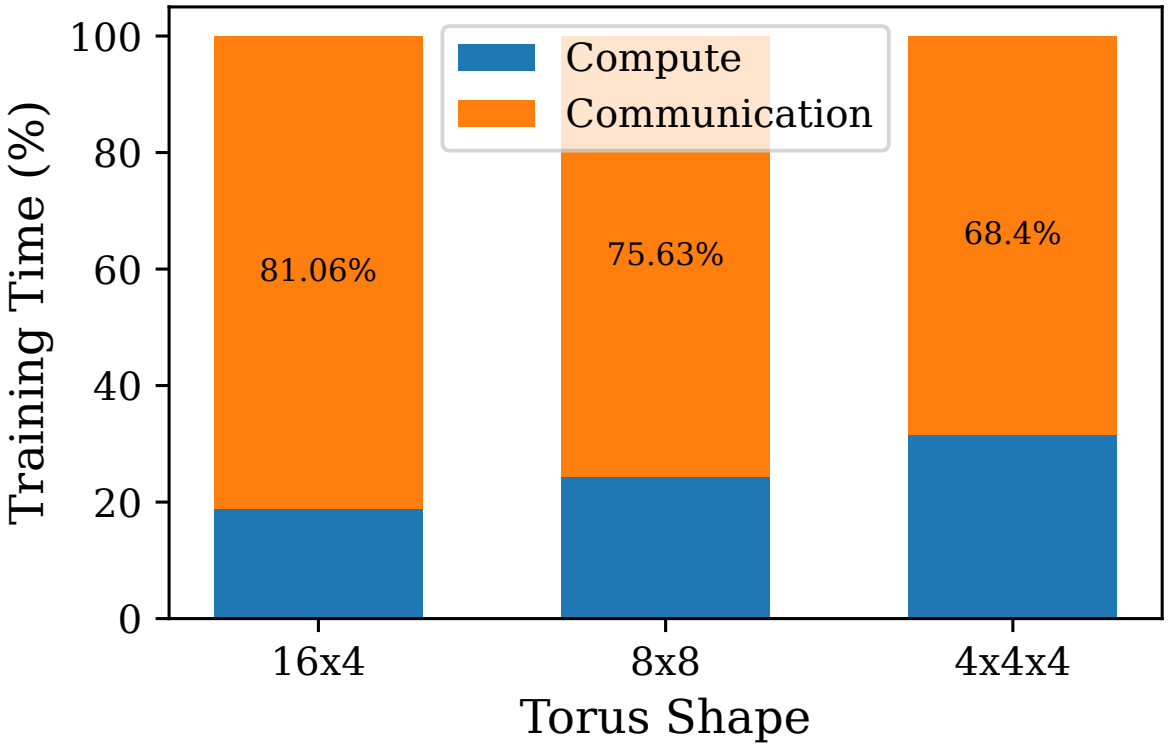


Figure 3.24. DLRM with a large data size

overhead of 10.8%, the reconfigurable links achieve a reduction of about 46%. This further proves the power of reconfiguration in ML cluster settings.

3.7 Conclusion

Addressing the problem of fragmentation is vital to improving the overall system utilization of a cluster. Given the cost of accelerators and the demand for ML workloads, it is important to make sure the cluster is utilized efficiently. In this paper, we investigate the extent of fragmentation in a block-wise reconfigurable cluster and show that relaxing the reconfigurable settings further can lead to reduced fragmentation and the job failure rate. We show that at 90% utilization, a 64-block block-wise reconfigurable cluster wastes, on average, 59% of its free space due to fragmentation, leading to roughly 38% of job requests being rejected despite sufficient overall resources. Increasing the flexibility of the cluster—particularly through node-

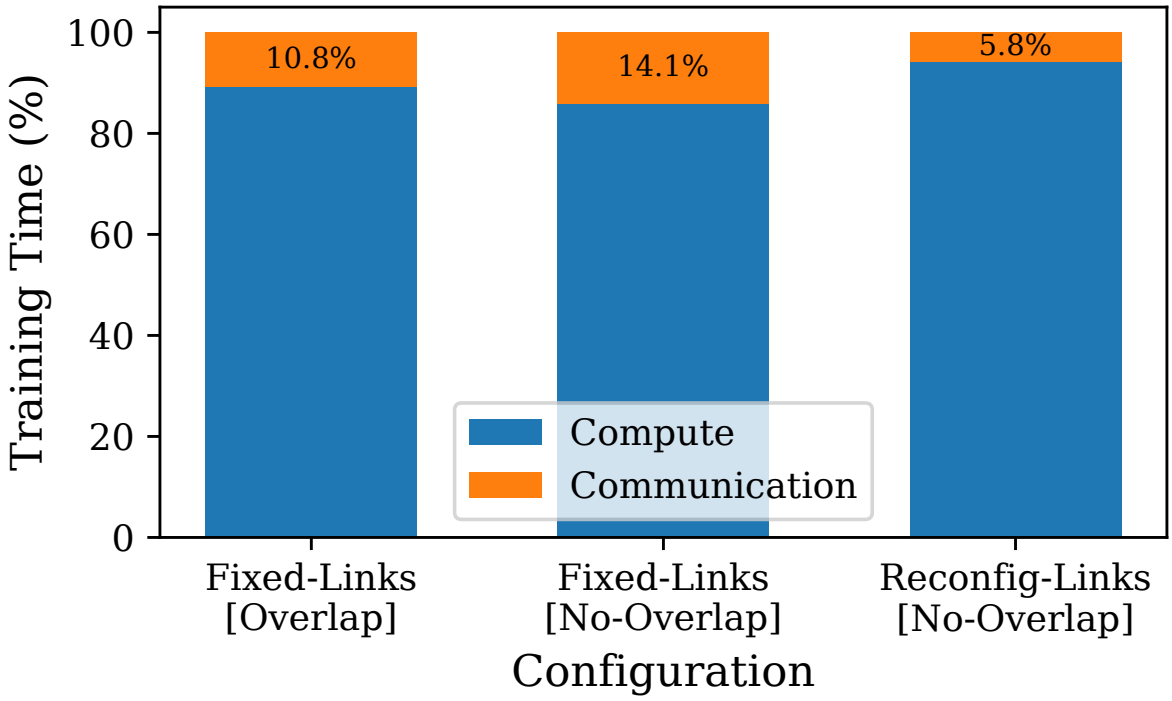


Figure 3.25. CosmoFlow model under different configurations

wise reconfiguration—significantly mitigates this issue, reducing external fragmentation by an average of 55% and improving successful job allocation rates by at least 15%. We also examined how reconfiguring allocated topology slices can improve individual job performance. Adjusting the torus shape reduces communication overhead by 15.6% for DLRM on large datasets, while reallocating unused torus links along the communication direction further cuts overhead by up to 46% for CosmoFlow.

Acknowledgments

Chapter 3, in full, is a reprint of the work submitted to 24th USENIX Symposium on Networked Systems Design and Implementation (NSDI '27) under the title "Flexible Torus Clusters: Less Fragmentation, Lower Communication Overhead". Rukshani Athapathu, George Porter. The dissertation author was the primary investigator and author of this paper.

Table 3.10. Distributions of TPUv4 slices employed in the simulations

Original (Slice:%)	Synthetic (Slice:%)
(1×1×1) : 2.1	(1×1×1) : 2.17
(1×1×2) : 0.4	(1×1×2) : 0.41
(2×2×1) : 6.7	(1×1×3) : 1.24
(2×2×2) : 4.7	(1×2×4) : 0.72
(2×2×4) : 6.4	(2×3×5) : 2.17
(2×4×4) : 8.9	(3×3×5) : 4.76
(4×4×4) : 14.4	(2×2×6) : 2.90
(4×4×8) : 19.5	(3×5×7) : 3.62
(4×4×12) : 0.7	(4×4×5) : 6.63
(4×8×8) : 10.7	(4×6×8) : 10.35
(4×4×16) : 1.0	(5×5×9) : 6.52
(4×8×12) : 0.1	(4×4×10) : 1.97
(8×8×16) : 3.2	(6×6×9) : 7.77
(4×16×16) : 0.3	(4×8×12) : 9.12
(8×8×8) : 9.6	(5×7×11) : 3.93
(4×8×16) : 1.7	(6×8×14) : 5.49
(4×4×32) : 0.6	(7×7×13) : 3.10
(8×8×12) : 0.7	(8×8×12) : 7.35
(4×4×64) : 0.1	(9×9×15) : 3.62
(4×8×32) : 0.1	(8×12×16) : 5.18
(8×12×16) : 0.1	(10×10×12) : 2.28
(4×4×96) : 0.1	(12×12×18) : 3.00
(8×8×24) : 0.1	(8×8×20) : 1.86
(8×16×16) : 1.7	(5×5×25) : 0.72
(12×16×16) : 5.7	(7×11×19) : 0.62
(4×4×192) : 0.4	(12×16×16) : 1.35
-	(16×16×12) : 1.14

Chapter 4

Rotor-XDP: Emulating a Circuit-Switched Network Stack with eBPF

Reconfigurable networks that uses Optical Circuit Switches (OCSes) are slowly paving their way into datacenters due to their ability to offer high bandwidth at low cost and power. Jupiter [50] is an example of a real-world cloud deployment that has at least replaced the spine layer switches with OCSes in their datacenters. However, their design still leaves a large number of ToRs and aggregate-level switches intact. There are two implications to this. 1) Existing protocols at end-hosts and lower layers do not need any modifications 2) The power of OCSes is not fully realized. The main reason for this slow adoption of OCSes by the datacenter operators is its complexity. It should be pointed out here that the OCSes are not a drop-in replacement for packet switches since they require assistance from the entire network. This is because the control plane complexity hidden in packet switches is now exposed to the other elements such as ToRs and end-hosts with OCSes.

However, there have been many proposals [15, 43, 24, 49, 41] for hybrid and all-optical designs over the years. Yet these proposals focus entirely on optical design decisions, ignoring the design's implications on end-host transport protocols. Therefore, there is a need for an emulator that mimics the OCS behavior while letting us easily tap into the existing transport protocols. Emulators are a great way to explore different designs faster and cost-effectively and offer many possibilities in terms of debugging compared to hardware. In our use case,

the emulator also let us verify the validity of the optical design before the actual OCS became available. Time synchronization is another important aspect of reconfigurable designs as nodes need to be aware of the switch state for traffic scheduling. Therefore, the emulator framework should also be able to work with popular time synchronization techniques such as PTP [32].

High-performance packet processing frameworks such as DPDK [2], BESS[19], and Click[30] are popular among researchers for building and testing novel protocols and emulator designs. eXpress Data Path (XDP) [23] is a programmable packet processing framework that falls under the same category and runs in the form of an eBPF code. Unlike other frameworks, the advantage of XDP is its ability to selectively choose kernel networking stack features without having to take total control of the NIC. XDP also works seamlessly with programmable hardware designs such as Corundum [13] without needing any driver support but at the cost of performance. AF_XDP [1] on the other hand makes use of XDP to bypass the kernel and redirect frames to a user-space application.

