

Reshaping Data Center Networks with Reconfigurability

Shawn Shuoshuo Chen

CMU-CS-26-124

August 2026

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Srinivasan Seshan, Co-Chair
Peter Steenkiste, Co-Chair
Tim Dettmers
Minlan Yu (Harvard)

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Copyright © 2026 Shawn Shuoshuo Chen

This research was supported by NSF grants CNS-1565343, CNS-1122444, U.S. Army Contracting Command Award W911QX20D0008, an Argonne Leadership Computing Facility award, a CMU Provost / Graduate Student Assembly funding award, and gifts from Google and Lambda Labs. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Data center networking, Optical circuit switching, Transport protocol, Traffic engineering, Scheduling, Topology reconfiguration, AI data center

For Ziyue, my love and my spark

Abstract

Every web search, video stream, and chatbot reply is served not by one computer but by thousands of them, packed into warehouse-sized data centers and stitched together by an internal network. For decades that network was scaled by brute force. When the machines needed to talk faster, operators bought faster switches and wired in more of them. This approach is now failing. The chips inside those switches have stopped getting faster the way they once did, and the standard design insists that every machine be able to reach every other at full speed at once, so keeping up means buying more hardware at prohibitively higher cost and power. Then came AI, whose appetite for shuffling data between chips dwarfs anything before it. The result is a widening gap between the bandwidth a data center needs and the bandwidth anyone can afford to build, a gap we call the data center scaling crisis.

Yet a data center rarely needs all that bandwidth at once. Traffic is often bursty and localized: a few machines communicating heavily while the rest stay quiet, so most of a uniform network's capacity sits idle. The smarter move is to shift bandwidth on the fly to wherever demand actually is, and that is exactly what a reconfigurable data center network (RDCN) does. To do so it relies on a kind of switch very different from the packet switches networks use today, the optical circuit switch (OCS), which is both cheaper and more energy-efficient. The catch is that this is a paradigm shift from today's packet-switched networks: decades of protocols quietly assume what a circuit-switched RDCN cannot promise, uniform bandwidth between every pair of machines and static wiring that never changes. When those assumptions break, the overall performance suffers.

This thesis redesigns the network protocol stack for the RDCN to reclaim the performance that the unmodified stack gives up. It tackles the three core components of that stack in turn: the transport protocol, routing and traffic engineering, and job scheduling. For transport, conventional congestion control stumbles when the path beneath it keeps changing, so our work Time-Division TCP (TDTCP) keeps separate state for each network configuration and reuses it across reconfigurations, delivering 24–41% higher throughput than existing TCP. For traffic engineering, a spine-free optical data center network offers uneven, shifting paths that overwhelm today's switch hardware, so our work PreciseTE computes routes that respect those hardware limits, holding precision loss below 7% where the state of the art loses up to 67%, and is now deployed in Google's production network. For job scheduling, ML jobs placed on a static non-reconfigurable torus must trade fragmentation against contention, and our work RFold co-adapts the job's shape and the topology to escape both, cutting average job completion time by more than $4\times$ at high load. Overall, reconfigurability is reshaping data center network infrastructure, but unlocking its performance benefits calls for a systematic redesign of the protocol stack above it.

Acknowledgments

As Alfred Tennyson once wrote, “I am a part of all that I have met.” My PhD journey was not mine alone, and the half-decade behind this thesis was never a solo effort. The debts I accrued along the way are the part of it I am most glad to record.

My deepest thanks go to my advisors, Srinivasan Seshan and Peter Steenkiste. Just about every thesis brags about having the world’s best advisor. I went one better and got two. Being co-advised by Srini and Peter gave me two different perspectives to draw on, which made every hard problem easier to see clearly. They are patient, supportive, and endlessly insightful. They gave me complete freedom to follow my curiosity and encouraged me to chase risky ideas that others might have waved off. That kind of trust is the greatest gift a student can receive. From Srini I learned to keep my interests wide and to appreciate good work far outside my own area. He taught me that inspiration often arrives from the side, in the way someone tackled a completely different problem, and that a researcher who reads broadly is never short of ideas. Peter has an almost magical ability to impose structure on clutter. Research has a way of sliding into a messy pile of technical details until it loses all sense of direction, and Peter is the one who could step back, cut through the noise, and set us back on course.

I am grateful to other members of my thesis committee, Tim Dettmers and Minlan Yu, for their time, their careful reading, the thought provoking feedbacks, and the questions that pushed this work past where I would have left it. Tim and Minlan’s expertise in machine learning systems gave me the confidence to step into a field I had never thought to try. What first looked like unfamiliar and intimidating territory later became an area I was eager to explore, and this thesis’s turn toward AI and machine learning workloads owes a great deal to their guidance and encouragement.

I owe a special thanks to my faculty mentors Vyas Sekar and Theo Benson, who patiently listened to my venting through the most uncertain and frustrating times of this journey, challenged me and my work to make it better, and guided me through the academic job search. I am also grateful to Daniel Alves, Wei Bai, Mallesh Dasari, Sadjad Fouladi, Manya Ghobadi, Olivia Hsu, Weirong Jiang, Wenqi Jiang, Srikanth Kandula, Moein Khazraee, Daehyeok Kim, Ana Klimovic, Ilias Marinos, Jeffrey Mogul, Todd Mowry, Michael Papamichael, Jignesh Patel, Adrian Perrig, Leon Poutievski, Justine Sherry, David Sidler, Arjun Singh, Alex Snoeren, Amin Vahdat, Lorenzo Vicisano, Rui Wang, Keith Winstein, Ji Yang, Juncheng Yang, and Liangcheng Yu for their generous research and career advice over the years.

To my cohort, labmates, and friends, at CMU and beyond, thank you for the whiteboard arguments, the coffee breaks, the group lunches, the friendship that outlasted every paper deadline, and a life waiting for me outside the lab. This goes to: Anup Agarwal, Daiyaan Arfeen, Sanjith Athlur, Nirav Atre, Sagar Bharadwaj, Chris Canel, Lucas Barbosa Castanheira, Brian Chang, Hilbert Chen, Patrick Coppock,

Arjun Devraj, Eric Ding, Miguel Ferreira, Keqiang He, Yige Hong, Ankush Jain, Sheng Jiang, Timothy Kim, Yiran Lei, Zhouzi Li, Haoyi Liang, Sherman Lim, Ningxi Liu, Shinan Liu, Pragna Mamidipaka, Adithya Abraham Philip, Devdeep Ray, Hugo Sadok, Nitika Saran, Yash Savani, Noah Singer, Milind Srivastava, Muyang Sun, Ranysha Ware, Mingkuan Xu, Yue Yao, Mingyang Zhang, Kaiyang Zhao, Zhipeng Zhao, Zhizhen Zhong, and Mingxun Zhou.

None of this would have happened without my parents, whose love and support prepared me to embark on this journey and kept me going. I also could not have finished this thesis, or my PhD, without my partner, Ziyue Qiu, who has been through every up and down with me and is always there for me. Finally, I want to thank myself, for deciding to do this six years ago, and for not giving up on the days when quitting seemed like the easy choice.

This is worth it.

Contents

Table of Glossary	xix
1 Introduction	1
1.1 Summary of contributions	3
1.2 Thesis organization	6
2 Background: The Case for Optical Circuit Switching	7
2.1 The scaling crisis and the bandwidth wall	8
2.1.1 Demand is outrunning silicon	8
2.1.2 The bandwidth wall	9
2.1.3 The “one big switch” abstraction and the cost of scaling out	9
2.2 Optical circuit switching	10
2.2.1 What optical circuit switching is	10
2.2.2 OCS technologies	11
2.2.3 Advantages over packet switching	12
2.2.4 Disadvantages relative to packet switching	13
2.3 Reconfigurable data center networks	13
2.4 Why OCS makes sense in data centers	17
3 Time-Division TCP	19
3.1 Motivation	21
3.1.1 Hybrid demand-oblivious RDCNs	21
3.1.2 Measuring TCP performance in RDCNs	22
3.2 Design	23
3.2.1 Per-TDN state variables	24
3.2.2 TDN change notification	24
3.2.3 Sequence numbering	25
3.2.4 Packet reordering	26
3.2.5 Fairness, generality, and limitations	28
3.3 Implementation	29
3.3.1 Packet format	29
3.3.2 Connection establishment	30
3.3.3 TDN state management	30
3.3.4 RTT estimation	31

3.4	Evaluation	32
3.4.1	Etalon testbed	32
3.4.2	Bandwidth and latency difference	34
3.4.3	Reordering	37
3.4.4	TDN change notification	37
3.5	Related work	38
3.6	Conclusion	39
4	Precise Data Center Traffic Engineering	41
4.1	Background	43
4.1.1	Traffic engineering in data center networks	43
4.1.2	Spine-free data center networks	44
4.2	TE precision loss challenges	46
4.3	Design	49
4.3.1	Shared heuristics	49
4.3.2	Direct mixed-integer reduction	51
4.3.3	Iterative greedy reduction	53
4.4	Evaluation	54
4.4.1	FabricEval evaluation framework	54
4.4.2	Experimental setup	55
4.4.3	Spread and path diversity	55
4.4.4	Heterogeneous fabrics	57
4.4.5	Traffic pattern	59
4.4.6	Execution speed	59
4.4.7	Impact on flow completion time	60
4.4.8	Group Pruning’s impact on path diversity	61
4.4.9	Ablation study	62
4.4.10	Quantization error	62
4.4.11	Table usage	63
4.4.12	Recommendation	64
4.5	Related work	64
4.6	Conclusion	65
5	Machine Learning Job Placement	67
5.1	Background	69
5.1.1	ML communication pattern	69
5.1.2	Reconfigurable torus	70
5.1.3	Job scheduling	71
5.2	Motivation	72
5.3	RFold resource allocation	74
5.3.1	Reconfiguration	75
5.3.2	Folding	76
5.3.3	Placement algorithm	78
5.4	Evaluation	81

5.4.1	Workload characteristics	82
5.4.2	Admission and placement policies	84
5.4.3	Cluster utilization	87
5.4.4	Failure resilience	88
5.4.5	Reconfigurability	90
5.4.6	More workload patterns	92
5.5	Related work	95
5.6	Conclusion	97
6	Conclusion	99
6.1	New use cases enabled by faster reconfiguration	101
6.1.1	Adapting the topology to collective communication	101
6.1.2	Pooling spare accelerators across scale-up domains	102
6.1.3	Unified scale-up and scale-out bandwidth	103
6.1.4	Scaling an all-optical data center	104
6.2	Challenges that must be addressed	105
6.2.1	Faster reconfiguration is an end-to-end problem	105
6.2.2	An all-optical data center needs hierarchical, multi-hop switching	107
6.3	Concluding remarks	108
	Bibliography	109

List of Figures

1.1	The bandwidth gap. Switch-chip supply grows more slowly than either general-purpose or AI-driven bisection-bandwidth demand, so the gap between what a network must deliver and what faster switches can provide widens year over year. (Data is compiled from Broadcom, NVIDIA, Google, and SemiAnalysis reports.)	2
3.1	A hybrid, demand-oblivious RDCN topology. The OCS network provides temporary high-bandwidth links that supplement the EPS network.	21
3.2	Sequence graph of TCP variants running in hybrid RDCN for 3 optical weeks.	22
3.3	Cross-TDN reordering scenarios. Time proceeds from top to bottom; arrows represent TDTCP segments transmitted between sender and receiver.	27
3.4	An example of relaxed reordering detection.	28
3.5	TDTCP packet formats: (a) ICMP for TDN change notification, (b) TCP option for the TD_CAPABLE handshake, (c) TCP option of TD_DATA_ACK exchange.	29
3.6	Our Etalon deployment. The emulated packet and circuit networks are 10 Gbps and 100 Gbps, respectively. Note Etalon only emulates the RDCN, the hosts are implemented as containers on physical machines.	32
3.7	TCP throughput and switch buffer utilization under both bandwidth and latency difference: (a) sequence graph of each TCP variant, (b) ToR VOQ utilization over time (capped at 16 for all but reTCP with dynamic buffers, <code>retcpdyn</code>).	33
3.8	TCP throughput and switch buffer utilization with only bandwidth difference: (a) sequence graph of each TCP variant, (b) ToR VOQ utilization over time (capped at 16 for all but reTCP with dynamic buffers, <code>retcpdyn</code>).	34
3.9	TCP throughput with only latency difference at 100 Gbps bandwidth. (TDTCP and CUBIC perform almost identically.)	35
3.10	ToR VOQ utilization for an RDCN with only latency differences (max VOQ size is 16 packets for all cases except reTCP with dynamic buffers, <code>retcpdyn</code>). Packet and optical RTTs are 20 μ s and 10 μ s, respectively. Bandwidth is fixed.	36
3.11	CDFs of (a) reordering events and (b) packets subject to potential spurious retransmission, per optical day. Note the y -axis does not start from 0.	37
3.12	TDTCP performance with and without all optimizations on TDN change notification.	38
4.1	Hardware representation of flows/groups, and a demonstration of group reduction.	43
4.2	Traditional Clos and spine-free DCN topology. A path segment is an abstraction of parallel physical links.	45

4.3	(a) 30-day time series of maximum group table usage in 3 arbitrary production fabrics. Usage is normalized to the table size of each switch. (b) Normalized link utilization CDF plotted from the TE solution and TE implementation in one of the spine-free fabrics. (c) Normalized link utilization CDF in a traditional Clos fabric of scale similar to the fabric in (b).	46
4.4	(a) Abstract view of a small spine-free network. (b) Cascading precision loss in a two-stage aggregation block. Desired traffic is in dashed blue, actual traffic is in solid red.	47
4.5	(a) 50% spread achieves best link utilization using production traffic trace and routing settings. (b) larger spreads are more robust against traffic spikes.	56
4.6	Median and tail link utilization with different path diversity.	56
4.7	Link utilization distribution in various network fabrics.	57
4.8	Link utilization distribution with different traffic patterns.	58
4.9	Comparison of group reduction speed.	58
4.10	<i>ns-3</i> FCT.	60
4.11	Production FCT vs. link load.	61
4.12	Impact of reduced path diversity.	61
4.13	Impact of Group Sharing.	61
4.14	Impact of Table Carving.	61
4.15	Impact of Group Pruning.	61
4.16	(a, b, c) Median and tail link utilization if running original group reduction algorithm vs. running it with infinite group table size, so that there is only precision loss caused by rounding weights. (d, e, f) Per-link utilization delta over the TE solution.	63
5.1	An 3D torus cluster (Google TPU v4) built from $4 \times 4 \times 4$ reconfigurable cubes.	70
5.2	Static contention of placed jobs. Baseline is where jobs have no background contention at all. Space-filling curve (SFC) is the default placement policy being used in the SLURM and ALPS schedulers.	73
5.3	Demonstration of 1D, 2D and 3D job folding in a 3D torus. Each $4 \times 4 \times 4$ cube is reconfigurable, with wrap-around links (not displayed) to itself on the X and Z axes, and to the other cube on the Y axis.	74
5.4	Joint distribution of job size (XPU's per job) and duration across five public ML cluster traces, drawn as per-trace log-log 2D heatmaps. Color encodes probability mass. In every trace the mass concentrates in the lower-left corner; most jobs are small and short.	82
5.5	Timeseries of cluster load for several public ML cluster traces. Offered load differs widely across clusters, from heavily loaded (Philly, Helios) to lightly loaded (MLaaS, FGD).	83
5.6	Average JCT versus load sweep of different placement policies. Subplot (a) uses FIFO as the admission policy, (b) uses Backfill, (c) uses SWF, and (d) uses LJSF.	84
5.7	(a) time-averaged cluster occupancy and (b) effective utilization versus offered load, for the different placement policies under FIFO admission.	87

5.8	Average JCT versus offered load under node failure, for the different placement policies at failure rates of 0%, 0.5%, and 1%, using FIFO admission.	88
5.9	Average JCT versus offered load for FirstFit and RFold at reconfiguration cube sizes of $1 \times 1 \times 1$, $2 \times 2 \times 2$, and $4 \times 4 \times 4$, under FIFO admission.	90
5.10	Shape placeability as a function of the reconfiguration cube size.	91
5.11	Average JCT versus offered load under three workload patterns: a uniform job-size distribution, the default small-heavy Pareto distribution, and a mono-shape workload in which every job requests the same $4 \times 2 \times 1$ shape. All runs use FIFO admission.	92

List of Tables

- 2.1 OCS technology capabilities across implementation families, reproduced from the OCP white paper [120]. 12
- 4.1 Switch hardware profile of an example deployment. 44
- 4.2 Summary of various fabrics used in FabricEval. 57
- 4.3 Group table space usage on sampled S1/S2/S3 switches by different group types. 63
- 5.1 Notation. The lower group are the rank-key components of Eq. 5.1; smaller is always better. 78
- 6.1 The end-to-end latency of a reconfiguration is a sum of four terms spanning hardware and software. Reducing only the OCS device (the first row) leaves the total dominated by the remaining terms. The magnitudes are representative orders of magnitude, not measurements. 105

Table of Glossary

Term	Meaning
ACK	Acknowledgment
AI	Artificial Intelligence
ASIC	Application-Specific Integrated Circuit
BGP	Border Gateway Protocol
CCA	Congestion Control Algorithm
CDF	Cumulative Distribution Function
CMOS	Complementary Metal-Oxide-Semiconductor
CPU	Central Processing Unit
DCN	Data Center Network
DCTCP	Data Center TCP
DMIR	Direct Mixed-Integer Reduction
DP	Data Parallelism
DPDK	Data Plane Development Kit
DTP	Datacenter Time Protocol
ECMP	Equal-Cost Multi-Path
EP	Expert Parallelism
EPS	Electrical Packet Switch
FCT	Flow Completion Time
FFD	First-Fit Decreasing
FIFO	First-In-First-Out
GPU	Graphics Processing Unit
HBM	High-Bandwidth Memory
HOL	Head-of-Line
HPC	High-Performance Computing
IAT	Inter-Arrival Time
ICMP	Internet Control Message Protocol
IGR	Iterative Greedy Reduction
IP	Internet Protocol
JCR	Job Completion Rate
JCT	Job Completion Time
LJSF	Largest Job Size First
LLM	Large Language Model

Term	Meaning
LPM	Longest Prefix Match
MCF	Multi-Commodity Flow
MEMS	Micro-Electro-Mechanical Systems
MIP	Mixed-Integer Programming
ML	Machine Learning
MoE	Mixture of Experts
MPTCP	Multipath TCP
MZI	Mach-Zehnder Interferometer
NIC	Network Interface Card
OCF	Open Compute Project
OCS	Optical Circuit Switch
PCIe	Peripheral Component Interconnect Express
PLB	Protective Load Balancing
PP	Pipeline Parallelism
QoS	Quality of Service
RDCN	Reconfigurable Data Center Network
RDMA	Remote Direct Memory Access
RTT	Round-Trip Time
SACK	Selective Acknowledgment
SDN	Software-Defined Networking
SerDes	Serializer/Deserializer
SFC	Space-Filling Curve
SOA	Semiconductor Optical Amplifier
SWF	Shortest Work First
TCP	Transmission Control Protocol
TDN	Time-Division Network
TDTCP	Time-Division TCP
TE	Traffic Engineering
TM	Traffic Matrix
ToR	Top of Rack
TP	Tensor Parallelism
TPU	Tensor Processing Unit
VLB	Valiant Load Balancing
VOQ	Virtual Output Queue
WCMP	Weighted-Cost Multi-Path
XPU	Accelerator Processing Unit (GPU, TPU, or similar)

Chapter 1

Introduction

Data center networks (DCNs) keep pace with application demands in the simplest way imaginable: when applications need more bandwidth, operators build a bigger network out of faster switches. This approach is now failing. The switch chips these networks are built from have run into physical limits that keep them from speeding up as they once did [178, 184]. Figure 1.1 shows the trend: the bandwidth supply of a switch chip cannot keep up with the bandwidth demand of modern data centers, and the gap between them has been widening over the past two decades. The recent boom in artificial intelligence (AI) demands has only amplified this gap, and as the trajectory shows, the gap will continue to widen in the near future. We refer to this problem as the data center scaling crisis.

This crisis has two roots, one in the hardware and one in the way we assemble it. The first is physical. A packet switch must inspect and forward every packet that crosses it, and the rate at which a single chip can do so is bounded by the transistors etched onto it. As Moore's law slows, the switching capacity of a chip grows far more slowly than the demand placed on it. The second root is architectural. Data centers are built as fat trees, a topology that gives every server the same full bandwidth to every other and so presents the whole network as one big switch to which all machines attach equally. This abstraction is a gift to application developers, who may place work anywhere and assume uniform, ample bandwidth between any pair of machines. But delivering it demands enough switching capacity to connect everything to everything at full rate, and preserving that guarantee as networks grow means piling on ever more switches whose cost and power climb faster than any single chip's capacity improves. Operators have kept pace with demand only by adding twenty to sixty percent more switches every year, and that pace cannot hold. Adding ever more switches not only drives up the cost of building the network but also raises its aggregate power consumption. The very abstraction that made data centers easy to program is the one that no longer scales.

The pressure is sharpest wherever AI is trained and served. A modern AI model is far too large for any one processor, so training and serving it means spreading the work across thousands of specialized chips that must constantly exchange data as they compute. The speed of the whole endeavor then depends less on the chips than on the network that binds them: when the network stalls, thousands of processors worth tens of millions of dollars sit idle waiting. The network, in other words, has become part of the computer, and the demand it must satisfy is climbing faster than ever [95, 198].

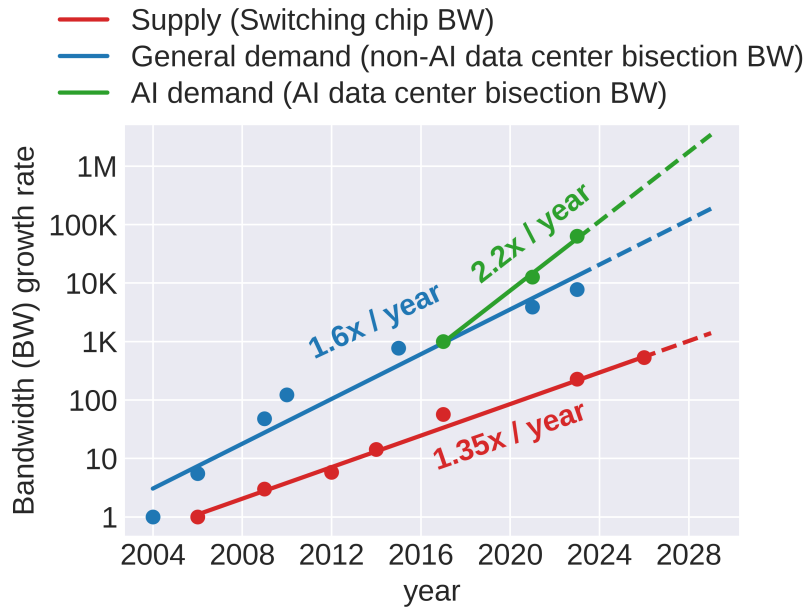


Figure 1.1: The bandwidth gap. Switch-chip supply grows more slowly than either general-purpose or AI-driven bisection-bandwidth demand, so the gap between what a network must deliver and what faster switches can provide widens year over year. (Data is compiled from Broadcom, NVIDIA, Google, and SemiAnalysis reports.)

If simply building bigger can no longer be sustained, the question becomes how to make more efficient use of the bandwidth we can afford, without giving up application performance. The answer begins from an observation the fat tree ignores: servers do not all need to talk to all other servers at once. Real workloads communicate in patterns that are sparse and localized, a handful of servers exchanging data heavily while most pairs exchange almost nothing, so much of the bandwidth a fat tree provisions for sits idle. A network that could take that idle bandwidth away from where it is not needed and place it where it is could deliver the same or better performance with far less hardware. Doing so, though, requires something a fat tree cannot do: a network that rewires itself to follow the traffic.

We therefore turn our attention to a different kind of network architecture, one built around a fundamentally different kind of switch. To see what sets it apart, consider how data moves through a data center today. When a server sends a packet, a transceiver converts the bits into light signals that travel over an optical fiber. At the far end the light arrives at an electrical packet switch (EPS), which converts it back into electrical form, reads the packet's destination address, and forwards it on toward that destination. An optical circuit switch (OCS), by contrast, does none of this: it simply steers the beam of light from one input port to another, without ever converting the signal or reading the packet it carries. Doing almost nothing turns out to be a considerable advantage: such a switch is far cheaper and draws far less power than its electrical counterpart, with roughly half the per-port cost and less than one percent of the per-port energy. Because OCS only redirects light, it can be re-aimed while the network runs, changing which machines are directly wired together from one moment to the next. A network assembled from these switches can therefore

rewire itself on demand. Rather than paying, up front and forever, for the ability to connect any machine to any other at full speed, it puts bandwidth only where traffic actually flows and moves it as the traffic moves. We call such a fabric a reconfigurable data center network (RDCN).

Adopting such a network is a deeper change than swapping one kind of switch for another. The software stack that runs a data center, the programs that decide how fast to send data, which routes each flow should follow, and which machines a job should run on, rests on two assumptions a fixed network has always satisfied: that the bandwidth between any two machines is the same, and that the set of connections holds still for as long as a program runs. An RDCN honors neither, and deliberately so: its efficiency comes precisely from being uneven, lavishing bandwidth on the connections that need it and withholding it from those that do not, and from changing those connections as demand shifts. Running the existing stack unchanged on such a fabric would suffer degraded performance or incorrect behavior, because neither assumption holds any longer.

These broken assumptions are only half of the difficulty. Reconfigurability comes not from an idealized switch but from the OCSes available today, which are limited in three concrete ways. First, an OCS offers only point-to-point connectivity: each circuit joins one endpoint to exactly one other, so one-to-many communication must be serialized over time. Second, it reconfigures slowly, on the order of seconds to minutes, and carries no traffic while it does, so the network cannot adapt to shifting demand on the fly. Third, a single OCS has a limited port count and cannot, by itself, connect all the endpoints of a large data center. One cannot, then, simply drop these switches into today’s data center and expect it to work; but neither must one wait for better hardware. This thesis works within these limits across three RDCN designs of increasing specialization. The first is a general-purpose RDCN that forgoes reconfigurability altogether, running the OCS network as a time-division multiplexed fabric that cycles through a fixed schedule to provide full-mesh connectivity over time. The second is a spine-free RDCN designed for general web traffic, in which an optical layer replaces the spine and reconfigures to follow the measured demand between aggregation blocks. The third specializes entirely, a dedicated RDCN for ML training workloads, whose communication pattern is known in advance and whose topology is shaped to match it. Each design exposes performance challenges in a different part of the network protocol stack, and we redesign the affected components for the RDCN to achieve better end-to-end performance.

Thesis statement. RDCNs built on OCSes can address the scaling crisis by adapting to application needs and providing better end-to-end performance. However, to unlock their full potential, fundamental redesigns are required across core network components, such as transport, traffic engineering, and job scheduling.

1.1 Summary of contributions

The thesis makes three contributions, one for each layer of the software stack that an RDCN disrupts, each redesigned to run above the same slow reconfigurable, limited optical hardware. Together they form a single progression of increasing awareness. Each contribution extracts more value from the same OCS by knowing more about which servers actually need to talk: a transport

that is oblivious to real-time demand and merely rides a fixed schedule, a traffic engineering (TE) system that allocates bandwidth in response to measured demand, and a placement system that knows the workload’s exact communication pattern in advance. This progression runs alongside a shift from the general-purpose toward the specialized, from a fabric that serves any traffic to one built for a single, well-understood workload, the AI supercomputer. The thesis takes up the works in that order, transport, then TE, then job placement, addressing at each step the layer of the stack that comes under the most pressure.

Transport for a schedule-driven optical network (Chapter 3). This layer is the most general and the least aware. A separate optical network sits beside the traditional packet-switched one and accelerates large transfers whenever a circuit happens to be available. Because a single OCS cannot wire every endpoint at once, it cycles rapidly through a fixed schedule, in the manner of a rotor-based network, offering each pair of endpoints a brief high-bandwidth window in turn rather than adapting to real-time traffic. Classical congestion control, assuming one stable path, collapses under this rotation, probing cautiously for capacity and then throttling at every abrupt change as though the network had failed. Chapter 3 presents a transport design that is aware of the rotation schedule, keeping what it learns about each configuration and reusing it to exploit the fleeting high-bandwidth windows rather than starting over each cycle. This transport, Time-Division TCP (TDTCP) [37], makes the following contributions.

- A measurement study on an emulated RDCN that pinpoints why existing transports fall short: single-path variants react too slowly to the rotation, while MPTCP stalls on flow control. The study motivates modeling each configuration separately and reusing the periodic schedule to converge quickly.
- A transport design that keeps a separate congestion-control state per configuration but draws all of them from one sequence-number space, so loss recovery and bookkeeping stay simple across reconfigurations and the receiver needs almost no changes.
- A full implementation in the Linux 5.8 kernel that scales to 100 Gbps and tracks microsecond-scale reconfigurations, driven by top-of-rack notifications that tell each host the moment its path changes.
- An evaluation showing that TDTCP delivers 24% higher throughput than single-path CUBIC and DCTCP and 41% higher than MPTCP in a representative setting, matching a switch-assisted baseline (reTCP) while keeping switch buffers nearly empty and requiring no active buffer management.

The takeaway is twofold: endpoints should coordinate with the RDCN control plane, for instance through proactive in-network signaling; and because existing TCP pays little attention to regular, drastic latency changes, an RDCN transport must be built to handle them.

TE for a spine-free fabric (Chapter 4). One step toward the specialized, this layer is demand-aware. In a production spine-free design, an OCS layer replaces the spine and connects the data center’s aggregation blocks directly. Because demand between blocks is non-uniform, bandwidth is allocated dynamically between them rather than provisioned as a uniform static spine. Classical load balancing assumes a uniform topology and cannot cope with this shifting, skewed allocation. Chapter 4 develops a TE system that is allocation-aware and efficient, computing and installing routes precisely over the uneven, moving topology and within the unforgiving limits of what the

switches can be told to do. This system [38], since deployed in Google’s production network, makes the following contributions.

- The first quantification of traffic-engineering precision loss across a range of DCN topologies, scales, and traffic patterns, tracing it to five root causes: network scale, skewed weights, hardware heterogeneity, path diversity, and the multiplicative cascade of errors through multi-stage fabrics.
- Two group-reduction algorithms, one favoring precision and one favoring speed, that round and merge routing weights to fit the switch’s group table while minimizing divergence from the intended traffic split, guided by three hardware-aware heuristics: traffic-aware table carving, path pruning, and group deduplication.
- FabricEval, an open-source evaluation framework that mirrors Google’s production DCN and TE system, used to run controlled experiments that would be unsafe on live traffic.
- An evaluation showing that the design holds precision loss below 7% where the state of the art loses up to 67%, runs 10× faster, and avoids the traffic loss that group-table exhaustion would otherwise cause.

The takeaway is twofold: network hardware handles heterogeneity inefficiently, and the heterogeneity an RDCN exposes must therefore be managed explicitly in software.

Machine learning (ML) job placement in a reconfigurable torus (Chapter 5). At the specialized extreme, this layer is workload-aware. An OCS layer assembles ML clusters into large tori, and each training job carries a three-dimensional shape, set by its data-, tensor-, and pipeline-parallelism degrees, that must be embedded in the torus. The communication that shape induces is near-neighbor: during collective communication a GPU exchanges data only with its immediate neighbors along each parallelism dimension, so a job runs efficiently only when its shape maps onto a contiguous, well-aligned region of the torus. Enforcing that shape leaves the network fragmented and nodes idle, while placing jobs with no regard to shape invites contention as they compete for shared links. Chapter 5 presents a placement system that uses reconfigurability to escape this trade-off, co-adapting the job and the topology to cut fragmentation and contention at the same time. This placement system, RFold [39], makes the following contributions.

- Measurements showing that on a torus, unlike a Clos, placement dominates performance: scattering a job’s accelerators forces its traffic through intermediate nodes, and contention on the shared links inflates communication time sharply, with one competing job that doubled its load lengthening another’s communication by 95%.
- Folding, which treats a job’s communication pattern as a graph and searches its isomorphic shape variants for a contiguous embedding that minimizes fragmentation without hurting performance.
- Reconfiguration, which, when no contiguous placement exists, uses the OCS layer to knit scattered free accelerators into a single allocation, wiring the job’s communication ring across cubes as optical circuits.
- A topology-aware, job-level simulator and an extended Microsoft Philly trace, used to show that RFold delivers the lowest average job completion time at every load in the sweep and sustains the highest load before saturating, with its knee near 58% offered load under

FIFO admission against 54% for the best shape-unaware policy and 44% for FirstFit, while plateauing near 57% effective cluster utilization.

The takeaway is that when the communication pattern is known in advance, co-adapting the job and the topology escapes the fragmentation-versus-contention trade-off that forces every fixed-topology resource allocator to sacrifice one for the other.

1.2 Thesis organization

The rest of the thesis proceeds as follows.

Chapter 2 lays the groundwork the later chapters build on. It first develops the scaling crisis in detail, tracing the two roots sketched above, the physical limits of a single switch chip and the architectural cost of the fat tree, to explain why building bigger can no longer keep pace with demand. It then explains what an OCS is and how it differs from an EPS, both in what it physically does to a signal and in the cost, power, and reconfigurability that follow from doing less. From there it introduces the RDCN and lays out the spectrum of forms it can take, from a demand-oblivious fabric that rides a fixed schedule to one that reconfigures in direct response to measured traffic. Finally, it details the three concrete limitations of today’s optical switches, point-to-point connectivity, slow reconfiguration, and bounded port count, that constrain every design in the thesis and that the later chapters must work within rather than around.

Chapters 3, 4, and 5 then present the three contributions in order of increasing workload awareness. Each is self-contained, with its own background, design, implementation, and evaluation, and each can be read on its own.

Chapter 3 takes up transport, the most general and least demand-aware layer. It shows why classical congestion control collapses when a rotor-like optical network cycles through a fixed schedule, and presents a transport design that is aware of the rotation, retaining per-configuration state and reusing it to exploit each fleeting high-bandwidth window instead of starting over every cycle.

Chapter 4 takes up TE for a spine-free fabric, one step toward the specialized. Here an optical layer replaces the spine and connects aggregation blocks directly, allocating bandwidth between them dynamically in response to non-uniform demand. The chapter develops a TE system that computes and installs routes precisely over this uneven, moving topology, within the unforgiving limits of what the switches can actually be told to do.

Chapter 5 takes up job placement in a reconfigurable ML torus, the specialized extreme. Each training job carries a three-dimensional parallelism shape that must be embedded in the torus, and the chapter presents a placement system that co-adapts the job and the topology, using reconfigurability to cut fragmentation and contention at the same time rather than trading one for the other.

Chapter 6 concludes by drawing the three contributions together and then looking ahead to a coming generation of optical switches with faster reconfiguration and higher port counts, and to the use cases their added capability would unlock, arguing that the redesigns in this thesis are steps along a longer trajectory rather than one-off fixes.

Chapter 2

Background: The Case for Optical Circuit Switching

DCNs are entering a period of rapid architectural change. For two decades, operators met growing bandwidth demand by scaling out electrical packet-switched Clos fabrics built from ever-faster merchant-silicon switch application-specific integrated circuits (ASICs) [3, 79, 202]. That formula is now under strain. The two trends the formula depends on have pulled apart. Over the past two decades the bandwidth of a single switch chip has climbed roughly 1000-fold, a compound rate of about $1.3\times$ per year, yet the bisection bandwidth a data center must deliver has grown even faster, at roughly $1.6\times$ per year for conventional workloads and faster yet for AI-oriented clusters [18, 31, 69, 184]. Because demand outpaces the switch silicon that serves it, the gap between supply and demand widens every year, and operators have closed it only by brute force, installing 20–60% more switches annually, a trajectory that cannot be sustained. This demand is driven by diverse applications, from ML to video streaming and other data-intensive services, and it now collides with hard physical limits: the per-chip bandwidth of packet switches is flattening, even as the power and capital cost of scaling the network outward become first-order concerns. Together, these pressures constitute a scaling crisis for data center networking.

OCS [120] has re-emerged as a leading response to this crisis. Once regarded as an academic curiosity, OCS is now deployed at hyperscale: it forms the interconnection layer of Google’s Jupiter fabric [182] and the reconfigurable interconnect of its TPU supercomputers [112]. This chapter provides the background needed for the rest of the thesis. Section 2.1 describes the scaling crisis and the bandwidth wall that motivate a rethinking of the network. Section 2.2 explains what OCS is and weighs its advantages and disadvantages against packet switching. Section 2.3 introduces the RDCN, the architecture that OCS makes possible and the setting for this thesis’s contributions. Section 2.4 then argues why, despite those disadvantages, OCS is a particularly good fit for the modern data center.

This shift is far more consequential than swapping one kind of switch for another. Circuit switching, and the OCS-based DCN it enables, is a fundamentally different technology from the packet-switched fabric our systems were built for, and today’s network software stack was never designed to be compatible with it. In particular, a reconfigurable optical network breaks two assumptions that packet-switched designs have long taken for granted: bandwidth uniformity (the premise that any pair of endpoints can communicate at the same rate) and connectivity stability

(the premise that the set of links stays fixed over a connection’s lifetime). Software written against these assumptions, from congestion control to routing and TE to cluster scheduling, therefore suffers performance degradation when run unmodified on an RDCN. The central argument of this thesis is that reclaiming this lost performance requires redesigning the stack layer by layer to embrace reconfigurability rather than resist it.

2.1 The scaling crisis and the bandwidth wall

2.1.1 Demand is outrunning silicon

Two distinct forces drive the demand behind this crisis. The first is the steady growth of conventional cloud and web services (search, storage, video streaming, and the like), whose aggregate traffic has for years strained Clos fabrics and accounts for the bulk of the $1.6\times$ annual demand growth noted above. The second, more recent force is AI/ML, which does not simply add more of the same traffic but changes its character, making the strain acute and, for the first time, exposing the network itself as part of the computer. Because the AI case is the most vivid, we use it to illustrate the pressures that follow; but the scaling problem, and the reconfigurable-network response this thesis develops, applies to conventional workloads just as well.