Our OCS design choice for the emulator is Opera [41]. Opera is a demand-oblivious, all-optical reconfigurable network design and we have three requirements. 1) Develop an OCS emulator based on Opera[41] to understand the end-host stack behavior and limitations of the current protocols. 2) For cloud datacenters to be able to utilize OCSes directly with end-hosts they need a controller at each end-host that interfaces with the OCS as well as the VMs and containers. 3) Recently developed real-world OCS (Rotor Switch[42]) which is based on Opera [41] needs to utilize host DRAM for part of the protocol which requires a software framework for interfacing with an FPGA-based NIC (Corundum [13]).

Before we reason out why XDP/AF_XDP is the right choice for the above requirements, we first describe the chosen OCS design and the features required by the actual Rotor Switch [42] and the emulator.

4.1 System Design

4.1.1 OCS Design

Opera [41] is a demand-oblivious reconfigurable network that follows a static schedule regardless of the traffic demand making way for a decentralized control plane. It makes use of an expander graph approach for its deterministic schedule. Our particular example of the design assumes to have eight OCSes (Rotor Switches) which rotate through a fixed set of configurations providing connectivity among all connected nodes. The emulator testbed consists of 32 nodes and each node is assumed to connect to the eight OCSes. At a given time we assume that one OCS is reconfiguring and the rest of the seven make up the topology which forms an expander. There are seven possible next-hops a given node can talk to directly in a topology. However, there are multi-hop paths between all nodes at all times. This gives the system great flexibility in deciding whether to send packets over multi-hop paths or wait for a direct connection. The previous experiments with Opera [41] have only been conducted on a simulation and with a custom network stack. The authors have not examined how the existing transport protocol would behave under a constantly changing topology like Opera [41] and whether the existing traffic shaping techniques would still hold in a multi-tenant environment. The goal of the emulator is to explore these unknowns.

4.1.2 OCS Emulator Requirements

The emulator needs to faithfully mimic the OCS behavior as well as the hardware that directly connects with the OCS. This becomes important to the end goal which is to analyze the end-host stack behavior for a given OCS design. All nodes need to be time synchronized with each other so that the packets are sent at the right times (in the correct topology) by all nodes. The emulator should also support a given number of containers that we later plan to utilize to understand how traffic shaping should be done on an ever-changing topology.

4.1.3 Why XDP/AF_XDP?

We rule out DPDK [2] and BESS [19] because they take total control of the NIC and in the case of hardware/software interaction they cannot seamlessly interact with the FPGA-NIC (Corundum [13]) without having to make extensive modifications. We also only require certain types of packets (GRE) [35] to bypass the kernel and need the rest to go through the normal network stack. This allows us to use PTP on the same NIC as the datapath. The emulator also needs to interact with multiple containers. This can be easily accomplished by hosting an XDP program on each VETH and redirecting packets to the emulator via AF_XDP. As for the software/hardware interface with the actual Rotor Switch [42], the requirement arises due to the limited amount of memory available at the FPGA. The store and forward protocol needed by the OCS design has to be supported by the host DRAM. Therefore, we need a framework to interact with FPGA without needing any driver support for fast development. SKB_MODE of XDP allows us just that.

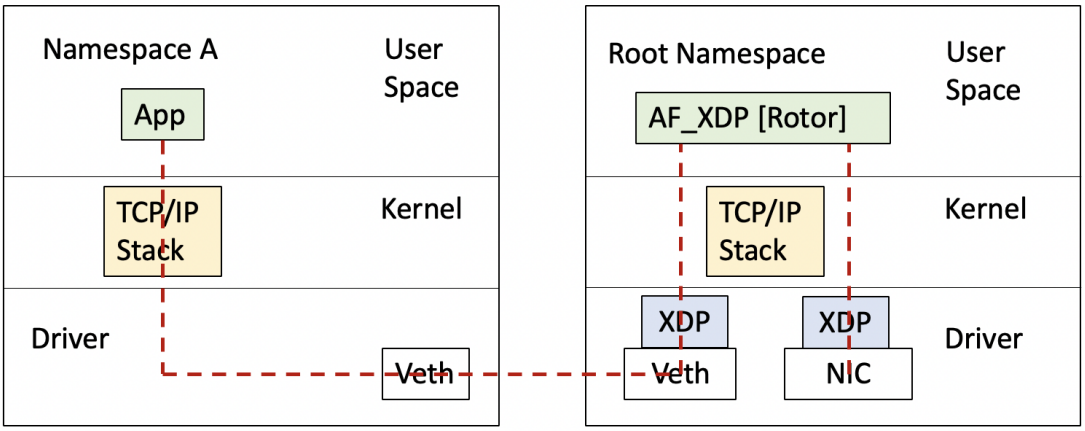


Figure 4.1. Use of XDP/AF_XDP in the Emulator

4.1.4 Emulator and the Actual System

A high-level view of the emulator and the actual system is shown in Figures 4.1 and 4.2 respectively. All 32 nodes in the emulator run their own instance of the rotor (in AF_XDP) and

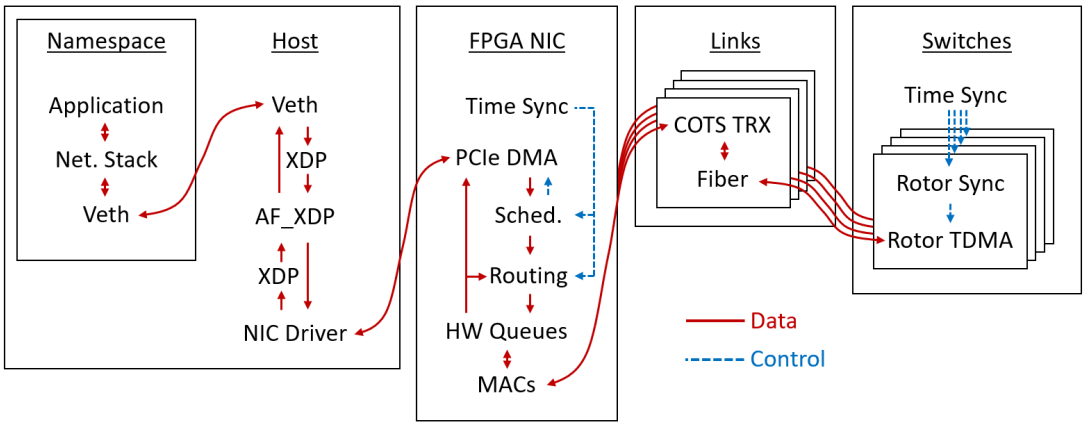


Figure 4.2. Use of XDP/AF_XDP in the Physical System (This figure is reproduced from one that appears in [42])

handle the scheduling, routing, and queue management. In the actual system, those functionalities are offloaded to the FPGA-NIC. In our design, we host our applications in network namespaces. We create a VETH pair for each namespace; One we place inside the namespace and the other one in the root namespace. The physical NIC and the VETH in the root namespace both have XDP programs running in the driver so that the incoming frames can be redirected to the AF_XDP program (Figures 4.1 & 4.2).

4.2 Implementation

4.2.1 Emulator

We host our emulator logic in AF_XDP [1]. It consists of two parts. A user-space application where the emulator controller resides and an XDP (eXpress Data Path) program that runs on each VETH and the physical NIC. The responsibility of the XDP program resides in each VETH is to redirect packets from applications to the emulator. XDP program in the physical NIC helps filter GRE-encapsulated ingress frames and redirect them to a memory buffer in the emulator. It lets all other packets go through the normal network stack which is invaluable when PTP is running on the same network interface. Each AF_XDP socket in the emulator is bound to a specific queue ID (HW Queue/VETH queue) and consists of two rings: RX and TX. This

enables the emulator to receive packets from the namespace, process them, and send them out via NIC and vice versa. The emulator acts as a bridge between applications hosted in namespaces and the physical NIC.

4.2.2 Memory Pool

TX and RX rings that are associated with AF_XDP sockets hold the descriptors that point to data buffers in a memory area called UMEM. UMEM consists of equal-sized chunks of 4096 bytes and is associated with two rings: a FILL ring and a COMPLETION ring. With the FILL ring, we can pass down the buffer addresses to the kernel so that they can be used to fill the RX packet data. The COMPLETION ring, on the other hand, contains the buffer addresses of transmitted packets and can be reused for either RX or TX. In our use case, we share the UMEM area with all the AF_XDP sockets hence each socket gets its own FILL ring and COMPLETION ring.

We have experimented with two memory pool implementations. 1) All ports (we refer to an AF_XDP socket as a port) share a single UMEM area. 2) Each port has its own UMEM area. With a single pool implementation when the packets need to be forwarded from/to between VETH and Physical NIC, descriptors can be passed between ports without ever having to copy the packet content. However, with a multi-pool approach where each port has its own UMEM, we cannot avoid the packet copy. We only considered this second approach because when the number of namespaces increases thread contention can become a bottleneck with a single pool. However, packet copying has a larger hit against the throughput hence we restored to the first approach for the emulator.

4.2.3 Data Path

The Emulation data path is shown in Figure 4.3. A packet coming from a VETH interface first needs to be classified into two categories; low latency or bulk traffic (Bulk-Local). Packets are then queued by their original destination in local per-dest queues. Then the next hop of

the packet is determined by the current topology the assumed OCS is in and the packet is encapsulated with GRETAP. The outer Ethernet header of GRETAP points to the next hop of the packet and might or might not be its final destination. Finally, the packets are sent out via the mapped hardware queue. In the RX path, when a packet is received from the physical NIC if the packet is destined for that node, first we decapsulate the outer headers (remove GRE headers) and move it to the corresponding VETH queue which is then forwarded to the application. If the packet is not destined for the node we first move it to the corresponding per-dest non-local queue and calculate the routing information to determine the next hop. Non-local traffic has a higher priority than local traffic to reduce the latencies of intermediate traffic.