The immediate driver of the crisis is the explosive growth of AI/ML workloads. The amount of compute used to train state-of-the-art models has grown by many orders of magnitude over the past decade, doubling far faster than the roughly two-year cadence of Moore’s law [10, 198]. This growth cannot be absorbed by a single chip. Instead, training and, increasingly, inference are distributed across thousands of accelerators (GPUs, TPUs, and other XPU) that must exchange enormous volumes of data during every step of computation. The network binding these accelerators together has become part of the computer itself.

At the same time, the traditional engine of performance growth has stalled. With the end of Dennard scaling and Moore’s law, general-purpose processors no longer deliver the exponential single-thread gains of the past, and the industry has pivoted to domain-specific accelerators to sustain performance [95]. Warehouse-scale machines now derive their capability from parallelism, connecting many specialized chips, rather than from faster individual chips. This shift places the interconnect on the critical path: the performance of a large training job is often limited not by the accelerators but by how quickly they can synchronize.

Distributed ML communication is unusually demanding. Training relies on collective operations such as `AllReduce` that are bulk-synchronous: a step cannot finish until every participant has exchanged its data, so the slowest path dictates the pace of the whole job [45]. These collectives move terabytes across the fabric in bursts and are acutely sensitive to both bandwidth and latency variability. Production accounts from large operators confirm that the network, and RDMA-style collective communication in particular, is a dominant factor in the performance of large training clusters [69, 184].

The structure of this synchronization is what leaves the network so exposed. In each training step, every accelerator computes an update to its slice of the model and must then combine that update with the results of all its peers before the next step can begin. Because the operation is a barrier, its completion time is dictated by the slowest participant and the slowest network path

serving it: the collective proceeds at the pace of its worst-case straggler, so it is tail latency, not average latency, that governs performance. If even one link in the network is congested or briefly disconnected, every accelerator stalls, and hardware worth tens of millions of dollars sits idle waiting for a single laggard. For these workloads the network is thus not a supporting component but a first-class determinant of end-to-end performance, and it is not enough for bandwidth to be merely high; it must also be predictable, because a single slow path taxes the entire job.

2.1.2 The bandwidth wall

Historically, the network kept up with demand because the bandwidth of a single switch ASIC doubled roughly every two years, tracking Moore’s law. That trajectory is now hitting physical limits, what practitioners call the bandwidth wall. A packet switch chip’s total capacity is the product of its port count (radix) and its per-port line rate, but both are constrained by the number of high-speed serializer/deserializer (SerDes) lanes that can be physically escaped from the package. As per-lane signaling rates rise, each additional doubling of SerDes speed becomes harder to achieve and more power-hungry, and the reach of the electrical channel shrinks. The practical consequence is a fundamental tension: at a fixed chip capacity, faster ports mean fewer of them, so raising per-port speed reduces switch radix [178]. Lower radix forces traffic through more switching hops to interconnect the same number of endpoints, adding latency, power, and cost at every tier of a Clos fabric.

A concrete illustration makes this tension vivid. Suppose a switch chip offers a fixed total capacity, say 51.2 Tb/s, representative of a current-generation ASIC. That budget can be divided into 128 ports of 400 Gb/s or 64 ports of 800 Gb/s, but not both: doubling the per-port speed halves the number of endpoints the chip can attach. Since the chip’s total capacity grows only slowly from one generation to the next while endpoint speeds keep climbing, each new switch tends to connect fewer neighbors at once. To wire up the same number of servers, operators must then stack more switches into more tiers, and every additional tier a packet crosses adds another store-and-forward hop, another optical-electrical-optical (O-E-O) conversion, and more power and latency. The bandwidth wall, in other words, does not merely cap a single chip; it pushes cost and complexity outward into the entire fabric.

Copper interconnects face a parallel wall. The distance a passive copper cable can carry a signal falls as the signaling rate increases, so at modern line rates copper is confined to within-rack reach. Extending connectivity beyond a rack requires optics, and in a packet-switched fabric every such link must be terminated by an O-E-O conversion at each switch it traverses. These conversions are a major and growing contributor to network power, cost, and latency [178, 182].

2.1.3 The “one big switch” abstraction and the cost of scaling out

Underlying this scaling strategy is a powerful and deliberate abstraction: the DCN presented to applications as “one big switch.” In this view every server is equally connected to every other, so an application that asks for a handful of servers need not care which servers it receives; from its perspective they are interchangeable. This uniformity is what makes the data center easy to program and able to host arbitrary workloads, and a non-blocking Clos fabric is engineered precisely to sustain it. But the abstraction is expensive to maintain: it is realized by statically

provisioning enough bisection bandwidth for the worst-case, any-to-any traffic pattern, whether or not any real workload exhibits it.

Because a single switch cannot grow arbitrarily, large fabrics are built as multi-tier Clos (fat-tree) topologies that scale out by adding stages of switches [3, 79, 202]. This works, but at a price. Every additional tier multiplies the number of switches, optical transceivers, and cables (and their associated power and capital cost) while adding hops and O-E-O conversions to end-to-end paths. Clos fabrics are also awkward to grow incrementally: because the topology and its bandwidth are provisioned uniformly, expanding capacity or refreshing to a faster generation of switches typically means overhauling the spine layer and re-striping the fabric, a disruptive operation for a live network [182].

Network power is now a first-order design constraint. In a hyperscale fabric the spine layer alone can draw on the order of a megawatt [120], and as line rates climb, the share of total facility power consumed by O-E-O conversion and packet processing grows. With AI infrastructure expanding under tightening energy and sustainability constraints, a network architecture that spends less power to move each bit is increasingly valuable.

The deeper issue is that the “one big switch” abstraction pays for a generality that most workloads do not need. Real applications rarely exhibit worst-case, any-to-any communication; their traffic is instead localized in space, stable in time, or structured by the application itself. A graph database communicates heavily within a partition and only lightly across partitions; web services show pronounced hot spots once traffic is aggregated at the leaf or aggregation-block level; and distributed ML jobs have deterministic near-neighbor communication fixed by their tensor-, pipeline-, and data-parallel structure [112]. If the network instead created bandwidth only where communication actually occurs, it could be at once cheaper and more scalable, escaping the static trade-off in which a network is either general but inefficient (the fat tree) or efficient but specialized and fragile (a topology hard-wired for one pattern). Realizing this requires a network that can change its own topology at runtime. That is the promise of an RDCN, and OCS is the hardware that makes it practical. These pressures (demand outstripping silicon, the bandwidth wall, and the rising cost of maintaining a uniform “one big switch”) set the stage for reconsidering how DCNs are built.

2.2 Optical circuit switching

2.2.1 What optical circuit switching is

An OCS establishes a direct, transparent light path between two endpoints and holds that connection open for the duration of a circuit. Unlike a packet switch, which receives a packet, buffers it, inspects its headers, and forwards it toward its destination, an OCS never interprets the traffic flowing through it. It behaves like a dynamically reconfigurable patch panel: light enters one port and is steered (by a mechanical, electro-optical, or free-space mechanism) straight out of another, entirely in the optical domain [120].

An everyday analogy helps separate the two styles of switching. A packet switch works like the postal system: each packet is an addressed envelope that is read, sorted, and forwarded on its own at every hop, so one wire can interleave mail bound for many different destinations. An

OCS works more like an old manual telephone exchange that physically patches one caller’s line straight through to another: while the connection is up it is a private, dedicated wire that faithfully carries whatever the two ends put on it, but it reaches exactly one place at a time, and patching a new connection takes a moment. Packet switching buys fine-grained flexibility at the cost of inspecting and queueing every packet; circuit switching buys raw speed and efficiency at the cost of that flexibility. The rest of this section makes the consequences of this bargain precise.

Three properties follow directly from this transparency. First, OCS performs no O-E-O conversion; because the signal is never converted to electrical form, there is no packet processing, no buffering, and correspondingly little power drawn to move data through the switch. Second, OCS is bufferless and therefore free of the queueing that causes congestion and jitter in packet networks: once a circuit is established, data traverses it with deterministic, near-zero added latency. Third, OCS is protocol- and data-rate-agnostic: since the switch merely guides light, the same device can carry any encoding at any line rate, and can be upgraded to faster endpoints without being replaced [120, 182]. This is why OCS is often described as “future-proof”: the optical layer does not need to be re-provisioned when the endpoints it connects get faster.

The trade-off inherent to circuit switching is that connectivity is established in advance and changed only by reconfiguring the switch. Setting up or tearing down a circuit takes a finite reconfiguration time, on the order of milliseconds for the Micro-Electro-Mechanical Systems (MEMS) switches used in production, and during a reconfiguration the affected ports carry no traffic. OCS is thus best suited to traffic that is either long-lived or predictable enough to be scheduled, and it is typically paired with a control plane that decides when and how to rewire the fabric.

The economic contrast with packet switching is stark, and it is what makes OCS attractive despite this rigidity. Because an OCS merely steers light, it needs none of the per-port electrical transceivers, packet-processing pipelines, or buffers that dominate a packet switch’s cost and power. Representative figures put an OCS port at roughly \$800 and 0.002 pJ/bit, against roughly \$1,800 and 15 pJ/bit for an EPS port, nearly four orders of magnitude less energy per bit [18, 120]. A network organized around OCS in this way is an RDCN. Its catch is not only reconfiguration latency but that an OCS violates assumptions baked deep into today’s transport, routing, and resource-management stacks, all of which presume packet-granular, any-to-any forwarding. Extracting the hardware’s benefit therefore demands software–hardware co-design across the stack, the recurring theme of the chapters that follow.

2.2.2 OCS technologies

Many physical mechanisms can implement optical switching, and they differ substantially in the metrics that matter for data centers: port count (radix), insertion (signal) loss, and reconfiguration time. Table 2.1 summarizes the main families surveyed by the Open Compute Project (OCP) community [120].

The technologies span a wide design space. Robotic switches act as automated fiber patch panels: motorized actuators physically re-plug fibers, giving very large radix, low loss, and latching connections that survive power loss, but reconfiguration takes seconds to minutes. 3D-MEMS switches steer collimated beams in free space using arrays of tiltable micromirrors; they reach large radix with moderate loss and millisecond switching, and are the technology behind

Table 2.1: OCS technology capabilities across implementation families, reproduced from the OCP white paper [120].

Feature	Robotic	Beam steering (MEMS / LC / piezoelectric)	Silicon photonics (MZI / MEMS)	Metasurface
Radix	Large	Med	Low–Med	Large
Insertion loss (dB)*	Low	Med	High	Med
Dark fiber switch**	Yes	Some	Yes	Yes
Reconfiguration time	Slow (sec–min)	Med (msec)	Fast (nsec– μ sec)	Fast (μ sec)
Simultaneous connections	No	Yes	Yes	Yes

*Insertion loss typically increases as radix increases, and can be reduced with an integrated semiconductor optical amplifier (SOA).

**Dark-fiber switching: no light need be present at the input port for the switch to establish a connection. This lets an operator provision, test, and reconfigure optical paths ahead of time, before the attached endpoints are powered on or transmitting, rather than only while live traffic is flowing.

Google’s Jupiter and TPUv4 deployments [112, 182]. Liquid-crystal (LC) and piezoelectric approaches trade moving mirrors for polarization control or precise actuation. Silicon-photonic switches route light through on-chip waveguides using Mach-Zehnder interferometers (MZIs) or ring resonators, achieving nanosecond-to-microsecond reconfiguration and CMOS-compatible mass production at the cost of higher insertion loss and crosstalk as radix grows. Metasurface and hybrid designs aim to combine fast switching with high radix and low loss. The right choice depends on the use case: large-radix, slow-switching devices suit static rewiring, while fast, lower-radix devices target dynamic, per-job or sub-microsecond reconfiguration [18, 178]. A long line of research systems explored these points in the design space, progressively driving reconfiguration times down from milliseconds toward nanoseconds: Helios [54], c-Through [223], REACToR [141], ProjecToR [73], RotorNet [152], Opera [155], and Sirius [18]. For this thesis the practical takeaway from the survey is narrower: today’s production RDCNs are built almost exclusively on large-radix, millisecond-scale MEMS switches, so that is the regime we assume throughout. The faster silicon-photonic and metasurface regimes matter chiefly for the future directions discussed in Chapter 6.

2.2.3 Advantages over packet switching

The benefits of OCS follow from its transparency and are consistent across deployments [120, 182]:

- **Lower power and cost.** Eliminating O-E-O conversion and packet processing removes the most power-hungry elements of the network. At-scale deployments report substantial savings: Google’s Jupiter Evolving reports a 41% reduction in network power and a 30% reduction in capex from introducing an OCS interconnection layer, together with a $5\times$ increase in capacity and faster deployment [182].
- **Ultra-low, jitter-free latency.** With no buffers and no store-and-forward processing, an established circuit adds almost no latency and no queueing variability. This determinism is

especially valuable for the bulk-synchronous collectives of ML training, which are stalled by even a single delayed packet [45].

- **Rate- and protocol-agnostic, future-proof.** Because the optical layer is oblivious to encoding and line rate, endpoints can be upgraded to faster speeds without replacing the switching layer, decoupling the fabric’s lifetime from the endpoints’ generation [182].
- **Flexibility and resiliency.** A reconfigurable optical fabric can change its physical topology to match workload demand or to route around failures, turning the network from a static transport into an active, programmable resource [112, 255].

2.2.4 Disadvantages relative to packet switching

These benefits come with real limitations, which explain why OCS complements rather than wholly replaces packet switching [120]:

- **Circuits, not packets.** An OCS provides one-to-one connectivity for the duration of a circuit. It cannot fan traffic out to many destinations or make the per-packet forwarding and load-balancing decisions that a packet switch makes naturally. Endpoints (or an accompanying packet layer) must supply that functionality.
- **No packet-level visibility or control.** Operating purely in the optical domain, an OCS cannot inspect, filter, buffer, or apply QoS to individual packets, nor can it provide per-packet telemetry. Functions that assume packet-level processing must live elsewhere.
- **Insertion-loss limits hops.** Each optical hop attenuates the signal, and without amplification, practical paths are limited to roughly one or two OCS hops. This ceiling is an optical power-budget limit rather than an architectural rule: every pass through a switch adds insertion loss, and once the accumulated loss exhausts the link’s power margin the signal must be optically amplified or regenerated to travel farther. Packet fabrics, by contrast, regenerate the signal electrically at every stage and can therefore scale across many hops [120].
- **Reconfiguration cost and control complexity.** Changing connectivity takes a finite reconfiguration time during which affected circuits carry no data, so OCS suits traffic that is predictable or schedulable rather than arbitrarily bursty. Realizing its benefits requires a capable control plane to decide the topology, and much of the engineering effort in production OCS systems lives in that control plane [182, 255].

2.3 Reconfigurable data center networks

The previous section described an OCS as a device. Assembling such devices into a working network yields an RDCN: a fabric whose physical topology is not fixed at deployment but can be rewired on demand to match the traffic it currently carries. In a conventional network the wiring is frozen the day the data center is built, and the control software’s only freedom is to choose paths over that fixed graph. An RDCN adds a second, more powerful degree of freedom, the ability to change the graph itself: it places OCSes wherever links would otherwise be hard-wired and lets a control plane decide, from minute to minute or from job to job, which endpoints are directly

connected. The network’s topology becomes a runtime variable rather than a build-time constant.

This capability is precisely what lets an RDCN escape the generality-versus-efficiency trade-off of Section 2.1. A static fat tree must provision for the worst case because it cannot know in advance which endpoints will need to talk; an RDCN can instead observe (or be told) the current demand and spend its bandwidth only where communication actually happens: many links between the few endpoint pairs that are busy, and none between pairs that are idle. The same physical hardware can therefore serve a graph database in one hour and a language-model training job the next, presenting each workload with a topology shaped to its needs. Reconfigurability converts the network from a fixed substrate that every workload must be squeezed onto into an adaptable resource that molds itself to the workload.

RDCN designs differ chiefly in how quickly and on what basis they reconfigure, and these choices follow directly from the switching technologies of Table 2.1. Two timescales are easy to conflate here. One is the physical reconfiguration time of the switch device: how long the hardware needs to move a mirror or retune a wavelength, the quantity tabulated in Table 2.1. The other is the scheduling granularity of the network built on top: how often the control plane actually chooses to rewire, which can be far coarser than the hardware permits. A fast device does not compel fast rewiring; it only makes it possible. Early hybrid systems such as Helios [54] and c-Through [223] bolted a slow optical path onto a conventional packet network, diverting only the large, long-lived “elephant” flows onto circuits while leaving small, bursty flows on the packet fabric. Later demand-oblivious designs such as RotorNet [152], Opera [155], and Sirius [18] instead cycle through a fixed schedule of topologies so rapidly (microseconds down to nanoseconds) that every endpoint pair is connected in turn, doing away with per-demand scheduling altogether. Production systems occupy the pragmatic middle ground: Google’s Jupiter fabric replaces the spine of its DCN with a slowly reconfigured OCS layer [182], while its TPU supercomputers reconfigure the optical fabric once per job to give each training run a bespoke topology [112]. What unites this diverse family is a single premise: that a network need not be uniform and unchanging in order to be general.

With that taxonomy in place, it is worth pausing on what, concretely, operators use OCS reconfigurability for today. Three recurring use cases illustrate how OCS reconfigurability is put to work, spanning both mature production practice and designs still confined largely to research; each exploits a different facet of the same underlying capability.

The first and most mature use case is spine replacement. Recall that a conventional fabric is a multi-tier Clos in which a top layer of spine switches stitches together the lower aggregation blocks. That spine is costly and power-hungry, and it must be upgraded in lockstep with the blocks beneath it: the moment a block moves to a faster line rate, the spine becomes a bottleneck unless it is upgraded too. An RDCN removes the spine switches outright and connects the aggregation blocks through a large-radix OCS layer instead, handing each pair of blocks a direct optical circuit sized to the traffic they actually exchange. Google’s Jupiter fabric took exactly this path and reported a 41% reduction in network power, a 30% reduction in capital cost, and a $5\times$ increase in delivered capacity. Because the optical layer is oblivious to line rate, it also let operators roll in faster blocks incrementally without ever rebuilding the spine [145, 182]. In this role the OCS behaves like a slowly adjusted patch panel: the aggregated inter-block traffic matrix drifts only over minutes to hours, so its circuits need to be recomputed just occasionally.

The second use case is runtime topology reconfiguration for AI training clusters. Because an

ML job’s communication pattern is fixed and known the moment its parallelization strategy is chosen, an RDCN can assemble a physical topology tailored to that pattern for the job’s lifetime and tear it down when the job finishes. Google’s TPU supercomputers do this by interposing MEMS OCSes between hardwired accelerator “cubes” and reconfiguring them on a per-job basis, giving each training run its own torus of exactly the right dimensions [112]. Matching the topology to the job improved training performance by up to $3.3\times$ and lowered power by 9% relative to a fixed fabric, and the very same optical layer doubles as an isolation and resiliency mechanism, walling jobs off from one another and rewiring around a failed cube without disturbing the healthy ones [112, 255]. Here too reconfiguration is coarse-grained: it happens once per job, and jobs run for hours or days, so millisecond switching is far faster than the workload requires.

The third use case is accelerating bulk data transfer. Large, long-lived transfers such as a database shuffle, a checkpoint write, or the bulk gradient exchange between training steps can monopolize a packet fabric and starve the small, latency-sensitive flows that share it. An RDCN can detect such a transfer and stand up a dedicated optical circuit just for it, shunting the heavy traffic onto a private, high-bandwidth path and leaving the packet fabric clear for everything else. This is the strategy pioneered by the early hybrid systems Helios and c-Through [54, 223], and it reappears in recent AI-fabric designs that dedicate optical bandwidth to the bulkiest communication phases of distributed training [139]. Unlike spine replacement and per-job topology reconfiguration, which are established production practice, dedicated bulk-transfer circuits remain largely the province of research prototypes and recent proposals. Because these transfers persist for many milliseconds or longer, a circuit that takes a few milliseconds to set up still repays its setup cost many times over.

A common thread runs through all three use cases: the traffic they serve is stable over comparatively long timescales, so the network need only reconfigure slowly, at millisecond granularity or coarser. This is exactly the regime that MEMS-based OCSes serve best. As Table 2.1 indicates, MEMS switches pair the large port counts these deployments demand with millisecond switching and field-proven reliability, and they are, accordingly, the technology behind essentially every large-scale production RDCN to date [112, 182]. Slow, MEMS-based reconfiguration is therefore the mainstream of RDCN practice today, and it is the regime this thesis primarily targets.

This need not remain the only regime. A faster OCS, one that reconfigures in microseconds or nanoseconds as the silicon-photonics, metasurface, and demand-oblivious designs of Table 2.1 aspire to, would unlock use cases that millisecond switching cannot reach: steering the individual bursts of a collective, scheduling circuits at the granularity of single flows, or dispensing with the packet fabric entirely [18, 152]. These fast-reconfiguration regimes are still largely experimental, and they place even sharper demands on the software stack; we return to them as an opportunity for future work in Chapter 6.

Reconfigurability, however, is not free at the software layer, and this is where the challenges this thesis addresses begin. Decades of network and systems software were written on two assumptions that a packet-switched fat tree satisfies almost by definition: that the fabric offers uniform bandwidth between any two endpoints, and that it offers stable connectivity that does not change beneath a running application. An RDCN violates both. Its bandwidth is deliberately non-uniform (that non-uniformity is the very source of its efficiency), and the set of live links shifts over time as the fabric rewires itself. Transport protocols that assume a single stable

path and probe cautiously for its capacity, routing and TE systems that compute over a fixed topology, and cluster schedulers that treat the network as an undifferentiated pool of bandwidth all lose performance, or misbehave outright, when the ground shifts under them. Bridging this gap between reconfigurable hardware and reconfiguration-oblivious software is the problem the remaining chapters take up, one layer of the stack at a time.

Which of the two assumptions bites hardest, and thus which layer of the software stack must be redesigned, depends on where a given RDCN sits on a spectrum that runs from the fully general to the narrowly specialized. The designs surveyed above do not form a single point on that spectrum but populate its whole length.

At the general end are fabrics that serve any workload while knowing nothing about it. The demand-oblivious designs introduced above, RotorNet, Opera, and Sirius, are the extreme case: they rotate through a fixed schedule of topologies so quickly that every endpoint pair is connected in turn, never matching the network to the traffic [18, 152, 155]. Such a fabric assumes nothing about the workload, but it offers each connection only a fleeting, ever-changing path; its defining hardship is connectivity that will not hold still.

In the middle lies the general-purpose production data center, exemplified by the spine-replacement fabric of Jupiter described above [182]. It still serves arbitrary workloads, but its bandwidth is now deliberately non-uniform, and traffic must be engineered across a topology that reconfigures within the hard limits of real switch forwarding tables.

At the specialized end is the network built for a single, well-understood workload: the AI training cluster, exemplified by the per-job TPU fabric described above [112]. Because the workload’s communication pattern is known in advance, the network can be co-designed with the job it serves, and the two become a single design problem rather than two separate ones.

This thesis traverses the spectrum from its general end toward its specialized one, taking up at each stop the layer of the stack that the local conditions stress most:

- **Transport (Chapter 3).** Lowest and most general in the stack, transport must keep a connection efficient across reconfiguration whatever fabric lies beneath it. This is where connectivity instability bites hardest, the assumption a demand-oblivious, rapidly rotating fabric breaks most violently.
- **Routing and TE (Chapter 4).** In the general-purpose reconfigurable data center, bandwidth non-uniformity becomes the central problem: traffic must be engineered across a spine-free fabric whose paths are unequal and whose topology reconfigures within the limits of switch forwarding tables.
- **Scheduling (Chapter 5).** In the specialized ML cluster, both assumptions dissolve into a single opportunity: because the workload is known, the cluster scheduler can co-adapt ML job shapes with the reconfigurable topology they run on, shaping each to the other.

Addressing these three problems, and so turning reconfigurability from a property the software must merely tolerate into a lever it can actively exploit, is the work of the chapters that follow.

2.4 Why OCS makes sense in data centers

The limitations above might seem disqualifying, but the data center is a uniquely favorable environment for OCS, for three reasons.

Traffic is stable and predictable where it counts. OCS is a poor fit for arbitrarily bursty, fine-grained traffic, but at the scale where OCS is deployed (between aggregation blocks, racks, or accelerator cubes), traffic is far more stable. Aggregated flows change slowly relative to individual packets, and the communication patterns of the dominant workload, distributed ML, are exceptionally regular: parallelization strategies produce structured, repeating collective-communication patterns that are known in advance from the job’s configuration [112]. This predictability is exactly what circuit switching needs, and it lets a control plane provision circuits ahead of demand rather than reacting packet by packet. Where full predictability is lacking, a hybrid design pairs OCS for the stable bulk of traffic with a packet layer for the rest [54, 223].

Reconfigurability is a feature, not just a workaround. Because ML jobs communicate in known patterns, an OCS fabric can match the physical topology to the workload. We saw this concretely in the TPU use case of Section 2.3: matching the topology to each job’s communication shape improved training performance and power efficiency over a static fabric, and the same optical layer improves job isolation and resiliency [112]. The same reconfigurability underpins failure resiliency: when a component fails, the control plane rewires around it at the physical layer, restoring full bandwidth rather than leaving the network in a degraded state, and doing so from a shared pool of spare paths rather than fully duplicated infrastructure [255]. This ability to reshape the network at runtime is the central theme of this thesis.

The economics are compelling at scale. The scaling crisis of Section 2.1 bites hardest exactly where OCS pays off most. As the spine-replacement use case showed, swapping the power-hungry electrical spine for a rate-agnostic optical layer removes a megawatt-scale power draw and lets operators upgrade endpoints without the disruptive, fabric-wide refreshes a packet Clos requires [145, 182]. The same rate-agnosticism and low latency make OCS attractive for emerging use cases such as resource disaggregation, where high-speed copper’s shrinking reach otherwise forces costly O-E-O conversions [120]. In short, OCS trades away packet-level flexibility (which the data center can afford to relocate to its endpoints) in exchange for dramatic gains in power, cost, latency determinism, and topological flexibility, precisely the axes on which packet fabrics are hitting a wall.

These properties have moved OCS from research prototype to production cornerstone in barely a decade [112, 145, 182], and recent systems continue to push it toward finer-grained, per-job, and even in-workload reconfiguration for AI clusters [139, 234]. The remainder of this thesis builds on this foundation, treating reconfigurability as a first-class design lever across the network stack.

Chapter 3

Time-Division TCP

DCNs connect a large number of end hosts over a high-bandwidth, low-latency fabric, traditionally using a Clos topology that offers the “big-switch” abstraction of a single full-bisection network. As Chapter 2 described, that approach is reaching its limit, and OCS-based RDCNs have emerged to bridge the widening gap between bandwidth supply and demand, dynamically allocating network resources over time at lower cost and energy than a statically wired fabric [81, 124, 203, 205]. Reconfigurability, however, introduces a challenge that Chapter 2 named but did not solve: it breaks the connectivity stability that transport protocols rely on. This chapter takes up that problem at the transport layer. This is the first and most general of the three redesigns the thesis statement calls for. Of the software layers an RDCN disrupts, transport sits farthest from the workload, knowing nothing about which servers actually need to talk, and the fabric it runs on is correspondingly the least aware, cycling through a fixed schedule that endpoints must ride rather than shape. Even at this remove, honoring the broken assumption requires rebuilding congestion control from the inside rather than tuning it from the outside, as the rest of the chapter shows.

While TCP congestion control is designed to optimize data transfers between end points with competing traffic, it assumes that the channel between them is relatively stable in terms of bandwidth and latency. In the context of RDCNs, however, this assumption does not hold. In hybrid electrical/optical networks, the path between a pair of hosts may alternate rapidly between a fast optical network and a slower electrical network, with orders-of-magnitude differences in bandwidth and/or latency. Even in proposals that are purely optical [156], indirect forwarding, where packets hop across several hosts when a direct link is unavailable, creates drastic latency changes. Our experiments show that existing TCP variants are unable to take full advantage of the additional capacity provided by these RDCN technologies. Because TCP infers network state and updates its congestion model iteratively on the timescale of multiple RTTs, senders cannot converge to the current path conditions before the network reconfigures again in highly dynamic RDCNs.

To exploit the full potential of RDCNs, we present Time-Division TCP (TDTCP), a new variant of TCP designed specifically for reconfigurable networks. We build on the insight that in RDCNs, as long as the network moves crisply between distinct configurations, it suffices to operate efficiently in each configuration separately; the full algorithm is then a piecewise function of individual models. In other words, in TDTCP, we multiplex independent network states over time, much like how MPTCP [231] multiplexes subflows over space. Unlike MPTCP, however, in

TDTCP only one subflow is active at a given time. As the network moves from one configuration to the next, senders switch between network models.

While this overall approach may seem straightforward, we must solve several challenges before TDTCP is a complete protocol. First, in today’s RDCNs, the duration in which a single optical configuration is in place is on the order of a few RTTs, leaving little time for the host to react to the new network environment. We address this issue by leveraging the fact that the top-of-rack (ToR) switches to which end hosts connect are directly involved in reconfiguring the network fabric. Therefore, we enlist ToR switches to directly notify their attached hosts when paths change. This signaling enables end hosts to view the reconfigurable network not as a single fabric, but rather as a series of time-division networks (TDNs) that transition periodically.

Second, when the network reconfigures, in-flight packets may not belong entirely to one TDN or the other. For example, data packets may traverse a different TDN than their ACKs, or a decrease in latency may reorder packets. Careful accounting is required to disambiguate transient reordering from true loss and credit the appropriate TDN, preventing unnecessary retransmissions. TDTCP tracks the sequence space associated with each TDN and uses heuristics and selective acknowledgments (SACK) to avoid severe performance penalties during TDN transitions.

Finally, implementing TDTCP in a modern operating system kernel is challenging because the congestion control state is tightly integrated into the networking stack. We identify the subset of CUBIC congestion control state that must be duplicated across TDNs and optimize the TDN switching process to ensure end hosts are promptly notified of network transitions. We show that our open-source implementation in Linux kernel 5.8 scales to 100 Gbps and supports reconfigurations on microsecond timescales.

Using our prototype, we evaluate the performance of TDTCP against existing TCP variants in the kernel as well as MPTCP and reTCP [169], an RDCN-specific approach that requires extensive switch support. Results show that long-lived flows achieve better throughput under TDTCP: 24% higher than single-path CUBIC and DCTCP in one representative RDCN setting, and 41% higher than MPTCP. Additionally, TDTCP matches the throughput performance of reTCP while exhibiting lower switch buffer occupancy and does not rely on active switch buffer management.

This chapter makes three contributions:

- Our measurement study on an emulated RDCN identifies TCP’s shortcomings: single-path variants react too slowly to changing conditions, but MPTCP is crippled by flow control stalls. We learn that a new TCP should leverage predictable reconfigurations to model TDNs separately and use a subflow scheme that bridges the gap between single and MPTCP.
- We present the design of TDTCP, which maintains a separate congestion control state corresponding to each TDN but uses a unified sequence number space to (1) simplify loss recovery and bookkeeping across reconfigurations and (2) minimize the need for receiver-side changes.
- We present a full-featured implementation of TDTCP in Linux kernel 5.8, including performance optimizations that support aggressive reconfiguration frequencies and modern (e.g., 100-Gbps) link rates.

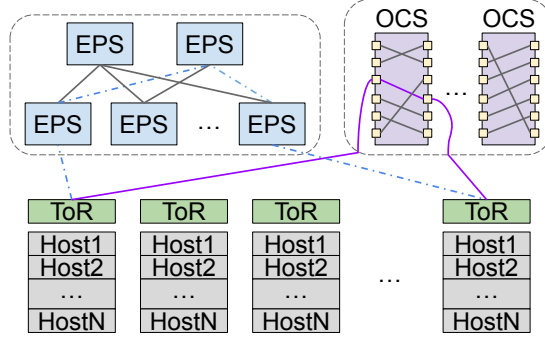


Figure 3.1: A hybrid, demand-oblivious RDCN topology. The OCS network provides temporary high-bandwidth links that supplement the EPS network.

3.1 Motivation

Chapter 2 introduced RDCNs in general and the spectrum of forms they take. This chapter targets one specific point in that spectrum, the hybrid, demand-oblivious RDCN, and asks why it degrades TCP performance. This section describes that variant and then explores why its characteristics yield poor TCP throughput. We leave a full description of other RDCN variants to §3.5.

3.1.1 Hybrid demand-oblivious RDCNs

In this chapter, we consider hybrid, demand-oblivious RDCNs. As illustrated in Figure 3.1, such networks are hybrids because they consist of both EPSes and OCSes. Each ToR is connected to several upstream EPSes as well as one or more OCSes. The EPS and OCS networks are isolated from each other.

Time-division multiplexing. When forwarding a packet, a ToR chooses either the EPS network or the OCS network. We assume that for a given destination, only one network is in use at a time: if a circuit exists, then the optical network is chosen, otherwise the ToR falls back to the packet network. During its lifetime, a flow may traverse either the packet network, the optical network, or both, depending on timing and flow duration. From a sender’s perspective, this is a time-division multiplexed path, which is conceptually in-between a single path and true multipath routing. We refer to each discrete network state as a time-division network.

Different network characteristics. The packet and optical networks have different bandwidth and end-to-end latency due to their physical characteristics and network architectures. EPS networks typically use 10–400 Gbps links, while OCS links have the potential to be an order of magnitude greater than that [142, 154]. Traversing an EPS topology may take 100 μ s with queueing, whereas the end-to-end OCS latency can be shorter because it includes fewer intermediate switches, does not queue inside the network, and only the destination switch decodes packets.

Stable characteristics within a TDN. Compared to the orders-of-magnitude differences across TDNs, within a TDN conditions are relatively stable: subject to background traffic, the bandwidth, latency, and loss rate on a path oscillate within a comparatively small range.

Predictable TDN pattern. OCSes follow a schedule that consists of several independent

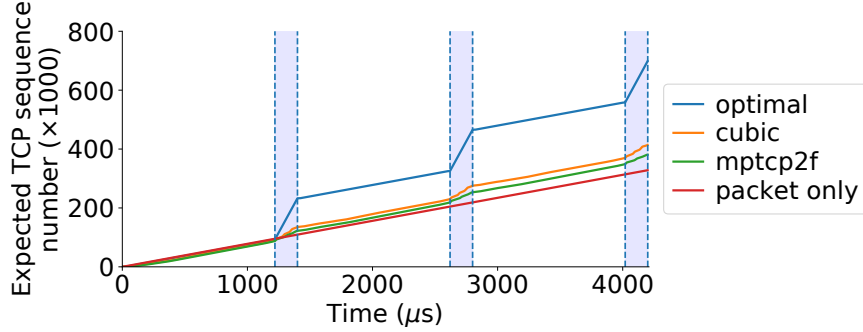


Figure 3.2: Sequence graph of TCP variants running in hybrid RDCN for 3 optical weeks.

configurations, which we refer to as days, each of which individually provides connectivity to a disjoint subset of ToR pairs. The full schedule is known as a week. Between two days, there is a blackout period we call a night during which no packets can be sent while the optical network reconfigures. The term demand-oblivious refers to how the configurations are generated [144]: the complete schedule provides eventual full-mesh reachability over a week, independent of the traffic demand.

Frequent path switches. To keep overall utilization high, the optical network must maintain a high duty cycle, defined as the ratio of uptime to downtime. Related work [169] assumes a 9:1 duty cycle, so a state-of-the-art reconfiguration time of $20 \mu\text{s}$ [142] yields a circuit day of $180 \mu\text{s}$, striking a balance between frequent reconfiguration (to serve all rack pairs often) and stability (to give flows time to converge). Assuming an RTT of $\approx 40 \mu\text{s}$, a circuit day is on the order of five RTTs. This is a concern since TCP receives congestion feedback and adjusts its rate on the timescale of RTTs, so a short circuit day means that common TCP variants like CUBIC and DCTCP may not have enough time to adapt to path characteristics before switching paths again.

3.1.2 Measuring TCP performance in RDCNs

TCP is the dominant transport protocol in data centers, so it is important to understand TCP’s performance on hybrid demand-oblivious RDCNs. In this section, we investigate the performance of two representative, widely-used TCP variants, single-path TCP CUBIC [87] and MPTCP [186, 231]. Our measurement and analysis addresses the following questions: *Can TCP take full advantage of RDCN capacity? Why or why not?*

We conduct all experiments using the Etalon RDCN emulator [169]. Specific testbed parameters are described in §3.4.1. We run CUBIC and MPTCP flows in the RDCN topology depicted in Figure 3.1. The open-source MPTCP implementation does not support our intended use: pinning one subflow (index 0) to the packet network and one subflow (index 1) to the optical network. Hence, we extended MPTCP with a prototype `tdm_schd` scheduler that steers packets between two subflows according to the RDCN schedule. For example, when the packet network is active, `tdm_schd` sends all packets to subflow 0, which is pinned to the packet network, and vice versa.

Figure 3.2 shows the throughput of single-path TCP CUBIC and MPTCP in the form of a sequence-number graph. To remove startup and transient effects, results across thousands of optical weeks are averaged to create this visualization. The shaded regions show when the

optical path is active while the unshaded regions correspond to the packet network. The slope of a sequence number curve gives the throughput. For example, for the optimal curve, which is computed analytically and assumes that a flow uses all available bandwidth, the slope is steeper in shaded regions than in unshaded ones, as one would expect due to the higher bandwidth of the optical network. On the other hand, the packet only curve specifies the best-case performance when the end host uses only the packet network. It has a constant slope equal to the rate of the packet network.