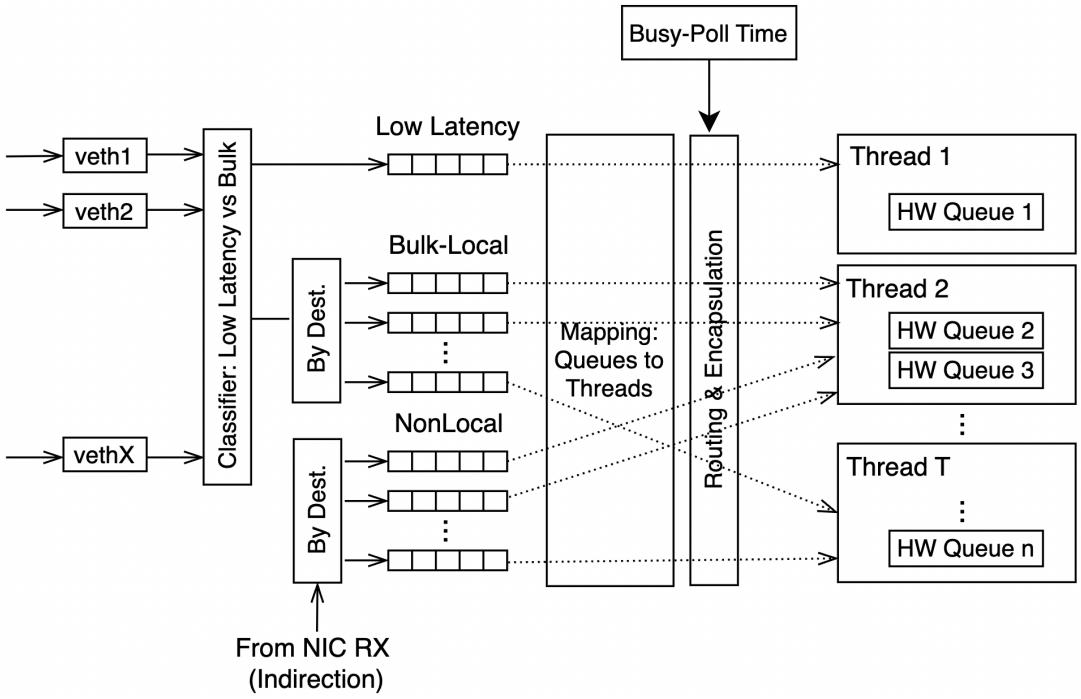


Figure 4.3. Emulation: Data Path

4.3 Evaluation

Before we start evaluating the end-host behavior with the protocols dictated by the chosen OCS (Opera & RotorLB [43]) we need to understand the capabilities of XDP/AF_XDP

in terms of latency and throughput. We evaluate our system on three types of servers. We use Cloudlab c6525-25g (AMD EPYC 7302P 16-Core Processor) servers that have two dual-port Mellanox ConnectX-5 25Gb NICs for the 32-cluster experiments. Dual-port NIC allows us to run PTP on a separate interface and busy poll the NIC clock from the emulator to determine the correct topology for each node. We use Cloudlab r650 nodes (Intel Xeon Platinum 8360Y) to stress test the XDP/AF_XDP with 100G capable NICs. We use campus LEED servers (Intel(R) Xeon(R) Gold 6150 CPU) to host the FPGA-NICs. We run Ubuntu 22.04.2 LTS with kernel 5.15.0-86-generic version on all servers. MTU for VETH is set to 3400 bytes and for the physical NIC, it is set to 3490 bytes to accommodate the GRE headers.

4.3.1 Node-to-Node Throughput

With the current version of the emulator, node-to-node experiments can be easily conducted by changing just the routing configuration file for each node. The goal of the node-to-node experiments is to uncover the limitations of XDP/AF_
XDP/VETH and figure out the thread model for the emulator.

Version 1

The first version of the emulator is tested with one network namespace and one HW queue with a single thread managing both the forward path and the return path as shown in *Design 1* of Figure 4.4. This gives us roughly ≈6Gbps throughput for TCP. Managing the forward path and return path with two separate threads (*Design 2* of Figure 4.4) increased the throughput to ≈9Gbps. Further separating the functionality by giving each port's RX and TX its own thread (*Design 3* of Figure 4.4) increases the throughput to ≈11Gbps. Further improvements by creating multiple AF_XDP sockets for each netdev/q failed due to the reasons mentioned in §4.4. The next step in the process is to see the scalability of the emulator with multiple namespaces but keeping the same thread count. This leads to no improvements because sending packets out to the applications on the return path via VETH TX bottlenecks the whole process. Because of the

way the shared memory pool works if the buffers cannot be released to the memory pool fast enough, the NIC side FILL queue runs out of descriptors and the kernel waits for the buffers to be released by VETH TX so that it can copy NIC RX packets to UMEM.

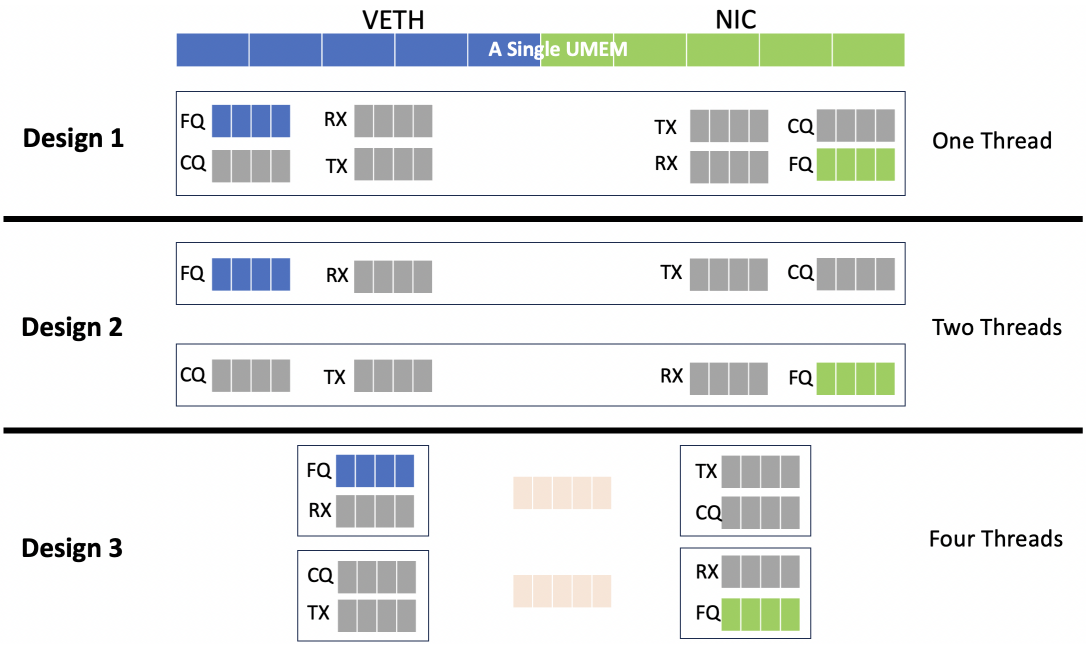


Figure 4.4. Node-to-Node (Version 1)

Version 2

Experiments with the second version are run on Cloudlab r650 Intel Xeon Platinum 8360Y servers which are more powerful than Cloudlab 6525-25g and have Mellanox ConnectX-6 100Gb NICs. In this version, first, we give each VETH's RX and TX its own thread and rerun the previous experiment by increasing the number of namespaces but keeping the hardware queue count to one. With two namespaces throughput jumps to ≈24Gbps but soon becomes bottlenecked by the hardware queue count when the namespace count increases beyond two. Next, we map the namespaces to the hardware queues 2:1 and 1:1 and also increase the number of iperf3 client processes that we ran on each namespace so that RSS hash has a better chance of utilizing all available NIC RX queues. 1:1 mapping gives us a higher throughput than the 2:1 mapping. We report only the former results (Figure 4.5) due to the limited space.

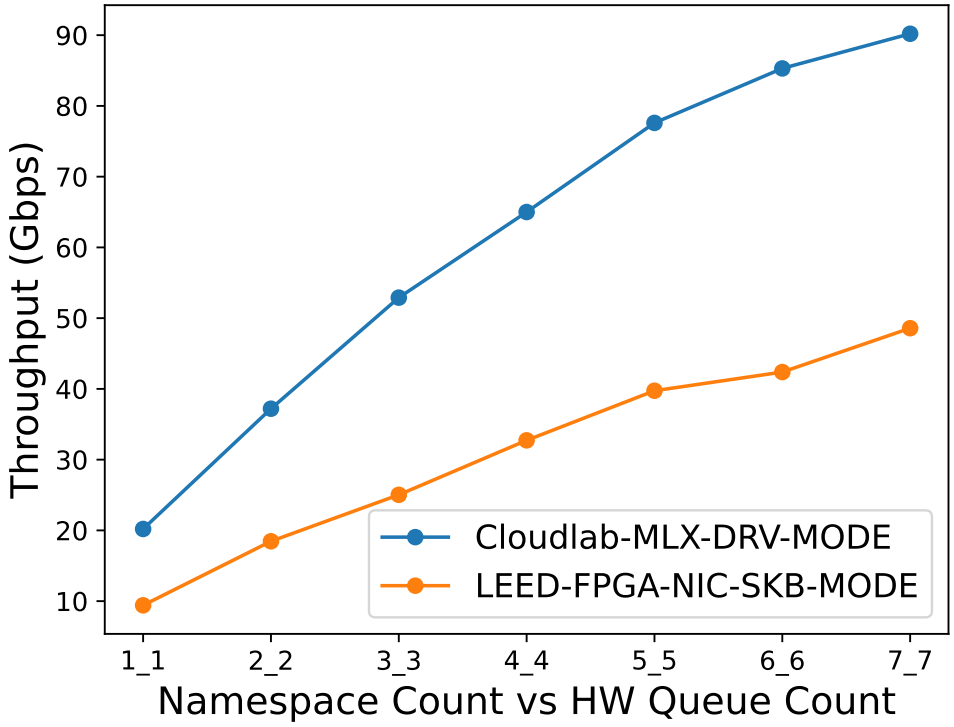


Figure 4.5. One-to-One Mapping between Namespaces and HW Queues (Version 2)

FPGA-NIC with SKB_MODE

Next, we want to see whether the SKB_MODE of XDP gives us a reasonable performance when used with FPGA-NIC. We conduct the same node-to-node experiment with FPGA-NIC on campus LEED servers. We see a maximum of ≈48Gbps throughput with SKB_MODE on FPGA-NIC (Figure 4.5). Even though the results for both Mellanox DRV_MODE and FPGA-NIC SKB_MODE are depicted on the same graph, we cannot do a direct comparison between them since the servers that have FPGA-NICs are less powerful than the Cloudlab r650 machines and cannot account for how much throughput reduction is due to SKB_MODE. However, with the actual system, to account for the loss throughput, we have the option of using XDP/AF_XDP only for the store and forward protocol (for hosting the intermediate traffic) and directly engage the host machine to run end-user applications instead of using a namespace. Therefore, to see

whether the send rate and receive rate to and from AF_XDP with SKB_MODE is good enough we created a custom UDP packet generator with AF_XDP. Results are shown in Figure 4.6. We also observed that the receive side throughput largely depends on how well the packets are distributed across all available RX queues.

4.3.2 Node-to-Node Latency

Latency is another important metric that lets us decide the slot duration of a topology for the OCS design. We first report the application latency (RTT) between namespace-to-namespace when used with AF_XDP. As a comparison point, we plot the root RTT as well (Figure 4.8). With AF_XDP, 99.99% of packets have less than $\approx 46\mu$s RTT. This is roughly a $\approx 16\mu$s increase compared to the root. We also record the duration a packet spends on AF_XDP (time spent between AF_XDP-send to AF_XDP-receive) in the namespace-to-namespace scenario (Figure 4.7). Results indicate that 99% of the packets can be routed between AF_XDP to AF_XDP in less than 15μs. This is roughly equivalent to the majority of 99% of packets spending $\approx 16\mu$s time inside the namespace on both the send and receive side.

4.3.3 Time Synchronization

The next important task is to verify that all nodes are synced with each other and are in the correct topology for the packet transmission and receive. We run PTP on all 32 nodes. We take advantage of the dual port NIC and run PTP on a separate interface from the datapath. In the emulator we busy poll the NIC clock from a different thread and use that timing information to determine the slot and the topology of the OCS.

To verify PTP along with AF_XDP with just a single experiment on all 32 nodes, we host a UDP client on node-1 (namespace) and a UDP server on node-32 (namespace) and send 90000 packets from node-1 to node-32 via all other nodes (2-31). The configuration file determines the next hop of the packet based on the original destination and the time slot the OCS is in. A slot duration is set to 1 ms and for each slot, packets are sent to an intermediate node. The

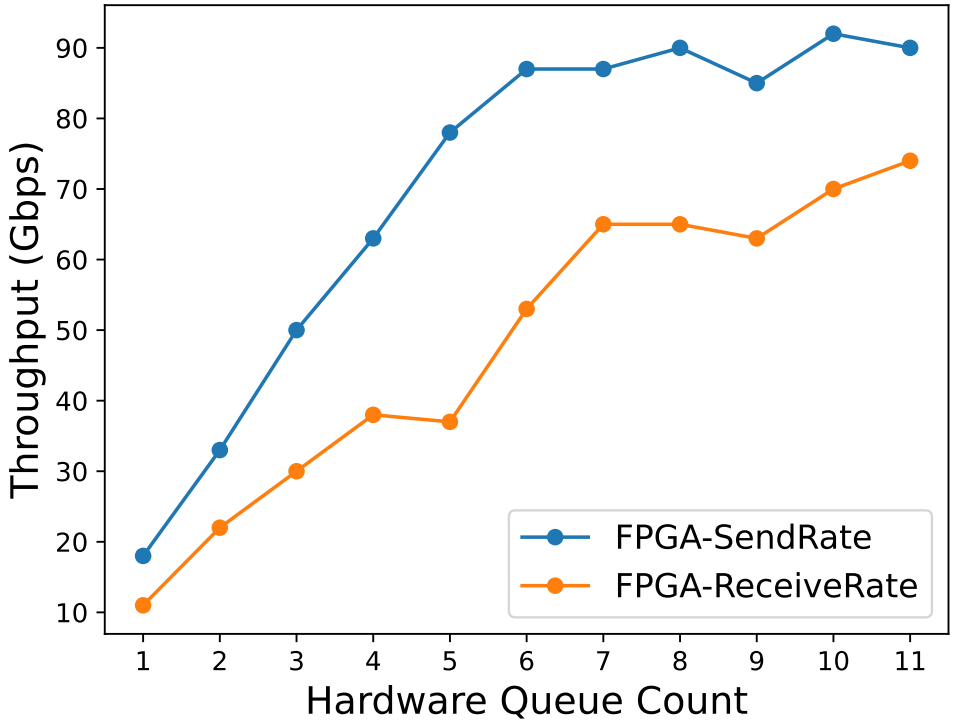


Figure 4.6. AF_XDP Packet Generator Results for FPGA-NIC SKB_MODE

intermediate node's responsibility is to receive packets via AF_XDP and send them out back via NIC to the final (original) destination. Also to mitigate the effect of batching that happens due to IRQ coalescing with AF_XDP sockets we use SO_PREFER_BUSY_POLL along with proper values set to *napi_defer_hard_irqs* and and *gro_flush_timeout*.

Then we plot the packet receive times on all intermediate nodes. We see that the packets are received in the correct order by all nodes and at the correct times. Due to the space constraint, we omit the graph.

4.3.4 Embedding OCS Behaviour

Now that we have all the pieces together we embed the OCS behavior into the emulator via a configuration file. Unlike the node-to-node experiments where we map namespaces to HW queues, mapping needs to happen between per destination local/non-local queues and HW

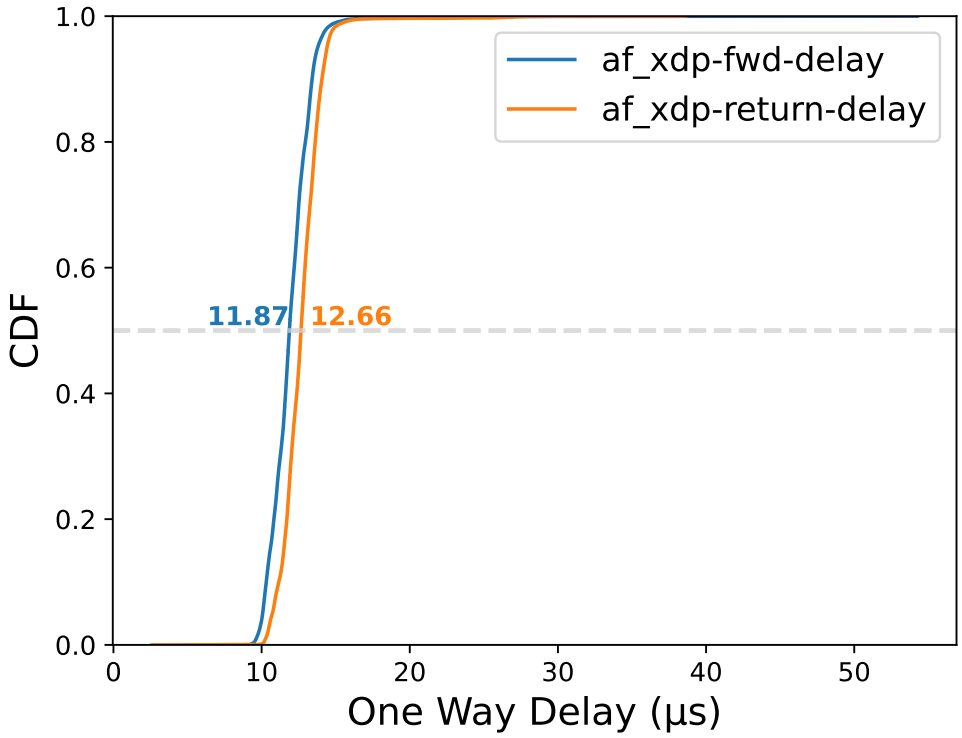


Figure 4.7. AF_XDP to AF_XDP One Way Delay

queues (Figure 4.3). We assume we have eight OCSes and that each OCS rotates through four matchings. We do not reconfigure the OCSes all at once to prevent total loss of connectivity. We stagger the reconfigurations so that at a given time only one OCS is reconfiguring and the rest of the seven OCSes make up the topology as is the case with Opera [41]. This gives us altogether 32 topologies. In each topology, each node will have direct access to seven nodes and indirect paths to all other nodes. This opens up various possibilities for packet scheduling. We can decide to send all packets immediately through multi-hop paths to the destination (Opera protocol) or decide to wait till we get a direct connection to the destination or a mixture of the two (RotorLB [43]). How the existing end-host stacks as well as many proposals of network stacks designed for OCSes would react under these conditions is an interesting question that we hope to explore.

In this paper, we only report the first few measurements we have taken with Opera where we route all packets as soon as they are received by VETH to the destination via either direct or

indirect paths based on the topology the OCS is in. The maximum hop count a packet takes on a one-way direction is three if the packet can get to the other end within the timeslot. Based on the tail latency we observed for the node-to-node scenario in §4.3.2 (Figure 4.8), we can assume that for the worst-case scenario with Opera where a packet takes three hops in each direction, RTT to come to about $\approx$420μs. Based on this number we set 1ms as the slot time for Opera where a packet has ample enough time to get back even under the worst-case scenario. However, if a packet is sent near the end of a slot there is a high chance that it might go through a few other hops since the next hop might have moved on to the next slot. Figure 4.8 shows the RTT for Opera when it's moving through all 32 topologies. We see that 99% of the packets are completed within $\approx$100μs time. However, the tail latency comes to about $\approx$373μs which is better than we expected. If all packets are able to route within a single topology, the maximum number of hops a packet traverses in each direction will be three for the worst-case scenario. However, we see a maximum of four hops for a few packets on the return path because as mentioned before if a packet gets caught up in multiple topologies it might take more hops.

4.4 Challenges and Limitations of XDP/AF_XDP

Our initial approach for the emulator is to see the feasibility of implementing part of the emulator logic (indirection) at the XDP layer since it can be run at the driver level closer to the hardware reducing the unnecessary latencies that might occur when having to send the packets to the user space and back out the NIC in indirection. However, we soon realized that we could not access the NIC clock (which is required to determine the next hop of a packet) in XDP which only supports CLOCK_MONOTONIC. The next attempt is to pass the timing information from the user space to the XDP program via a BPF map. However, we could not get the accuracy we wanted because the BPF update took a considerable amount of time.

With *Design 3* of Version 1 (Figure 4.4) of the emulator, to increase the throughput of a single container, we tried to bind more sockets to both netdev1(VETH)/q1 and netdev2(NIC)/q2.

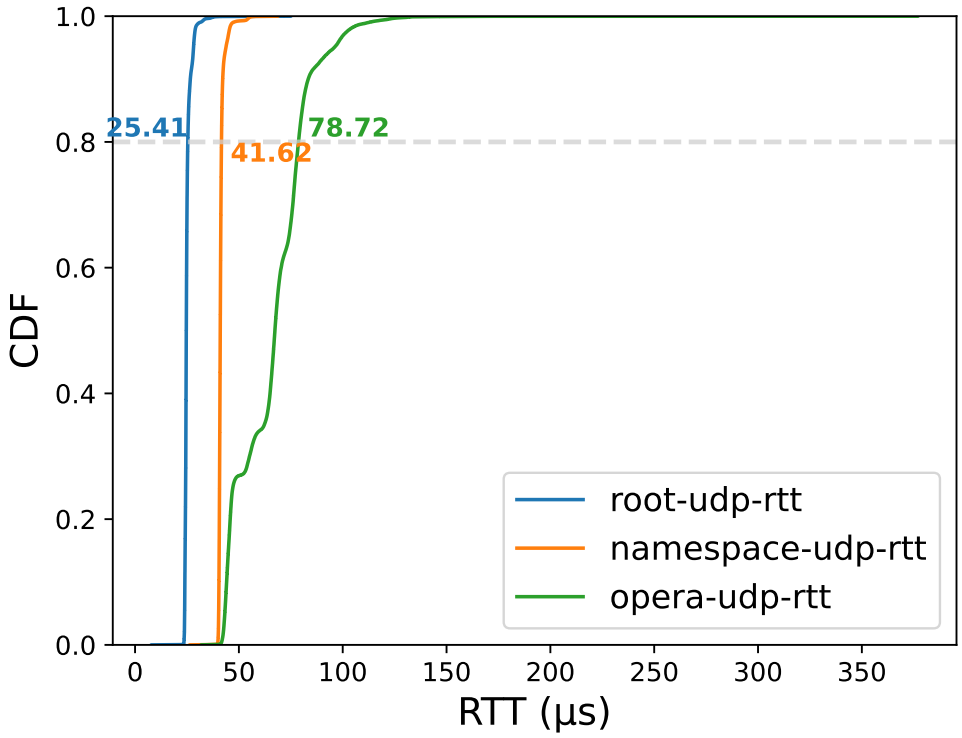


Figure 4.8. Node-to-Node and Opera RTT

However, XDP only allows the creation of multiple sockets to just one netdev/q1 when multiple sockets share the UMEM. We report the issue in the GitHub repo. Another limitation we came across is the fact that sharing UMEM with VETH side sockets alongside the NIC sockets makes us lose the zero-copy advantage we have with physical NIC sockets.