Both single-path TCP CUBIC and MPTCP fall far below the optimal line, showing that neither takes full advantage of the RDCN bandwidth. While their sequence number curves in the packet network (unshaded) are parallel to the optimal line, their curves in the optical network (shaded) are shallow and only slightly better than in the packet network. MPTCP’s performance is lower than that of CUBIC, despite the fact that it has been made RDCN-aware, likely because of the overhead of subflow management and the reinjection of lost segments that could not be ACKed before a path change.

Single-path TCP CUBIC has two shortcomings that contribute to its sub-optimal performance. First, it assumes a single path with conditions that change relatively slowly, but this assumption does not hold in RDCNs. After a path change, single-path TCP must spend multiple RTTs probing the path to converge to a new state that is appropriate for the new bandwidth and RTT, which can differ by orders of magnitude. Given the short duration of each optical day (≈ 5 RTTs in this example), there is insufficient time for single-path TCP to ramp up. Second, it is ignorant of the cyclic, deterministic RDCN schedule; instead, it must rediscover each change and adapt using mechanisms designed for traditional networks.

MPTCP’s low throughput is caused by the fact that the connection is frequently stalled due to the strict isolation between the two subflows. Packets sent on one subflow may not be acknowledged for a while if the receiver is blocked from sending on that subflow because the corresponding TDN is not active. This means that the sender stalls until MPTCP’s connection-level reinjection mechanism [188] attempts to retransmit the unacknowledged packets from one subflow on the other subflow, the one that is currently active. This retransmission also introduces additional overhead.

In summary, our investigation reveals three findings: (1) TCP should model the characteristics of each TDN separately, (2) TCP should leverage the repeated, deterministic RDCN schedule to speed up congestion control convergence, (3) MPTCP-style independent subflows are not suitable for the RDCN environment where only one path is active at a time.

3.2 Design

The lessons above motivate a TCP variant designed for networks with fast, periodic changes in path characteristics. Our goal for TDTCP is concrete: handle each of the network’s distinct paths efficiently, but in an interleaved rather than simultaneous fashion, since only one path is active at a time. To meet this goal, we believe the answer is TDTCP, designed specifically for such use cases. TDTCP takes advantage of the best of both worlds, leveraging building blocks and concepts from both single-path TCP and MPTCP. In this section, we explore our design decisions for TDTCP.

3.2.1 Per-TDN state variables

A time-division multiplexed path, with distinctive traits setting itself apart from a stable single path, should be treated like distinct paths. Particularly, TDTCP should model the path characteristics of different TDNs independently. Traditional single-path TCP lacks the ability to distinguish its own operating states in different TDNs. Upon a path change, traditional single-path TCP has no accumulated state information about the new TDN; all of its state is modeling the previous TDN. Instead of invalidating this model, it tries to reconverge with samples taken in the new TDN. For instance, using TCP's standard exponentially weighted moving average technique to merge bandwidth estimates for two different TDNs results in a value that is too large for the packet TDN and too small for the optical TDN. A natural fix is to maintain independent copies of internal state and dynamically swap them in when the relevant TDN is active. These sets are kept isolated so that samples from different TDNs do not pollute each other.

TCP models path state using many different internal variables. We group these into three categories according to their main functions and for brevity only list a few of the variables themselves. “Pipe” variables such as `pkts_out`, `lost_out`, and `retr_out` are used to estimate the number of bytes in flight, lost, and retransmitted, respectively. This gives TCP a sense of how full the “pipe” is. Congestion control variables such as `cwnd`, `ssthresh`, and `ca_state` are used to control how many more bytes are allowed into the network and whether the sender should slow down. Delay/RTT variables such as `srtt`, `rttvar`, and `mdev` are used to track the path latency. All three categories of path state variables require per-TDN copies to make TDTCP truly model each TDN separately.

In stable state, each per-TDN set of variables closely tracks its corresponding TDN. When the network path changes, TDTCP swiftly replaces the variables in use with another set tracking the new TDN. The new set of variables already contains a snapshot view of the new TDN when it was last active. After the replacement, TDTCP continues to operate on the new TDN as if it has just resumed from a checkpoint. In principle, variables of inactive TDNs should not be modified when that TDN is inactive to avoid unexpected state corruption. However, there are certain key exceptions that require updates to the state of inactive TDNs. These exceptions are to handle reordering scenarios and to make sure that when the inactive TDN becomes active again, it is modeled accurately. As a simple example, suppose that a packet is sent on TDN 0 and its ACK returns on TDN 1. TDTCP needs to decrease the in-flight count for TDN 0 even though TDN 1 is active. This ensures that when TDN 0 becomes active again, TDTCP has accurate knowledge about the in-flight packets. In addition to the above state variables, TDTCP also tags each packet and ACK with a TDN ID and keeps track of it throughout the lifetime of the packet. TDTCP can find out on which TDN a particular packet is sent by examining this field. This information enables some heuristics for managing cross-TDN reordering and spurious retransmissions. We discuss reordering scenarios in more detail in §3.2.4.

3.2.2 TDN change notification

By modeling path states separately, TDTCP implicitly requires the ability to promptly determine when network transitions occur. Instead of relying on in-band notifications like packet marks (e.g., explicit congestion notification [63]) or iteratively probing the path, TDTCP relies on an

out-of-band hint to decide when to use which set of states. Given that all switches in the RDCN know when the path change will happen and what the new path will be, it is reasonable to have the switches at the network edge, i.e., ToRs, propagate this information to the end hosts using proactive notifications. We employ a dedicated Internet Control Message Protocol (ICMP) [181] packet to carry this notification. We prefer this alternative to an in-band signal, such as piggybacking the notification in packet headers, because our out-of-band ICMP approach avoids potentially deferring the notification indefinitely in the absence of a continuous flow of packets. The ICMP packet simply carries an integer index indicating which network path is currently active. We refer to this path index as the TDN ID.

One major concern is whether TDTCP can obtain the most up-to-date TDN ID in a timely fashion, as this governs TDTCP’s operating efficiency. For example, if a packet-to-optical TDN change notification arrives at the sender when the optical day is about to end, then TDTCP will miss the opportunity to send at higher speeds. Vice versa, TDTCP could experience excessive packet drops if it continues to send at the optical bandwidth after the network has transitioned into a packet TDN. While the delivery latency differs network-by-network, our design of ToR-generated notifications should keep it generally low since ToRs and end hosts are located within the same rack. In §3.4.4, we evaluate several optimizations that further minimize this latency.

Delivery assumptions. TDTCP assumes that ToR-generated notifications are delivered reliably and in order within the rack. Because the ToR and its attached hosts share a single hop, we expect loss, duplication, delay, and reordering of notifications to be rare. Our current prototype does not implement explicit recovery for these cases, and a bare integer TDN ID is not by itself sufficient to distinguish a current notification from a stale one. A lost or late notification therefore leaves a host operating with an out-of-date TDN ID until the next notification arrives; in the interim, the per-TDN reordering heuristics (§3.2.4) limit the resulting damage by avoiding spurious retransmissions across TDNs. Hardening notification delivery, for example by sequencing notifications so that stale ones can be detected and discarded, is a worthwhile extension we leave to future work.

3.2.3 Sequence numbering

TDTCP maintains a single, connection-level sequence number space despite viewing each TDN independently. This decision contrasts with MPTCP’s two-level design for its subflows. Three reasons lead to this design decision: (1) flow control stalls that occur due to ACK delays after a switch, (2) it does not require inter-subflow coordination to facilitate loss recovery, and (3) only minimal receiver-side modifications are needed.

Before we discuss the benefits of TDTCP’s single sequence number design, let us summarize why MPTCP uses a separate sequence number space for each path. MPTCP benefits from this design in two ways. First, MPTCP assumes multiple paths that operate (if one ignores potentially shared bottlenecks) completely independently, thus using full-fledged TCP connections as subflows is a natural fit. Second, and perhaps more pragmatically, MPTCP sought to keep each subflow compatible with various Internet middleboxes. Subflows with discontinuous sequence numbers may be incorrectly identified by middleboxes and blackholed or reordered. Fortunately, such universal compatibility is not a design requirement for TDTCP. Data center operators have significant control over network functions, and we assume they can make whatever changes may

be necessary to support TDTCP, as they have for DCTCP and other recent TCP variants.

Avoiding flow control stalls after a TDN switch. As we observe in §3.1.2, MPTCP has poor performance after a TDN switch because of flow control. The reason is that ACKs for packets sent over the old TDN right before the switch may be stuck in the send queue of the receiver, since they must be transmitted over the same TDN as the data packets they acknowledge. It is only after the packets have been reinjected over the active TDN that transmission can continue. With a single uniform sequence number space, packets received on one TDN can be ACKed on another TDN, and they naturally contribute to the overall sequence-space progress and drive transmission forward regardless of the currently active TDN.

Coordination. TDTCP’s single sequence number space eliminates the need for inter-subflow coordination. MPTCP requires a scheduler to serve as the central point of coordination. It makes a decision on which subflow to use when sending out a packet. In TDTCP, the active TDN is dictated by the network: there is only one TDN available at any point in time. Because all TDNs draw from one sequence space and one send buffer, a segment lost in one TDN need not wait for that TDN to become active again. It is retransmitted at the next send opportunity, on whichever TDN is active at that time, and immediately contributes to the connection’s forward progress. The loss is still charged to the congestion state of the TDN on which the segment was originally sent (§3.2.4), so that TDN’s window, and only that TDN’s window, is reduced. This is precisely the behavior that lets TDTCP avoid the connection-level reinjection MPTCP uses to remap lost segments across subflows [188], eliminating that mechanism’s implementation complexity and performance overhead.

Receiver processing. MPTCP’s two-level sequence number space design requires runtime translation between data sequence number and subflow sequence number. This means the receiver also needs extensive modifications. However, with a single uniform sequence number space, no receiver-side modification is needed to support TDTCP sequencing. The major downside of TDTCP’s single sequence space and simple receiver behavior is cross-TDN reordering. We describe the impact of this phenomenon and how it can be mitigated below.

3.2.4 Packet reordering

Since the days of Reno, TCP has employed a fast-recovery mechanism to ameliorate the impact of packet loss. Specifically, segments deemed lost due to the receipt of three consecutive dupACKs are queued for immediate retransmission. This time-tested heuristic builds upon an assumption that no longer holds true in RDCNs: that a network path only slightly reorders packets, if at all. In fact, most reordering seen in RDCNs is cross-TDN reordering, where packets and their corresponding ACKs traverse TDNs with different latencies. Cross-TDN reordering does not signal loss; ACKs are simply delayed due to increased path latency. If TDTCP were to continue following TCP Reno’s fast-retransmit heuristics, it would spuriously retransmit a large number of packets at each TDN transition (indeed, over 100 in our setting according to measurements in §3.4.3). Notably, cross-TDN reordering arises only when the schedule transitions from a high-latency to a low-latency TDN; transitions in the opposite direction cause packets to arrive farther apart but still in order.

Cross-TDN reordering. Occurrences of cross-TDN reordering can be grouped into two categories: data reordering and ACK reordering. Figure 3.3 shows an example of each: scenario

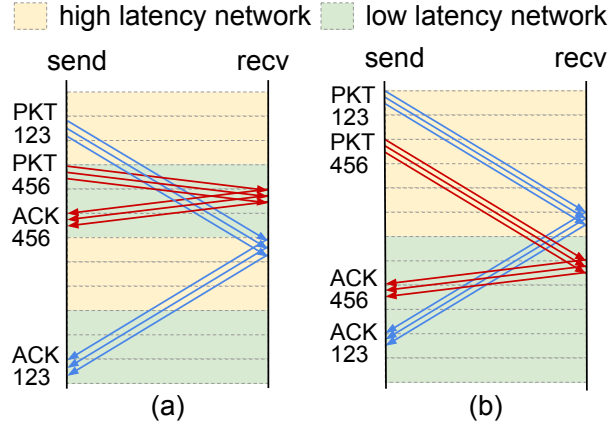


Figure 3.3: Cross-TDN reordering scenarios. Time proceeds from top to bottom; arrows represent TDTCP segments transmitted between sender and receiver.

(a) corresponds to data reordering, and scenario (b) represents ACK reordering. Every horizontal slot in Figure 3.3 represents a unit of time t . One-way delays of the high-latency (yellow) network and low-latency (green) network are $5t$ and t , respectively. Consecutive slots of identical color represent the active duration of a TDN. For example, in scenario (a), the active TDN switches every $3t$. We label each data segment and its corresponding ACK with individual sequence numbers for clarity. (In practice TDTCP continues to use payload-byte-based sequence numbering and cumulative acknowledgments.)

Data reordering typically happens when the last few segments of a sending episode enter the network after the new TDN becomes active. In scenario (a), segments 4–6 are sent later but arrive before segments 1–3, causing reordering at the receiver. The receiver acknowledges segments 4–6 and 1–3 in the order they are received. The arrival of ACKs 4–6 before 1–3 would be interpreted by TCP Reno as a loss event, causing the sender to enter fast recovery mode and retransmit spuriously.

In scenario (b), the receiver sends out ACKs in the expected order. However, half of the ACKs are transported by a different TDN with lower latency. ACKs 4–6 end up arriving at the sender before ACKs 1–3. Unlike data reordering, ACK reordering is largely harmless because the latter ACKs carry a higher acknowledgment number than the former ACKs. Following TCP’s cumulative ACK semantics, upon receiving ACKs 4–6, the sender assumes segments 1–6 have all been received, which implicitly nullifies the in-flight ACKs 1–3. When ACKs 1–3 arrive, they are simply discarded.

Relaxed detection. In order to avoid frequent spurious retransmission, TDTCP relaxes the triple dupACK heuristic with the help of per-TDN state and SACK support [62]. Whenever TDTCP detects a gap in the sequence number space, it first inspects the associated TDN IDs of all segments falling in the missing sequence range and compares them with a TDN change pointer. The TDN change pointer tracks the final sequence number of the previous TDN/the first sequence number of the new TDN. When packets in the “hole” have a different TDN ID than the ACK(s) that triggered the heuristics, the sender suspects cross-TDN reordering. As a result, TDTCP does not consider such segments to be lost and retransmits only those with a matching TDN ID (if any).

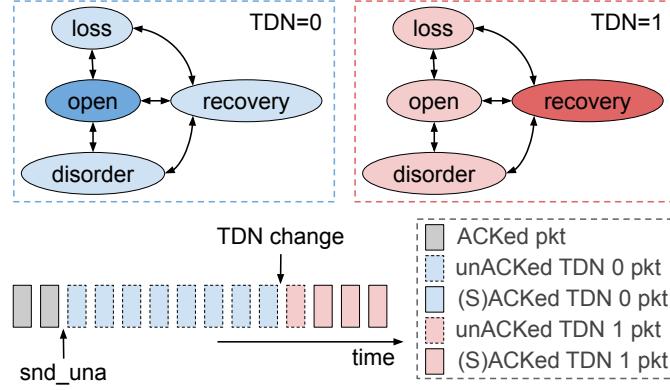


Figure 3.4: An example of relaxed reordering detection.

TDTCP also updates the per-TDN congestion state machine associated with the lost segments; TDNs with retransmissions will enter the recovery state. For cases where lost segments with a different TDN ID are true tail losses (i.e., the last segments of a prior TDN really were dropped by the network), TDTCP relies on RACK-TLP [40] to recover efficiently.

Figure 3.4 illustrates TDTCP’s relaxed reordering detection heuristic. When (pink) segments sent over TDN 1 are acknowledged before outstanding (blue) segments from TDN 0, SACK marks the successfully acknowledged segments and invokes fast recovery. All segments between the `snd_una` pointer and the highest SACKed sequence number (dashed rectangles) might be lost and potentially need to be retransmitted. However, TDTCP inspects the associated TDN IDs of these packets and compares them with the TDN change pointer. (Dashed blue) segments from TDN 0 are ignored since they belong to a different TDN and their ACKs are very likely just delayed. Only one (dashed pink) segment belonging to TDN 1 is confirmed as a true loss, which will be retransmitted. TDN 0 remains in `Open` state and is allowed to continue sending at full speed; TDN 1, on the other hand, enters `Recovery` state due to the loss.

3.2.5 Fairness, generality, and limitations

Fairness. TDTCP does not propose a new congestion control algorithm (CCA). Rather, it simply implements one of the available CCAs in each TDN. In principle, TDTCP could use multiple, different CCAs within a single flow. In our current implementation, however, CUBIC is used in all TDNs. We expect CCAs used within each TDN to have similar fairness properties as their single-path siblings, over a long period of time. Short-term anomalies could however exist. An in-depth investigation regarding fairness across TDNs and disparate CCAs remains as future work.

Generality. Hybrid, demand-oblivious RDCNs motivate our work, but TDTCP is potentially applicable to any environment that satisfies a small set of conditions: (1) the network moves between a fixed set of internally coherent conditions; (2) each condition recurs during the lifetime of a connection and is stable and distinguishable enough to be modeled separately; (3) only one such condition (path) is relevant at a time; (4) the two endpoints use consistent TDN identifiers; and (5) hosts receive timely transition notifications. Environments beyond the setting we evaluate are therefore potentially applicable rather than demonstrated. Satellite-based communication, for

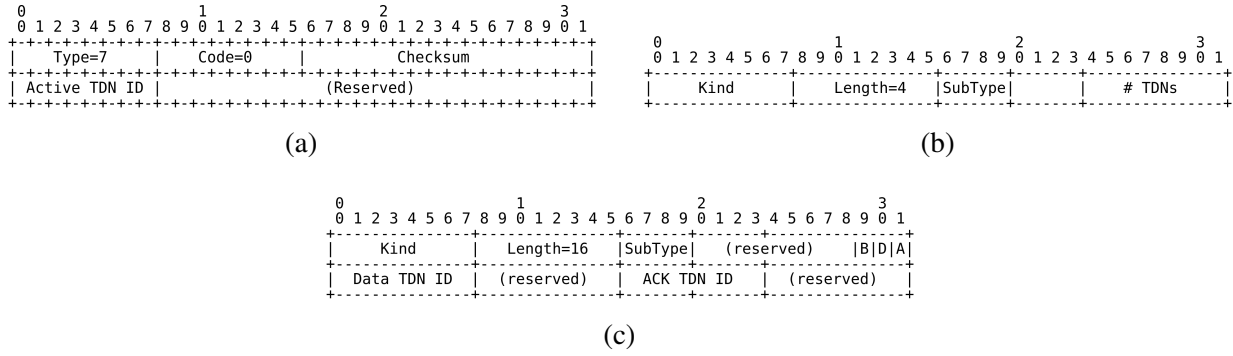


Figure 3.5: TDTCP packet formats: (a) ICMP for TDN change notification, (b) TCP option for the TD_CAPABLE handshake, (c) TCP option of TD_DATA_ACK exchange.

instance, appears to be a good fit: satellite signal coverage has a periodic strong-weak pattern as satellites orbit the earth, links are used when a strong signal is present, and fiber links between ground stations serve as a backup when the signal is weak [92], with only one link selected at any time.

Limitations. The main limitation of TDTCP is its operating regime. TDTCP is only useful when the network conditions change with a certain frequency. Consider two extreme cases: (1) TDNs change every few hours, (2) TDNs change every few nanoseconds. In case 1, a regular single-path TCP performs well because few connections will ever experience a change in TDN, and those that do can amortize any performance degradation over a very long period of time. In case 2, traditional TCP variants can perform near-optimally because nanosecond-level TDN changes will appear similar to per-packet load balancing. TDTCP is therefore intended primarily for networks where the interval between TDN changes is on the order of 1–100 path RTTs; we offer this range as a heuristic rather than a sharp boundary.

3.3 Implementation

We have implemented TDTCP in Linux 5.8 kernel with 11,798 lines of C code changes. The code is open source and available at <https://github.com/shuoshuc/TDTCP>. In this section, we describe a few implementation-specific considerations. More details can be found in the source code, accompanying comments and corresponding documentation.

3.3.1 Packet format

TDTCP defines three types of packets for its operation. An ICMP packet is used for path change notifications. A TCP handshake option is used to negotiate and establish TDTCP connections. Another TCP option is used to inform the peer of the TDN path taken by the packet.

As discussed in §3.2.2, TDTCP uses a switch-generated path change notification to learn about RDCN schedule changes. The active TDN ID field is carried as the first byte of the ICMP payload. We assume the number of distinct paths in an RDCN is no more than 256, hence allocating a

single byte in the packet. Should the use case require a larger integer, we can easily allocate more bytes for it. Figure 3.5a shows the ICMP packet format of the notification.

3.3.2 Connection establishment

A TDTCP-speaking end host uses TCP header options inside of normal TCP connections. TDTCP exposes the same application-level API as traditional TCP so it is fully compatible with existing applications. TDTCP is also transparent to the application layer: it does not require multiple network interfaces nor require the applications to make any decisions regarding path selection.

To enable hosts to establish a TDTCP connection, TDTCP defines a new TCP option (Figure 3.5b) of subtype `TD_CAPABLE`, indicating that the end host supports TDTCP. The option also carries the number of TDNs observed by the end host. In our current implementation, the two ends must both agree on the number of TDNs. Agreeing on the count alone, however, does not guarantee that a given ID denotes the same network condition at both ends. The mapping from TDN ID to network condition is established out of band by the RDCN control plane, which assigns each configuration a globally consistent index and programs the ToRs that generate the notifications accordingly. Both endpoints therefore inherit the same ID-to-condition mapping from the network rather than negotiating it during the TCP handshake. On an established TDTCP connection, all segments carry a TCP option with subtype `TD_DATA_ACK` (shown in Figure 3.5c) that contains the ID of the TDN on which the segment was sent. If the segment contains data, the data TDN ID field should be properly filled and the `D` bitflag set, likewise for an ACK segment. This option is used by the TDTCP state machines and heuristics to identify segment reordering.

Our current TDTCP implementation supports runtime RDCN schedule changes, but the new number of TDNs seen by both the sender and receiver must remain equal. TDTCP automatically initializes a new set of state variables upon being notified of a new TDN for the first time. We have also implemented the ability for a host to downgrade established TDTCP connections to regular TCP. When doing so, only the local side is affected; the peer can continue to send TDTCP-enabled segments but only regular ACKs will be returned. We find this feature quite helpful for debugging.

The initial `SYN` packet requires special handling during connection setup. Although it is counted as a segment and increments `packets_out`, the Linux TCP stack does not keep a local copy of it as it does for other segments. Moreover, there is not yet a notion of TDNs at this point because the TDTCP negotiation has not completed. Consequently, when the `SYN/ACK` arrives, `packets_out` should be decremented to reflect the delivery of the `SYN`, but TDTCP does not know which per-TDN counter was incremented. To resolve this ambiguity, TDTCP treats the first `SYN` as a special case and always tracks it under TDN 0.

3.3.3 TDN state management

Mapping per-TDN state variables from design to implementation is not entirely straightforward. Due to the way connection state is maintained in the Linux TCP stack, there are subtle semantic differences affecting how per-TDN state should be managed, requiring case-by-case inspection. At a high level, state variables fall into one of four different classes: current TDN, all TDNs, any TDN and specific TDN. We provide examples of each below.

Current TDN. The most common and natural semantic is for state to be accounted entirely within the current TDN. For example, when transmitting new data, each segment is tagged with the current TDN ID. Relevant state variables that track in-flight data such as `packets_out` should be incremented for the current TDN only.

All TDNs. On the other hand, sometimes the state of other TDNs must also be considered. Linux uses the variable `packets_out` in its `tcp_ack()` function to filter out ACKs that do not need to be processed. If `packets_out` is 0, the ACK must be either stale or malicious because no data is pending; hence, no ACK is expected. In the context of TDTCP, `packets_out` must be 0 across all TDNs because an ACK can acknowledge data sent in any TDN. This means the sum of all per-TDN `packets_out` should be used to verify ACKs.

Any TDN. TDTCP retransmits lost segments at the earliest opportunity, regardless of which TDN was used originally to transmit them. The Linux TCP stack maintains variables `ca_state` and `lost_out` to decide if a retransmission should be scheduled. If `ca_state` is Loss or Recovery and `lost_out` is non-zero, Linux prioritizes retransmitting the lost segment(s). In the context of TDTCP, retransmission needs to be scheduled if the above condition is true for any TDN, which means a logical OR should be applied to the two variables of all TDNs.

Specific TDN. In the receiving path, an incoming (cumulative) ACK can acknowledge data sent over multiple TDNs. When segments are acknowledged, their corresponding tracking variables such as `packets_out` and `retrans_out` must be decreased to reflect the progress. In the context of TDTCP, we identify the set of segments being acknowledged by scanning the retransmission queue and obtaining the TDN ID associated with each one of them. The variables `packets_out` and `retrans_out` are updated for the specific TDN associated with each.

3.3.4 RTT estimation

RTT estimation faces the same problem as in §3.2.4. Regular TCP assumes stable routing when collecting RTT measurements, i.e., that the samples always reflect properties of the same network condition. However, TDTCP may sample three distinct conditions even with only two TDNs. Let the one-way delay of TDN i and TDN j be denoted as $\frac{1}{2}RTT_i$ and $\frac{1}{2}RTT_j$, respectively. Type-1 samples measure RTT_i , type-2 samples measure RTT_j , and type-3 samples measure $\frac{1}{2}RTT_i + \frac{1}{2}RTT_j$. Type-3 samples arise when a segment and its ACK traverse different TDNs.

We seek to discard type-3 samples and match type-1 and type-2 samples to their corresponding TDNs. To achieve this goal, TDTCP makes use of the `TD_DATA_ACK` option and its local per-segment state tagging. TCP already uses the Karn-Partridge algorithm [119] to filter out retransmitted segments in RTT measurement; TDTCP further ignores samples with different data and ACK TDNs. The remaining samples are matched to the appropriate TDN based on information in the option.

One special consideration is the timeout value used by the retransmission timer. TDTCP only knows which TDN a segment is sent on, but cannot predict in general upon which TDN the corresponding ACK will return. In order to avoid premature timeouts, our current TDTCP implementation pessimistically assumes ACKs will return on the TDN with largest RTT. The timeout value of a segment sent on TDN n is based on a synthesized RTT of $\frac{1}{2}RTT_n + \frac{1}{2}RTT_{slowest}$.

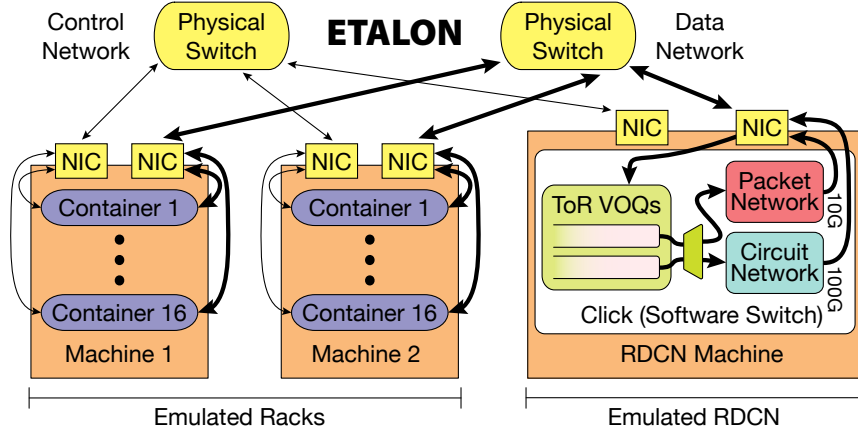


Figure 3.6: Our Etalon deployment. The emulated packet and circuit networks are 10 Gbps and 100 Gbps, respectively. Note Etalon only emulates the RDCN, the hosts are implemented as containers on physical machines.

3.4 Evaluation

We evaluate TDTCP using our in-kernel Linux implementation on a small-scale testbed using Etalon [169], an RDCN emulator. First, we compare TDTCP performance to leading data center TCP variants in an RDCN setting: §3.4.2 shows that TDTCP achieves high throughput and low switch-buffer occupancy for long-lived flows despite both bandwidth and latency differences. Then, we revisit design decisions on reordering and TDN change notification in §3.4.3 and §3.4.4 through microbenchmarks.

3.4.1 Etalon testbed

Etalon is a software RDCN emulator based on the Click modular router [127] and DPDK [64]. We configure Etalon to emulate a simple two-rack demand-oblivious hybrid RDCN topology, as illustrated in Figure 3.6.

Hardware. We use two physical servers to emulate two racks of hosts; Etalon itself runs on a third server. We fix the source and destination of all connections in our experiments to use hosts attached to one of these two racks. We can emulate any scale of RDCN using this topology by pinning flows between this pair of racks. Each server runs 16 Docker containers [158] to emulate 16 distinct hosts residing in the rack. All servers have two 20-core Intel Xeon E5-2680 CPUs, 128 GB of memory, a 10 Gbps NIC for the control network, and a Mellanox ConnectX-3 40 Gbps NIC for the data network. We use *flowgrind* [253] to generate cross-rack flows over the data-plane network. Note that TDTCP is running in the actual Linux kernel networking stack, not through emulation or simulation; only the network itself is emulated.

RDCN configuration. In order to emulate a realistic target RDCN, we set the path-related parameters in Etalon according to §3.1.1. The number of TDNs is set to 2: TDN 0 corresponds to the EPS network and TDN 1 to the OCS network. The bottleneck bandwidth/average RTT of TDNs 0 and 1 are 10 Gbps/100 μ s and 100 Gbps/40 μ s, respectively. Etalon emulates two ToR

virtual output queues (VOQs) for each rack, one for each traffic direction, which can be either static or reconfigurable at runtime. As a baseline configuration, we set the VOQ size to 16 packets (with jumbo frames [51] enabled). This is slightly larger than the packet network bandwidth-delay product, which has been shown to be sufficient for both TDNs [13]. One limitation of our testbed is the 40-Gbps hardware. In lieu of 100-Gbps network devices, we leverage time dilation [84] to “slow down” the end hosts. As a result, we are able to emulate a 100-Gbps network using only 5 Gbps of physical network capacity by dilating time by $20\times$.

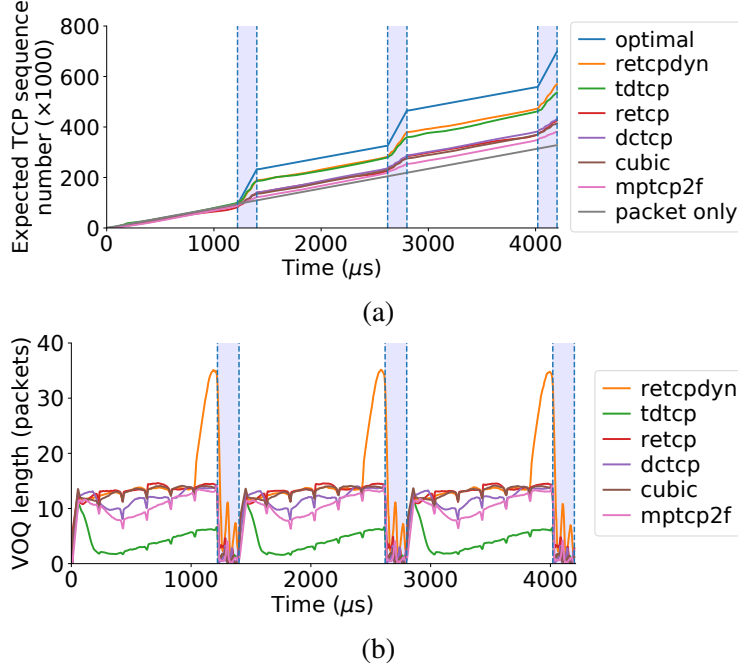


Figure 3.7: TCP throughput and switch buffer utilization under both bandwidth and latency difference: (a) sequence graph of each TCP variant, (b) ToR VOQ utilization over time (capped at 16 for all but reTCP with dynamic buffers, `retcpdyn`).

We measure long-lived cross-rack flow throughput for the various TCP implementations under a single representative RDCN schedule; all 16 hosts in one source rack transfer bulk data to their corresponding hosts in the destination rack. All flows are configured to start at the same time and last for 40 seconds. We deploy a pre-computed static schedule (in the fashion of RotorNet [154]) that consists of a variable number of 180-microsecond-long configurations. In every configuration, the racks are either connected directly to one another through the OCS (i.e., TDN 1) or to all other racks through the packet-switched network (TDN 0). Each configuration is separated out by a blackout (reconfiguration) period of 20 microseconds. Our experiments focus on a schedule with a 6:1 ratio of packet to optical TDNs, which naturally corresponds to an 8-rack hybrid RDCN where each rack is directly connected to the others in 1 out of every 7 OCS configurations. TDTCP has the most advantage over other TCP variants with ratios on this order. We leave it as future work to study TDTCP’s performance when operating under extreme ratios.

We focus exclusively on long-lived flows because short-lived flows are unlikely to benefit from TDTCP. For example, remote procedure call workloads that last a few RTTs likely only exist

during one TDN. Even flows lasting tens of RTTs would terminate before exiting slow start in either TDN. In such cases, a larger initial `cwnd` would be more helpful than TDTCP. Overall, we do not expect TDTCP to impact the completion time of short-lived flows but a full treatment is outside the scope of this chapter.

3.4.2 Bandwidth and latency difference

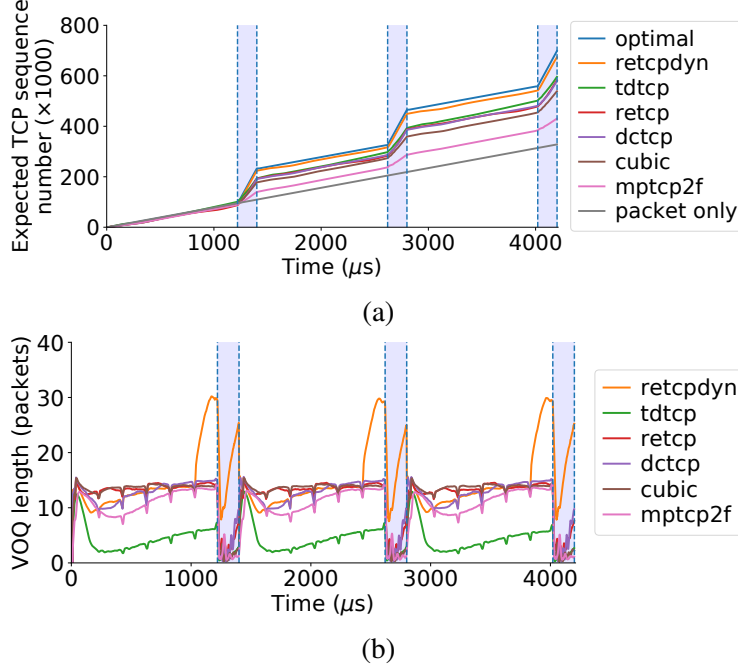


Figure 3.8: TCP throughput and switch buffer utilization with only bandwidth difference: (a) sequence graph of each TCP variant, (b) ToR VOQ utilization over time (capped at 16 for all but reTCP with dynamic buffers, `retcpdyn`).

To understand the reasons behind TDTCP’s superior performance we compare TDTCP with a wide variety of common TCP variants used in data centers: single-path TCP CUBIC and DCTCP [8], and MPTCP with two subflows, one for each network condition. In addition, we also compare to reTCP [169], an approach to improving single-path TCP performance in RDCNs that requires active switch buffer management.

Figure 3.7 shows a side-by-side comparison of how each TCP variant performs when an RDCN oscillates between two conditions with differences in both bandwidth and latency. Specifically, the shaded area shows time periods when the source and destination racks are directly connected by the high-bandwidth/low-latency (optimal) path; otherwise the connections are over the low-bandwidth/high-latency (electrical) path. Figure 3.7a shows the relative sequence number increase over time, which reflects the overall flow throughput. The “optimal” line plots the upper bound where an idealized TCP fully utilizes both conditions. The “packet only” line shows the same idealized TCP using the packet network only (i.e., it remains on the packet-switch fabric even with the optical path being available). The difference between these two lines represents the

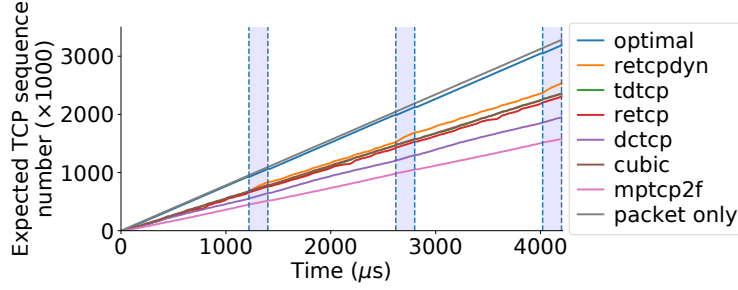


Figure 3.9: TCP throughput with only latency difference at 100 Gbps bandwidth. (TDTCP and CUBIC perform almost identically.)

potential gain a TCP connection can capture in this RDCN by making use of the additional optical capacity. Note that the plots represent a ≈ 4 -ms period during the experiment, not the absolute start (although both x and y axis values are normalized to the start of the plotted period). As a result, the performance of each variant is not necessarily equal even in the first TDN due to the state (e.g., send window and RTT estimates) accumulated earlier in the trace (c.f. Figure 3.9).

TDTCP substantially outperforms CUBIC, DCTCP, and MPTCP, as none of the traditional variants are able to effectively make use of the additional capacity made available by the optical link. Quantitatively, TDTCP achieves 24% higher throughput than single-path CUBIC and DCTCP and 41% higher throughput than MPTCP in this representative setting. The only competitive alternative is reTCP, but it requires sophisticated dynamic switch buffer resizing [169] in order to be effective. The reTCP-with-dynamic-buffers line (labeled `retcpdyn` in the plots) corresponds to a setting where the ToR enlarges its VOQ size to 50 packets at 150 microseconds ahead of the TDN change, and notifies reTCP to ramp up its congestion window. Thus, reTCP is able to pre-fill the VOQ and starts bursting at high bandwidth immediately after the TDN switch. In contrast, reTCP without dynamic buffer resizing (“retcp”) sees a similar throughput to other single-path TCP variants. TDTCP, on the other hand, does not require any buffer resizing at runtime. In fact, Figure 3.7b shows that TDTCP’s VOQ utilization is the lowest among all of the TCP variants we consider.