4.5 Related Work

Etalon is a reconfigurable datacenter network emulator that works for a hybrid network model, consisting of both packet switches and OCSes. It is based on Click software switch that uses DPDK for packet processing. The main idea of the paper is to dynamically resize top-of-rack switch virtual output queues to help ramp up TCP quickly enough to suit the reconfigurable network. Time-division TCP (TDTCP), is another new TCP variant designed for reconfigurable data center networks that track the state variables across all time-division networks (TDNs). It is

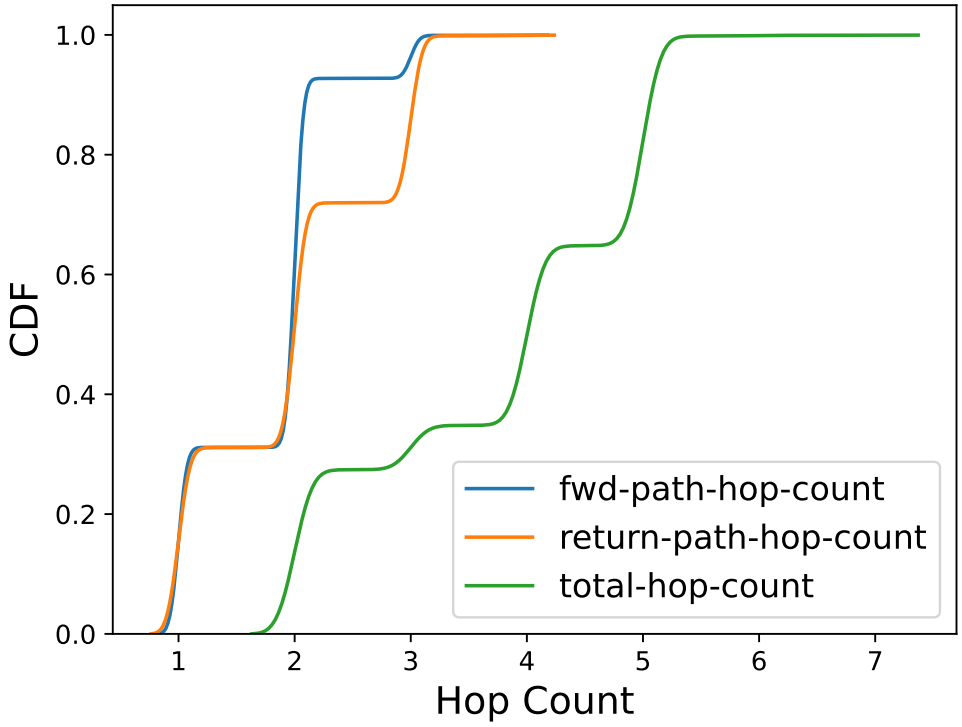


Figure 4.9. Number of hops a packet takes when running with Opera

also designed for hybrid networks. Even though there are many OCS designs proposed over the years they are yet to explore the impact of their design on end host stack behavior.

4.6 Conclusion

Emulators are essential in computer science research to test the feasibility of novel ideas cost-effectively. They greatly help speed up and simplify system development and debugging. We use XDP and AF_XDP to build an all-optical circuit switch emulator on a 32-node cluster. We further utilize XDP/AF_XDP as a software interface to the FPGA-based NIC that directly interacts with an actual Optical Circuit Switch. Our goal is to use the XDP/AF_XDP emulator to verify the practicality of the Optical Circuit Switch design and understand the behavior of the end-host network stack. In this chapter, we recorded our use case and our experience in using XDP/AF_XDP along with the challenges, and limitations we encountered during development.

Chapter 5

Mitigating Spurious Retransmissions In Opera

Now that we have the Rotor-XDP emulator, the question remains about Opera's system-wide performance. We would like to ask whether TCP is good enough for a network design that has an ever-changing topology in microseconds time scale. Prior experiments with Opera and RotorNet have only been conducted on a simulation and with a custom network stack. Although the recently developed rotor-based optical circuit switch which is based on the aforementioned papers runs with an unmodified TCP, it only shows RTT measurements with Ping and UDP and not a comprehensive study of the endhost stack behavior. All of this prior work focused entirely on the OCS design shedding very little light on the endhost stack behaviour. We plan to remedy that with this chapter.

When packets belonging to a TCP flow are sent over multi-hop paths due to the nature of Opera's design, they are reached at the destination out of order. Therefore, handling reordering becomes an important part of Opera. To that end, in this chapter, we explore the features of TCP to minimize the spurious retransmissions that occur due to reordering.

5.1 TCP Loss Recovery Algorithms

To detect and repair packet losses TCP uses various loss recovery algorithms. We first give an overview of these algorithms as they are essential in handling spurious retransmissions.

Fast Retransmission with SACK - The receiver sends TCP Selective Acknowledgments to inform the sender of the data that has been received. The sender maintains a scoreboard with this information to keep track of duplicate acknowledgments and retransmits the missing data segments when the number of DupAcks reaches a threshold.

RACK-TLP - This has two parts. Recent Acknowledgment (RACK) keeps a virtual timer for every transmitted data segment and inferences losses when the timer expires. This virtual timer is set dynamically based on the minimum RTT plus an additional delay budget to account for reordering. Tail Loss Probe (TLP) on the other hand uses RACK to trigger ACK feedback by sending a probe packet to avoid expensive retransmission timeouts (RTO).

RTO recovery - Sender restores to RTO recovery when all else fails to repair the losses. RTO is initially set to one second and then decreases according to connections smoothed RTT (SRTT) plus four times the RTT variation. Once a retransmitted packet gets lost, RTO exponentially backs off. Maximum and minimum allowed timeout are 120s and 200ms respectively.

Next, we evaluate the extent of spurious retransmissions in Opera and explore how to mitigate them by tuning TCP loss recovery algorithms.

5.2 Evaluation

We evaluate our system on Cloudlab c6525-25g (AMD EPYC 7302P 16-Core Processor) servers that have two dual-port Mellanox ConnectX-5 25Gb NICs. Dual-port NICs allow us to run PTP on a separate interface and busy poll the NIC clock from the emulator to determine the correct topology for each node. We run Ubuntu 22.04.2 LTS with kernel 5.15.0- 86-generic version on all 32 servers. MTU for VETH is set to 3400 bytes and for the physical NIC, it is set to 3490 bytes to accommodate the GRE headers. We describe the testbed in the following section.

5.2.1 Testbed

Our testbed consists of 32 nodes and the rotor controller runs on each node faithfully mimicking the OCS behavior as well as the hardware that directly connects with the OCS. All nodes are time synchronized with each other using PTP so that the packets are sent at the right times (in the correct topology) by all nodes. The rotor controller in each node handles the scheduling, routing, and queue management. We host our applications in containers with an unmodified TCP stack to understand the host stack behavior when it interacts with the Rotor design. We create a VETH pair for each container; One we place inside the container and the other one in the root namespace. The physical NIC and the VETH in the root namespace both have XDP programs running in the driver so that the incoming frames can be redirected to the emulator which is an AF_XDP program. Each AF_XDP socket in the emulator is bound to a specific queue ID (HW Queue/VETH queue) and consists of two rings: RX and TX. This enables the emulator to receive packets from the container, process them, and send them out via NIC and vice versa. The emulator acts as the rotor controller and forms a bridge between applications hosted in containers and the physical NIC.

A packet coming from a VETH interface first needs to be classified into two categories; low latency or bulk traffic (Bulk-Local). Packets are then queued by their original destination in local per-dest queues. Then the next hop of the packet is deter- mined by the current topology the assumed OCS is in and the packet is encapsulated with GRETAP. The outer Ether- net header of GRETAP points to the next hop of the packet and might or might not be its final destination. Finally, the packets are sent out via the mapped hardware queue. In the RX path, when a packet is received from the physical NIC if the packet is destined for that node, first we decapsulate the outer headers (remove GRE headers) and move it to the corresponding VETH queue which is then forwarded to the application. If the packet is not destined for the node we first move it to the corresponding per-dest non-local queue and calculate the routing information to determine the next hop. Non-local traffic has a higher priority than local traffic to reduce the latencies of

intermediate traffic.

5.2.2 Topology and Routing

Unlike most of the OCS designs which assume to have a blackout period during circuit reconfiguration, Opera provides connectivity among all nodes at all times by staggering the reconfigurations. We assume we have eight rotor switches and that each rotor rotates through four matchings. We do not reconfigure the rotors all at once to prevent total loss of connectivity. We stagger the reconfigurations so that at a given time only one rotor is reconfiguring and the rest of the seven rotors make up the topology. This gives us altogether 32 topologies. In each topology, each node will have direct access to seven nodes and indirect paths to all other nodes.

The slot duration of a single topology is determined by the accuracy of PTP and the node-to-node ping latency (RTT) between containers. Opera routes all packets as soon as they are received by VETH to the destination via either direct or indirect paths based on the topology the rotor switch is in. The maximum hop count a packet takes on a one-way direction is three if the packet can get to the other end within the timeslot (Fig. 5.5). However, if a packet is sent near the end of a time slot there is a high chance that it might go through less or more than the intended number of hops since the next hop might have moved on to a different topology. Based on the ping RTT we observed for the direct container-to-container scenario (Fig. 5.1), we experimented Opera with 100μs, 200μs, and 1ms time slots.