One pattern worth noting in TDTCP’s VOQ curve is the spike when the RDCN transitions from high bandwidth/low latency to low bandwidth/high latency. We refer to the spike as the initial burst. The initial burst occurs because TDTCP is switching to a wide-open congestion window with near-zero inflight for the new active TDN. During the previous (optical) TDN, no new segments were arriving at the ToR’s packet queue while virtually all in-flight packets have been drained. This process opens up the packet TDN’s congestion window, hence the sender transmits a `cwnd`-sized burst upon receiving the TDN change notification. Techniques such as sender pacing can help prevent the potential switch buffer overflow.

To differentiate the impact of bandwidth difference versus latency difference on flow throughput, we repeat the same experiments but configure Etalon with only bandwidth difference or only latency difference at a time. Figure 3.8 shows the results for an RDCN with only bandwidth differences. Figure 3.8a suggests both CUBIC and DCTCP are able to adapt to bandwidth variation alone; they only slightly underperform TDTCP in this setting (although MPTCP still struggles). Interestingly, the reTCP results suggest that dynamic buffer resizing, in the absence of latency

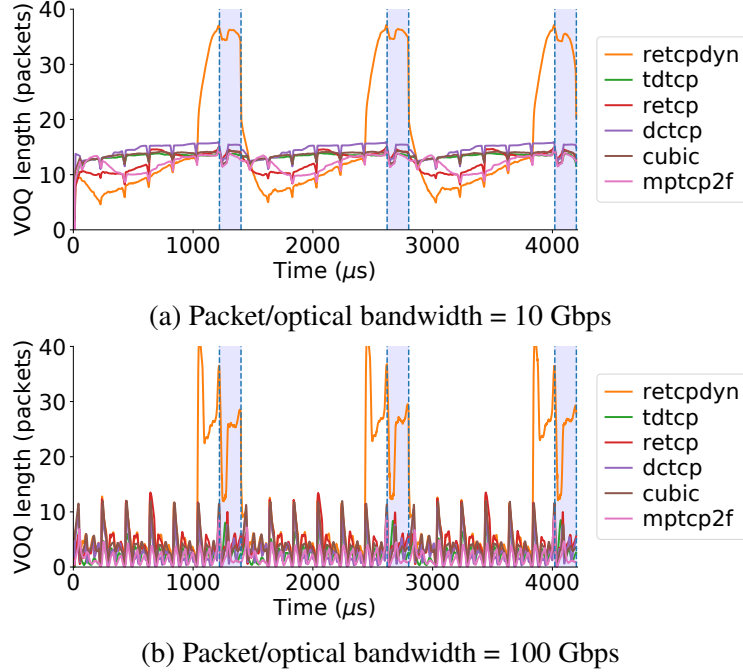


Figure 3.10: ToR VOQ utilization for an RDCN with only latency differences (max VOQ size is 16 packets for all cases except reTCP with dynamic buffers, `retcpdyn`). Packet and optical RTTs are $20\ \mu\text{s}$ and $10\ \mu\text{s}$, respectively. Bandwidth is fixed.

variation, can achieve near-optimal throughput. Without it, however, reTCP falls back in line with the other variants. Figure 3.8b shows that VOQ occupancy is largely unchanged in this scenario, with TDTCP still the clear winner.

Next, we consider a setting where the two TDNs have equal bandwidth but differing latency, such as in a multi-hop Opera-style [156] RDCN. Figure 3.9 illustrates the throughput when link bandwidth is 100 Gbps. Because both TDNs have identical capacity, the optimal line and packet-only line almost overlap; the packet-only performance is slightly higher because flows in packet-only mode do not experience the reconfiguration blackout period. All of the buffer-filling TCP variants see similar throughput while DCTCP, a latency-sensitive variant, performs worse than its peers (with MPTCP again bringing up the rear). We hypothesize that the TDTCP approach could allow even latency-sensitive CCAs to perform well in such RDCN setting, but leave this topic to future work.

The VOQ behavior in this latency-only regime, shown in Figure 3.10 for both 10 and 100 Gbps, reinforces TDTCP’s low-buffer advantage. TDTCP’s buffer occupancy stays in line with CUBIC, DCTCP, and MPTCP, whereas reTCP again builds up large queues ahead of circuit start. Because bandwidth is fixed here, the circuit bandwidth-delay product is actually smaller than that of the packet network (the circuit has lower latency), so reTCP’s queue-building is mismatched, whereas TDTCP attains high throughput without it.

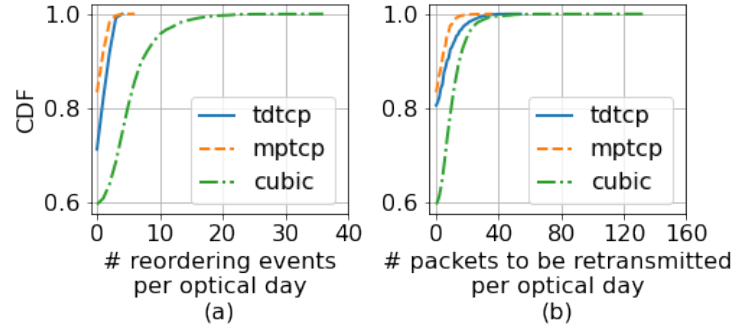


Figure 3.11: CDFs of (a) reordering events and (b) packets subject to potential spurious retransmission, per optical day. Note the y -axis does not start from 0.

3.4.3 Reordering

At first glance, CUBIC’s performance under latency variation is surprising, as TDTCP includes numerous heuristics (§3.2.4) to avoid packet reordering that one would expect to result from transitioning from a high-latency (packet) TDN to one with much lower latency (optical). To understand this behavior, we plot the number of reordering events in each optical day and the total number of packets impacted by this reordering. In one reordering event, multiple packets could be affected if their sequence numbers all fall into the gap between cumulative ACK sequence number and SACK sequence number. The total number of affected packets is the number of segments that will be resent as spurious retransmissions (if the congestion window permits). The measured reordering includes both cross-TDN reordering and intra-TDN reordering, which is an intrinsic property of the TDN itself.

Figure 3.11 depicts the cumulative distributions of reordering events and retransmitted packets per optical day, for 3 representative TCP variants: CUBIC, MPTCP and TDTCP. CUBIC does, in fact, experience significant reordering and spurious retransmission: 15 packets are retransmitted per TDN transition at the 90th percentile and 133 at maximum. TDTCP effectively cuts off the tail, as only 7 packets are retransmitted at the 90th percentile with a maximum of 54. Moreover, 80% of the transitions to the optical TDN do not experience any reordering or retransmission at all. Hence, we conclude that CUBIC’s impressive performance is a result of the sophisticated heuristics implemented in the Linux stack that are able to maintain an appropriate congestion window and high utilization despite significant reordering. Less finely-tuned stacks are unlikely to perform as well. We note that MPTCP maintains separate subflows for each of the two TDNs, so there should be no reordering due to TDN transitions. Hence, one can interpret the MPTCP line in these plots as a measure of the baseline intra-TDN reordering during the experiments.

3.4.4 TDN change notification

Following discussions in §3.2.2, we analyze some TDTCP traces and identify the component breakdown of end-to-end notification delivery latency. Barring the physical propagation delay, top contributing components are ToR-side packet generation and host-side notification processing. We list 3 concrete optimizations below that prove effective in reducing the component latency. These

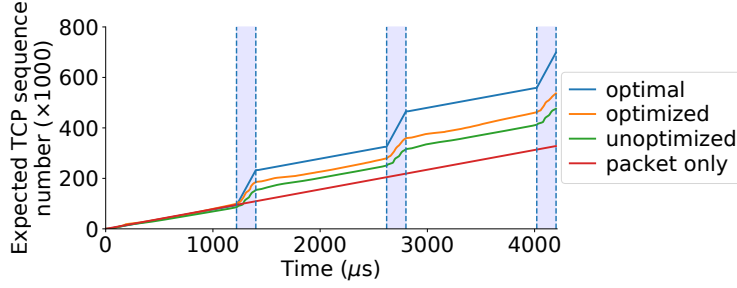


Figure 3.12: TDTCP performance with and without all optimizations on TDN change notification.

optimizations combined improve TDTCP’s throughput by 12.7%, as demonstrated by Figure 3.12.

First, ToRs can use caching to minimize delay: by constructing an empty ICMP packet in advance and keeping it cached in memory, ToRs can just fill in the TDN ID and immediately send the packet out when needed. Compared to constructing a new packet from scratch every time, caching reduces the delay in sending out the notification in our software switch by $8\times$ at the 50th percentile and $2.7\times$ at the 99th.

Second, our initial implementation follows a “push” model to distribute new TDN IDs: the kernel loops over all established TDTCP flows and updates their current TDN IDs one by one. This scales poorly and the unlucky flows which see the TDN update after others get less time to send. Instead, we change to a “pull” model where each TDTCP flow checks a global TDN variable protected by a read-write lock when they need to send or process incoming packets. The time to update all TDTCP flows is reduced by 3 orders of magnitude.

Finally, using a separate dedicated network to send and receive the ICMP packets can eliminate the extra queueing delay on a busy data plane interface with a lot of incoming/outgoing data packets. This optimization lowers the one-way delay between sending out an ICMP packet and completion of receiver processing by $5\times$ at both the 50th and 99th percentiles.

3.5 Related work

Existing TCP variants rely on in-band signals such as packet loss [86, 87], delay [30, 34], and explicit switch feedback [8] to estimate a fair share of the path bandwidth. TDN changes interfere with these signals, leading to estimates that tend to average across TDNs rather than track individual TDNs. TCP variants for high-speed networks ramp up more aggressively [61, 87] and can recover more quickly from estimation errors, but do not address the root of the problem. Wireless networks, with highly-variable bandwidth and frequent packet drops, have spurred research on modeling bandwidth fluctuations [230] and differentiating between congestion loss and random loss [65, 150]. RDCNs, on the other hand, can provide certainty regarding the current network conditions, which should in theory enable better performance.

TCP over multiple networks. MPTCP [231] is similar in spirit to TDTCP in that it divides a connection into multiple subflows that traverse (potentially) different paths, such as simultaneous WiFi and cellular connections on a mobile device. While theoretical analysis shows MPTCP can perform optimally by leveraging the least-congested path, our experiments show it is not well equipped to switch rapidly between subflows when the subflows experience periods of

disconnectivity as they do in RDCN.

One recent proposal, reTCP [169], focuses specifically on the RDCN environment and advocates for explicit support from ToR switches, which entice senders to ramp up their transmission rate before additional bandwidth becomes available by dynamically increasing the size of switch buffers in advance of circuit establishment. Furthermore, reTCP requires ToR switches to mark packets to inform senders when their flows traverse an optical network; senders then react by multiplicatively increasing their congestion window. TDTCP builds on the idea of explicit notification regarding network conditions but eschews such elaborate switch support. Moreover, TDTCP is general, supporting an arbitrary number of distinct TDNs with various properties, not just the bimodal fabric reTCP presumes. Even still, our experiments show that TDTCP’s performance is competitive with reTCP in precisely the environments for which reTCP was designed without requiring its extensive changes to ToR switches.

RDCN technologies. Our evaluation focuses on one particular class of RDCN, but there are many others in the literature. In hybrid RDCNs [55, 88, 144, 224], the OCS network is an accelerator for an existing packet network, and thus uses the packet network to hide the OCSes’ limited connectivity. Alternatively, OCS-only RDCNs [20, 35, 74, 89, 154, 156] do not include a separate packet network; instead, ToRs with no direct connectivity send traffic through transit ToRs or hold traffic until direct connectivity is restored. In either case, the number and frequency of TDN changes can vary. In demand-aware RDCNs [35, 55, 74, 88, 89, 142, 180], a controller collects real-time traffic demand information and calculates a schedule that serves the current demand. More recent demand-oblivious RDCNs [154, 156] cycle through a set of configurations that together provide all-to-all connectivity, avoiding runtime calculations by using a fixed schedule. TDTCP is applicable in either case; all that is required is that ToRs notify the senders of the upcoming TDN when they change configurations.

3.6 Conclusion

TDTCP is a new TCP variant designed to make efficient use of a broad range of recent RDCN proposals that provide time-division multiplexed network paths between communicating end hosts. TDTCP monitors path characteristics of different TDNs using independent copies of TCP state variables, leverages switch-generated path change notifications to rapidly converge to the optimal congestion state, and relaxes TCP’s fast-recovery heuristics to avoid spurious retransmissions when network conditions change. We have implemented TDTCP in Linux kernel 5.8 and evaluated its performance on a small-scale (emulated) RDCN testbed. Results show that TDTCP substantially decreases queue occupancy within the network and outperforms traditional single-path TCP variants such as CUBIC and DCTCP for long-lived flows. TDTCP is competitive with reTCP, a proposal targeting a particular class of RDCNs, but does not require switches to dynamically adjust buffer sizes as reTCP does.

Seen against the thesis’s arc, TDTCP shows what the thesis statement asks of the transport layer: the redesign that lets an RDCN deliver on its promise is not a tuning of existing congestion control but a structural change to it, one that models each configuration on its own and swaps between them as the network rotates. Two lessons carry beyond transport. First, endpoints cannot infer a reconfigurable fabric’s behavior from the traffic alone; they must be told, and the RDCN’s

own control plane, which already decides every reconfiguration, is the natural source of that signal. Second, because existing protocols were built for networks whose latency and bandwidth hold still, they falter precisely where an RDCN changes them most, so an RDCN redesign must confront those regular, drastic changes head-on rather than treating them as anomalies. TDTCP does both while leaving the network oblivious to real-time demand, riding whatever schedule the OCS happens to present. The next chapter takes the first step away from that obliviousness: it lets the fabric allocate bandwidth in response to measured demand, and in doing so trades transport's broken assumption of stable connectivity for a new one, that the bandwidth between any two points is uniform, moving the redesign up the stack from transport to traffic engineering.

Chapter 4

Precise Data Center Traffic Engineering

The previous chapter redesigned transport for a fabric that rode a fixed schedule, oblivious to which servers actually needed to talk. This chapter takes the first step away from that obliviousness, moving one notch toward the specialized end of the spectrum of Chapter 2. Here a general-purpose reconfigurable data center replaces its Clos spine with an OCS layer that allocates bandwidth between aggregation blocks in response to measured demand rather than provisioning it uniformly and statically. That shift breaks a different assumption than the one transport relied on: not the stability of connectivity, but the uniformity of bandwidth, since a spine-free fabric offers unequal, reconfigurable paths the network can no longer treat as interchangeable. Making the fabric demand-aware is thus the second of the thesis's three redesigns, and it falls not on transport but on traffic engineering, the layer that decides how traffic is spread across those uneven paths. What makes this more than a routing problem, and what the thesis statement anticipates, is that a reconfigured, non-uniform topology also changes what the control plane can feasibly install in real switch hardware: the difficulty lies not in computing a good routing plan but in realizing it within the finite forwarding tables of real switches.

DCNs use high-bandwidth links, richly connected topologies, and large-scale deployments to meet growing application needs. However, these additional resources come at a significant cost and DCN operation must ensure efficient utilization of the network infrastructure. TE plays a critical role in addressing this efficiency need by routing traffic over carefully chosen paths. Existing TE designs consist of two parts: (1) calculating the optimal plan for mapping of flows to links, which is called the TE solution; (2) implementing the TE solution on data plane switches, which we call TE implementation. Past TE research [4, 5, 9, 24, 80, 102, 121] has mostly focused on the TE solution, but the TE implementation impacts operational efficiency greatly and, despite this, has received much less attention.

Equal-Cost Multi-Path (ECMP) [101] is the most widely adopted method to implement TE solutions because of its universal hardware support. Weighted-Cost Multi-Path (WCMP) [249] is an extension of ECMP that supports a weighted split of traffic among paths by using forwarding table entry replication. This enables finer-grained traffic control but can consume much more space in the switch hardware table. As a result, two problems occur. First, some groups (the data structure that implements weight distributions in the switch) used by the TE solution cannot be installed in the switch after the switch group table becomes full, leading to traffic loss. Second, if the weights are adjusted to use less space in the group table, the resulting TE implementation may

have poor TE performance. We refer to the difference between link utilization of the TE solution and of the TE implementation as the precision loss.

Recently, spine-free DCN designs [19, 153, 157], the OCS-based spine replacement introduced in Chapter 2, replace the traditional Clos topology with clusters connected as a full mesh. Spine-free design reduces the total cost of ownership and accelerates DCN evolution, but we found that it also increases usage on the group table, exacerbating WCMP’s performance issues.

We seek to answer this question: *how to implement TE solutions with high precision given constrained hardware space?* Adjusting weight distributions of groups proves effective in lowering table space requirement. However, when we compared the groups retrieved from production switches with the TE solution, we found that the precision loss is often high. We analyzed the differences, which led to several insights on how to reduce the group table usage with lower precision loss: (1) Groups that serve significant traffic contribute disproportionately to the precision loss. They should be given more group table space. (2) Some groups have very skewed weight distributions and consume many table entries. Normal weight adjustment that preserves all paths struggles to avoid high precision loss. However, certain paths can be pruned from the distribution to significantly lower the table space requirement, often with little impact on precision loss. (3) After adjusting weights, some groups can become identical. Deduplicating such groups frees up group table space without incurring precision loss.

We develop three heuristics based on the insights: (1) traffic-aware resource allocation, (2) prune paths, and (3) deduplication. We then implement two algorithms that map TE solutions to switch hardware, applying the above heuristics. They both aim to minimize precision loss while adjusting group weights to meet the hardware resource constraint. Heuristic 2 is invoked conditionally when the resource constraint is otherwise impossible to satisfy. The two algorithms make different trade-offs: one has very low precision loss, while the other runs faster but with higher precision loss in some cases.

Since evaluation in production DCNs could impact user traffic, we built a data center TE evaluation framework named FabricEval that allows us to run controlled experiments using production-like DCN topologies and traffic loads. Our implementations of the algorithms are compared to the state-of-the-art approach [249] under both common and extreme DCN configurations. Results show that our design implements TE solutions with no more than 7% precision loss, while using the state-of-the-art approach can have up to 67% precision loss. Furthermore, our design achieves a 10 $\times$ execution speed improvement over the existing approach and successfully avoids traffic loss triggered by exhausted hardware resources.

In summary, this chapter makes the following contributions:

- We quantify the TE precision loss problem for a variety of DCN topologies, scale, and traffic patterns. Our study identifies five root causes of precision loss (§4.2).
- We present two algorithms with different speed and optimality trade-offs that can generate high precision TE implementations, at the same time complying with hardware resource constraints (§4.3).
- We develop a TE evaluation framework that mirrors Google’s production DCNs and TE system and use it to evaluate our design (§4.4).
- We release the source code of our algorithm implementation and the evaluation framework FabricEval [1].

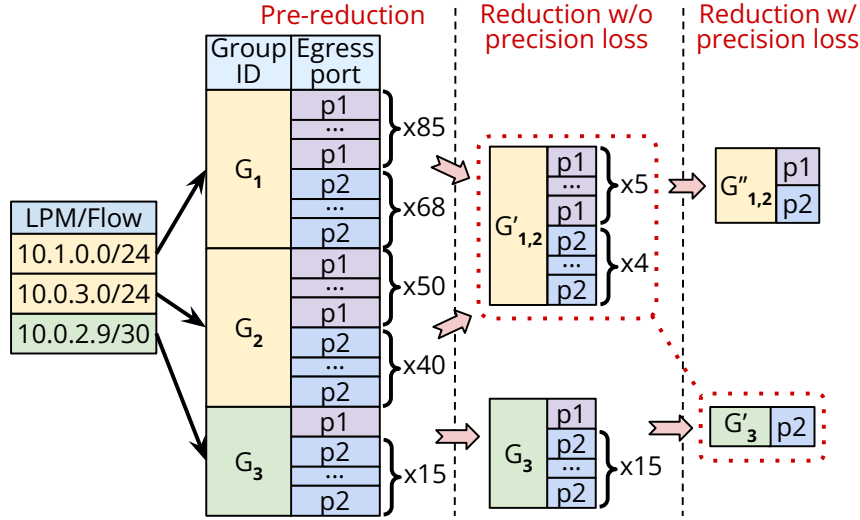


Figure 4.1: Hardware representation of flows/groups, and a demonstration of group reduction.

4.1 Background

We provide an overview of how data center TE systems are designed, as well as the hardware resource constraint these systems face. We then describe how the emerging spine-free DCN architecture differs from the traditional Clos-based design from a TE perspective.

4.1.1 Traffic engineering in data center networks

End hosts in DCNs are densely connected by many paths traversing different switches. The TE system acts as a centralized controller that manages all links/switches and routes traffic end-to-end via available paths. It looks for an optimal plan to fulfill a list of goals, including serving all traffic demands (bytes to send between each source-destination pair), maximizing flow throughput, balancing link loads etc. The TE problem is typically formulated as a multi-commodity flow (MCF) optimization problem. Though it is theoretically possible to obtain a TE solution by solving an MCF with comprehensive constraints, e.g., switch group table size limits, the scale of today's DCNs makes this approach prohibitively expensive. Common TE systems break the task into two steps: first generate a *TE solution* to the MCF problem without hardware constraints, and then map the TE solution to a data plane *TE implementation*. Some TE systems generate solutions hierarchically [60] for better scalability, but this does not impact the TE implementation step, the focus of this chapter.

The TE solution specifies for each demand which paths to use and how much traffic to place on each path. When paths diverge at a switch, the demand is split by a ratio (weight distribution) according to the routing decision. The TE system updates each switch along the paths to reflect the routing decision. To map the TE solution to a TE implementation, the end-to-end TE solution first needs to be broken down into switch-local routing decisions. Next, switch-local decisions are translated to switch rules, namely longest prefix match (LPM) entries and groups, and installed on switches using switch control plane APIs (e.g., OpenFlow/SDN [151]).

An LPM/flow entry points to a group that is used for packet forwarding (see the pre-reduction

Table 4.1: Switch hardware profile of an example deployment.

Switch generation	Port speed	Group table size
Gen. 1	40 Gbps	4096 entries
Gen. 2	100 Gbps	16384 entries
Gen. 3	200 Gbps	32768 entries

phase in Figure 4.1). A group consists of a set of ports, each with an integer weight reflecting the rounded fractional traffic volumes assigned to each path. Weight distributions are achieved via port/entry replication since all entries in a group are selected by uniform hashing with equal probability at runtime. When ECMP is used, each port in a group uses exactly one entry in the group table, which is very efficient. With WCMP, the entries a port uses is proportional to its weight in the group. It is easy to see that the number of entries needed depends strongly on the weight ratios, e.g., $G_1=\{p1:85;p2:68\}$ will require a lot of entries.

The group table space is a scarce resource. Table 4.1 lists the group table capacity of a few types of switches found in production DCNs. Given the large number of flows, and the use of WCMP in today’s DCNs, it is hard to avoid exhausting the group table. A group reduction algorithm is needed to reduce group sizes, the sum of entries used by each group, so that the TE solution fits in the group tables. The group reduction algorithm should generate a new weight distribution that closely approximates the original one. For example in Figure 4.1, groups G_1 - G_3 can be reduced to $G'_{1,2}$ and G'_3 with minimal precision loss, while lowering table usage from 259 to 10 entries (more details given in §4.2 and §4.3). The state-of-the-art group reduction algorithm is called *WCMP TableFitting* [249].

4.1.2 Spine-free data center networks

Until recently, DCNs have used a traditional Clos topology that connects clusters to a spine layer, as illustrated in Figure 4.2 (top left). Inside each cluster, a non-blocking tree topology connects a large number of hosts. In this chapter, we consider a three-stage cluster layout, as illustrated in Figure 4.2 (right). Stage 1 (S1) switches are also called ToR switches. Stage 2 (S2) and stage 3 (S3) switches are aggregation switches. The S2 and S3 switches in each cluster also form an aggregation block.

Clos DCNs can use all links leaving a cluster to reach another cluster (assuming no failure) because they belong to the same shortest path. For example, for cluster 1 to reach cluster N in the traditional Clos DCN in Figure 4.2, there is only one shortest path: ①→② through the spine. The spine is often the link speed bottleneck of all paths because of infrequent upgrades. TE in Clos DCNs is thus straightforward, with all links running at the same speed after speed auto-negotiation, flows are spread across links in the shortest path in the form of either ECMP or very simple weight distributions (e.g., due to asymmetric link striping or some out-of-service links). In a cluster with egress demands, each switch has one flow entry matching the aggregated IP prefix of each destination cluster. Every flow references an individual group that contains ports

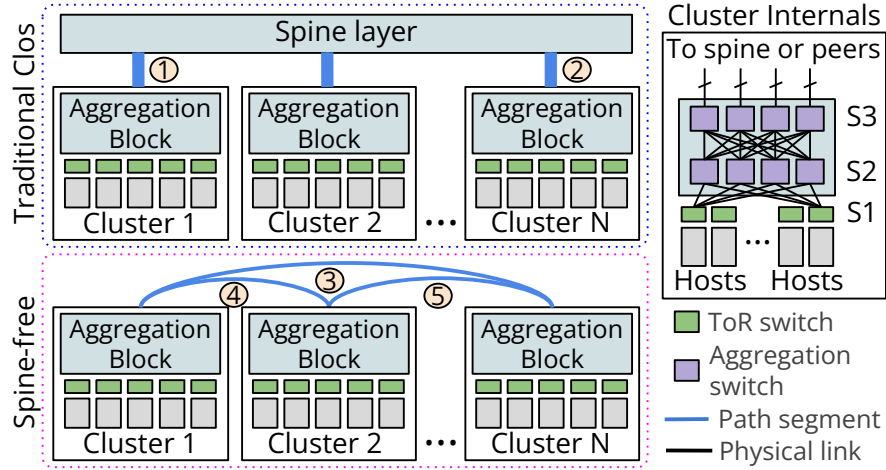


Figure 4.2: Traditional Clos and spine-free DCN topology. A path segment is an abstraction of parallel physical links.

used for reaching the next-stage switches, e.g., $S1 \rightarrow S2$ and so on¹. Groups on S3 switches contain ports to the spine.

The spine layer is expensive and requires frequent upgrades to keep up with the growing traffic demand. As Chapter 2 described in its spine-replacement use case, DCNs overcome this cost by moving to a spine-free topology enabled by OCS [216] that directly connects the clusters using a full mesh, as illustrated in Figure 4.2 (bottom left). Note that the intra-cluster topology and number of links coming out of a cluster remain the same.

TE in spine-free DCNs is different and more challenging to implement than in Clos DCNs in two ways. First, link speed differences hidden by the slow spine layer are now exposed. This translates to more skewed weight distributions in groups, which consume more hardware resources. Second, spine-free DCNs have far fewer direct-connect links in the shortest path between clusters. In order to support high traffic volumes, non-shortest-path forwarding must be adopted to utilize all links. In the spine-free DCN of Figure 4.2, cluster 1 can reach cluster N via the direct path ③, as well as multiple indirect paths such as ④ $\rightarrow$ ⑤ through cluster 2. However, graph theory shows that longer paths lead to higher bandwidth overhead and lower flow throughput [206]. Therefore, the length of non-shortest paths is limited to two hops, i.e., a flow can transit through at most one intermediate cluster to reach the destination. We term these two types of paths direct and indirect paths. In practice, TE avoids using all indirect paths simultaneously since this leads to a Valiant Load Balancing (VLB) scheme [245] with higher link loads, although VLB does reduce the group table footprint. The benefits of combining VLB with WCMP are left for future work.

For both direct and indirect paths, switches of the source cluster need to install flows and groups the same way as in Clos DCNs. We call these `src`-typed flows/groups. In addition, the intermediate cluster in an indirect path uses separate `transit`-typed flows/groups to forward traffic from source clusters to the final destination. Indirect-path traffic only traverses the aggregation block. `transit` flows are installed on all S2/S3 (but S1) switches to match on the

¹A small, constant number of static flows/groups are installed on each switch to handle intra-cluster and ingress (destination) traffic. They are not an important factor in the discussion of this chapter, hence ignored.

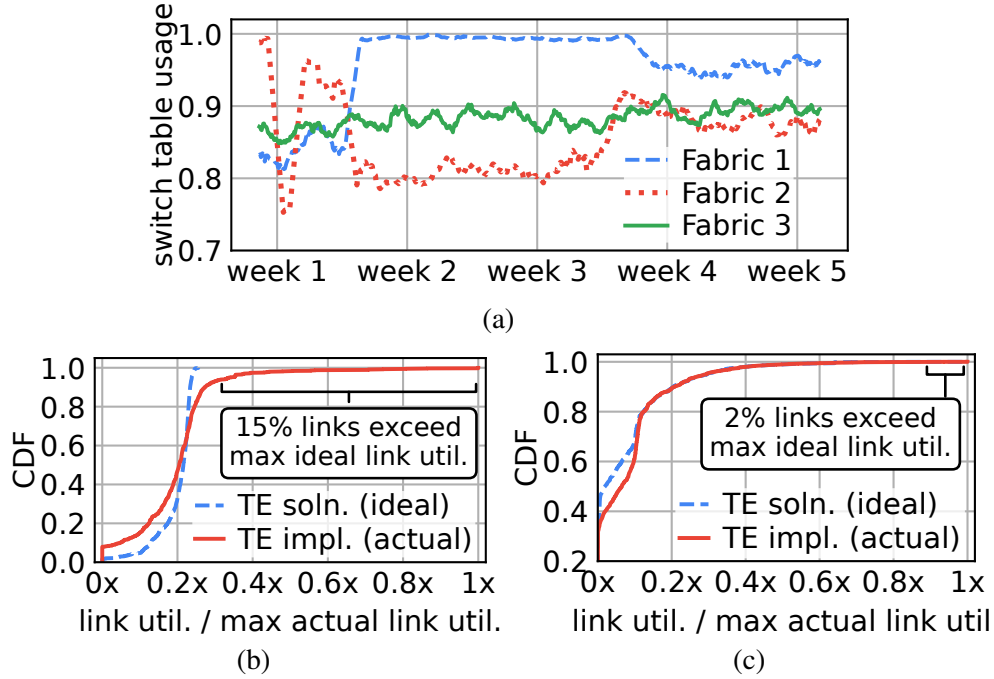


Figure 4.3: (a) 30-day time series of maximum group table usage in 3 arbitrary production fabrics. Usage is normalized to the table size of each switch. (b) Normalized link utilization CDF plotted from the TE solution and TE implementation in one of the spine-free fabrics. (c) Normalized link utilization CDF in a traditional Clos fabric of scale similar to the fabric in (b).

source and destination prefixes. Corresponding `transit` groups reflect indirect-path traffic in the direction of $S3 \rightarrow S2 \rightarrow S3$.

4.2 TE precision loss challenges

To illustrate the pressure on group tables, Figure 4.3a shows the group table utilization on Google’s production spine-free fabrics. We see that group table usage can be as high as 90–100%. However, high table usage does not directly represent the negative impact on traffic. The key network-level metrics are the traffic load and loss on each link. We observed that high table usage strongly correlates with high link utilization. For example, Figure 4.3b shows the actual link utilization in a spine-free fabric in which the TE implementation produces a maximum table utilization of 90%. We see that the TE implementation results in a much more imbalanced link utilization distribution compared to the (ideal) TE solution. Moreover, 15% of the links exceed the maximum link utilization. The worst few links see an actual utilization 5 times higher than that of the TE solution, resulting in congestion loss. Figure 4.3c shows that a Clos fabric of similar scale also experiences precision loss due to group table constraints, although the problem is less severe. We focus our discussion on a spine-free TE setting, though our findings also apply to Clos DCNs.

We compared a large number of TE solutions and their resulting switch implementations and identified five key challenges that exacerbate precision loss: (1) Large network scale leads to more groups sharing hardware resources; (2) TE solutions are generated without considering hardware constraints and often contain skewed weights that are difficult to reduce; (3) Heterogeneous DCN

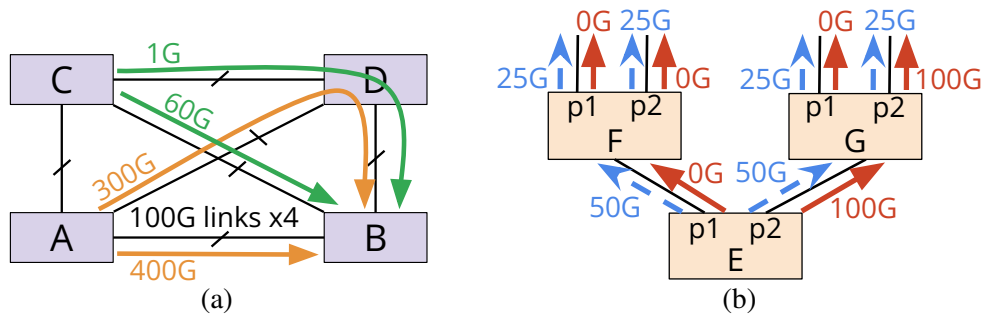


Figure 4.4: (a) Abstract view of a small spine-free network. (b) Cascading precision loss in a two-stage aggregation block. Desired traffic is in dashed blue, actual traffic is in solid red.

hardware subjects parts of the network to higher precision loss; (4) Diverse paths in spine-free TE result in larger group sizes; and (5) The cascading impact of errors in one stage to the next in multi-stage DCNs increases precision loss multiplicatively. We now explain these challenges in detail.

① Scalability challenge. The total number of flows/groups to install on a switch increases with the number of clusters because each cluster has demands for virtually all clusters. As the number of groups in a table increases, each group has to use fewer entries, resulting in higher precision loss overall.

Considering the Clos and spine-free DCNs in Figure 4.2, hosts in each cluster are usually assigned continuous IP addresses that aggregate into a single prefix. The number of distinct prefixes equals the number of clusters N . In Clos DCNs, each switch needs to support $N-1$ `src` flows/groups, one for each destination cluster. In spine-free DCNs, each switch not only has $N-1$ `src` groups, but also one `transit` group for each indirect path traversing it. In the worst case, a cluster can be used by all $(N-2)$ source clusters to reach $(N-1)$ destination clusters indirectly. This sums up to $(N-1)(N-2)$ `transit` groups on each aggregation switch.

② Skewed weights challenge. Since the TE system is not aware of any hardware resource constraint, the TE solution it generates may include highly skewed traffic distributions. Consider the example in Figure 4.4a, demand $C \rightarrow B$ has a total volume of 61G. The TE solution assigns 1G to path $C \rightarrow D \rightarrow B$ and 60G to path $C \rightarrow B$, a rather skewed split because path $C \rightarrow A \rightarrow B$ has no slack capacity left and at least two paths must be used for diversity. Aggregation blocks A, B, C, and D each has four physical S3 switches. The four switches in each block are connected pair-wise (via four links) to the four switches in each of the other blocks. As a result, the `src` group for demand $C \rightarrow B$ on the S3 switches in C has a weight distribution of 0.25 : 15, which is rounded to 1 : 15.

Demand $C \rightarrow B$ experiences precision loss for two reasons. First, rounding weights to integer as required by the hardware triggers precision loss. We could also use 1 : 60, but this increases the group size. Second, if we cannot afford 16 entries, we can reduce the larger weight while retaining all member ports in the group, which is a common group reduction behavior. However, this further increases precision loss and leads to oversub(scription) on link $C \rightarrow D$. In the worst case, 1 : 15 is reduced to 1 : 1 (ECMP), resulting in an oversub of $\frac{\text{actual volume}}{\text{desired volume}} = \frac{1/2 \cdot (0.25 + 15)}{0.25} = 30.5$ times more traffic on link $C \rightarrow D$. Alternatively, precision loss can be lowered if certain ports are removed from the group. We discuss this option as a heuristic in §4.3.

③ **Heterogeneity challenge.** Cloud data centers are expanded incrementally to deal with growing traffic demand. Newly installed clusters are always equipped with the latest generation switch hardware, which means different generations of hardware co-exist in the DCN. Switches differ in link speeds and group table sizes, as demonstrated in Table 4.1. This heterogeneity aggravates precision loss in two ways: (1) The smaller table size of older switches forces groups to be reduced more heavily; (2) Mixed link speeds make group sizes larger on newer generation switches.

Gen. 3 switches have bandwidth $5\times$ higher than Gen. 1 switches, but group table size is $8\times$ larger. This means newer switches can support more groups with less group reduction if flow/group counts scale with bandwidth; however, older switches may struggle to store the necessary state.

In spine-free DCNs, a switch can see multiple speeds across its ports since they may peer with switches of different generations. The TE system seeks to balance link loads by placing more traffic on links with higher speed. A switch with mixed link speeds (e.g. Gen 3 switches with connections to slower hardware) increases the skew of group weights. This is defined as the maximum-to-minimum weight ratio of a group. The larger the ratio, the more skewed a group is. Our experiments (§4.4.4) show that in a baseline spine-free topology, the average skew ratio on Gen. 1 switches, which don't suffer from this issue, is 484 and the maximum ratio is 1176. In contrast, for Gen. 3 switches, the average ratio is 1187 and the maximum ratio is 2676.

④ **Path diversity challenge.** Group pruning can result in a significant reduction of path diversity which may result in high link utilization when there are traffic shifts or link failures. [183] proposes a technique that accounts for potential demand shifts by imposing a minimum level of path diversity for each flow. We incorporate this technique as a path diversity constraint into our MCF formulation that generates TE solutions. With this constraint, TE solutions must use at least a certain percentage of the available paths to serve each demand. This percentage is referred to as the spread. While this constraint makes the TE solution more resilient, it degrades the solution optimality.

We compared the TE solutions generated by the MCF without and with a path diversity constraint for the same traffic load. Findings suggest that the standard formulation tends to use as few paths as possible. Many groups in the standard solution contain only one member port because the demand is small enough to fit onto one path. These groups use only one entry in the group table. Solution to the modified formulation uses several paths as the diversity constraint mandates. With a *spread*=50%, flows use more paths, which increases the pressure on the group table. As mentioned in the Heterogeneity challenge, larger groups will be reduced more heavily to fit into the same table space, hence causing high precision loss. We present a quantitative study in §4.4.3.

⑤ **Cascading precision loss challenge.** The multi-stage DCN topology has a cascading effect that amplifies precision loss multiplicatively as traffic flows from upstream switches to downstream switches.

To understand the cascading effect, consider the example in Figure 4.4b. Switches E, F and G form a two-stage topology inside an aggregation block. A demand of 100G needs to be sent out from switch F and G. The TE solution (illustrated in dashed blue) divides the demand uniformly on upstream (S2) switch E, i.e., it sends 50G to both F and G. Next, downstream (S3) switches F and G each uniformly split the traffic between their two ports. However, to conserve group table

entries, the TE implementation places all 100G on link $E \rightarrow G$ (shown in solid red) while switch F receives zero traffic. The oversub of link $E \rightarrow G$ is 2, while the undersub of link $E \rightarrow F$ is 0. This oversub/undersub is inherited by the downstream links on switches F and G. The two ports on switch F are undersubscribed with zero traffic, regardless of the groups installed on F. On the other hand, even if switch G implements a perfect ECMP group, port G-p1 and G-p2 will each carry 50G, which yields an oversub of 2. This is the baseline precision loss inherited from switch E.

This example only shows the cascading precision loss inside one aggregation block. For indirect paths, the aggregation block of intermediate clusters inherits the baseline precision loss from the upstream aggregation block in source clusters.

4.3 Design

We now present two algorithms that lower the TE precision loss compared to current group reduction solutions. Our algorithms, called Direct Mixed-Integer Reduction (DMIR) and Iterative Greedy Reduction (IGR), aim to convert a set of original input groups to weight-reduced output groups so that the sum of the group sizes does not exceed hardware limit. DMIR builds on top of mixed-integer programming (MIP) to approximate groups to the TE solution. Specifically, it uses an MIP solver (e.g., Gurobi [85]) to directly find the optimal weight assignment for each group. Since MIP-based solutions have exponential time complexity, we also develop IGR which greedily searches for the smallest-sized groups that best approximate the TE solution in polynomial time. IGR is similar to WCMP TableFitting but uses additional heuristics.

Both algorithms achieve group reduction by decreasing group weights relatively. They also use three heuristics to deal with challenging scenarios (§4.2) in the reduction process. We first describe the heuristics in §4.3.1 and then introduce the algorithms in §4.3.2 and §4.3.3.

4.3.1 Shared heuristics

The heuristics described next are used in both algorithms. The first two, Group Sharing and Group Pruning, directly reduce the number of entries used in the group table. The third heuristic, Table Carving, ensures that high-volume flows have sufficient group entries to accurately distribute traffic.

Group Sharing. Group table usage depends on both the number of groups and the number of entries per group. As the network size increases, the number of groups to install on a switch increases, requiring more hardware resources (see §4.2 Scalability challenge). This heuristic reduces the number of groups by eliminating duplicates. Two groups are identical if they use the same ports and weights for each port. While `src` and `transit` flows/groups must be kept isolated by a design requirement, flows of the same type can share the same group entries to distribute their traffic. This optimization does not incur any penalty, since it does not change the traffic distribution. For example, G_1 and G_2 in Figure 4.1 can be reduced to the same $G'_{1,2}$, which is shared by the two flows.

We find that while `src` groups generally differ, `transit` groups often become identical after group reduction. For instance, the TE solution in Figure 4.4a leverages both direct and

indirect paths to fulfill demands $A \rightarrow B$ and $C \rightarrow B$. Two `transit` groups are installed on switches of aggregation block D, one for path $C \rightarrow D \rightarrow B$ and one for path $A \rightarrow D \rightarrow B$. Since $D \rightarrow B$ is a common segment, the same ports are used in both `transit` groups. The two groups only differ in pre-reduction port weights. Considering any S3 switch in D, it only has one port connected to B, as explained in the Skewed Weights challenge. The `transit` group for $C \rightarrow D \rightarrow B$ places 0.25G on this port while the `transit` group for $A \rightarrow D \rightarrow B$ puts 75G on it. Both groups can be reduced to single-port ECMP groups with weight 1, so they become identical. Generally speaking, `transit` groups for indirect paths with a common segment are identical post-reduction.

In production DCNs, groups are reduced batch by batch as the TE system generates new groups in response to the ever-changing network conditions. Since Group Sharing itself does not incur TE precision loss, it should always be invoked to conclude a round of reduction. The saved table space can subsequently benefit the next batch.

Group Pruning. Sometimes not all groups in the TE solution fit in the table, even if they are reduced to ECMP. In this case, some groups are not installed, and their corresponding traffic is dropped by the switch. Barring this heuristic, group reduction algorithms do not proactively reduce groups beyond ECMP because they aim to preserve the path diversity. But as a last resort to avoid dropping traffic, our algorithms invoke this Group Pruning heuristic that prunes ports from groups.

The key question is which port to prune. There exist multiple victim selection policies, e.g., prune the first port or a random port. The strategies are easy to implement, but they are not necessarily good choices. For example, a port carrying 90% of the group traffic might be pruned, impacting precision loss significantly. Our pruning policy considers both group size reduction (which impacts the path diversity of the group) and the increase in precision loss. Similar to WCMP TableFitting, our goal is to limit the maximum port oversub in a group. For IGR, we prune the port that results in the least increase in maximum oversub across all member ports, i.e., the port with the smallest weight (traffic volume). For example, p_1 in G_3 of Figure 4.1 can be pruned, after which the size of pruned G'_3 becomes one, while the remaining p_2 is merely oversubscribed by $1.016\times$. As discussed later in §4.3.2, the best port to prune in DMIR is not always the one with the smallest weight, but rather the port that yields the optimal objective.

We expect Group Pruning to be especially useful in addressing the Skewed Weights and Heterogeneity challenges. The reason is that both challenges lead to small weights on some ports, which can be pruned with minimal impact on precision loss. Our evaluation in §4.4.8 shows that Group Pruning has very little impact on path diversity.

Table Carving. This heuristic addresses the Path Diversity and Cascading Precision Loss challenges. Groups are forced to use more egress ports to satisfy the path diversity constraint. This means groups used by flows with a low traffic volume may have a lot of non-zero weights and large sizes, even though their contribution to the overall precision loss is more limited than high-volume groups. We develop Table Carving to ensure high-volume groups are prioritized over others and can receive sufficient resources during group reduction.

Table Carving allocates table entries exclusively to each group. A minimum number of entries are allocated per-group to avoid that low-volume groups end up with too few entries and little path diversity. Remaining entries are allocated to groups in proportion to their traffic volume. Note, a group will receive an allocation at least equal to the number of entries required by its ECMP form. Some low-volume groups are likely to be reduced to ECMP (or even get pruned) if they are

Algorithm 1 DMIR($\{G_i\}, T$), $1 \leq i \leq n$. $\{G_i\}$ is a set of `src` or `transit` groups. T is the available table space.

```

1: // Step 1: Table Carving.
2: for  $i = 1$  to  $n$  by 1 do
3:    $T_i = \text{MAX}(\text{len}(G_i), \left\lfloor \frac{\text{SUM}(G_i)}{\sum_i \text{SUM}(G_i)} \cdot (T - \sum_i \text{len}(G_i)) \right\rfloor)$ 
4: // Step 2: single-group optimization.
5:  $\{G'_i\} = \{G_i\}$ 
6: for  $i = 1$  to  $n$  by 1 do
7:    $G'_i = \text{SINGLEGROUPMIP}(G_i, T_i)$  // Equation 4.1.
8: // Step 3: reclaim and redistribution.
9: for  $G'_i = G'_1, \dots, G'_n \in \text{SORT}(\{G'_i\})$  do
10:    $\text{unused} = \text{RECLAIMUNUSED}(\{G'_i\})$ 
11:   if  $\text{unused} > 0$  then
12:      $G'_i = \text{SINGLEGROUPMIP}(G_i, T_i + \text{unused})$ 
13: // Step 4: Group Sharing.
14: return  $\text{DEDUP}(\{G'_i\})$ 

```

only allocated ECMP-sized entries. Nevertheless, the minimum allocation is a trade-off between protecting high-volume groups and avoiding starving low-volume groups.

While $G'_{1,2}$ in Figure 4.1 can be pruned to $G''_{1,2}$ to further reduce table usage without hurting path diversity, reducing G_3 to G'_3 is preferred because it carries less traffic. In addition to the Path Diversity challenge, this heuristic also helps address Cascading Precision Loss. Protecting the high-volume groups in the upstream switches through Table Carving effectively limits the baseline precision loss on the downstream switches.

4.3.2 Direct mixed-integer reduction

DMIR is a parallel group reduction algorithm. For a given set of input groups, DMIR performs a reduction on each group individually while ensuring the overall resource constraint is not violated. Algorithm 1 presents the structure of DMIR. Reduction happens in four steps: (1) The Table Carving heuristic allocates table space T to each group (line 2-3). (2) DMIR instantiates a number of single-group MIP optimization problems (described below) and solves them in parallel to get final reduced groups (line 5-7). (3) Table entries unused by reduced groups are reclaimed (line 10) and redistributed to other groups, potentially allowing precision improvements over that of the original allocation (line 12). (4) Final groups are deduplicated using the Group Sharing heuristic (line 14).

The objective of single-group MIP optimization is to minimize the difference between an original group G and the reduced group G' such that G' uses no more than its allocated resources. Equation 4.1 is the single-group MIP formulation. w_i and w'_i are port weights of groups G and G' . p is the number of ports on the switch. T_G is the allocated table entries for group G' . The formulation treats G and G' as two p -dimensional vectors such that $G = (w_1, w_2, \dots, w_p)$ and $G' = (w'_1, w'_2, \dots, w'_p)$. This brings G and G' into the same vector space $\mathbb{R}^p$ for convenient

Algorithm 2 IGR($\{G_i\}, T$), $1 \leq i \leq n$.

```

1: // Step 1: Table Carving, see Algorithm 1
2: // Step 2: single-group reduction.
3:  $\theta = 1.00$  // oversub limit.
4:  $\{G'_i\} = \text{SORT}(\{G_i\})$  // sort by descending group size.
5: while  $\text{SUM}(\{G'_i\}) > T$  or  $\text{SUM}(G'_i) > T_i, \exists i$  do
6:   for  $i = 1$  to  $n$  by 1 do
7:      $G'_i = \text{REDUCE\_SINGLE\_GROUP}(\text{SORT}(G_i), T_i, \theta)$ 
8:     if  $\text{SUM}(\{G'_i\}) \leq T$  and  $\text{SUM}(G'_i) \leq T_i, \forall i$  then
9:       return  $\text{DEDUP}(\{G'_i\})$ 
10:    $\theta += 0.05$ 
11:    $\text{PRUNE\_GROUP\_IF\_STUCK}(G_i\_not\_fit)$ 
12: // Step 3: Group Sharing.
13: return  $\text{DEDUP}(\{G'_i\})$ 
14: function  $\text{REDUCE\_SINGLE\_GROUP}(G = \{w_i\}, T_i, \theta)$ 
15:    $G'^* = G' = \{w'_i\} = (1 \dots 1)$  //  $G'^*$ : optimal  $G'$ .
16:    $oversub_{min} = \infty$ 
17:   for  $w'_1 = 1$  to  $wmax = \left\lceil \frac{w_1 \cdot T_i}{\text{SUM}(G)} \right\rceil$  by  $\left\lceil \frac{wmax}{ITER} \right\rceil$  do
18:     for  $i = 2$  to  $p$  by 1 do
19:        $w'_i = \left\lceil \frac{w'_1}{w_1} \cdot w_i \right\rceil$ 
20:        $oversub = \text{MAX\_PORT\_OVERSUB}(G, G')$ 
21:       if  $oversub < oversub_{min}$  then
22:          $\text{UPDATE}(oversub_{min} = oversub, G'^* = G')$ 
23:       if  $\text{SUM}(G') > T_i$  or  $oversub_{min} \leq \theta$  then
24:         break
25:   return  $G'^*$ 

```

comparison. Note that this formulation also organically integrates Group Pruning with the common path-diversity-preserving reduction behavior by allowing w'_i to be 0.

$$\begin{aligned}
& \underset{\text{minimize}}{\sum_{i=1}^p \left| \frac{w_i}{\sum_{i=1}^p w_i} - \frac{w'_i}{\sum_{i=1}^p w'_i} \right|} \\
& \text{s.t.} \quad \sum_{i=1}^p w'_i \leq T_G \\
& \quad \quad w'_i \in \mathbb{Z}^+, \forall i \in \{1, \dots, p\}
\end{aligned} \tag{4.1}$$

The difference between G and G' can be measured by various metrics, such as cosine similarity, L1-norm, Kullback-Leibler divergence etc. We choose L1-norm $\|G - G'\|_1$ since finding the best G' is a combinatorial optimization problem and L1-norm's linearity makes it computationally tractable. This metric is different from that used by WCMP TableFitting and IGR: restricting the maximum port oversub of a group. While restricting maximum port oversub is an intuitive

objective, since the overall precision loss is bounded as long as no port exceeds the oversub limit, subjecting all ports to the same oversub limit does not always produce the optimal group reduction outcome (as seen in the Skewed Weights challenge).

In the third step, DMIR collects the unused entries from each group after reduction. This is in order to make further improvements on some groups by rerunning reduction with extra hardware resources. Some groups have unused entries because these entries do not significantly improve the solution quality, so we reclaim them for other groups as follows. After the initial reduction, we sort groups by their objective metric (line 9). The group with the largest difference from its original group is ranked on top. All the unused entries are allocated to the top group, and group reduction is rerun with the larger space limit. It is possible that the top group only consumes a few or none of the unused entries. If so, unused entries are reclaimed again and redistributed to the next group in line. This process repeats until either all groups are reduced again or all previously unused entries have been used.

An alternative to the per-group single-group MIP is to directly solve all groups in a single monolithic formulation. Unlike Equation 4.1, this formulation combines all the single-group reduction problems into one monolithic optimization that can be solved with a single call to the MIP solver. Since the monolithic formulation needs a single optimization objective, we construct a weighted sum of L1-norm based on each group’s individual L1-norm, where the weights are the total traffic volume carried by each group. The intuition is that groups with more traffic contribute more to the overall precision loss metric, so the MIP solver should spend more effort optimizing those groups. Equation 4.2 shows the comprehensive formulation.

$$\begin{aligned}
& \text{minimize} && \sum_{i=1}^n \left(\sum_{j=1}^p w_{ij} \right) \cdot \sum_{j=1}^p \left| \frac{w_{ij}}{\sum_{j=1}^p w_{ij}} - \frac{w'_{ij}}{\sum_{j=1}^p w'_{ij}} \right| \\
& \text{s.t.} && \sum_{i=1}^n \sum_{j=1}^p w'_{ij} \leq T \\
& && w'_{ij} \in \mathbb{Z}^+, \forall i \in \{1, \dots, n\}, j \in \{1, \dots, p\}
\end{aligned} \tag{4.2}$$

$G_i = (w_{i1}, w_{i2}, \dots, w_{ip})$ is the original group i , w_{ij} is the original weight on port j of group i . Similarly, w'_{ij} is the reduced weight on port j of group i . T is the total available table space on the target switch. This formulation only enforces the sum of group sizes across all reduced groups to meet the table space limit; there is no per-group space limit and the Table Carving heuristic is not invoked.

The monolithic formulation is nevertheless infeasible in practice. With tens of thousands of decision variables and constraints, we fail to find a solution for a large-scale production DCN even after running the latest version of the Gurobi solver for days. With a one-hour timeout on Gurobi, the resulting solution is considerably worse than that obtained from single-group MIP.

4.3.3 Iterative greedy reduction

IGR has two objectives: a primary one to keep reduced group sizes under resource limit, and a secondary one to restrict the maximum port oversub in each group to an upper bound. Algorithm 2 describes the three-step structure of IGR. Step 1 is the same Table Carving heuristic of Algorithm 1.

In step 2, IGR iteratively reduces all groups (line 5-11) until both conditions are met: (1) sum of all group sizes meets the table limit T , and (2) each group size meets its allocated space limit T_i . Each group is reduced individually by the REDUCESINGLEGROUP function. IGR is greedy because reduction could terminate in the middle of an iteration once the two conditions are met (line 8-9), leaving some groups reduced more than the rest. Some groups may fail to meet the upper bound θ on port oversub, especially when θ is very tight in the initial iterations. IGR relaxes θ in each iteration (line 10) by a constant step size. A more relaxed θ enables REDUCESINGLEGROUP to reduce the group size more. If the reduced group size stops decreasing after θ has been relaxed for $thresh=3$ consecutive times, we consider the reduction “stuck”. $thresh$ is a tunable parameter that allows us to balance between aggressive reduction and more port oversub tolerance. When the reduction is stuck, the Group Pruning heuristic is invoked (line 11).

REDUCESINGLEGROUP starts the search for a final group from the original group’s ECMP form (line 15). It locks the relative weight ratio between ports in the group under reduction G' . In each iteration, ports in G' are sorted by their weights in descending order (line 7). Once the assigned weight of the first port w'_1 is determined, all the remaining port weights are determined based on the relative weight ratio (line 18-19). The number of iterations performed on each group is limited to a small constant $ITER$. REDUCESINGLEGROUP conducts the search in increments of $\lceil \frac{w_{max}}{ITER} \rceil$ instead of the smallest increment of one (line 17), to ensure the search process finishes within $ITER$ iterations. This effectively mitigates the long convergence time (see §4.4.6) that iterative-reduction-style algorithms suffer from, since there are many groups (Scalability challenge) and many ports per group (Path Diversity challenge) to iterate through. REDUCESINGLEGROUP makes the best effort to meet the oversub limit and returns the smallest group it has found when it hits the group size limit (line 21- 24).

4.4 Evaluation

In §4.4.1, we describe FabricEval, a data center TE evaluation framework that we created. The DMIR and IGR algorithms are compared to WCMP TableFitting under a wide range of network conditions in §4.4.3-§4.4.5. We also present group reduction speed as another metric in §4.4.6. §4.4.7 investigates the application layer impact. §4.4.8 inspects the side effect of Group Pruning. §4.4.9 discusses the contribution of each heuristic.

4.4.1 FabricEval evaluation framework

FabricEval is a network-level data center TE evaluation framework. It models the entire data center TE pipeline end-to-end: from taking snapshots of network state and traffic demands to generating TE solutions, group reduction and finally implementing the solution on data plane. FabricEval employs a modular design that allows us to plug in different TE solving algorithms, group reduction algorithms, and switch hardware models with different configurations. This enables us to compute both the ideal link utilization expected by the TE solution and the actual link utilization from different group reduction algorithms, on all links. It also tracks table usage and traffic loss due to missing forwarding entries from switches.

In FabricEval, topology inputs are represented as Protobuf [163]. Traffic demands are represented as plain matrices. Inputs are passed to a TE solver module, which by default runs the same TE algorithm used in our production fabrics. The generated TE solution is organized by cluster and forwarded to a group reduction module of each cluster. Finally, reduced groups are installed on the corresponding switches. FabricEval is implemented in 15,820 lines of Python code. Since FabricEval is not a packet-level simulator, we use *ns-3* [171] to study application layer metrics.

Open source releases. The source code of FabricEval, DMIR, IGR and our implementation of WCMP TableFitting are released online [1].

4.4.2 Experimental setup

All configurations used in the evaluation are developed from a baseline production-like configuration that includes (1) a 65-cluster heterogeneous spine-free fabric, (2) a path diversity spread of 50%, and (3) a traffic matrix (TM) representing the total egress/ingress demands between ToR pairs. Due to security and privacy reasons, we cannot publish most results from production fabrics and traffic traces. However, we carefully tuned FabricEval to match production results. Using FabricEval also avoids the difficulty of running controlled experiments on live fabrics. All experiments in this chapter are conducted on a Dell PowerEdge R720 server, with two 20-core Intel Xeon E5-2680 CPUs, 128 GB of memory and 1.2 TB of disk.

Network topology. The baseline fabric features 2.6K switches and 83.2K links. Each cluster in the fabric has 32 S1 switches. Each S1 has 32 ports, 8 facing the S2 switches and 24 facing the host machines. The aggregation block in each cluster has 4 S2 and 4 S3 switches interconnected as a nonblocking FatTree; each has 128 ports, 64 facing up and 64 facing down. The 64 up-facing ports on S3 are evenly spread to 64 peer S3 switches. The S2 switches connect to each S1 with 2 links.

Traffic matrix. The baseline TM captures properties found in production traffic traces. Clusters have unequal egress and ingress demands. Storage clusters, for example, typically have a small ingress but a large egress because applications often read more data than they write. The demands follow an exponential distribution. The traffic volume between a cluster pair is characterized by the Gravity model [183, 192, 215, 243]. A cluster’s total egress/ingress volume is distributed proportionally among peer clusters, weighed by each peer’s total volume. Incremental data center expansion could leave a few clusters deployed but having zero demand. They can still be used by other clusters for transit.

4.4.3 Spread and path diversity

The Path Diversity challenge in §4.2 focuses on balancing resilience to traffic changes and optimal efficiency. This trade-off depends on the spread. To determine an appropriate spread for our evaluation, we run production trace using realistic routing settings (i.e., TM fed into TE comprises peak demands of the past 24 hours and TE is recomputed every hour) on a large spine-free production fabric. Figure 4.5a shows the link utilization for this realistic scenario, normalized relative to the optimal TE solution computed against the instantaneous demand (i.e., assuming traffic measurement and TE solution programming have zero delay). We find that 50% spread achieves link utilization closest to OPT ($1.14 \times \text{OPT}$) because higher spreads rely on costly indirect

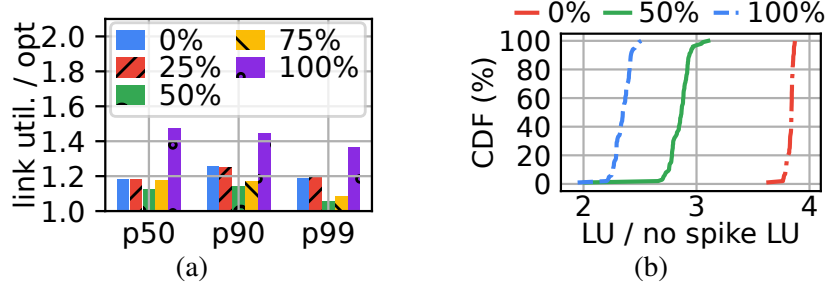


Figure 4.5: (a) 50% spread achieves best link utilization using production traffic trace and routing settings. (b) larger spreads are more robust against traffic spikes.

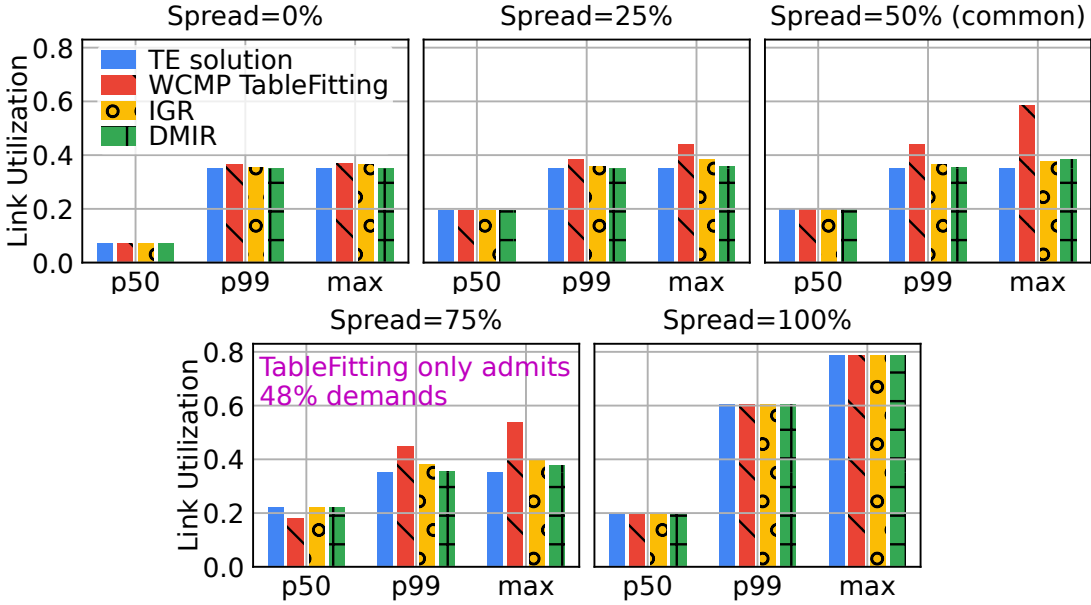


Figure 4.6: Median and tail link utilization with different path diversity.

paths and lower spreads are fragile to TM changes. To understand how spread values affect TE solution's robustness, we further study more extreme cases, such as when 25% of the original demands spike by $4\times$. We define robustness as the link utilization post-spike divided by the pre-spike link utilization for the same spread. Figure 4.5b shows this ratio for each link in the same production fabric for 0% (no) spread, 50% and 100% (full) spread. As expected, 100% spread is most robust since spikes are spread throughout the network. 50% spread is significantly more robust than 0%. Overall, 50% spread proves to be a reasonable trade-off between TE optimality and robustness.

Figure 4.6 compares the link utilization of IGR and DMIR with that of the TE solution (assuming no table size constraint) and WCMP TableFitting (TF) for five spread values. The results are collected on the baseline fabric and TM. In each graph, the y-axis shows the link utilization of the 50th (p50) and 99th (p99) percentile, and the maximum (max) link in the network. We see that with 0% spread, all algorithms come close to the link utilization of the TE solution, which is not surprising since most flows have a group size of one. As the spread increases, the link utilization difference between the TE solution and the group reduction algorithms increases.

Table 4.2: Summary of various fabrics used in FabricEval.

Name	Topology	# Gen1	# Gen2	# Gen3	Link failure
Fabric 1	spine-free	22	22	21	none
Fabric 2	spine-free	11	11	11	none
Fabric 3	spine-free	65	0	0	none
Fabric 4	spine-free	22	22	21	1%
Fabric 5	Clos	21	21	22	1%
Fabric 6	random graph	22	22	21	none

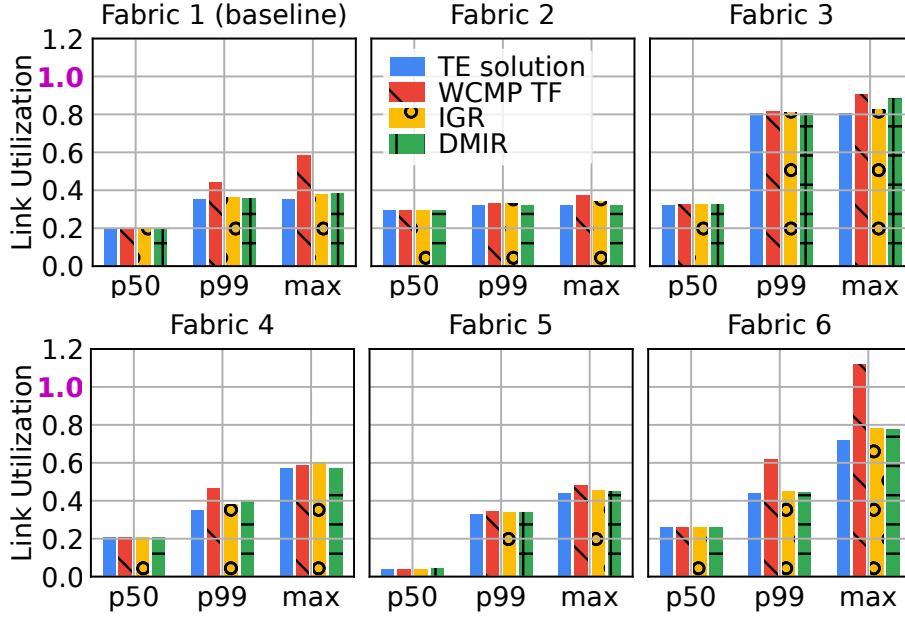


Figure 4.7: Link utilization distribution in various network fabrics.

With a 50% spread, WCMP TF’s link utilization is 25.4% higher at p99 and 67.1% higher at max. DMIR sees link utilization 1.6% higher at p99 and 9.1% higher at max. IGR increases over the TE solution by 4% and 7.4%, respectively. The result at 75% spread largely resembles that at 50% spread, except for a seemingly counter-intuitive link utilization improvement from WCMP TF².

As the spread further increases, link utilization continues to deteriorate due to increased pressure on the group tables. At 100% spread, all algorithms achieve a link utilization equal to the TE solution. Since 100% spread forces demands to spread across all paths, groups become identical and after deduplication, they easily fit in the table with minimum precision loss.

4.4.4 Heterogeneous fabrics

In order to understand how different fabric configurations affect TE precision, we construct a list of fabrics that differ in scale, hardware and/or topology, as described by Table 4.2. Fabric 1 is

²WCMP TF’s result at 75% spread does not reflect a true improvement. The reason is that the groups generated by WCMP TF do not fit in the switch table. The overflow groups cause the associated traffic being dropped. This benefits the groups that do fit.

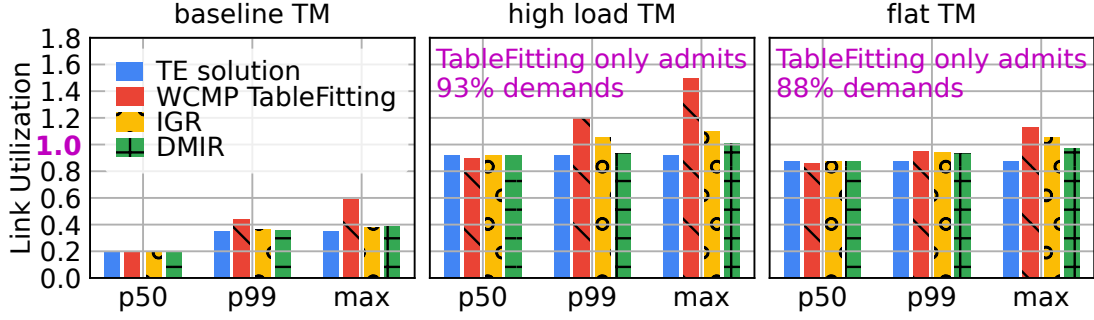


Figure 4.8: Link utilization distribution with different traffic patterns.

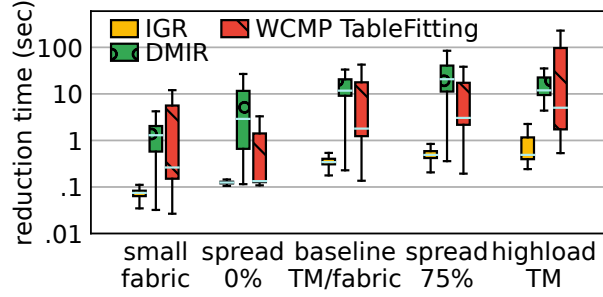


Figure 4.9: Comparison of group reduction speed.

the baseline fabric. Fabric 6 is a Jellyfish-like fabric with random graph topology [206] using the same hardware as fabric 1.

Figure 4.7 shows how results differ across these fabrics. Compared to fabric 1, all three algorithms in fabric 2 and 3 operate close to the TE solution in terms of both median and tail utilization. This confirms our analysis in §4.2 that Scalability and Heterogeneity both have an impact on precision loss. The tail link utilization in fabric 3 is noticeably higher than other fabrics of same scale because the same TM is applied to a fabric with lower capacity. Note that DMIR’s maximum link utilization in fabric 3 is 10% higher than the TE solution, while WCMP TF and IGR are 12.4% and 3% higher, respectively. This is caused by a 120-second timeout on DMIR. Due to the nature of MIP problems, we use a timeout to ensure DMIR terminates within a duration comparable to the other two algorithms, which can lead to a suboptimal solution. Without this timeout, DMIR can still find an optimal solution within 10 minutes in all the experiments we run.

Looking at fabric 1 and 4, we find that the impact of failure is mostly on tail link utilization. The maximum link utilization under failure has increased by up to 63% over that without failure. Among the three algorithms, DMIR and IGR are more affected than WCMP TF. Compared to fabric 1, precision loss in fabric 5 is moderate, as WCMP TF is 9% higher than the TE solution, yet IGR and DMIR are still effective in bringing this down to 1.3%. This aligns with our production observation that precision loss in Clos fabrics is not as grave as spine-free fabrics. Precision loss in fabric 6 is high. This is due to the asymmetry and imbalance introduced from random link assignment. WCMP TF has a maximum link utilization 55.1% higher than the TE solution, while IGR and DMIR lower this gap to 8.4% and 7.5%. Generally speaking, IGR and DMIR are more effective when the topology is irregular.

4.4.5 Traffic pattern

The baseline TM has an average cluster ingress/egress volume that is about 40% of the cluster's bisection bandwidth. 15% of the latest generation clusters are configured to be in expansion mode with empty demands. For comparison, we created two more TMs: a high load TM and a flat TM. The high load TM is generated in the same way as the baseline TM, except that its average cluster volume is raised to 70% of the bisection bandwidth and all clusters have non-empty demands. The flat TM does not follow the Gravity model. Instead, each cluster has an equal ingress and egress demand of 7.68 Tbps (240 Gbps per ToR) that is uniformly spread across all peer clusters. There are no empty clusters in the flat TM. We run all three TMs on the baseline fabric with 50% spread.

With the high load and flat TMs, WCMP TF results in traffic loss due to group table overflow, as shown in Figure 4.8. As much as 7% and 12% of the total demands are dropped, respectively. In the high load TM scenario, even with a fraction of groups rejected, WCMP TF increases the actual link utilization over the TE solution by up to 62.1%. On the other hand, DMIR increases actual link utilization by only 9.4% over the TE solution at max, and IGR increases by 19%. This result is consistent with the increase found in the baseline TM scenario. The flat TM results are slightly better: WCMP TF has a maximum link utilization 28.4% higher than the TE solution, while it is 11.1% for DMIR and 20.2% for IGR.

All three algorithms will result in congestion loss due to oversub on some links. For example, in the high load TM scenario, WCMP TF's actual maximum link utilization is 1.49, almost 50% over the physical link capacity, while that of DMIR and IGR are 1.007, and 1.096. This is however expected given that the maximum link utilization of the TE solution is over 0.9. Any error in group reduction could lead to traffic exceeding the link capacity. While the average cluster volume is configured to 70% of the bisection bandwidth, link utilization of the TE solution ends up much higher. We attribute this to the blocking property of the spine-free topology.

4.4.6 Execution speed

The execution speed of the IGR and DMIR algorithms is important since they are used online. Each switch runs group reduction twice, first on the `transit` groups and then on the `src` groups. We measure the total time to complete both reductions on each switch. A number of representative configurations are selected to cover the common and corner cases. They include: (1) fabric 2 (small-scale) in §4.4.4, (2) 0% spread in §4.4.3, (3) the baseline fabric/TM in §4.4.5, (4) 75% spread in §4.4.3, and (5) the high load TM in §4.4.5.

Figure 4.9 plots the distribution of per-switch group reduction time as standard boxplots with max/min, first/third quartile and median. As can be seen, the reduction time on each switch spans a wide range: there is about two orders of magnitude difference. The reduction time differs significantly across switches as well as between different group types. On switches with smaller tables it takes longer to fit the groups since there is a tighter table size constraint. The `transit` groups are easier to reduce than `src` groups because they are mainly ECMP-like groups. If a cluster has a zero demand, there is no `src` group to be installed on the switches in this cluster. The execution time is then determined solely by `transit` reduction.

DMIR and WCMP TF have overlapping ranges. In some scenarios, such as 0% and 75% spread, DMIR is slower than WCMP TF, while in other cases, they are on par. IGR is the fastest

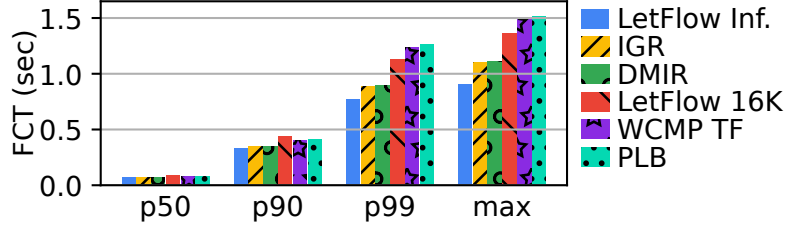


Figure 4.10: *ns-3* FCT.

among all three. It generally completes group reduction in under a second. Overall, IGR is $17\text{--}42\times$ faster than DMIR and $10\times$ faster than WCMP TF by comparing the median reduction time.

4.4.7 Impact on flow completion time

Thus far, we have shown our approach is effective at meeting TE’s link utilization objectives. We now explore the impact our approach has on flow completion time (FCT). We extend our FCT comparison to include flowlet-based and host-based schemes. The traffic trace is constructed by sampling the flow sizes and start times from published production measurement studies [23, 117, 136, 193, 242] while ensuring that the total flow size of each cluster adds up to its demand in the TM.

In contrast to ECMP/WCMP, TE approaches such as [9, 121, 122, 218] balance load at the flowlet level [116]. While this finer grain load balancing could improve FCT, most large-scale DCN operators adopt some form of ECMP/WCMP [2, 80, 159, 199, 204] instead of a flowlet-based approach. We attribute this to the universal vendor support for ECMP/WCMP and the simplicity of deployment, e.g., no parameter tuning is required. We focus on LetFlow [218], a recent flowlet proposal that provides significant FCT performance gains.