95th percentile ping RTT for direct container to container (between node-1 and node-2) is $\approx$41μs. We compare this result with Opera under various time slots between the same two nodes (Fig. 5.1). The increased RTT we see for Opera is due to packets taking multiple hops to reach the destination as per the topology rotation (Fig. 5.2). We can see that the RTT distribution closely follows the hop count distribution. The maximum number of hops a packet takes during a full RTT for Opera between node-1 and node-2 is five if it follows the intended Opera path meaning that the current topology has not changed during the RTT. However, we see a maximum of eight hops for 100μs slot length and seven for 200μs and 1ms slot lengths within a single RTT

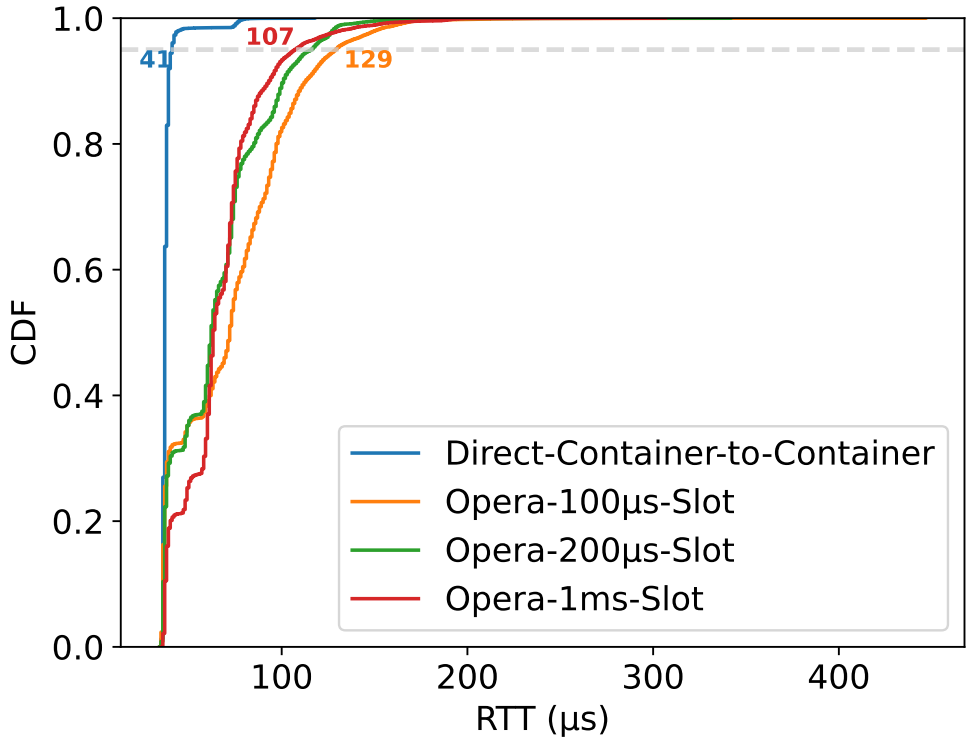


Figure 5.1. Ping RTT Distribution when Opera is moving through all 32 topologies between node-1 and node-2

indicating that a topology change has occurred amid RTT for some packets. If a topology change has occurred during a packet journey to the destination we define that one-way path (forward path or return path) as a rogue path. The highest number of rogue paths (30%) are seen for the 100μs slot experiment followed by the 200μs slot experiment (13%). Only 3% of rogue paths are observed for the 1ms slot length experiment. The percentages are consistent across multiple runs of the experiment. The higher rogue path count of shorter time slots can be explained by the fact that the 100μs slot length is not enough for a packet to get back to the source under the same topology if has to follow multiple hops, given the node-to-node RTT of $\approx$41μs. The overhead of the software NIC (Rotor Controller) also contributes to the high RTT of Opera. If the routing logic can be implemented in hardware where packets hop between hardware NICs within nanoseconds timescale without going back up the host, RTT can be drastically reduced

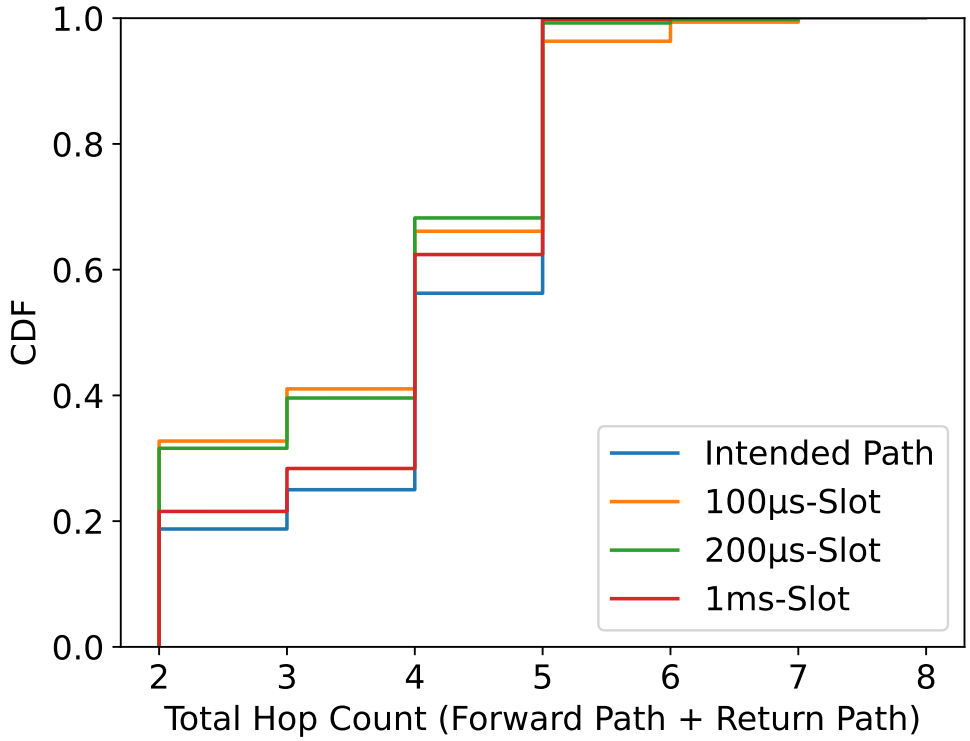


Figure 5.2. Distribution of hop counts between node-1 and node-2 for ping experiments conducted in Fig. 5.1

even if a packet has to take multiple hops.

5.2.3 TCP Stack Analysis

We use 200μs slot length for TCP stack analysis with Opera. This is a trade-off between cycle time and the intensity of rogue paths. 1ms slot length has only 3% rogue paths but has a high cycle time of 32ms. 100μs slot length on the other hand has a high rogue path percentage (30%) but a small cycle time of 3.2ms. 200μs gives us the best of both worlds with just 6.4ms cycle time with only 13% rogue paths.

We run a single instance of `iperf3` with an unmodified TCP stack inside a single container. The Rotor controller has 32 local and non-local queues and the traffic is mapped to a single hardware queue with AF_XDP. We rate limit each container to 11 Gbps and the goodput achieved between two nodes for this simple setup without Opera for a single container is 10

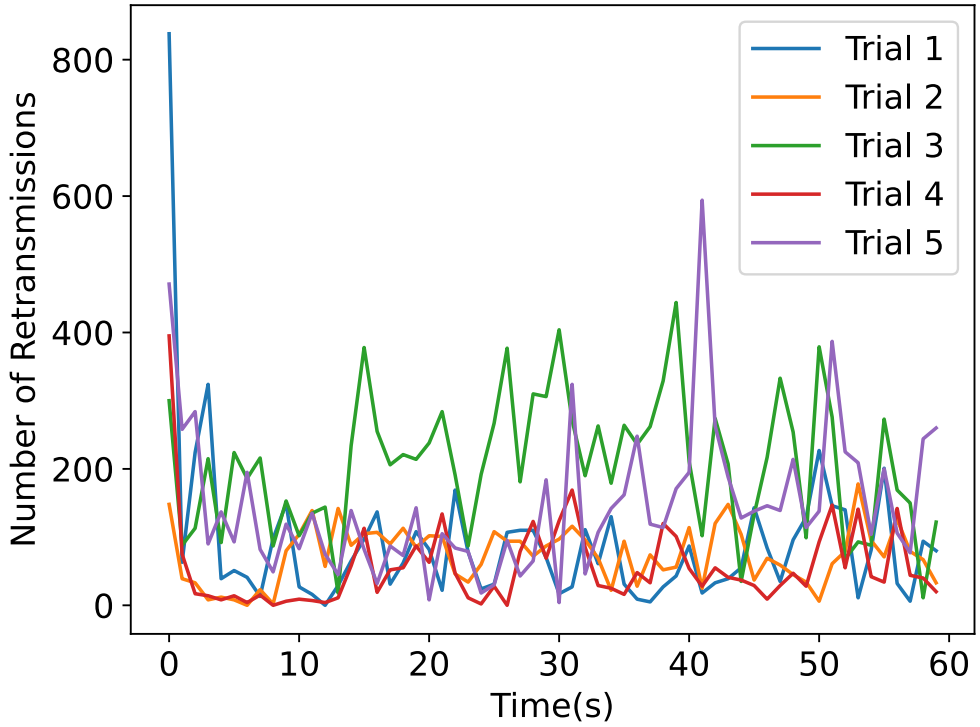


Figure 5.3. Number of spurious retransmission observed in each second for five trials of `iperf3` during a 60s period with Opera when loss recovery algorithm is RACK.

Gbps with zero retransmissions. Throughput can be linearly increased with hardware queues and containers. The main goal of this seemingly simple experiment is to uncover the behavior of the traditional TCP stack when it interacts directly with an ever-changing topology like Opera.

We run `iperf3` for 60s and this includes ≈9375 rotor cycles. As expected packet reordering becomes the number one issue with Opera when running with the traditional TCP stack. Packets belonging to a single flow traverse through multiple topologies following different paths and end up in different order at the destination causing spurious retransmissions from the sender. Fig. 5.3 shows the number of spurious retransmissions observed for five trials of `iperf3` during a 60s period.

Question: Can we eliminate spurious retransmissions with existing features of TCP?

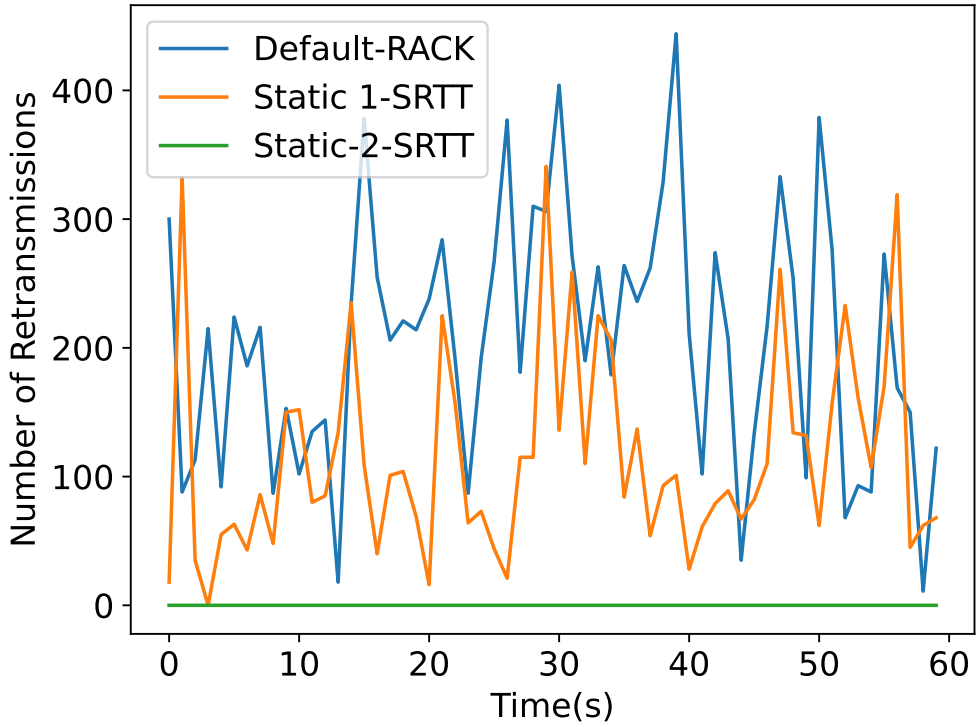


Figure 5.4. Number of spurious retransmissions observed for different reordering windows. The reordering window for default RACK is dynamically adjusted between MIN_RTT/4 and 1-SRTT based on the window step value.

RACK is the default TCP loss recovery algorithm used in Linux. To be more reorder resilient, RACK uses a reordering window of `min_rtt/4` as a settling delay. This reordering window can be dynamically increased up to smoothed RTT (SRTT) with the help of a window step value maintained by RACK. RACK increases this step value every time it sees a Duplicate Selective Acknowledgement (DSACK). A DSACK suggests that the sender has made a spurious retransmission and that the reordering window might be too small as a settling delay. Therefore, the reorder window is allowed to increase up to 1 SRTT according to the window step value. Finally, RACK marks a packet as lost if it has not been sacked beyond the recent RTT plus the reordering window [64].

All of the spurious retransmissions we see in Fig. 5.3 are due to the expiry of this RACK virtual timer. To figure out whether it is the minimum RTT or the Smoothed RTT that has been

used for the reordering window calculation, we keep track of the window step value maintained by RACK with a BPF hook (Table 5.1). The RACK window step spends the majority of the time in its maximum allowed value (between 128 and 255) indicating that the reordering window uses SRTT. Currently, there is no mechanism in Linux to increase the reordering window beyond 1 SRTT. Therefore, we modified the kernel to test multiples of SRTT values with Opera. Fig. 5.4 shows that at least 2-SRTT timeout is needed to bring down the spurious retransmission count to zero.

When RACK is disabled it falls back to DupAck threshold. However, there is no DupAck counter when SACK is enabled. Therefore, TCP uses the total number of SACKed segments for DupAck heuristics. For fast retransmission with SACK, both DupAck heuristics and reorder detection are needed. TCP determines that a reorder has occurred when a higher sack sequence has been delivered before a lower never-retransmitted sequence [64]. The reorder distance is approximated in full-mss packet distance and is configurable in Linux with `tcp_reordering` and `tcp_max_reordering` configurations. `tcp_reordering` indicates the initial reorder level (default 3) and is allowed to go up to `tcp_max_reordering` (default 300). TCP dynamically adjusts the reorder level between these two values. If the DupAck count exceeds the reordering level, the connection enters the recovery stage, reducing the congestion window.

With Opera the number of spurious retransmissions we see with the SACK DupAck threshold with default reorder levels is no different from RACK (Fig. 5.3). Therefore, we set up various values for `tcp_reordering` and `tcp_max_reordering` to track the reordering distance needed by Opera to reduce the spurious transmissions with SACK. We observe that the spurious retransmissions can be reduced to zero for `iperf3` if the maximum reorder distance is set to at least 4095 (Table 5.2).

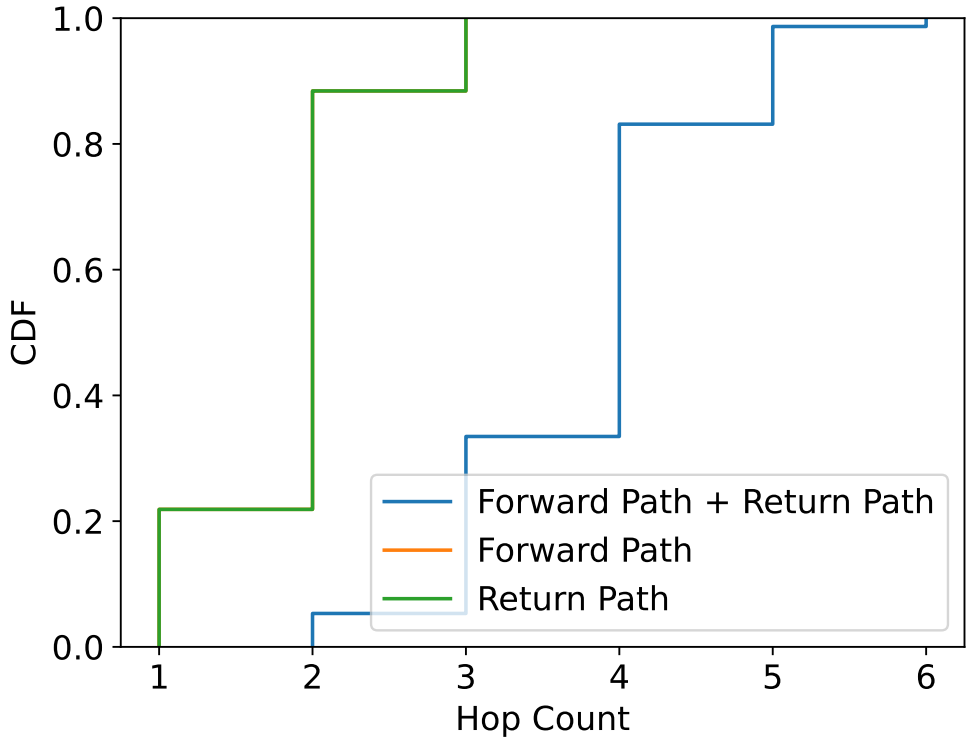


Figure 5.5. Distribution of hop counts for intended paths between all 32 nodes (32x32)

5.3 Conclusion

In this chapter, we demonstrate that unnecessary retransmissions caused by dynamic reconfiguration can be mitigated using existing TCP loss recovery mechanisms. However, in Linux, there is no built-in support for increasing the reordering window beyond one SRTT when using RACK, the default loss recovery algorithm, without modifying the kernel. In contrast, adjusting the SACK DupAck threshold provides an alternative approach to reducing spurious retransmissions. This method is less intrusive than modifying RACK, which requires kernel-level changes to extend the reordering window and eliminate such retransmissions.

Table 5.1. RACK Window Step Distribution

Window Step	Count
0 - 1	271
2 - 3	324
4 - 7	504
8 - 15	1881
16 - 31	22003
32 - 63	318420
64 - 127	653476
128 - 255	2190298

Table 5.2. Histogram for Reorder Distance (SACK)

Reorder Distance	Count
0 - 1	0
2 - 3	58
4 - 7	0
8 - 15	0
16 - 31	1363
32 - 63	1673
64 - 127	145
128 - 255	639506
256 - 511	314243
512 - 1023	1545893
1024 - 2047	51
2048 - 4095	1294703

Chapter 6

Conclusion

In this dissertation, we explored how reconfigurable networks can be leveraged to improve the communication performance of large-scale DNN training workloads while also reducing overall system fragmentation. We first investigated the benefits of dynamically reconfiguring network links at every step of collective communication algorithms to maximize performance gains. While fine-grained reconfiguration at every step provides the highest potential performance improvement, an important direction for future work is to investigate whether selectively reconfiguring only the most congestion-prone steps can still deliver significant performance gains while relaxing reconfiguration latency requirements. This highlights an important tradeoff between performance improvement and reconfiguration flexibility that warrants further exploration in future work. We then demonstrated how increasing the flexibility of Torus networks through reconfiguration can help mitigate system fragmentation in multi-tenant ML clusters. Finally, we extended the concept of flexibility even further by enabling tenants to directly control how their allocated topology slices are reconfigured, allowing them to optimize the network for the communication characteristics of their individual workloads. Together, these contributions illustrate the potential of reconfigurable network architectures to address both the performance and resource efficiency challenges of next-generation AI datacenters.

Bibliography

[1] AF_XDP — The Linux Kernel documentation. https://www.kernel.org/doc/html/next/networking/af_xdp.html.

[2] DPDK Community. DPDK - DataPlane Development Kit. https://www.dpdk.org/.

[3] 3D PHOTONICS FOR AI. Passage™. https://lightmatter.co/products/passage/, March 2025.

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, page 63–74, New York, NY, USA, 2008. Association for Computing Machinery.

[5] Rukshani Athapathu and George Porter. Reconfigurability within collective communication algorithms. In *Proceedings of the 2nd Workshop on Networks for AI Computing*, NAIC '25, page 43–49, New York, NY, USA, 2025. Association for Computing Machinery.

[6] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 782–797, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Hitesh Ballani, Paolo Costa, Istvan Haller, Krzysztof Jozwik, Kai Shi, Benn Thomsen, and Hugh Williams. Bridging the last mile for optical switching in data centers. In *2018 Optical Fiber Communications Conference and Exposition (OFC)*, pages 1–3, 2018.

[8] M. Barnett, R. Littlefield, D.G. Payne, and R. van de Geijn. Global combine on mesh architectures with wormhole routing. In *[1993] Proceedings Seventh International Parallel Processing Symposium*, pages 156–162, 1993.

[9] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 62–75, 2021.

[10] Minsik Cho, Ulrich Finkler, David S. Kung, and Hillery C. Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. In *Proceedings of the 2nd SysML Conference (MLSys)*, pages 241–251, Palo Alto, CA, USA, 2019. MLSys.

[11] Hyunseung Choo, Seong-Moo Yoo, and Hee Yong Youn. Processor scheduling and allocation for 3d torus multicomputer systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(5):475–484, 2000.

[12] EpiPhotonics. Nano-Second Speed PLZT Photonics. http://epiphotonics.com/products.html, March 2025.

[13] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. Corundum: An open-source 100-gbps nic. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 38–46, 2020.

[14] Johannes Gerstmayr, Peter Manzl, and Michael Pieber. Multibody models generated from natural language. *Multibody System Dynamics*, 62(2):249–271, 2024.

[15] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 216–229, New York, NY, USA, 2016. Association for Computing Machinery.

[16] Google. Accelerate AI development with Google Cloud TPUs. https://cloud.google.com/tpu, January 2026.

[17] Google. TPU v4. https://docs.cloud.google.com/tpu/docs/v4, January 2026.

[18] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. The architectural implications of facebook's dnn-based personalized recommendation. *CoRR*, abs/1906.03109, 2019.

[19] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. Number UCB/EECS-2015-155, May 2015.

[20] Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, USA, 1st edition, 2013.

[21] HLC-Lab. Swing Allreduce Simulator. https://github.com/HLC-Lab/swing-allreduce-sim, March 2025. original-date: 2025-03-15T17:21:48Z.

[22] Torsten Hoefler, Tommaso Bonato, Daniele De Sensi, Salvatore Di Girolamo, Shigang Li, Marco Heddes, Jon Belk, Deepak Goel, Miguel Castro, and Steve Scott. Hammingmesh: A network topology for large-scale deep learning. *CoRR*, abs/2209.01346, 2022.

[23] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery.

[24] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*, pages 1–15, St. Louis, MO, USA, November 2021. Association for Computing Machinery (ACM).

[25] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. *GPipe: efficient training of giant neural networks using pipeline parallelism*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[26] Sylvain Jeaugey. Distributed Deep Neural Network Training: NCCL on Summit. https://www.olcf.ornl.gov/wp-content/uploads/2019/12/Summit-NCCL.pdf, 2019. Presentation slides.

[27] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 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*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[28] Debajyoti Kar, Arindam Khan, and Malin Rau. Improved approximation algorithms for three-dimensional bin packing. In *Proceedings of the 52nd International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 334 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 104–? Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025.

[29] Mehrdad Khani, Manya Ghobadi, Mohammad Alizadeh, Ziyi Zhu, Madeleine Glick, Keren Bergman, Amin Vahdat, Benjamin Klenk, and Eiman Ebrahimi. Sip-ml: high-bandwidth optical network interconnects for machine learning training. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 657–675, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, aug 2000.