We implemented IGR, DMIR, WCMP TF and LetFlow in *ns-3*, and use the baseline fabric/TM. As per [218], each switch uses a flowlet table that maps a 5-tuple hash to an egress port id. ECMP groups are still required to track ports that can reach the destination for each flow aggregate. When the inactive interval in a flow exceeds the pre-configured timeout, the flowlet entry expires and a new egress is randomly selected from the corresponding group. The best timeout value is picked by sweeping through a range of timeout values. We assume the flowlet table is subject to hardware resource constraints similar to WCMP. A 32K-entry memory is split in half, 16K for groups and 16K for flowlets³.

Figure 4.10 shows that IGR and DMIR reduce the p99 FCT of WCMP TF by 28.1% and 27.8%, respectively. LetFlow with an unlimited table size outperforms all other candidates, shortening the p99 FCT of WCMP TF by 37.8%. With a 16K table size constraint, LetFlow underperforms IGR and DMIR but remains better than WCMP TF by 8.7% in p99 FCT. To put this improvement into perspective, we run flows of two sizes in two production fabrics, then we steadily increase average link utilization of the fabric, the p50/p99 FCTs at every link load are collected. Figure 4.11 illustrates how FCT normalized to each flow’s min RTT correlates with

³Cisco ACI fabric with Nexus 9300 switches supports flowlets under the feature named Dynamic Load Balancing [42]. A total of 4096×8 -way ECMP entries (see Table 7 in [43]) are available on chip.

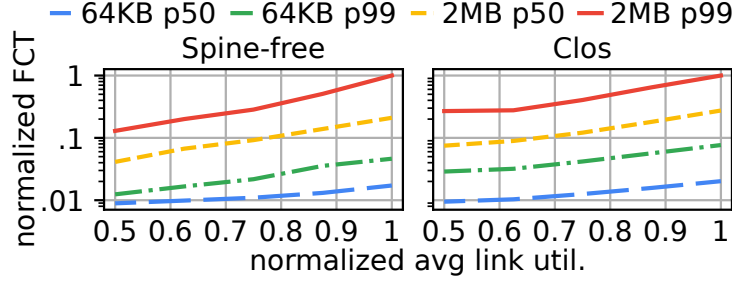


Figure 4.11: Production FCT vs. link load.

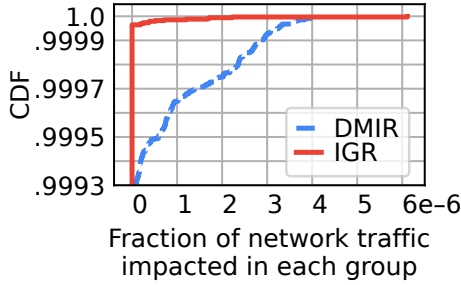


Figure 4.12: Impact of reduced path diversity.

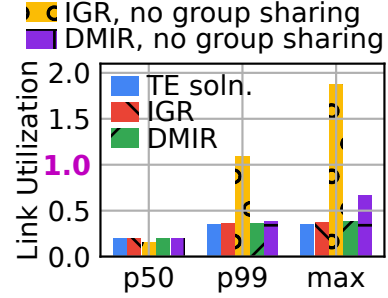


Figure 4.13: Impact of Group Sharing.

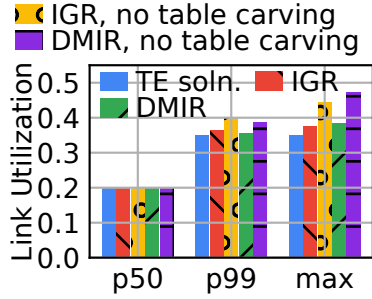


Figure 4.14: Impact of Table Carving.

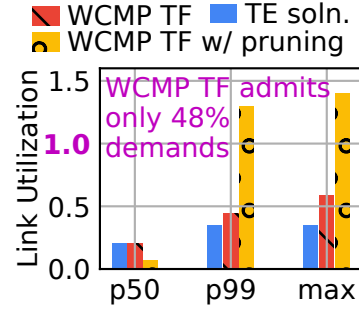


Figure 4.15: Impact of Group Pruning.

link utilization. On the more loaded side of the spectrum, a 10% reduction in link utilization can translate to a 20–40% reduction in p99 FCT.

Another class of designs that has reported significant FCT gains is host-based TE (e.g., PLB [185]). [185] shows that PLB can fully correct load imbalance caused by ECMP between two links with 1:2 capacity difference. However, with capacity difference of 1:100, PLB can only correct it to 1:3.5, and p99 FCT is $2.3\times$ higher than that of a load balanced scenario. Figure 4.10 indicates that PLB alone achieves FCT on par with WCMP TF and we hold the same view as [185] that in-network WCMP and host-based PLB are complementary in performance benefits.

4.4.8 Group Pruning’s impact on path diversity

A potential side effect of Group Pruning is that heavily pruned groups might lose a lot of paths. This reduced path diversity could impact the network’s resilience to link failures. To understand

the impact of pruning, we record the paths/ports used by all groups in the baseline fabric/TM experiment. Figure 4.12 shows that IGR prunes less than 0.01% of the total groups, while DMIR prunes 0.07% of them. DMIR’s pruning decision is more aggressive than IGR: 48 groups (out of 1350624) are pruned down to a single port. In contrast, groups pruned by IGR only lose 1-3 ports.

Figure 4.12 also shows the fraction of traffic impacted by Group Pruning. Impacted means that the traffic is steered away from pruned paths, but is not dropped. Despite DMIR’s more aggressive pruning, the total traffic impacted (area under curve) is lower than IGR. We attribute this to DMIR’s trade-off between path diversity and lower traffic impact.

Overall, the small number of aggressively pruned groups and tiny fraction of traffic impacted suggests that the negative impact of Group Pruning is minor.

4.4.9 Ablation study

Now, we evaluate how much each of the heuristics in §4.3 contributes to the overall precision improvements. We use the baseline fabric/TM and 50% spread. First, the contribution of Group Sharing is evaluated by disabling this heuristic in both DMIR and IGR. Figure 4.13 illustrates the significant impact of Group Sharing on IGR: $5\times$ decrease in maximum link utilization. DMIR is improved by $1.7\times$. Additionally, Group Sharing reduces the $O(N^2)$ `transit` group entries (e.g., 162K) to no more than 64 (see §4.4.11 for details). Next, the contribution of Table Carving is demonstrated in Figure 4.14. Instead of standard Table Carving, the algorithms are modified to carve the table in constant sizes: each group receives an equal share of the table space. IGR and DMIR with standard Table Carving perform $1.2\times$ and $1.24\times$ better than the constant version. Finally, we look at the contribution of Group Pruning in Figure 4.15. 75% spread is used to reflect how Group Pruning avoids traffic loss. The pruning logic is hard to completely disable in both algorithms, so instead, we add Group Pruning to WCMP TE. With Group Pruning, WCMP TE can avoid group installation failures and the consequent traffic loss entirely, at the cost of a $2.4\times$ increase in link utilization. However, this increase is specific to certain scenarios and can be offset by other improvements.

4.4.10 Quantization error

Rounding fractional weights in groups to integers introduces a minor yet acceptable quantization error. We choose to express weights in Mbps so that the quantization error of rounding is at most ± 1 Mbps. Weights in Kbps or bps can further reduce quantization error, but it comes at a cost where the Protobuf is bloated with larger integers. Figure 4.16 shows that the quantization error of Mbps granularity is sufficiently small.

We repeat the baseline fabric/TM experiment using three group reduction algorithms, WCMP TableFitting, DMIR and IGR, with unlimited group table space on each switch. In other words, the original fractional weights in each group only need to be rounded to integers, but will not be reduced in size, as there is infinite group table space to store any arbitrary number of entries. The precision loss over the TE solution from each group reduction algorithm with unlimited table space reflects the pure impact of rounding.

For instance, WCMP TableFitting is significantly improved with unlimited table space. The TE implementation is almost identical to the TE solution. Per-link utilization delta over the TE

Table 4.3: Group table space usage on sampled S1/S2/S3 switches by different group types.

Stage	Group type	Original usage	Post-reduction
S1	src	8 entries	8 entries
S2	src	1746624 entries	64 entries
S3	src	2241913 entries	18285 entries
S1	transit	0 entry	0 entry
S2	transit	162112 entries	64 entries
S3	transit	792738 entries	60 entries

solution is also reduced to close to zero, which means there is very minimum oversub and the implementation overall is accurate. Similarly, for DMIR and IGR, the implementation is also improved, although the improvement is not as large as WCMP TableFitting. The TE precision loss (i.e., quantization error) achieved by all three group reduction algorithms is generally under 0.01%.

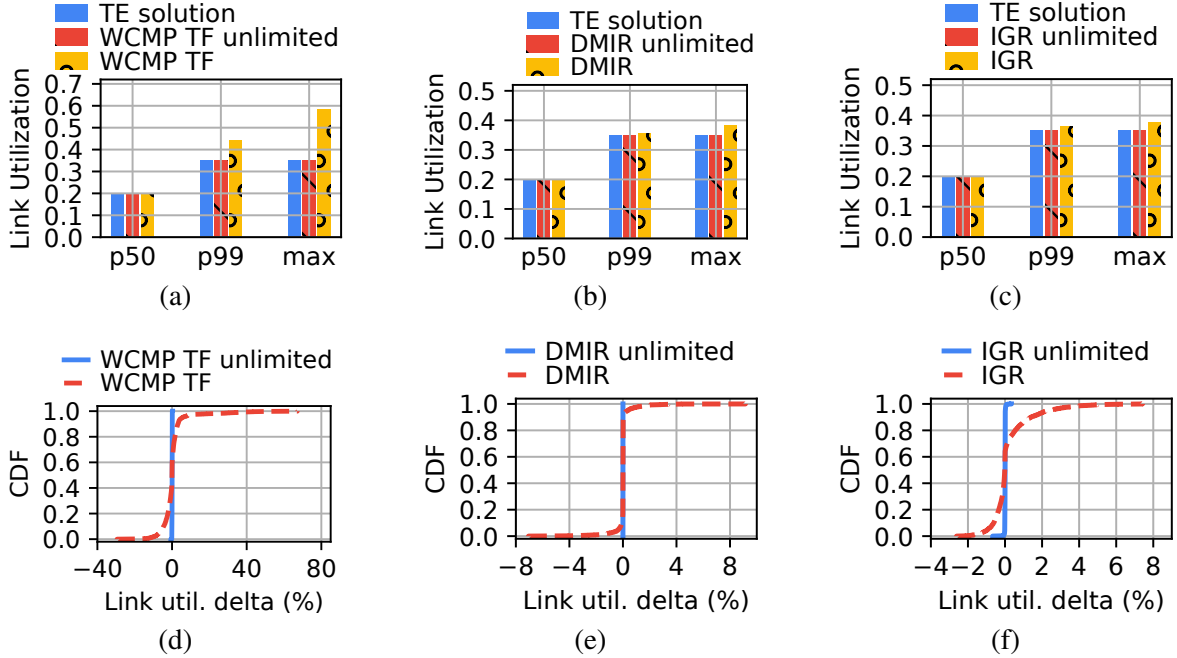


Figure 4.16: (a, b, c) Median and tail link utilization if running original group reduction algorithm vs. running it with infinite group table size, so that there is only precision loss caused by rounding weights. (d, e, f) Per-link utilization delta over the TE solution.

4.4.11 Table usage

Table 4.3 lists the group table usage statistics from sampled S1/S2/S3 switches in the baseline fabric from §4.4.4. The original usage column indicates the number of entries required if the table space is unlimited (pre-group-reduction). The post-reduction column indicates the actual number of entries consumed found on the switch (after running group reduction). We can see

that table usage on S1 switches is low and group reduction is irrelevant here. This is because the groups are always ECMP on S1 switches and a shared group is used for serving all demands. As mentioned in §4.3, the Group Sharing heuristic significantly reduces the number of entries required by `transit` groups to no more than 64, on both S2 and S3 switches. The `src` groups on S3 switches see a table usage reduction from 2241913 to 18285, a $122\times$ reduction.

4.4.12 Recommendation

While both our algorithms perform well, DMIR generally has lower precision loss than IGR. Though its advantage is usually small, some extreme scenarios can benefit from DMIR. For example, in the high load TM scenario (§4.4.5), DMIR is the only option that avoids link utilization above 1.0, except for a slight violation on the worst link. In contrast, IGR is always faster than DMIR, so it responds better to traffic load changes. Because of that, we recommend generally using IGR, except for difficult scenarios where low precision loss matters.

4.5 Related work

Data center TE. Many DCN designs [2, 4, 5, 80] leverage flow-level ECMP [101] to achieve failure resilience and efficient bandwidth utilization. However, ECMP is known to perform poorly with asymmetry and heterogeneity. WCMP [249] improves over ECMP. Niagara [118] distributes traffic using flow rules instead of group rules. DASH [103] splits traffic via comparison-based hash space partition. It avoids WCMP-style entry replication but requires arithmetic operation and P4 [29] support. Our work assumes fixed-function switches but also applies to programmable switches.

Flare [116], CONGA [9], HULA [121], Clove [122], LetFlow [218], and Contra [102] leverage the elasticity of flowlets to balance traffic load across paths. In networks with different path latencies [37, 56, 143, 157, 168, 225, 227], one has to carefully select the flowlet detection timeout to avoid reordering. Flowlets are also susceptible to traffic characteristics, e.g., [147, 207] find that RDMA traffic [16, 71, 83, 94, 123, 159, 250] exhibits weak flowlet pattern. As described in LetFlow [218], flowlet switching requires switch support [42], which is not as widely available as ECMP. There are also works like DRILL [75] and RPS [48] that operate at packet-level.

Host-based TE approaches like Presto [93], Hermes [241], MPTCP [187], FlowBender [115], RePaC [244] and PLB [185] assume either no in-network support or merely ECMP. PLB [185] shows that our work complements host-based TE.

Spine-free data center. Dragonfly [125], Dragonfly+ [200], Slim Fly [25], Slingshot [44], Aquila [76] have studied direct-connect topology networks. However, they primarily focus on HPC networks. Meanwhile, most deployed DCNs [2, 80] employ a Clos topology with a spine. Harsh et al. [91] discusses ECMP-based TE in spine-free data centers. Sirius [19] and RotorNet [153] are spine-free but spread traffic using extended VLB [245]. Jupiter [183] explores advanced TE in spine-free DCNs. We believe our work is among the first to improve TE precision loss in spine-free DCNs.

Wide area network TE. Wide area network TE faces similar switch hardware constraints. SWAN [99] allocates traffic demands to a set of tunnels, which are limited by the number of

supported switch rules. B4 TE [100] generates groups that consume $14\times$ the switch table space. It has to decouple traffic splitting rules across two stages in its Clos fabric to reduce per-switch table consumption. Meta’s Express Backbone [161] and Edge Fabric [196] run centralized TE on BGP and ECMP. We found no available data on their TE precision loss.

4.6 Conclusion

Precision loss is an inherent problem when implementing TE with limited switch hardware resources and the shift towards spine-free topologies exacerbates the problem. We introduce two group reduction algorithms that offer different trade-offs in terms of precision and execution time. Both algorithms use heuristics to address challenges, such as the hardware-agnostic nature of TE algorithms, switch heterogeneity, and path diversity. Our evaluation shows they achieve significant improvements in various network scenarios compared to the current solution.

More broadly, this chapter argues that the two halves of TE, computing a routing plan and realizing it in switch hardware, cannot be designed in isolation. The conventional pipeline treats the optimizer’s fractional solution as the answer and hardware installation as a mechanical afterthought, but on a spine-free reconfigurable fabric the finite group table is exactly where an optimal plan can fail to land: a solution that cannot be installed delivers none of the utilization it promises. Treating optimization and hardware realization as a single coupled problem is therefore what lets the network deliver the bandwidth a TE solution intends rather than a degraded approximation of it. This coupling only tightens as fabrics reconfigure more often, since every reconfiguration re-derives the routes that must then be squeezed back into the same limited tables.

Seen against the thesis’s arc, this chapter’s lesson is that the heterogeneity a reconfigurable fabric exposes cannot be left to the hardware to absorb. Fixed-function switches handle uneven, skewed traffic splits inefficiently, spending table entries lavishly on the very distributions a spine-free topology makes common, so that heterogeneity must be managed explicitly in software, by the same TE system that computes the routes. Where the transport of Chapter 3 rode a fixed schedule and this chapter made the fabric demand-aware, both still serve a general-purpose network carrying arbitrary traffic. The final redesign abandons that generality: it specializes the fabric to a single, known workload, machine-learning training, whose communication pattern is fixed and known before the job even starts, and moves the redesign one more layer up the stack, from how traffic is routed to where the job is placed.

Chapter 5

Machine Learning Job Placement

The previous two chapters redesigned a general-purpose reconfigurable network from below, first its transport and then its traffic engineering, while leaving it able to carry any traffic. This chapter reaches the specialized end of the spectrum of Chapter 2: an ML cluster whose optical fabric is reconfigured for each job. Here both of the assumptions the earlier chapters had to defend, stable connectivity and uniform bandwidth, give way to a single opportunity. Because a training job’s communication pattern is fixed the moment its parallelization strategy is chosen, the scheduler knows in advance which accelerators must exchange data and how much, and can co-adapt the job’s shape and the cluster’s topology, shaping each to the other. Reconfigurability is thus no longer only a property of the fabric, as it was for the transport and TE of Chapters 3 and 4, but a lever the scheduler wields to place jobs the fabric could not otherwise host. This is the third and most specialized of the redesigns the thesis statement calls for, and the layer under the most pressure is no longer transport or routing but job placement itself.

To see why placement carries that pressure, consider how such a cluster is used. An ML cluster is a pool of connected accelerators, also known as XPU’s, typically shared by many concurrent jobs. Because these clusters require substantial investment, there is strong interest in their efficient operation, which depends on supporting as many jobs as possible while minimizing the contention between them. Adding a job involves two sub-tasks, admission and placement. An admission policy determines whether to admit a job and how many XPU’s to allocate, while the placement policy identifies the most suitable set of XPU’s to use. Existing ML schedulers [82, 133, 149, 164, 170, 179, 208, 235, 247] primarily focus on admission, since placement is not a significant concern in Clos-topology GPU clusters. However, placement is an increasingly important problem as cluster designs move away from Clos topologies. In such settings, suboptimal placement often leads to severe resource underutilization and degraded job performance.

Recently, ML clusters based on torus topologies have been deployed in production [15, 113]. Unlike switched topologies, torus-based clusters are constructed with directly connected XPU’s. This allows them to scale easily with minimal rewiring. Torus clusters also have lower construction costs compared to switched topologies while supporting the neighbor-centric communication patterns of typical ML workloads. On the other hand, only neighboring XPU’s can communicate with non-blocking, full bandwidth. Data transmission between more distant XPU’s must route through intermediate XPU’s, leading to competition for link bandwidth and making a job’s performance sensitive to its placement in the torus cluster.

Two main approaches have been proposed to handle job placement in torus clusters. The first employs a FirstFit algorithm (or its variants) [41] to search for a sufficient number of contiguous XPU, and assigns a job to the first location discovered. The second approach attempts to find XPU that are close to each other on a best-effort basis [130, 146], without guaranteeing contiguity. Both approaches suffer from significant drawbacks.

FirstFit often leads to resource fragmentation and low utilization in multi-tenant torus clusters. This issue arises because users submit jobs of varying sizes, which arrive and complete asynchronously. In addition, the communication patterns inherent to ML jobs also demand XPU allocations that conform to specific rectangular or cuboid shapes (§5.1). This shape constraint inevitably strands idle XPU between allocations. Consequently, a new job might not be scheduled even if sufficient XPU are available, simply because of non-contiguity. If no placement is found, the job must be queued until resources free up. The queueing delay can be pronounced since ML jobs typically last for long periods.

In contrast, best-effort placement, despite its higher utilization of cluster XPU, may assign spatially scattered XPU to a job. The resulting network contention is an especially prominent problem for ML workloads, since their constituent XPU frequently exchange immense volumes of data. This contention prolongs job completion times (JCT).

We set out to create a more flexible cluster resource allocation, guided by two key insights. The first insight is that ML job communication patterns are regular and rearrangeable. A job’s shape can be mapped to the torus topology in multiple contiguous ways without compromising performance. This insight inspires folding, a technique that exhaustively evaluates all feasible shape mappings and identifies the one that minimizes fragmentation. Our second insight reveals that adapting the topology to a job shape can significantly improve the chance of successful allocation. This can be accomplished using OCSes [47, 217], the reconfigurable fabric of Chapter 2, to reconfigure the topology at runtime.

We explore the possibility of combining job shape alteration (folding) and topology alteration (reconfiguration) through the design of RFold, a new resource allocation scheme that aims to achieve high resource utilization while avoiding network contention. RFold attempts all legitimate shape variations for a given job, those that can limit network contention, by formulating shapes as graphs, and then invokes graph libraries to check for homomorphism. For each shape variation, it generates an allocation plan by virtually reconfiguring the underlying topology to best match the shape. Finally, RFold ranks these plans and commits to one that minimizes fragmentation.

To evaluate RFold, we developed a topology-aware, job-level simulator that simulates large-scale 3D torus clusters with reconfigurable OCS links. We also extend public ML job traces, such as the Microsoft Philly trace [109], based on our observations in a university-level cluster. We compare RFold to the shape-aware FirstFit placement policy and to four shape-unaware policies that place a job wherever XPU are free. Our results show that RFold delivers the shortest average JCT at every load we sweep and sustains the highest load before saturating, with its knee near 58% offered load under FIFO admission against 54% for the best shape-unaware policy and 44% for FirstFit, and that it converts more of the cluster into finished work than any baseline, plateauing near 57% effective utilization against 52% and 45% respectively. These findings show a strong potential for performance improvements through placement, topology design, and reconfigurability.

5.1 Background

5.1.1 ML communication pattern

The placement problem this chapter addresses is dictated by how ML training jobs communicate, so we begin there. A training job’s computation is parallel only within a single step; between steps, the participating XPU’s must reconcile their partial results, and it is this recurring exchange, rather than the arithmetic itself, that most often bounds how fast a job runs. How the work is partitioned across XPU’s therefore fixes both which XPU’s must talk to one another and how much data they move, and so determines what a good placement looks like.

ML training jobs often run on multiple XPU’s to reduce running time. Three parallelization strategies are commonly used to carry out distributed ML training: (1) data parallelism (DP), (2) tensor parallelism (TP), and (3) pipeline parallelism (PP). If the mini-batch of input data is too large to fit in the memory of a single XPU, DP shards it across multiple XPU’s, with model replica per XPU. If the model itself is too large, TP and PP shard it across multiple XPU’s. Many tools [111, 226, 248, 251] can determine the optimal combination of parallelization strategies for each ML job and how many XPU’s to use for each dimension of parallelism. The resulting plan can be expressed as a job “shape” requirement. For example, a job with a $4 \times 6 \times 1$ shape signifies that its model is sharded into six parts across six XPU’s (six-way TP), and the model itself is replicated four times (four-way DP). This configuration results in two independent communication phases. One phase synchronizes the calculated model gradients across four DP-dimension XPU’s corresponding to each model shard using six parallel ring-based AllReduce collective communication operations [172, 197]. The other phase synchronizes the six TP-dimension XPU’s of each model replica with four parallel AllReduce operations. The DP and TP parallelism are orthogonal to each other, and their communication patterns are isolated to the participating XPU’s. Similarly, a $18 \times 1 \times 1$ shape indicates DP-only, and $4 \times 4 \times 4$ denotes DP+TP+PP.

Each parallelism dimension induces a communication pattern of its own, and the three differ sharply in frequency and volume. TP is the most demanding: every layer must combine its sharded activations through a collective such as AllReduce or an AllGather paired with a ReduceScatter, so TP traffic recurs many times within a single training step and dominates the bandwidth budget, which is why TP degrees are kept small and confined to the most tightly coupled XPU’s. DP is coarser. Replicas synchronize their gradients through a single AllReduce at the end of each step, moving a volume proportional to the model size but amortized over the whole step and thus more tolerant of latency. PP is the lightest. Adjacent pipeline stages exchange only activations in the forward pass and gradients in the backward pass, as point-to-point transfers between neighboring stages rather than collectives.

Crucially, each of these collectives is realized as a ring over the XPU’s of a single parallelism dimension [172, 197]: an XPU sends only to the next member of its ring and receives only from the previous one. Communication is therefore near-neighbor by construction, and it stays cheap only when ring neighbors are also physical neighbors. When a job’s cuboid shape is embedded into the torus so that each parallelism dimension aligns with a torus axis, every ring closes over direct links and runs at full, non-blocking bandwidth. When the shape is scattered or misaligned, the same ring traffic must detour through intermediate XPU’s and compete for their links, inflating communication time even though the computation is unchanged. This sensitivity of

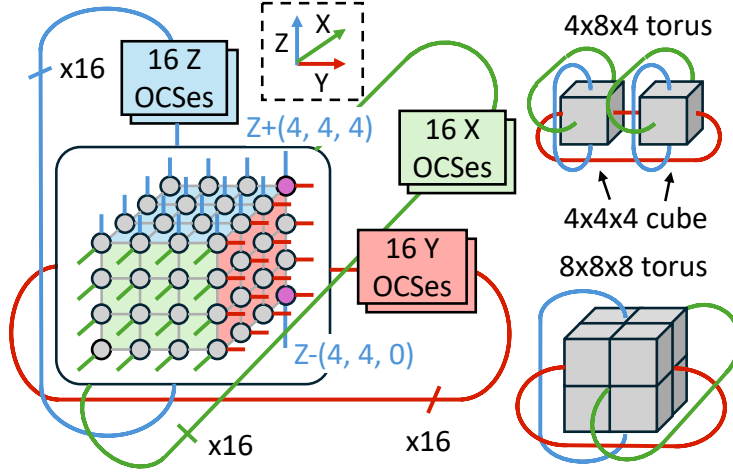


Figure 5.1: An 3D torus cluster (Google TPU v4) built from $4 \times 4 \times 4$ reconfigurable cubes.

communication to layout is what makes placement, not merely admission, the decisive scheduling problem on a torus.

The three strategies above are not the only ones in use. Mixture-of-Experts (MoE) models have grown increasingly popular, and they add a fourth dimension, expert parallelism (EP), whose communication does not fit the near-neighbor mold. In EP, each token is routed at runtime by a gating function to one or more experts, so it may be dispatched to any XPU rather than to a fixed neighbor. The resulting collective is an all-to-all, in which every XPU exchanges data with an arbitrary, runtime-determined set of peers, which at first glance seems incompatible with a torus whose strength is near-neighbor traffic. In practice an all-to-all can still be realized on a torus in one of two ways. The first maps it onto a one-dimensional ring, exactly as AllReduce is mapped; this is not latency-optimal, but for training, where the all-to-all overlaps with computation and does not sit on a tight critical path, the extra latency is rarely a problem. The second spreads a single all-to-all across a two-dimensional plane of the torus [129], raising the achievable bandwidth and lowering latency by treating the EP dimension as two torus axes rather than one. Under this second mapping a job that combines data and expert parallelism still presents a cuboid, three-dimensional shape, its DP degree along one axis and its EP degree factored across the other two, and so falls within the same shape-based placement framework as the DP+TP+PP jobs above.

5.1.2 Reconfigurable torus

A torus interconnect wires the XPUs directly to one another rather than through a separate switching fabric. Each XPU connects to its immediate neighbor along every dimension, and wrap-around links close each row into a ring, so an n -dimensional torus is a stack of interlocking rings. Neighboring XPUs communicate at full, non-blocking bandwidth over these direct links, whereas a pair that is not adjacent must forward its traffic hop by hop through the XPUs in between, conventionally along dimension-order routes [165]. The ring collectives of the previous subsection map onto this structure exactly. A ring laid along a torus dimension uses only direct

neighbor links and requires no forwarding at all.

Since ML workloads heavily rely on ring-based communications, many existing and proposed system designs [14, 15, 66, 78, 98, 113, 114, 128, 140, 219] adopt a mesh or torus topology to match their communication pattern. The bandwidth demands of each parallelism differ, with TP being the most bandwidth-intensive and PP the least. Torus system designs provision for and ensure uniform worst-case bandwidth. Among these, earlier designs such as the Google TPU v2/v3 [114] and Amazon Trainium1 [14] clusters were based on 2D torus. As cluster scale expands, 3D torus becomes advantageous, as it offers lower network diameter and higher bisection bandwidth than 2D torus.

This directness is what makes a torus economical. By dispensing with the spine of switches a Clos uses to guarantee uniform any-to-any bandwidth, a torus is far cheaper to build and wire and scales with little rewiring, but it provides high bandwidth only between neighbors. For a general-purpose workload that trade would be unattractive; for ML workloads it is a good match, precisely because the traffic is near-neighbor to begin with. The match holds, however, only when jobs are placed so that their rings align with the physical rings of the torus, which is what turns layout into a first-class concern.

A notable constraint of statically wired 3D tori is the immutable nature of their topology, diameter, and dimensional sizes. In response, the Google TPU v4 cluster [113, 254], the per-job reconfiguration use case of Chapter 2, breaks a large 4096-XPU static torus into 64 hardwired cubes and introduces OCSes to dynamically reconfigure the links between the cubes. This not only improves availability but also proves to enable considerable flexibility for job placement.

As Figure 5.1 shows, each cube contains $4 \times 4 \times 4$ XPU, with 6 ports per XPU. The 16 ports on the opposite faces are connected to the same group of 16 OCSes. More specifically, two opposing ports at the same position are connected to the same OCS. For example, we label each XPU in a cube with an (X, Y, Z) coordinate, and the two Z-dimension ports of an XPU are named Z+ and Z-. Port Z+ of (4, 4, 4) and port Z- of (4, 4, 0) are connected to the same Z-dimension OCS. All 64 cubes are connected to these three groups of OCSes. OCSes allow the face ports of each XPU to either connect to the opposite face port of the same cube and form wrap-around links, or connect to the corresponding face port of another cube to form larger shapes, e.g., $4 \times 8 \times 4$ and $8 \times 8 \times 8$ tori.

Reconfiguration thus grants the operator a limited but powerful freedom: the same pool of cubes can be wired into different torus shapes and sizes to suit the jobs currently running, and a job that would not fit the standing topology can sometimes be accommodated by rewiring around it. That freedom is bounded, however, by the granularity of a cube and by the port count of the OCSes, which caps how many cubes a single optical group can join. These limits, rather than the abstract ideal of an arbitrarily reconfigurable fabric, define the space in which the placement algorithm of §5.3 must operate.

5.1.3 Job scheduling

What is commonly referred to as job scheduling in fact comprises two distinct decisions, namely admission, which chooses whether to start a job and how many XPU to grant it, and placement, which chooses which XPU it receives. On a Clos cluster the big-switch abstraction makes the second decision almost irrelevant, since any set of XPU enjoys the same bandwidth to any other,

and the scheduling literature has accordingly concentrated on admission. On a torus the two swap in importance. As the previous subsections showed, the same allocation can run efficiently or badly depending only on where it lands, so placement becomes the harder and more consequential problem, and it is the one this chapter takes up. We discuss the two in turn.

On the admission side, a scheduler decides when to start each job and how many XPU's to grant it. Common policies range from first-come-first-served queues to backfill [167], which lets later jobs jump ahead to fill idle XPU's, shortest-job-first ordering [21], and more elaborate schemes that elastically resize jobs or preempt them for fairness or faster completion. Admission is not the focus of this chapter. We treat it as a knob that sits above placement and study it only insofar as it interacts with placement.

Placement is where a torus differs most from a Clos cluster, and it is the focus of this chapter. Placement policies fall into two broad families. Topology-unaware policies ignore the network structure and hand a job any set of free XPU's, wherever they happen to be. They are simple and keep utilization high, since almost any free nodes will do, but they scatter a job's ranks across the fabric, so its collective traffic contends on shared links and the job slows down. Topology-aware policies instead place a job on a compact, well-aligned region so that communicating ranks stay close and contention stays low. They protect performance but are far more restrictive: when no contiguous region of the right shape is free, the job cannot be placed even though enough XPU's sit idle elsewhere, which strands capacity and makes jobs wait. This tension, contention on one side and fragmentation on the other, is the central difficulty this chapter confronts.

The schedulers deployed in industry sit at different points on this spectrum. Some are topology-unaware, such as Kubernetes, while some HPC schedulers are more topology-aware and support 3D torus allocation. None of them, however, treats the topology itself as adjustable, which is the third option RFold exploits. We defer a detailed comparison to §5.5.

5.2 Motivation

The question we seek to answer is how to achieve both contention-free job performance and efficient resource utilization in torus clusters. Can we leverage ML job characteristics and OCS reconfigurability to place jobs intelligently, and how? A persistent obstacle is the mismatch between a job's logical shape and the physical topology. Logical job shapes always assume rings (at any granularity), but a statically wired physical topology cannot always provide them.

To better understand the impact of suboptimal placement, we ran experiments on a TPU v2 machine in Google Cloud. We place a two-TPU ML job first on a row (ideal) of a 2×2 grid, and then on the diagonal line (non-ideal). The job's communication time in the diagonal configuration is 17% longer than that on a row. We then placed two identical ML jobs, each assigned 2 TPU's, on the two diagonal lines of the grid, forcing them to compete for a common link. This resulted in a 35% longer communication time compared to when a single job ran in the same diagonal configuration. When one job's communication load was doubled (or tripled), the other measured job's communication time escalated by 95% (and 186%), respectively. These results show that the degree of performance degradation caused by suboptimal placement is unpredictable and can be very significant. The culprit is dimension-order routing. A job whose XPU's are spread across non-adjacent positions must forward its collective traffic through intermediate nodes, so every link

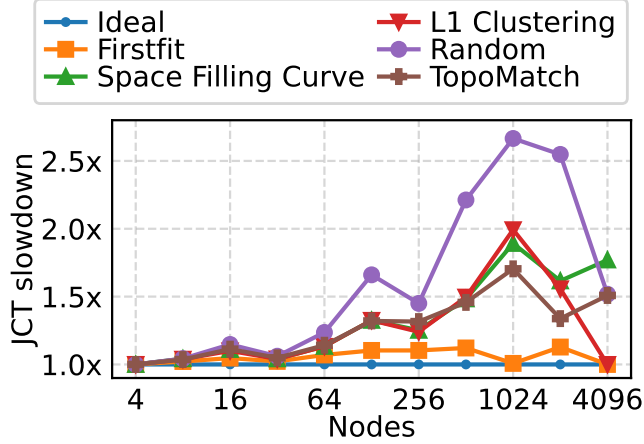


Figure 5.2: Static contention of placed jobs. Baseline is where jobs have no background contention at all. Space-filling curve (SFC) is the default placement policy being used in the SLURM and ALPS schedulers.

it shares with another job becomes a point of contention whose cost compounds as the competing load grows. Crucially, none of these penalties is visible to the scheduler at placement time, which is why we cannot simply budget for them and must instead rule them out by construction.

The TPU study isolates a single pair of jobs; Figure 5.2 asks how the placement heuristics shipped in production schedulers contribute to this contention at cluster scale. We sweep a single main job from a handful of nodes up to the full 4096-node cluster, a $16 \times 16 \times 16$ three-dimensional torus, and measure the contention that job experiences as it grows. We pack the idle nodes that are not part of the main job with $2 \times 2 \times 1$ background jobs that generate their own collective traffic, so the experiment captures how that background communication contends with the main job on shared links. The ideal policy runs the main job with no background jobs at all, and hence no competing traffic. The five baselines we compare against, here and throughout the evaluation, fall into two families, and we introduce them once here for reference.

The shape-unaware, or scatter, policies place a job on a set of nearby (or arbitrary) XPU without guaranteeing the contiguous ring that its communication pattern assumes. SFC [130], the space-filling-curve policy that is the default in SLURM and ALPS, maps the three-dimensional torus onto a one-dimensional Hilbert curve and packs each job into a contiguous segment of that curve; L1 Clustering, also known as MC1x1 [148], instead greedily gathers the nearest available XPUs around a seed node; and TopoMatch [108] embeds the job’s communication graph onto the torus to minimize aggregate communication cost. Random, included as a worst-case reference, simply assigns each job an arbitrary set of free XPUs with no regard for locality.

The shape-aware family instead requires that an allocation reproduce the job’s logical shape exactly. Its representative is FirstFit [41], which scans the torus and takes the first contiguous region that matches the shape; a job it places runs contention-free, but the constraint is stringent enough that it often finds no fit even when ample XPUs sit idle elsewhere.

None of these policies can reshape the underlying fabric, and for the scatter family that is decisive: a statically wired torus cannot always supply the ring that a job’s logical shape assumes,

and the surplus traffic spills onto links shared with the background jobs. The trend is evident in Figure 5.2. As the main job grows and occupies more of the cluster, it contends with more background traffic, and its JCT slowdown relative to the ideal placement climbs to as much as $2.5 \times - 3 \times$. This performance gap is the penalty of suboptimal job placement.

FirstFit is the exception in this figure, and instructively so. It stays within 13% of ideal across the sweep, because a job it does place is contiguous and runs without contention. What the experiment cannot show is the price of that guarantee: a single main job on an otherwise empty cluster is the easiest case FirstFit will ever face. Once a stream of jobs arrives and departs asynchronously, its shape constraint strands idle XPU and leaves jobs queued behind allocations that will not fit, which is the regime §5.4 measures. Neither family is satisfactory, then, and the two fail in opposite ways: the scatter policies pay in contention, FirstFit pays in placeability, and no policy confined to a static fabric escapes the trade.

Therefore, our goal is to minimize contention by enforcing the job shape so that allocated XPUs and links are always exclusive to the job, and to do so without inheriting FirstFit’s placeability penalty.

5.3 RFold resource allocation

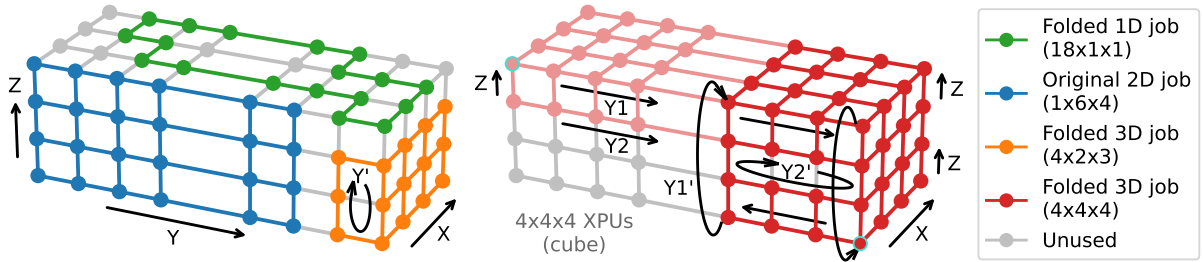


Figure 5.3: Demonstration of 1D, 2D and 3D job folding in a 3D torus. Each $4 \times 4 \times 4$ cube is reconfigurable, with wrap-around links (not displayed) to itself on the X and Z axes, and to the other cube on the Y axis.

In this section we introduce RFold, a resource allocation scheme that treats the cluster topology as an adjustable resource rather than a fixed one. Its goal follows from the tension identified in §5.1.3: place every job on a compact, well-aligned region so that its collective traffic stays fast, while leaving the unused capacity in shapes that later jobs can still occupy. On a static torus these two aims pull against each other, because the only way to keep a job compact is to find a contiguous region of exactly the right shape, and such regions grow scarce as the cluster fills. RFold eases the conflict by changing what “the right shape” means and, when needed, by changing the fabric itself.

RFold rests on two complementary techniques, reconfiguration (§5.3.1) and folding (§5.3.2). Reconfiguration adapts the underlying topology to the job by rewiring the optical circuit switches, so that cubes sitting apart in the machine can be joined into the torus a job needs. It is the capability that prior torus schedulers lack, and the one that lets RFold host shapes a static torus can never place, restore wrap-around links to jobs that would otherwise run without them, and defragment

the cluster by re-joining scattered cubes. Folding, in turn, rewrites a job’s logical shape into equivalent physical shapes that are easier to place, giving reconfiguration more footprints to try and wasting fewer cubes on partial occupancy. Reconfiguration changes the fabric to fit the job; folding changes the job to fit the fabric.

Both techniques spend a scarce resource. A reconfigurable OCS link is worth more than a plain link, because only OCS links can be rewired to change the topology or to defragment, and every OCS port a job ties up is one unavailable for a future reconfiguration. This leads to the core heuristic of RFold: among all the placements a job admits, the best one consumes the fewest cubes and the fewest OCS links. Reconfiguration and folding generate the placements, the heuristic chooses among them, and §5.3.3 makes that choice precise.

5.3.1 Reconfiguration

A static torus is commonly constructed with an equal number of XPU in every dimension. For example, a 4096-XPU cluster would be built as $16 \times 16 \times 16$. The static nature of this design limits the job shapes that can be supported. Consider a job that requires $4 \times 4 \times 32$ XPU. While the total number of XPU required can be met, with the shape constraint, this job can never be placed because one of its dimensions exceeds the maximum dimension size of the torus ($32 > 16$).

Reconfigurable torus offers more flexibility in job placement, i.e., it can support more shapes by adapting the runtime topology to the job shape. Recall the TPU v4 cluster described in §5.1. It contains 64 $4 \times 4 \times 4$ hardwired cubes, and any two cubes can be connected via the OCSes to form a larger torus. Hence, to place the $4 \times 4 \times 32$ job, we do not have to find 32 contiguous XPU in one dimension. Instead, we only need eight $4 \times 4 \times 4$ cubes to be reconfigured side-by-side.

Supporting otherwise-unplaceable shapes is only the first of three things reconfiguration buys. The second is wrap-around links. A ring collective needs its two ends joined, and on a static torus that closing link exists only when the job spans an entire dimension. Because RFold makes the reconfigurable unit a $4 \times 4 \times 4$ cube rather than the whole machine, any job whose sides are multiples of four can close its rings through the OCS, so far more jobs run with the low-diameter communication a torus is meant to deliver. The third is defragmentation and failure handling. Since the OCS decides at runtime which cubes are adjacent, cubes freed by departing jobs can be rejoined into a fresh contiguous region instead of being stranded as scattered holes, and a cube lost to a hardware fault can be switched out of the torus and replaced by a spare without renumbering the survivors. We quantify these last two benefits in §5.4.4 and §5.4.5.

When job shapes are not a multiple of four, for example $4 \times 4 \times 34$, the mapping results in at least one partially used cube. To address this fragmentation, we could construct smaller cubes, like $1 \times 1 \times 1$, to ensure full utilization. Yet, such a design would constrain the cluster’s overall scale because if all torus links are OCS links, the OCSes would quickly run out of ports. We primarily focus on $4 \times 4 \times 4$ cubes in our proposal since it is practical with current technology. The cube size is thus a design knob that trades placement flexibility against optical cost. Finer cubes can be rejoined more independently and make more shapes placeable, but they demand more OCS ports, and port count is what ultimately bounds how fine the tiling can be; §5.4.5 examines this trade-off in depth.

Although reconfiguration generally makes it easier to satisfy the job shape constraints, it is not unconstrained: the TPU-style fabric can rewire only certain links, and three limits follow that

RFold must design around. First, each cube is only reconfigurable on its faces. If a cube has available XPU in the core but all face XPU are used, its available XPU cannot be connected to other cubes to host a larger job. Second, an XPU on the face can only connect to the corresponding XPU in another cube at the same position. Thus, even with enough available XPU in two combined cubes, the two halves might not be allowed to connect if they are misaligned. Third, jobs in a reconfigurable torus only receive wrap-around links when their shapes are a multiple of the cube dimension size N . Nevertheless, in cases where the job shape is not a multiple of N , jobs suffer from poor communication performance. This last limit, rings that cannot close, is precisely what folding is built to repair, which we turn to next.

5.3.2 Folding

Folding changes the job shape to make the job easier to place, given the existing topology and resource availability. Thus, it complements reconfiguration and helps overcome the limitations just described. It is worth noting that rotation [52, 67] is a widely used technique in multi-dimensional bin packing. When placing a job within a 3D torus, rotation offers $3! = 6$ distinct placement options while preserving the job shape. We assume rotation is a default behavior incorporated into all placement policies and is therefore not considered a specific aspect of folding.

When a job arrives, the scheduler leverages folding to find all the shape variants homomorphic to the original shape requested by a job. An original shape can be folded to another shape as long as the communication pattern can be faithfully mapped onto the new shape. Then the scheduler evaluates each of these shape variants by invoking reconfiguration. The shape variant consuming the least amount of cubes is chosen and committed. Since jobs in a 3D torus usually communicate in one, two, or all three dimensions, they are henceforth referred to as 1D/2D/3D (shaped) jobs. We illustrate how folding works for these jobs using the examples in Figure 5.3.

1D folding. 1D jobs have a shape of $A \times 1 \times 1$, which indicates that the communication happens between A XPU along a one-dimensional ring (e.g., data-parallel only). A naïve scheduler that treats all jobs in a 3D torus as 3D would allocate A XPU along a straight line for these jobs. If A is not a multiple of cube size N , these jobs do not get wrap-around links, and cannot form a ring. In such unfortunate cases, the resource allocation and job shape mismatch would result in performance penalties. However, shape $A \times 1 \times 1$ does not necessarily require all A XPU to be placed on the same dimension. Rather, these A XPU can be arbitrarily allocated as long as they form a cycle/ring. Therefore, the main task in 1D folding is to find a simple cycle of length A given the grid graph of currently available XPU.

As an example, the green job in Figure 5.3 (left) is a 1D job of shape $18 \times 1 \times 1$. There are only two available $4 \times 4 \times 4$ cubes and they are reconfigured to form a larger $4 \times 8 \times 4$ 3D torus. No single dimension in this torus can provide 18 consecutive XPU. With folding, we are able to find 18 scattered XPU forming a cycle to meet the job shape requirement. Note that this example demonstrates how a 1D job is folded to 2D, but 1D jobs can also be folded to 3D in the same way. The only requirement is to find an A -XPU cycle.

2D folding. 2D jobs (of shape $A \times B \times 1$) manifest a two-dimensional communication pattern, e.g., DP along one dimension and TP along the other dimension. For instance, the blue job in Figure 5.3 is a 2D job of shape $1 \times 6 \times 4$, which requires the XPU to communicate along the Y and Z dimensions. Though this 2D job can be placed as shown, it leads to two issues: (1) The 6

XPU along the Y dimension are not a multiple of 4, so there is no wrap-around link. (2) There is resource fragmentation along the X dimension. The remaining 3 XPU along the X dimension are unable to serve jobs of even shape sizes, which are the majority of jobs.

Folding can mitigate these two issues. Specifically, we can fold the original 2D job to a 3D job of shape $4 \times 2 \times 3$ (orange job). This is feasible because shape $1 \times 6 \times 4$ is graph-homomorphic to shape $4 \times 2 \times 3$. Communication along the Z dimension in the original 2D job can be mapped to communication along the X dimension in the folded 3D job. Communication along the Y dimension in the original 2D job now becomes communication along the Y' direction (circular). This is how folding forms a cycle for non-multiple-of-4 shape sizes even if there is no wrap-around link. This technique of forming a cycle is applicable to most jobs with even shape sizes. Note that a 2D job can also be folded to another 2D job. 2D-to-2D folding works similarly to 3D-to-3D folding. Next, we discuss how 3D-to-3D folding works.

3D folding. True 3D jobs have shape sizes greater than 1 in all three dimensions, and communicate in all dimensions. This characteristic makes them the least “foldable” compared to 1D and 2D jobs. Without loss of generality, it is easy to fold an M-dimensional job to an (M+1)-dimensional torus because the extra dimension can accommodate communication requirements from the lower dimensions. But folding an M-dimensional job in an M-dimensional torus is non-trivial because of the communication requirements.

Nonetheless, certain 3D jobs can be folded, such as the red 3D job in Figure 5.3 (right). The original 3D job has a shape of $4 \times 8 \times 2$. To place it as is, we need two $4 \times 4 \times 4$ cubes reconfigured to form a $4 \times 8 \times 4$ torus. Half of the job will be placed in the left cube and the other half in the right cube. Through folding, it is possible to place the entire job in one single $4 \times 4 \times 4$ cube. Essentially, the left half of the job (light red) is folded to the bottom half of the right cube, e.g., the front-top-left XPU maps to the front-bottom-right XPU.

The original shape $4 \times 8 \times 2$ is homomorphic to the folded shape $4 \times 4 \times 4$ with special communication mapping. More specifically, communication along the X dimension remains unchanged. Communication along the Z dimension also remains the same, since the top two layers and bottom two layers of the cube communicate between themselves along the Z dimension. Hence, the Z-dimension links in the center between the middle two layers are not needed (greyed out). Communication along the Y dimension is mapped differently for the two layers of the original job shape. We label them Y1 and Y2. Y2 is mapped to Y2' (circular), similar to the 2D folding example. For communication along Y1, we cannot find a direct cycle like Y2'. So we take advantage of the wrap-around links and map Y1 to Y1'. The XPU on the top and bottom layers first communicate along the Y dimension, then use the wrap-around links to communicate with each other without traversing the middle two layers.

On the other hand, a job of shape $4 \times 8 \times 3$ cannot be folded to $4 \times 4 \times 6$ in the same way because only the innermost two layers and the outermost two layers can form a cycle to communicate. In the case of $4 \times 8 \times 3$, the middle layer cannot be mapped to any cycle in the $4 \times 4 \times 6$ shape.

Overall, jobs can be ranked by their “foldability” in the following order: 1D > 2D > 3D. When placing multiple new jobs, we should prioritize 3D jobs and then 2D jobs because 1D jobs can fit into the available space with minimum effort.

Table 5.1: Notation. The lower group are the rank-key components of Eq. 5.1; smaller is always better.

Symbol	Meaning
J, s	arriving job; its logical shape $s_x \times s_y \times s_z$
D, N	torus dimensions $D_x \times D_y \times D_z$; cube side (the torus is tiled into $N \times N \times N$ cubes)
F	set of currently free XPU's
V, C	fold variants of s ; candidate placement pool
$\perp$	search failure (no placement found)
$frag$	# cubes newly fragmented: fully idle before, partially occupied after
$cubes$	# distinct cubes touched
$multihop$	# ring edges left to multi-hop routing
$cost$	compactness-scorer tiebreak

5.3.3 Placement algorithm

The previous two subsections describe reconfiguration and folding as separate capabilities. This subsection ties them into a single placement engine and makes precise the core heuristic of §5.3, which is to prefer, among all placements a job admits, the one that fragments the fewest idle cubes, then the one using the fewest cubes, then the one with the best communication. The hardware model shapes every rule here. Only the links inside a cube are fixed wiring, while every inter-cube link is an optical circuit, so any two cubes are equally reachable and circuits themselves cost nothing. The one physical scarcity is that each face port holds a single circuit at a time. Table 5.1 collects the notation used throughout.

Algorithm 3 presents RFold, the scheduler’s placement engine. It has two objectives. It must never admit a job later than a folding-only placer would, and it must admit strictly earlier whenever the optical fabric offers a placement that folding cannot express. The design lesson is that reconfiguration must be available as a mechanism at every level of the search, while the objective that arbitrates among placements must be protecting future placeability. Every preference we tried that overrides it for another virtue (circuit frugality, shape fidelity, even a preference for contiguity itself) lost more than it gained.

Candidate generation (lines 1–7). Line 2 decouples the job’s logical shape from its physical footprint. By re-shaping s into other cuboids of equal size (e.g., $1 \times 1 \times 6 \rightarrow 2 \times 3 \times 1$), folding lets a job whose native shape fits nowhere fit after deformation. Lines 6–7 then probe every variant with both placement mechanisms, a solid free cuboid (Algorithm 4) and a non-contiguous placement across cubes (Algorithm 5), and pools the results as equals. Nothing prefers one mechanism over the other by fiat. Where a solid box wins, it wins because it fragments fewer cubes or communicates better, judged by the same key as every other candidate.

Selection (lines 8–12). Line 10 chooses the candidate that minimizes the rank key

$$\text{RANKKEY}(p) = (frag, cubes, multihop, cost) \quad (5.1)$$

Algorithm 3 RFOLD(J, F): place arriving job J on the free XPU's F ; notation in Table 5.1. Called once per scheduling event; returns PLACED, DEFER, or DROP.

```

1: // Step 1: enumerate candidate physical footprints.
2:  $V = \{s\} \cup \text{FOLDVARIANTS}(s)$  // equal-size cuboid re-shapings of  $s$ 
3:  $C = \emptyset$ 
4: for  $v \in V$  do
5:   // Step 2: both mechanisms feed one candidate pool.
6:    $C = C \cup \{\text{CONTIGUOUSSCAN}(v, F)\} \setminus \{\perp\}$  // Algorithm 4
7:    $C = C \cup \{\text{NONCONTIGUOUSPLACE}(v, F)\} \setminus \{\perp\}$  // Algorithm 5
8: // Step 3: choose by the fragmentation-led ranking of Eq. 5.1.
9: if  $C \neq \emptyset$  then
10:   $p^* = \arg \min_{p \in C} \text{RANKKEY}(p)$ 
11:   $\text{WIREOCS}(p^*)$  // install the OCS circuits realizing the winner's ring edges
12:  return PLACED( $p^*$ )
13: // Step 4: 1-D last resort, then the defer/drop oracle.
14: if  $\text{ONEDIMENSIONAL}(s)$  and  $(p = \text{SNAKEID}(J, F)) \neq \perp$  then
15:  return PLACED( $p$ ) // one ring can thread arbitrary free XPU's
16: if not  $\text{PLACEABLEONIDLE}(J)$  then
17:  return DROP // memoized: Steps 1–2 on an idle torus; impossible forever
18: return DEFER

```

(components in Table 5.1; k_i denotes the i -th) under the lexicographic order

$$p \prec p' \iff k_i(p) < k_i(p') \text{ at the first } i \text{ where they differ.} \quad (5.2)$$

Eq. 5.2 turns four numbers into one verdict, and it does so deliberately without a weighted sum, because each component is an absolute veto over everything after it and there are no weights to tune. *frag* leads because idle cubes are the currency every future job draws on. A placement that consumes whole cubes, or adds to cubes that are already fragmented, costs the future nothing; each cube newly taken from fully idle to partially occupied is one fewer home for the jobs to come. *cubes* and *multihop* follow, asking the placement to use less of the machine and then to communicate in fewer hops. Notably absent is any key that prices circuits or distance. On this hardware a circuit between far cubes costs exactly what one between neighbors costs, so frugality with either would optimize a quantity that does not exist. Port exclusivity, the one real scarcity, enters as a feasibility rule during wiring (Algorithms 4 and 5), not as a preference. We considered other heuristics before settling on this key. One is an occupancy-adjacency tiebreak, which, among candidates that tie on the components above, prefers the placement sitting closest to already-occupied XPU's, on the reasoning that packing work against work leaves the remaining free space in larger pieces. Another is a contiguity-first tier, an extra leading component that ranks every contiguous placement above every non-contiguous one, on the reasoning that a solid cuboid always communicates best. Neither survives measurement. Occupancy adjacency judges a placement by the shape of the current boundary rather than by the cubes it consumes, so it steers work into partially occupied cubes without regard for whether the remainders it leaves are usable,

Algorithm 4 CONTIGUOUSSCAN(v, F). The best placement of fold variant v inside one solid free cuboid.

```

1:  $best = \perp$ 
2: for  $\rho \in \text{ROTATIONS}(v)$  do // up to 6 axis permutations
3:   for anchor  $a \in D$  do // every position, wrapping around the torus
4:     if  $\text{XPUS}(\rho, a) \subseteq F$  then
5:        $p = \text{MAKEPLACEMENT}(\rho, a)$ 
6:       for ring edge  $e \in \text{RINGEDGES}(p)$  do
7:         if  $\text{TORUSADJACENT}(e)$  then pin  $e$ 
8:         else if  $\text{OCSREALIZABLE}(e)$  and  $\text{PORTSFREE}(e)$  then wire  $e$ 
9:         else  $\text{multihop}(p) += 1$  // rides the shared fabric
10:       $best = \text{MINBY}(\text{RANKKEY}, best, p)$ 
11: return  $best$ 

```

and over a long sequence of greedy decisions it carves the free space into pieces no future job fits. The contiguity-first tier fails for the opposite reason, because it makes the mechanism the objective. Once cubes may be shared, a non-contiguous placement that consumes no idle cube is worth more to the future than a contiguous one that breaks a fresh cube open, and a tier that vetoes the former on principle forfeits that trade whenever it arises. Both are the same mistake, which is overriding placeability for a virtue that only resembles it. Line 11 installs the winner by wiring its OCS circuits, so communication quality becomes a property of the fabric, not of routing luck.

Last resorts and the oracle (lines 13–18). A one-dimensional job is a single ring, which can thread arbitrary free XPUs as long as consecutive ranks stay adjacent; line 14 runs this snake search under a fixed budget. Finally, lines 16–18 separate “not placeable now” from “not placeable ever”. The oracle replays Steps 1–2 against an idle torus (memoized per shape), and only a job that fails even there is dropped. Without it a permanently impossible job would defer forever and, under first-come-first-served admission, stall the entire cluster behind it.

Contiguous search (Algorithm 4). Lines 2–3 make the cuboid search exhaustive, trying all axis rotations of the footprint anchored at every torus position with wraparound. Lines 7–9 classify each ring edge instead of gating the placement on it. Edges between torus neighbors are pinned directly, edges the OCS can serve are wired, and the remainder increment $\text{multihop}(p)$. This keeps feasibility and preference apart, so placeability stays maximal while Eq. 5.1 expresses taste. An edge is OCSREALIZABLE when its endpoints lie on cube faces of one axis at the same within-face position. Between two distinct cubes either face may pair with either (an optical light path has no polarity), while within one cube only the opposite-face pair exists, which is the wrap that closes a one-cube-wide ring. PORTSFREE enforces the one hardware scarcity. A face port holds a single circuit, and an edge that rides a cross-cube torus adjacency is itself occupying the default circuit on both of its ports, so a would-be wire whose port is already spoken for rides the shared fabric instead of double-booking it.

Non-contiguous search (Algorithm 5). Line 3 cuts the footprint into cube-sized tiles, and line 4 packs those tiles into the free space under a search budget, with three freedoms the placement of a solid cuboid does not have. A tile may land in any cube, at any within-cube offset that fits, and several tiles, of this job or of jobs already running, may share one cube on disjoint XPUs.

Algorithm 5 NONCONTIGUOUSPLACE(v, F). Places fold variant v across cubes connected through the OCS.

```

1:  $best = \perp$ 
2: for  $\rho \in \text{ROTATIONS}(v)$  do // all 6 axis permutations
3:    $tiles = \text{TILE}(\rho, N)$  // cut the footprint into cube-sized tiles
4:    $A = \text{PACKTILES}(tiles, F)$  // budgeted DFS: each tile onto any (cube, offset) whose
   XPU's are free; tiles may share a cube; fragmented cubes tried before idle ones
5:   if  $A \neq \perp$  then
6:      $p = \text{MAKEPLACEMENT}(A)$ 
7:     for ring edge  $e \in \text{RINGEDGES}(p)$  do
8:       if  $\text{TORUSADJACENT}(e)$  then pin  $e$ 
9:       else if  $\text{OCSREALIZABLE}(e)$  and  $\text{PORTSFREE}(e)$  then wire  $e$ 
10:      else  $\text{multihop}(p) += 1$  // rides the shared fabric
11:     $best = \text{MINBY}(\text{RANKKEY}, best, p)$ 
12: return  $best$ 

```

The candidate order concentrates fragmentation, trying cubes that are already partially occupied before idle ones, so remainders pile into existing fragments instead of opening new ones. This is the capability folding lacks, and it lets RFold place jobs whose sides do not divide the torus axes, jobs longer than an axis, and jobs arriving to fragmented or partially failed fabric. Lines 8–10 classify ring edges exactly as the contiguous search does, and the classification never vetoes a placement. An earlier revision rejected any assignment whose ring edges could not all be wired, which silently restricted non-contiguous side lengths to powers of two, so that a shape with a side of 6 could only place contiguously or defer. Treating unwirable edges as *multihop* makes placeability maximal here too, while Eq. 5.1 charges the communication debt.

Algorithm 4 alone is the folding-only placer, and RFold degenerates to exactly it when the cube is the whole torus and no OCS exists, so RFold’s candidate pool is a strict superset of folding’s at every decision. Every use of the optical fabric beyond the default configuration is therefore a measured improvement. Wiring circuits closes partial-axis rings at one hop, which folding cannot express even for the same cuboid, and non-contiguous placement admits jobs in states where folding-only would have deferred them and idled the fabric. What keeps this power disciplined is not a rule against using it but the ranking itself, since *frag* makes every candidate, solid or not, pay for the idle cubes it breaks, so the greedy trajectory preserves whole cubes for the jobs to come while spending reconfiguration wherever it buys admission.

5.4 Evaluation

The goal of this evaluation is to understand, quantitatively, where RFold helps, where it does not, and how it compares against the placement policies already deployed in torus clusters. We are interested in RFold’s behavior across the range of conditions a shared cluster actually encounters, from light to near-saturating load and under several different admission policies.

Our study is simulation-based. Reproducing these effects on real hardware would demand a

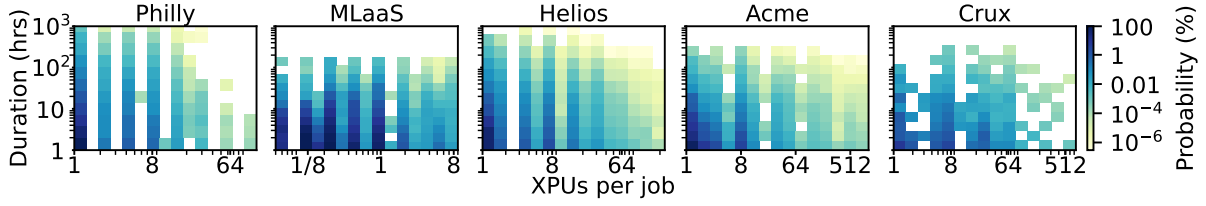


Figure 5.4: Joint distribution of job size (XPU per job) and duration across five public ML cluster traces, drawn as per-trace log-log 2D heatmaps. Color encodes probability mass. In every trace the mass concentrates in the lower-left corner; most jobs are small and short.

reconfigurable, multi-thousand-XPU cluster and weeks of trace replay for every configuration, which is impractical; simulation instead lets us sweep loads, cluster sizes, and policies freely. We build on *astra-sim* [190, 232], a widely used simulator of distributed ML workloads that models per-rank job traces and collective communication in detail, and extend it in three ways for our setting. First, we add multitenancy support, so that many jobs are admitted, placed, and run concurrently on one shared cluster rather than a single job in isolation (which is what *astra-sim* is originally designed for). Second, we add a reconfigurable network backend that models OCS reconfiguration and lets the topology change at runtime. Third, we add a scheduling runtime that implements the admission and placement policies compared throughout this section.

Unless otherwise noted, our main results target an $8 \times 8 \times 8$ 3D torus built from reconfigurable cubes, the same cube-based reconfigurable architecture used in production TPU clusters [113]. This is the configuration in which RFold’s folding and reconfiguration both have room to operate, and it is representative of the systems that motivated this work.

5.4.1 Workload characteristics

Before sweeping the load, we characterize the workloads that drive our experiments through a short study of publicly available ML cluster traces [36]. Two properties matter for placement: the distribution of job sizes and shapes, which sets how hard jobs are to pack, and the offered load, which sets how heavily the cluster is contended.

Figure 5.4 shows, for five public traces (Philly [109], MLaaS [229], Helios [104], Acme [105], and Crux [33]), the joint distribution of job size (XPU per job) and duration as a log-log two-dimensional heatmap. Across every trace the probability mass concentrates in the lower-left corner, where the overwhelming majority of jobs request only a handful of XPU and run for a short time, while a heavy tail stretches toward large, long-running jobs. Job sizes thus follow a small-heavy, Pareto-like distribution, and durations are similarly short-heavy. This heavy-tailed shape is exactly what makes placement hard. The many small jobs fragment the cluster, while the rare large jobs are the ones most likely to be unplaceable on a static torus.

To quantify how heavily a cluster is used, we adopt the standard queueing-theory notion of offered load. The load ρ is

$$\rho = \frac{\sum_T S}{NT} \quad (5.3)$$

where T is a time duration to observe, S is the job size in $XPU \cdot hours$, and N is the number

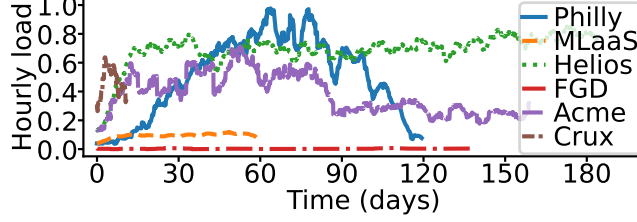


Figure 5.5: Timeseries of cluster load for several public ML cluster traces. Offered load differs widely across clusters, from heavily loaded (Philly, Helios) to lightly loaded (MLaaS, FGD).

of XPU's in the cluster. $\sum_T S$ is the total amount of work to be done within the observed time duration, and $N \cdot T$ is the available capacity of the cluster.

Note that we also have

$$\sum_T S = \lambda \cdot E[S] \cdot T \quad (5.4)$$

where λ is the job arrival rate. Given a target ρ , e.g., 0.5, we can derive the mean inter-arrival time as

$$E[IAT] = \frac{1}{\lambda} = \frac{E[S]}{\rho N} \quad (5.5)$$

Figure 5.5 plots the hourly load over time for each trace. The striking feature is how much it varies from cluster to cluster. Some clusters are heavily contended, sitting at a load of roughly 0.7 to 0.8 for long stretches (Philly and Helios), whereas others are far more lightly loaded, operating around 0.2 to 0.4 (Acme) or even lower (MLaaS and FGD). No single load level is representative of production. We therefore do not evaluate at one fixed load; instead, the experiments that follow sweep the offered load across the whole range, so that our conclusions cover both the congested and the underutilized regimes that real clusters exhibit.

From these observations we construct the synthetic traces that drive the rest of the evaluation, as follows. Job sizes are drawn from a Pareto distribution over the integers from 1 to 128, reproducing the small-heavy shape of Figure 5.4. Not every integer in this range is an admissible legal size. ML job shapes must map onto a 1D, 2D, or 3D shape, so its size must factor into at most three dimensions. Each dimension is either one or an even size. A size is therefore legal only if it admits at least one such factorization. We draw job sizes from the Pareto distribution restricted to these legal sizes. Once a job's size is fixed, we enumerate all of its legal shapes under the same per-dimension rule and draw the job's shape uniformly from that set. A job's duration is represented by its number of iterations, sampled from a Pareto distribution truncated to between 1 and 10 iterations, which keeps the simulation tractable. However, note that real-world ML jobs could run for thousands of iterations. To assemble these jobs into a trace at a target load, we assume a Poisson arrival process and invert the load relation of Equation 5.5. Given each job's size and duration, we set the mean inter-arrival time so that the offered load matches the target, and emit the jobs in arrival order. Each constructed trace contains 100K jobs, long enough for the cluster to reach steady state so that the warmup and drain phases in the simulation are negligible.

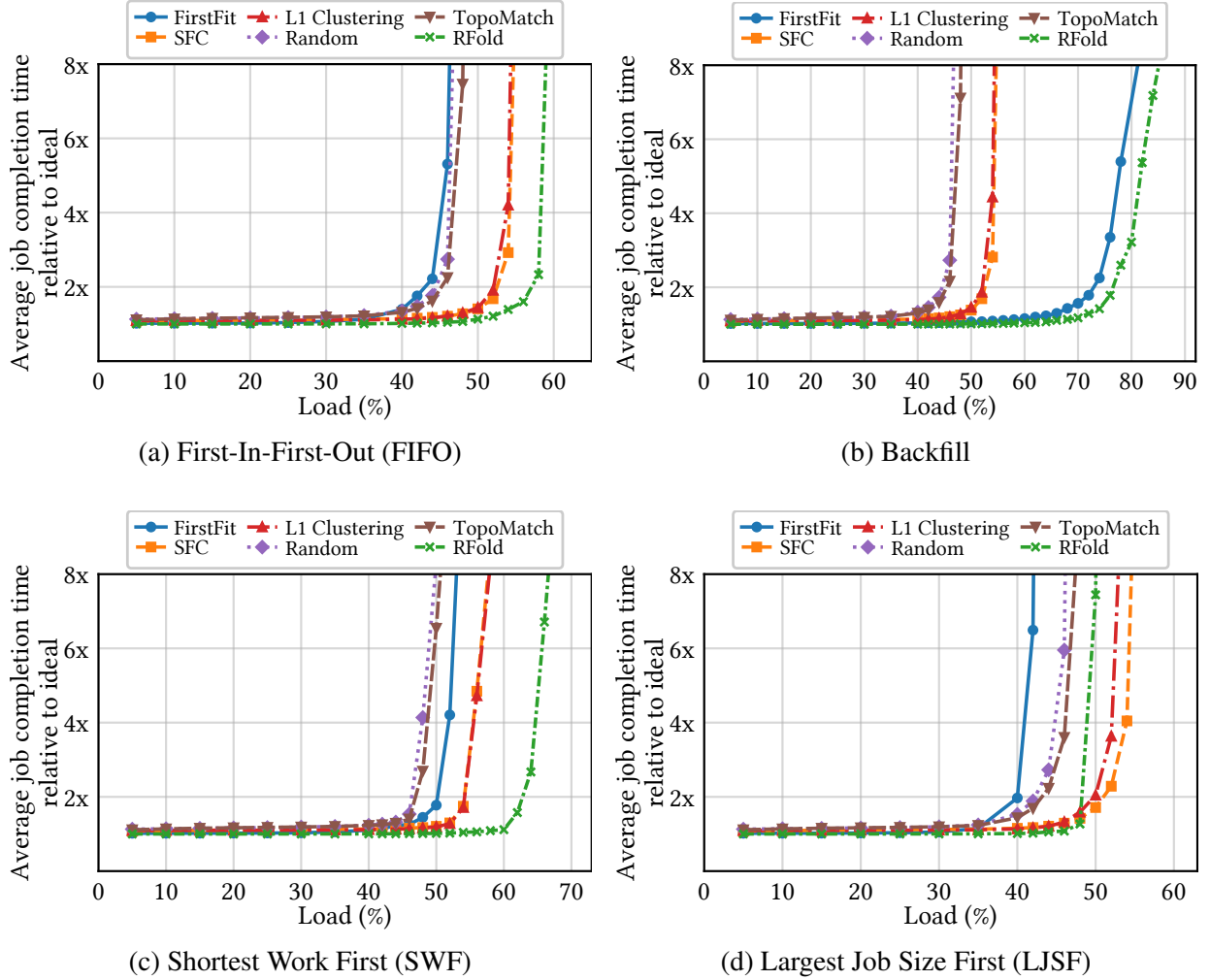


Figure 5.6: Average JCT versus load sweep of different placement policies. Subplot (a) uses FIFO as the admission policy, (b) uses Backfill, (c) uses SWF, and (d) uses LJSF.

5.4.2 Admission and placement policies

A production cluster often observes varying load levels. Sometimes it is lightly loaded, and at other times jobs arrive in bursts that create high loads. To understand how job scheduling behaves across this whole range, we sweep the load from low to high and measure the performance of each admission and placement policy. Two questions guide the study. First, how do shape-aware and shape-unaware placement policies compare across the full load spectrum, and how much load can each absorb before its job completion time runs away? Second, how sensitive is the gap between them to the admission policy the operator happens to run? A placement policy is only useful in production if its advantage holds up under whatever admission policy sits on top of it.

Using the offered-load definition of §5.4.1, we sweep the load ρ from 0.05 to 1.0 and plot the average JCT that each policy delivers. To put the different policies and their JCT numbers into perspective, we normalize the average JCT at every load to that of an “ideal” placement. The ideal placement is a hypothetical, fully connected mesh over the same 512 XPU, in which every

XPU sits exactly one hop from every other. A job’s logical neighbors are therefore always directly connected, and no two flows ever share a physical link. Hence, a job experiences no network contention at all and runs at its contention-free minimum. Such a full mesh is too expensive to build at this scale, so it only serves to demonstrate the best theoretical performance. A curve sitting at $1\times$ means the policy matches this contention-free performance thus is optimal, while any excess above $1\times$ is the combined penalty of network contention and queueing delay. Each point is an average over independently generated traces.

The load sweep compares six placement policies. Five are the baselines introduced in §5.2: the shape-unaware, or scatter, policies SFC [130], L1 Clustering [148], TopoMatch [108], and Random, none of which enforces contiguity; and the shape-aware FirstFit [41], which places a job only on a contiguous region of the static $8\times 8\times 8$ torus that matches its shape exactly. The sixth is RFold. RFold is shape-aware as well, but it is the only policy in the comparison that is reconfiguration-ready. Instead of searching a fixed topology, it folds the job shape and reconfigures the underlying fabric together, obtaining a contention-free contiguous allocation without being blocked by the fragmentation that stalls FirstFit. Every other policy operates on the fixed torus and cannot rewire a single link.

Figure 5.6a sweeps the load under FIFO admission, which admits jobs in strict arrival order and lets an unplaceable job block every job behind it, which is often referred to as head-of-line (HOL) blocking. Every curve shows the classic queueing signature, a long, nearly flat region at low load followed by a sharp, near-vertical shoot-up. The flat region sits just above $1\times$ because at low load a job rarely overlaps with others, so its only penalty is the modest contention of its own placement. The shoot-up marks the stability knee. As the offered load approaches the effective service rate of the cluster under that policy, jobs begin to arrive faster than they drain, the queue builds without bound, and the average JCT diverges. The horizontal position of the knee is thus the maximum load the policy can sustain.

The knees sit well below 100% load, for two distinct reasons that separate the two families. A scatter policy loses capacity to contention. Every admitted job runs longer than its ideal time due to network contention, so it holds its XPU’s longer than the offered load budgeted for, and the cluster’s effective service rate drops below its raw capacity. The heavier the contention, the earlier the knee, which is why Random saturates first, near 46% load, TopoMatch shortly after near 48%, and SFC and L1 Clustering later, both near 54%. A shape-aware policy loses capacity for the opposite reason. FirstFit introduces no contention, but its shape constraint leaves many jobs unplaceable, and under FIFO a single unplaceable job at the head of the queue blocks every job behind it. This HOL blocking idles XPU’s and throttles throughput, dragging FirstFit’s knee down to about 44%, as early as the worst scatter policy, even though its jobs run at near-ideal speed once placed. This indicates that under FIFO, the shape constraints and HOL blocking degrade performance more than contention does.

RFold escapes both failure modes. Folding preserves the contention-free shape, so its jobs run at near-ideal speed like FirstFit’s, while reconfiguration removes the fragmentation that would otherwise block them. Its effective service rate is consequently the highest of the group, and its knee sits furthest to the right, near 58% load. Across the whole sweep RFold’s curve is also the lowest, meaning that at any given load it delivers the shortest average JCT. Under FIFO, then, RFold sustains a meaningfully higher load than every baseline before saturating.

We next ask whether this advantage is an artifact of FIFO. FIFO is the strictest admission

policy, honoring arrival order absolutely and, as we saw, letting one stuck job stall the queue. The remaining panels replace it with three policies that relax that ordering. Backfill, shown in Figure 5.6b, keeps the arrival order for the job at the head of the queue but lets later jobs jump ahead to fill otherwise-idle XPU, as long as they do not delay the head job’s reservation; this directly targets HOL blocking. Shortest Work First (SWF), in Figure 5.6c, serves the job with the smallest amount of work in the queue first, where work is defined as the number of XPU requested times the job duration, which both lowers the mean waiting time and keeps a large or unplaceable job from monopolizing the front of the queue. Largest Job Size First (LJSF) does the reverse, prioritizing the job requesting the largest number of XPU in the queue. It is inspired by online bin-packing heuristics [52, 67]. A cluster can serve more jobs in parallel if it packs them tightly, and heuristics such as First-Fit Decreasing (FFD) [49] place jobs from largest to smallest to increase packing density.