[31] Abhishek Vijaya Kumar, Eric Ding, Arjun Devraj, Darius Bunandar, and Rachee Singh. Morphlux: Transforming torus fabrics for efficient multi-tenant ml, 2025.

[32] Kang Lee and J Eldson. Standard for a precision clock synchronization protocol for networked measurement and control systems. Proceedings of the 2004 Conference on IEEE 1588, Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems, Gaithersburg, MD, 2004-09-27 2004.

[33] Kangkang Li, Maciej Malawski, and Jarek Nabrzyski. Reducing fragmentation on 3d torus-based hpc systems using packing-based job scheduling and job placement reconfiguration. In *Proceedings of the 16th IEEE International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 34–43. IEEE, 2017.

[34] Kangkang Li, Maciej Malawski, and Jarek Nabrzyski. Topology-aware job allocation in 3d torus-based hpc systems with hard job priority constraints. In *Proceedings of the International Conference on Computational Science (ICCS)*, volume 108, pages 515–524. Elsevier, 2017.

[35] Tony Li, Dino Farinacci, Stanley P. Hanks, David Meyer, and Paul S. Traina. Generic Routing Encapsulation (GRE). Request for Comments RFC 2784, Internet Engineering Task Force, March 2000.

[36] Xudong Liao, Yijun Sun, Han Tian, Xinchen Wan, Yilun Jin, Zilong Wang, Zhenghang Ren, Xinyang Huang, Wenxue Li, Kin Fai Tse, Zhizhen Zhong, Guyue Liu, Ying Zhang, Xiaofeng Ye, Yiming Zhang, and Kai Chen. Mixnet: A runtime reconfigurable optical-electrical fabric for distributed mixture-of-experts training. In *Proceedings of the ACM SIGCOMM 2025 Conference*, SIGCOMM '25, page 554–574. ACM, August 2025.

[37] Hong Liu, Ryohei Urata, Kevin Yasumura, Xiang Zhou, Roy Bannon, Jill Berger, Pedram Dashti, Norm Jouppi, Cedric Lam, Sheng Li, Erji Mao, Daniel Nelson, George Papen, Mukarram Tariq, and Amin Vahdat. Lightwave fabrics: At-scale optical circuit switching for datacenter and machine learning systems. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 499–515, New York, NY, USA, 2023. Association for Computing Machinery.

[38] Xuting Liu, Behnaz Arzani, Siva Kesava Reddy Kakarla, Liangyu Zhao, Vincent Liu, Miguel Castro, Srikanth Kandula, and Luke Marshall. Rethinking machine learning collective communication as a multi-commodity flow problem. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 16–37. ACM, 2024.

[39] Arun Majumdar and Jennifer C. Ricklin. Free-space laser communications : principles and advances. 2008.

[40] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Kärnä, Diana Moise, Simon J. Pennycook, Kristyn Maschhoff, Jason Sewall, Nalini Kumar, Shirley Ho, Michael F. Ringenburg, Prabhat, and Victor Lee. Cosmoflow: using deep learning to learn the universe at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2019.

[41] William M. Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C. Snoeren, and George Porter. Expanding across time to deliver bandwidth efficiency and low latency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1–18, Santa Clara, CA, February 2020. USENIX Association.

[42] William M. Mellette, Alex Forencich, Rukshani Athapathu, Alex C. Snoeren, George Papen, and George Porter. Realizing rotornet: Toward practical microsecond scale optical networking. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 392–414, New York, NY, USA, 2024. Association for Computing Machinery.

[43] William M. Mellette, Alex Forencich, Rukshani Athapathu, Alex C. Snoeren, George Papen, and George Porter. Realizing rotornet: Toward practical microsecond scale optical networking. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 392–414, New York, NY, USA, 2024. Association for Computing Machinery.

[44] Samuel K. Moore. Another step toward the end of moore's law: Samsung and tsmc move to 5-nanometer manufacturing - [news]. *IEEE Spectrum*, 56(6):9–10, 2019.

[45] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, Krishnakumar Nair, Isabel Gao, Bor-Yiing Su, Jiyan Yang, and Mikhail Smelyanskiy. Deep learning training in facebook data centers: Design of scale-up and scale-out systems. *CoRR*, abs/2003.09518, 2020.

[46] NVIDIA. NVIDIA H100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/h100/, March 2025.

[47] NVIDIA. NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100/, January 2026.

[48] NVIDIA Corporation. *NVIDIA Collective Communications Library (NCCL) Documentation*, 2024. Accessed: 2024-05-16.

[49] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Integrating microsecond circuit switching into the data center. *SIGCOMM Comput. Commun. Rev.*, 43(4):447–458, aug 2013.

[50] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter evolving: Transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 66–85, New York, NY, USA, 2022. Association for Computing Machinery.

[51] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. CASSINI: Network-Aware job scheduling in machine learning clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1403–1420, Santa Clara, CA, April 2024. USENIX Association.

[52] R. Ryf, J. Kim, J.P. Hickey, A. Gnauck, D. Carr, F. Pardo, C. Bolle, R. Frahm, N. Basavanhally, C. Yoh, D. Ramsey, R. Boie, R. George, J. Kraus, C. Lichtenwalner, R. Papazian, J. Gates, H.R. Shea, A. Gasparyan, V. Muratov, J.E. Griffith, J.A. Prybyla, S. Goyal, C.D. White, M.T. Lin, R. Ruel, C. Nijander, S. Arney, D.T. Neilson, D.J. Bishop, P. Kolodner, S. Pau, C. Nuzman, A. Weis, B. Kumar, D. Lieuwen, V. Aksyuk, D.S. Greywall, T.C. Lee, H.T. Soh, W.M. Mansfield, S. Jin, W.Y. Lai, H.A. Huggins, D.L. Barr, R.A. Cirelli, G.R. Bogart, K. Teffeau, R. Vella, H. Mavoori, A. Ramirez, N.A. Ciampa, F.P. Klemens, M.D. Morris, T. Boone, J.Q. Liu, J.M. Rosamilia, and C.R. Giles. 1296-port mems transparent optical crossconnect with 2.07 petabit/s switch capacity. In *OFC 2001. Optical Fiber Communication Conference and Exhibit. Technical Digest Postconference Edition (IEEE Cat. 01CH37171)*, volume 4, pages PD28–PD28, 2001.

[53] Daniele De Sensi, Tommaso Bonato, David Saam, and Torsten Hoefler. Swing: Short-cutting rings for higher bandwidth allreduce. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1445–1462, Santa Clara, CA, April 2024. USENIX Association.

[54] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. Taccl: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 593–612. USENIX Association, 2023.

[55] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake A. Hechtman. Mesh-tensorflow: Deep learning for supercomputers. *CoRR*, abs/1811.02084, 2018.

[56] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[57] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A network architecture for disaggregated racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 255–270, Boston, MA, February 2019. USENIX Association.

[58] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter

network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 183–197, New York, NY, USA, 2015. Association for Computing Machinery.

[59] Brian Slechta, Nick Comly, Ashraf Eassa, Joe DeLaere, and Shivam Raj. NVIDIA NVLink and NVIDIA NVSwitch Supercharge Large Language Model Inference. https://developer.nvidia.com/blog/nvidia-nvlink-and-nvidia-nvswitch-supercharge-large-language-model-inference/, Aug 2024. NVIDIA Technical Blog.

[60] Shunichi Sohma, Toshio Watanabe, Tomohiro Shibata, and Hiroshi Takahashi. Compact and low power consumption 16 × 16 optical matrix switch with silica-based plc technology. In *Optical Fiber Communication Conference and Exposition and The National Fiber Optic Engineers Conference*, page OThV4. Optica Publishing Group, 2005.

[61] SST Simulator. The Structural Simulation Toolkit. http://sst-simulator.org/, April 2025. Accessed: (2025-04-30).

[62] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, February 2005.

[63] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, February 2005.

[64] The Linux Kernel Organization. Linux Kernel Source Code. https://www.kernel.org/, 2026. Version 6.x.

[65] Jesper Larsson Träff, Antoine Rougier, and Sascha Hunold. Implementing a classic: zero-copy all-to-all communication with mpi datatypes. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, page 135–144, New York, NY, USA, 2014. Association for Computing Machinery.

[66] Ryohei Urata, Hong Liu, Kevin Yasumura, Erji Mao, Jill Berger, Xiang Zhou, Cedric Lam, Roy Bannon, Darren Hutchinson, Daniel Nelson, Leon Poutievski, Arjun Singh, Joon Ong, and Amin Vahdat. Mission apollo: Landing optical circuit switching at datacenter scale, 2022.

[67] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, 1982.

[68] Richa Verma, Aniruddha Singhal, Harshad Khadilkar, Ansuma Basumatary, Siddharth Nayak, Harsh Vardhan Singh, Swagat Kumar, and Rajesh Sinha. A Generalized Reinforcement Learning Algorithm for Online 3D Bin-Packing. *arXiv preprint arXiv:2007.00463*, 2020.

[69] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil R. Devanur, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. In *Proceedings of the Third Conference on Machine Learning and Systems (MLSys 2020)*, pages 172–186. MLSys, 2020. Presented March 2020.

[70] Leilei Wang, Manhang Zheng, Xin Lu, Wenhe Yin, Xuwei Xue, Qi Sun, Zhenxing Sun, Jiaqiang Nie, Zhiqian Yin, Rulei Xiao, Mi Li, Yunshan Zhang, Jun Lu, Yuan Du, Li Du, Tao Fang, Chunhui Zhang, Feng Wang, Yashe Liu, and Xiangfei Chen. Enhanced-performance tunable sources for fast awgr-based optical switching data center networks. *Journal of Lightwave Technology*, 42(24):8598–8605, 2024.

[71] Weitao Wang, Dingming Wu, Sushovan Das, Afsaneh Rahbar, Ang Chen, and T. S. Eugene Ng. RDC: Energy-Efficient data center network congestion relief with topological reconfigurability at the edge. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1267–1288, Renton, WA, April 2022. USENIX Association.

[72] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739–767, Boston, MA, April 2023. USENIX Association.

[73] William Won, Midhilesh Elavazhagan, Sudarshan Srinivasan, Swati Gupta, and Tushar Krishna. Tacos: Topology-aware collective algorithm synthesizer for distributed machine learning. In *57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 856–870, 2024.

[74] Wencong Xiao, Rishabh Bhardwaj, Deepak Narayanan, Qizhen Chen, Kay Ousterhout, and Ion Stoica. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610. USENIX Association, 2018.

[75] John T. W. Yeow, K. L. Eddie Law, and Andrew A. Goldenberg. Mems optical switches. *IEEE Commun. Mag.*, 39:158–163, 2001.

[76] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, Steve Lacy, Hang Wang, Aaron Wisner, Chris Lewis, and Henri Bahini. Resiliency at scale: Managing google's tpuv4 machine learning supercomputer. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 761–774, Santa Clara, CA, USA, 2024. USENIX Association.