Comparing the panels shows that the admission policy reshapes the ranking most for the shape-aware baseline. FirstFit is by far the most admission-sensitive policy. Its knee moves from about 44% load under FIFO to about 50% under SWF, and beyond 72%, the highest load we sweep, under Backfill. Under SWF the rise is gradual rather than near-vertical, since SWF sheds pressure by deferring the largest jobs. The reason is precisely HOL blocking. FirstFit’s ceiling under FIFO is set not by contention but by the jobs it cannot place, and both Backfill and SWF let the queue flow around such a job instead of stalling on it, recovering most of the lost throughput. The scatter policies, in contrast, barely move between panels, because their ceiling is contention, which no admission policy can undo, so SFC, L1 Clustering, TopoMatch, and Random all saturate in the same 46–56% band regardless of the admission policy.

LJSF, in Figure 5.6d, is the one panel in which RFold does not come out best. Here SFC sustains the highest load, near 55%, while RFold saturates near 48% and FirstFit near 40%. The cause is an interaction between LJSF’s admission policy and RFold’s contiguity requirement. LJSF admits with a strict-stop sweep: it selects the largest pending job as the head, and if that job cannot be placed, admission halts entirely and no smaller job may bypass it. Under FIFO the head is a fresh draw from the size distribution, and most jobs are small, so blocking episodes are rare and brief. Backfill explicitly routes around a blocked head, and SWF picks the smallest job. LJSF is the only policy that maximizes the head’s size on every sweep, and at these loads a large job is almost always pending, so the head persistently demands a large contiguous region.

This is the worst possible input for a contiguity-constrained placement. RFold defers whenever it cannot find a foldable block, so it repeatedly stalls the whole cluster until completions coalesce a large region, and its ceiling drops to about 48%, below the contention ceiling of SFC, which never defers on geometry and so cannot be fragmented by admission order at all. The reversal is narrow rather than general. It does not make the scatter policies better, only RFold worse, and LJSF remains the worst admission policy for every scatter placement’s average JCT, since largest-first is close to the opposite of the shortest-job-first policy that minimizes mean waiting time. The takeaway is that although LJSF resembles the FFD-based online bin-packing heuristics in concept, largest-first admission does not carry over well to a shape-aware scheduling setting, where insisting on the largest job at the head of a queue introduces excessive HOL blocking.

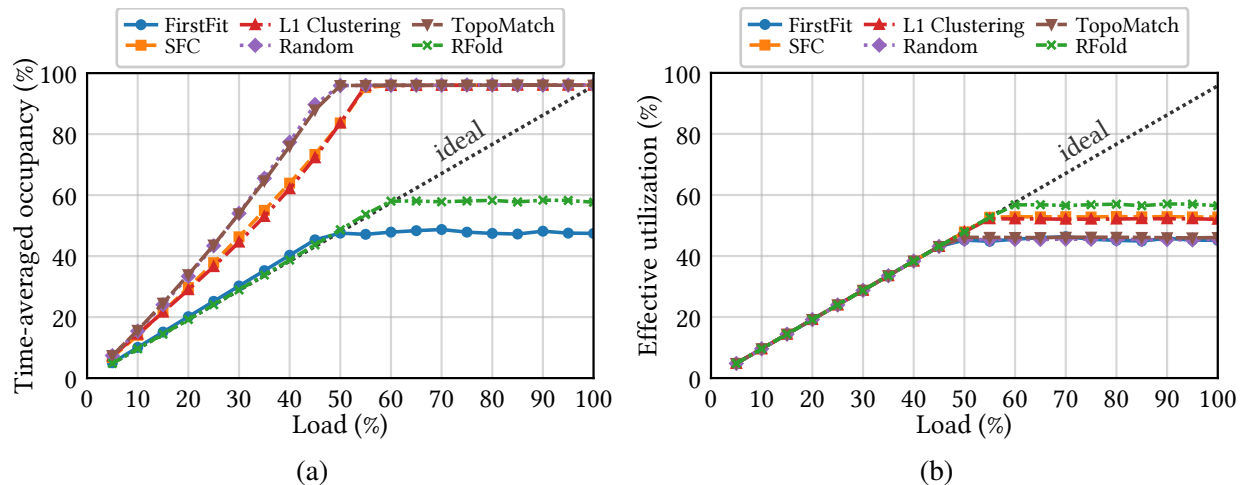


Figure 5.7: (a) time-averaged cluster occupancy and (b) effective utilization versus offered load, for the different placement policies under FIFO admission.

5.4.3 Cluster utilization

How much of the cluster a placement policy keeps busy can be read in two ways, and the two tell very different stories. The first is raw occupancy, the time-averaged fraction of XPU that are allocated to a running job, shown in Figure 5.7a. At any instant, some of the cluster’s XPU are assigned to admitted jobs while the rest sit idle, and the instantaneous occupancy is simply the ratio of the former to the total. Because this ratio fluctuates as jobs arrive, run, and depart, we average it over the whole simulated run, weighting each occupancy level by the time the cluster spends at it. An occupancy of 0.8 thus means that, averaged over time, four out of every five XPU were held by some job. Note that an XPU counts as occupied the moment it is allocated, regardless of whether the job on it is making progress or stalled waiting on a congested link, a distinction that becomes central below. The dotted ideal line marks where occupancy would equal the offered load, the level a cluster would reach if every job ran at its contention-free speed, so that each unit of offered work kept exactly its own share of XPU busy and no more.

Read this way, FirstFit and RFold behave as one would expect. Their occupancy climbs with load along the ideal line and then plateaus once the cluster saturates and no further work can be admitted. The scatter policies, however, look superficially far better, with occupancy that climbs steeply and approaches almost 100% at high load. Because these policies enforce no shape constraint, they can pack XPU densely and keep nearly the whole cluster allocated. The catch is that their occupancy rises faster than the ideal line and pushes well above it. Occupancy should never exceed the offered load, since the offered load already accounts for all the work the admitted jobs would do at full speed. A curve above the ideal line therefore signals that jobs are holding their XPU longer than their contention-free runtime would require. In other words, the scatter policies’ high occupancy is a symptom of contention rather than efficiency. Their XPU spend extra cycles fighting over shared links, so each job lingers and inflates the fraction of the cluster that appears busy.

Raw occupancy is thus a misleading measure of how much a cluster actually accomplishes,

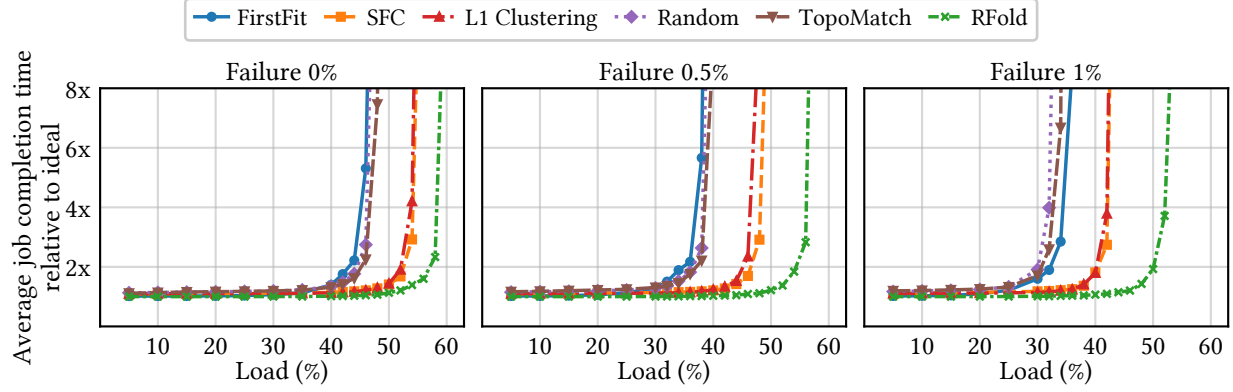


Figure 5.8: Average JCT versus offered load under node failure, for the different placement policies at failure rates of 0%, 0.5%, and 1%, using FIFO admission.

because it credits contention-inflated residency as if it were useful work. The honest measure is effective utilization, shown in Figure 5.7b, which counts only the productive fraction of each occupied XPU’s time. We discount every job’s occupancy by its slowdown, so an XPU running a job that contention has slowed by $2\times$ contributes only half a unit of effective utilization rather than a full one. Effective utilization therefore measures the rate of genuinely useful work, and by construction it can never exceed the offered load, which makes the ideal diagonal a true ceiling. This strips away exactly the contention inflation that flattered the scatter policies under occupancy.

Under this metric the ranking inverts. All six policies track the ideal diagonal until roughly 45% load and then plateau. The scatter policies’ near-full occupancy collapses to the mid-40s and low 50s, erasing the large lead they showed in Figure 5.7a. RFold plateaus highest, near 57%, keeping more of the cluster doing useful work than any baseline. FirstFit, though contention-free, is held down near 45% by the fragmentation that its rigid shapes create, and Random and TopoMatch land at about the same level, while SFC and L1 Clustering sit modestly above them near 52%. Across the saturated regime RFold delivers the highest effective utilization of the group, outperforming the weakest policies by as much as 25%.

Taken together, the two panels make an argument that raw occupancy cannot. Occupancy rewards the very contention that RFold is designed to eliminate, so it flatters the scatter policies that spread jobs across the fabric. Effective utilization, which credits only useful work, instead shows RFold converting more of the cluster’s raw capacity into completed jobs than every baseline. This is the throughput-side counterpart to the JCT results of §5.4.2, since the same folding and reconfiguration that hold job completion time low also let RFold turn more of the hardware into finished work. All occupancy and utilization results reported here are collected under FIFO admission, so that the differences across the curves reflect the placement policy alone rather than the order in which jobs are admitted.

5.4.4 Failure resilience

Node failure is a particular challenge for a torus-based cluster because it disrupts both placement and routing. In a static torus the wiring is fixed, so each node is hard-wired to the handful of

nodes physically adjacent to it, and those wired connections are the only way traffic can flow. When a node fails, the links attached to it become unusable as well, and the connectivity around it is lost with it. Communication that originates from or is destined for a failed node is blackholed, since there is no longer any endpoint to send it to or receive it from. Traffic that merely transits the failed node is not lost but must be rerouted around the hole, lengthening paths and pushing extra load onto the surviving links that absorb the detour. A single failure therefore perturbs a whole neighborhood of the fabric rather than just one XPU.

The effect on placement is equally direct. A job that would previously map onto a perfect contiguous shape can no longer do so once a failed node punches a hole in the region that shape needs. Different resource managers react to this in different ways. Some simply reject the job, refusing to place it until an intact region of the required shape becomes available. Others relax the shape requirement, keeping the rest of the mapping intact and remapping only the rank that landed on the failed node to some other surviving node, at the cost of the contiguity and the contention-free communication that the original shape guaranteed. Either response degrades the outcome, whether by rejecting work the cluster could otherwise run or by admitting it into a placement that now suffers contention.

This subsection quantifies how failure affects the performance of each placement policy and how much failure a given policy can tolerate before it can no longer place work. We repeat the load sweep of §5.4.2 under FIFO admission and simulate failure by randomly selecting some number of XPUs in the cluster and marking them as failed. A failed XPU advertises its failed status, so the placement policies can no longer select it for use, exactly as a real resource manager would exclude a dead node. We study three failure rates, 0%, 0.5%, and 1% of the 512 XPUs, where 0% reproduces the intact-torus baseline and 1% represents a fairly extreme failure scenario for a fabric of this size. All other parameters are held fixed. We use the same workload, the same $8 \times 8 \times 8$ torus, and the same set of placement policies as in the previous subsections, with RFold configured with a $4 \times 4 \times 4$ reconfiguration block.

Figure 5.8 shows how the average JCT of each policy responds as the failure rate rises. As the failure rate grows, more XPUs in the cluster become unusable. Each dead XPU punches a hole in the fabric, and enough such holes leave the cluster no longer a clean, regular torus but a shape riddled with gaps. Since placement depends on finding a contiguous region that matches a job’s shape, these gaps block placement. Jobs that would have been placed on an intact torus can no longer find a legal region, so the cluster admits work more slowly and its achievable stable load drops. This shows up in the figure as the shoot-up shifting toward lower load: each policy saturates at a lower load under a 0.5% failure rate than under 0%, and at a lower load under a 1% failure rate than under a 0.5% failure rate. Nevertheless, RFold continues to sustain the highest load of any policy at every failure rate, and it is also the one that degrades the least. Because a failed node is confined to its own cube and folding can route around the damage, RFold’s knee retreats only modestly as failures accumulate, whereas a policy like FirstFit, whose rigid shapes leave it little slack to begin with, sees its sustainable load fall sharply. To put a single number on this, we measure the $2 \times$ -JCT knee, the offered load at which a policy’s average JCT first reaches twice the ideal, linearly interpolated between samples. By this measure RFold is substantially more failure-resilient. Its knee sits at 57% load on the intact torus and falls only to 54% at a 0.5% failure rate and 50% at a 1% failure rate, a total loss of about 7 load points, or 12%.

The baselines give up far more. SFC starts from a lower knee of 53% and slides to 47% at

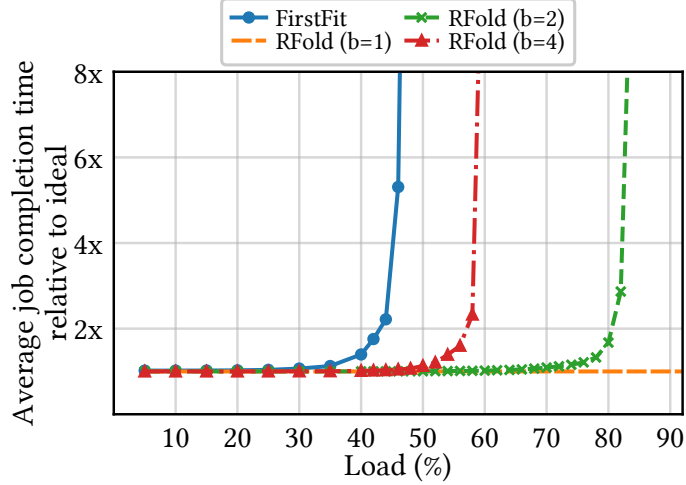


Figure 5.9: Average JCT versus offered load for FirstFit and RFold at reconfiguration cube sizes of $1 \times 1 \times 1$, $2 \times 2 \times 2$, and $4 \times 4 \times 4$, under FIFO admission.

a 0.5% failure rate and then to 40% at a 1% failure rate, shedding 23% of its sustainable load. FirstFit is worse off from the outset: its knee begins at only 43% and drops to 35% and then 32% across the same rates, a loss of 25%. In other words, the very policies that already sustain the least load are also the ones that lose the most of it to failure, while RFold both starts highest and gives up the smallest fraction.

These numbers reinforce the point. At a 0.5% failure rate RFold’s knee moves by 3 load points, versus 6 points for SFC and 8 points for FirstFit. The ranking is not an artifact of the exact threshold: any knee defined between $1.5 \times$ and $4 \times$ the ideal still identifies RFold as the policy that degrades the least. Extending the comparison to the remaining baselines, the total knee loss from a 0% to a 1% failure rate is about 12% for RFold, 23% for SFC and L1 Clustering, 25% for FirstFit, and 32% for Random and TopoMatch.

5.4.5 Reconfigurability

Introducing OCSes into the torus opens a new dimension of placement flexibility, and it is one that none of the fixed-topology policies can use. Only RFold reconfigures the fabric. FirstFit and the scatter policies all operate on whatever wiring the torus happens to present and cannot rewire a single link, so reconfigurability is a lever that benefits RFold alone. This subsection therefore studies RFold in isolation and asks how the degree of reconfigurability, that is, how many OCSes the fabric is given, shapes its performance.

We view the degree of reconfigurability quantitatively as the ratio of OCS links to static links. For a given torus, this ratio is set by the size of the hardwired cube at which the fabric is partitioned. In the $8 \times 8 \times 8$ torus of our evaluation, the default $4 \times 4 \times 4$ cube, the same granularity the Google TPU cluster uses, leaves only the surface ports of each cube reconfigurable and yields just 8 reconfigurable cubes across the whole torus. With more OCSes, each carrying more ports, the operator can partition the torus into smaller hardwired cubes, for example $2 \times 2 \times 2$. Smaller cubes mean more reconfigurable cubes and more surface links under OCS control, so RFold can

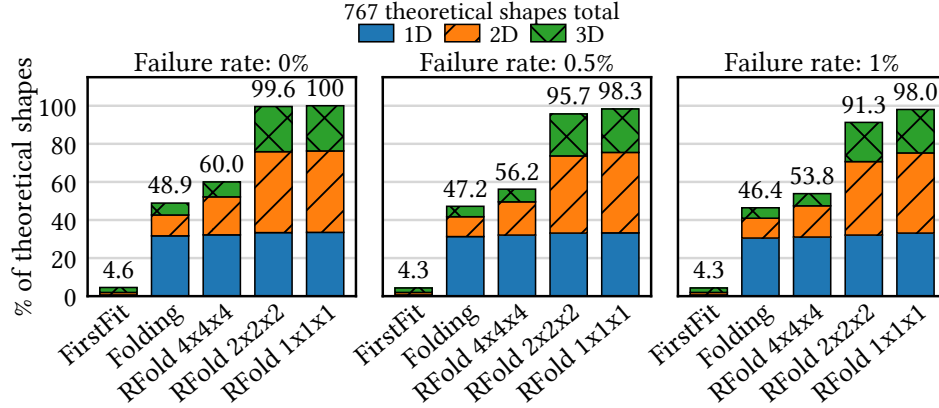


Figure 5.10: Shape placeability as a function of the reconfiguration cube size.

wire the free nodes together in more variations and match a wider range of job shapes.

In the most extreme case, each XPU becomes its own $1 \times 1 \times 1$ cube and every link in the fabric is reconfigurable. The torus then degenerates into a fully reconfigurable mesh, in which any XPU can be wired directly to any other. This gives RFold the maximum possible placeability, since any shape can be realized wherever free capacity exists.

This spectrum is of course bounded by hardware. The OCS port count caps how many cubes a torus can be split into, because every reconfigurable link consumes optical ports, and a fine partition such as $1 \times 1 \times 1$ demands far more ports than a coarse one. For the study in this subsection we set that constraint aside and assume an unlimited OCS port budget, so the cube can be made as small as $1 \times 1 \times 1$ and the full reconfigurability spectrum is reachable.

To trace out this spectrum we fix FIFO admission and replay the same trace under a sweep of cube sizes. We compare FirstFit against RFold at cube sizes of $1 \times 1 \times 1$, $2 \times 2 \times 2$, and $4 \times 4 \times 4$ (the default), increasing the cube edge in powers of two. On this $8 \times 8 \times 8$ torus the $4 \times 4 \times 4$ cube is the largest configuration that leaves any surface left to reconfigure; a larger cube would turn the reconfigurable torus into a static one. FirstFit, which never reconfigures, serves as the fixed-topology reference across the sweep.

Figure 5.9 plots the resulting average JCT against offered load for each cube size. The JCT of each curve is normalized to the ideal JCT at that load, so a value of $1 \times$ marks a placement that adds no queueing or slowdown beyond what the workload itself demands. The $1 \times 1 \times 1$ cube is the fully reconfigurable mesh and already attains this ideal at every load, so the RFold ($b=1$) curve is a flat horizontal line at $1 \times$ and serves as the best-case reference for the sweep.

Against that reference the configurations separate cleanly, and the ordering follows reconfigurability directly. Ranked by the load each can sustain before its JCT begins to climb, FirstFit comes lowest, then RFold at $4 \times 4 \times 4$, then $2 \times 2 \times 2$, and finally $1 \times 1 \times 1$. As the cube shrinks the fabric exposes more surface links to the OCSes, so RFold can wire the free nodes into more shapes and pack more jobs into the same torus, and the cluster absorbs a higher offered load before queueing sets in. The gains are large: with $2 \times 2 \times 2$ cubes RFold sustains roughly 80% offered load before its JCT departs from the ideal, well beyond the point at which FirstFit and the

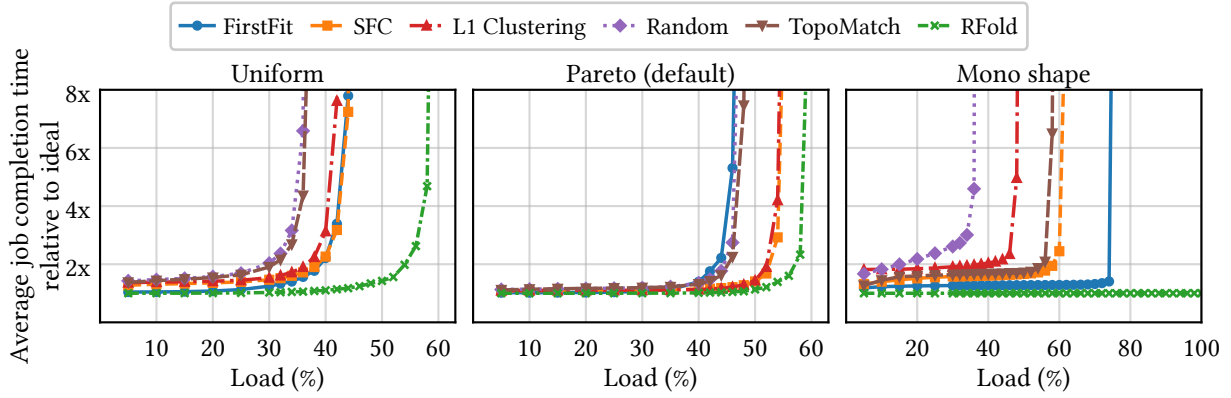


Figure 5.11: Average JCT versus offered load under three workload patterns: a uniform job-size distribution, the default small-heavy Pareto distribution, and a mono-shape workload in which every job requests the same $4 \times 2 \times 1$ shape. All runs use FIFO admission.

default $4 \times 4 \times 4$ cube begin to degrade.

Beyond load and failure, reconfigurability brings a further benefit: it widens the set of job shapes the cluster can support. The trace we have used so far draws from only a subset of the shapes whose total size is at most 512 XPU. Those shapes are representative of what a static $8 \times 8 \times 8$ torus is asked to run, but they are far from the whole set that RFold can accommodate. By tuning the reconfigurability level, RFold can in fact place almost every shape that is theoretically expressible on this many XPU.

Figure 5.10 measures this directly. Here we set load and admission aside and ask a purely structural question: attempting to place a single job into an empty torus, can each candidate shape fit at all? We enumerate every shape of the form $A \times B \times C$ with $A \cdot B \cdot C \leq 512$, where each of A , B , and C is either 1 or an even number, which yields 767 distinct shapes in total. The metric is simply the fraction of these 767 shapes that a policy can place.

The differences are stark. FirstFit, confined to the static torus, can place only 4.6% of the shapes. Folding alone, without any reconfiguration, raises this to 48.9%. RFold with the default $4 \times 4 \times 4$ cube improves it further to 60%, and as the cube size shrinks RFold eventually places all 767 shapes, reaching 100% placeability. The breakdown within each bar is equally telling. Folding is what rescues the 1D shapes, while reconfigurability is what unlocks the 2D and 3D shapes, precisely the shapes a static torus struggles to host.

Failures erode these numbers, since a torus with holes can no longer host every shape it otherwise could. Reconfigurability blunts that damage as well. Because a failed node is confined to its own cube and does not disturb the wiring of the others, the loss of placeable shapes stays local to the affected cube rather than rippling across the fabric, so RFold’s shape coverage degrades only gently as failures accumulate.

5.4.6 More workload patterns

The evaluation so far has driven the cluster with a single workload pattern, the small-heavy, Pareto-like distribution that the public traces of §5.4.1 exhibit. In theory, clusters could face

various workloads whose jobs differ in size and shape, and a policy that wins on one pattern may not win on another. This subsection therefore varies the workload pattern to build a fuller picture of how the different characteristics of a workload affect placement performance. We compare three workloads, holding FIFO admission and the load sweep fixed throughout so that the workload is the only variable. The Pareto workload is the default trace used everywhere else in this chapter, in which job sizes follow a truncated Pareto distribution over 1 to 128 ranks and each job draws its shape uniformly from the legal factorizations of its size. The uniform workload keeps that same shape rule but draws sizes from a uniform distribution over the same range, so it admits many more large jobs than Pareto. The mono-shape workload instead fixes every job to a single $4 \times 2 \times 1$ shape.

The uniform workload isolates the effect of the size distribution, since it shares the Pareto workload’s shape rule and differs only in how much weight it places on large jobs. The mono-shape workload isolates something different. Because sixty-four $4 \times 2 \times 1$ jobs tile the 512-XPU torus exactly, with no XPU left over, it packs perfectly and removes fragmentation from the comparison altogether; any gap a policy shows here is not a packing gap but the price its placements pay in communication efficiency, which exposes how much each policy loses to the static topology alone and how completely RFold’s reconfiguration erases that loss. Figure 5.11 reports the resulting average JCT for all three patterns.

Figure 5.11 shows that RFold is the only policy that consistently delays the sharp JCT blow-up as offered load rises, and that its margin over the baselines widens as the workload grows harder to pack. On the uniform workload, where medium and large jobs are common, RFold holds average JCT below $2\times$ ideal through 54% load, whereas the best non-RFold policy crosses $2\times$ just after 38%. The gap is starkest in the knee of the curve. At 50% load RFold sits at just $1.42\times$ ideal while every other policy has exploded beyond the plotted range. On the default small-heavy Pareto workload the baselines fare better, because most jobs are small enough to slot into whatever fragments the torus has left, yet RFold keeps the lead. It stays below $2\times$ through 56% load against 52% for the strongest baselines, SFC and L1 Clustering, and at 60% load it reaches $14.2\times$ ideal where the best baseline has already climbed to $63.7\times$.

Why do the baselines gain so unevenly when the workload softens from uniform to Pareto? Pareto is the easier of the two traces, since it swaps many medium and large jobs for small ones, but that ease reaches each policy only through its own bottleneck, and the two baseline families are bottlenecked by different quantities. FirstFit is bottlenecked by the tail. Its jobs are not slow, because its uncontended service time is near ideal in both traces, $1.04\times$ under uniform and $1.02\times$ under Pareto. What hurts it is placement stalls, because a large job needs one contiguous box, and under FIFO an unplaceable large job blocks the whole queue until enough neighbors drain. That cost is set by the largest jobs in the mix, not the average one. Pareto shrinks the mean job from 36.7 to 9.2 ranks but keeps sending jobs of up to 128 ranks, and 2.6% of arrivals exceed 64. Each of those rare giants is as hard to place as it was under uniform, so the severity of a blocking event does not change and only its frequency falls. The lower frequency sharply reduces the damage past the knee, the load at which average JCT turns sharply upward, cutting queue wait from 1469s to 375s at 50% load, but the knee itself barely moves, from roughly 44% to 46% load, because an occasional giant is enough to destabilize a FIFO queue. One hard job in a hundred stalls the line nearly as effectively as one in five.

The scatter policies are bottlenecked by the mean instead, and Pareto slashes the mean. SFC

and L1 Clustering do not require contiguity, so they almost never stall on placement; their tax is communication dilation, as a job’s ranks spread across the torus, its collectives crawl, and the sprawl clogs links for every other job. That tax grows with job size. A scattered 8-rank job is mildly dilated, while a scattered 96-rank job smeared across the 512-XPU torus is ruinous, and under uniform a great deal of job size is in flight at any moment. The uncontended numbers show it directly, as SFC runs at $1.28\times$ ideal under uniform but only $1.08\times$ under Pareto. So when the mass of the distribution slides from 36.7 to 9.2 mean ranks, the scatter policies’ effective capacity genuinely rises and their knee moves with it, from roughly 46% to 56% load. FirstFit and SFC therefore sit at nearly the same knee under uniform, around 44 to 46% load, for opposite reasons, FirstFit from tail-driven head-of-line stalls and SFC from mean-driven communication dilation, and Pareto relieves only the second.

RFold is bottlenecked by neither, so Pareto has nothing to give it. It pays no shape penalty, since it places almost any shape on almost any fragmented free set, and no dilation penalty, since its service time equals the ideal in both traces. Its knee near 58 to 60% load is set by raw capacity alone. Offered load is calibrated against total offered work at ideal service times, and that calibration is identical in the two traces by construction, because Pareto’s jobs are smaller but proportionally more numerous. A policy already bounded only by whether free hardware exists cannot gain from smaller jobs when the total work arriving per second has not changed; it already rests against the shared ceiling that the other two families are climbing toward from below. The whole picture compresses to a single rule. Each policy’s knee is roughly the shared capacity ceiling divided by that policy’s own inefficiency factor, which is tail-frequency-driven for FirstFit, mean-size-driven for the scatter policies, and near one for RFold. Moving from uniform to Pareto changes the mean sharply and the existence of the tail not at all, so the scatter policies gain, FirstFit barely gains, and RFold has no inefficiency left to reclaim.

The mono-shape workload is the clearest case of all, because it strips away the fragmentation that the other two patterns mix into the result. Since a $4 \times 2 \times 1$ job tiles the torus exactly, FirstFit packs it with no waste at all. Its queue wait is zero all the way to 74% load, so placement is not what holds it back. What holds it back is that every job it runs is about 30% slower than the same job under RFold. With no queueing in either system, FirstFit’s median JCT is 0.87s against RFold’s 0.61s. The cause is the static torus. A four-node segment of an eight-node ring is an open line rather than a closed ring, as §5.1.2 describes, so the ring collective’s wrap-around step must travel three hops back across the fabric instead of one and the ring’s effective bandwidth falls to match. The penalty deepens under concurrency, from 0.80s to 0.89s, as neighboring jobs’ routes begin to share links. RFold’s reconfiguration removes it entirely. By rewiring the OCS it gives every placed job a true closed one-hop ring, so its service time equals the ideal in every row of the sweep.

The knee position then follows from arithmetic. Offered load ρ is calibrated against ideal service times, so a policy whose jobs each run about $1.3\times$ longer operates at an effective utilization near 1.3ρ , which reaches one at $\rho \approx 0.75$. The data land exactly there. FirstFit’s queue wait is zero at 74% load, jumps to 46s at 76%, and from there climbs linearly in the manner of a saturated queue drained by a finite trace, 101s, 153s, 203s, and on to 563s at full load. FirstFit does not fail to place jobs near 78% load; it runs out of capacity, because every job it runs spends roughly a third of the machine’s communication budget and at that offered load no slack remains to absorb the tax. This is why its average JCT reaches $274.7\times$ the ideal at full load. RFold pays no such

tax. Its service time is the ideal, its effective utilization is exactly ρ , and it holds flat at 0.61s until it grazes saturation near $\rho \approx 1.0$, where it rises to 1.13s in step with the ideal curve itself. The remaining baselines tell the same story with a heavier tax, and each one’s knee falls close to the reciprocal of its effective communication slowdown. SFC, about $1.32\times$ slower even uncontended and worse under contention, knees near 62% load, TopoMatch near 58%, L1 Clustering near 48%, and Random near 36%. The mono-shape sweep thus reduces the whole comparison to one number per policy, the fraction of communication capacity its placements preserve, which is the placement-dependent capacity that reconfiguration exists to reclaim.

Taken together, the three patterns show that RFold’s advantage does not rest on the small-heavy size mix of the public traces. Whenever a workload’s size or shape distribution drives a static placer to either fragment the torus or run its jobs on open, low-bandwidth rings, folding and reconfiguration recover the lost capacity, and the harder the pattern is for a fixed topology, the larger that recovery becomes.

5.5 Related work

Topology reconfiguration. The idea of treating the interconnect as a mutable resource has gained traction as optical switching has matured. Google’s TPU v4 supercomputer [113] places optical circuit switches (OCSes) between $4\times 4\times 4$ cubes so that a large torus can be sliced into per-job sub-tori at allocation time, and Mission Apollo [217] reports landing OCS at data center scale. The same OCS layer is used defensively in Resiliency at Scale [254], which reconfigures around failed cubes to keep the machine available. In the ML setting, TopoOpt [228] co-optimizes the network topology and the parallelization strategy of a single training job on a reconfigurable optical fabric, rewiring an optical patch panel once per job and holding that topology fixed for the job’s duration. The HPC community has pursued the same idea on other fabrics: GRAP [58] allocates groups on a reconfigurable Dragonfly network, and bandwidth steering with silicon nanophotonics [160] rewires the lower layers of a hierarchical topology to recover from fragmentation. A parallel line of hardware work pushes reconfigurability further down the stack, from the optical circuit switch that PipSwitch [47] builds out of programmable integrated photonics to MorphLux [128], a programmable photonic fabric that rewires the links between the accelerators inside a single server. MorphLux shares our target setting most closely: it augments torus-based ML data centers to raise the bandwidth of tenant allocations and to cut compute fragmentation, and it can swap a failed accelerator for a healthy one in 1.2 seconds. RFold builds on this hardware substrate but differs in how it uses it. TPU v4 reconfigures coarsely and roughly once per job, carving out a fixed sub-torus at admission, and TopoOpt optimizes the fabric for one job in isolation; RFold instead treats reconfiguration as a fine-grained, multi-job placement primitive, folding each job’s shape and rewiring cubes so that many jobs share the fabric contention-free. MorphLux attacks fragmentation from below, rewiring within a server so that the accelerators of an allocation behave as though they were adjacent; RFold attacks it from above, folding the job’s shape onto the cluster-scale torus and rewiring whole cubes. The two are complementary, since they reconfigure at different layers of the same machine.

Direct-connect topologies. A second line of work leaves the fabric statically wired and shapes it to the workload at design time instead. HammingMesh [98] joins board-local 2D meshes of

accelerators with row and column packet switches, trading global bandwidth for cost while keeping enough allocation freedom to place jobs of many shapes, and Aquila [77] builds a low-latency cell-switched Dragonfly from a custom ToR-in-NIC ASIC. Others wire the accelerators together more directly: UB-Mesh [138] exploits the locality of LLM traffic with a recursive nD-FullMesh of passive electrical links, switch-less Dragonfly on wafers [59] realizes the topology through wafer-scale integration, and Dojo [210] and Bow [78] connect their chips in a fixed mesh and torus, respectively. LIBRA [233] is the design-time counterpart to these architectures, apportioning bandwidth across the dimensions of a multi-dimensional training fabric to suit a target workload. None of these fabrics can be rewired once deployed, so a job’s shape has to be accommodated by allocation alone, which is precisely the constraint under which the static baselines of §5.4 operate. RFold is complementary to this line of work rather than opposed to it: where these designs invest in the wiring, RFold assumes an OCS layer and spends it on placement.

Job admission. A large body of schedulers improves job performance in multi-tenant clusters by deciding when to start each job and how many accelerators to grant it. Optimus [179] and Gandiva [235] elastically resize jobs to improve JCT, Tiresias [82] preempts to reduce JCT, and Themis [149] preempts for fairness. Synergy [164], Gavel [170], Sia [208], Lyra [133], and Muri [247] schedule heterogeneous or multi-dimensional resources but treat the interconnect as a flat pool. General-purpose cluster managers such as Borg [212, 221], TetriSched [213], YARN [220], Mesos [96], Kubernetes [32], Sparrow [175], and SLURM [238] pack several resource types, e.g. CPU, memory, and disk, but assume the big-switch abstraction of a Clos fabric, under which any set of nodes is equivalent and placement is bandwidth-oblivious. These policies are orthogonal to RFold and complement it. They decide which jobs to admit, while RFold decides how to lay an admitted job onto a torus. As §5.4.2 shows, RFold’s advantage persists across FIFO, Backfill, and SWF admission, so it can sit beneath any of them.

Shape-aware placement and packing. On a torus the placement decision dominates, and it has a long lineage in high-performance computing. Classic allocators embed a job into a static torus or mesh to keep its communicating ranks close: FirstFit for 3D tori [41], noncontiguous allocation [146], space-filling-curve mapping [130] (the default in SLURM via a Hilbert curve [6]), contention-minimizing clustering [148], and process mapping with TopoMatch [108]. A broad literature formulates placement as graph embedding [7, 53, 239, 240] or as topology-aware heuristics [26, 28, 46, 68, 72, 97, 214, 252], generally aiming to reduce the average hop count a message travels, a goal motivated by measurements showing that jobs sharing a torus interfere and slow one another down [27]. Locality-aware schedulers such as that of Yang et al. [236] then weigh the per-job benefit of a compact allocation against the utilization it costs the system. A related thread improves cluster utilization directly, through packing, defragmentation, and backfill [17, 134, 167, 211] and through classical bin-packing analysis [49, 52, 67]. In the ML setting specifically, Cassini [189] and Crux [33] schedule communication to relieve network contention on GPU clusters. All of this work assumes a fixed interconnect, which forces the fragmentation-versus-contention trade-off at the heart of this chapter: an allocator can honor a job’s shape and risk leaving it unplaceable, or scatter the job and pay contention, but it cannot escape both. RFold can, because it reshapes the topology as well as the job. Its design (§5.3) can be read as a constrained form of graph embedding, but one tuned to ML rather than to HPC. Although ML and HPC workloads both decompose computation across nodes that exchange neighbor traffic and global collectives [126, 191, 209], their communication differs in kind. ML jobs have

deterministic, regular communication volumes and cuboid shapes [12, 135], whereas many HPC codes change message sizes and communication partners at runtime [107, 110, 194, 246] and produce unstructured allocation shapes [106, 176, 237]. RFold exploits exactly the regularity of the ML setting that a general graph embedder cannot assume.

Workload analysis. Our workload model is grounded in public measurements of production ML and GPU clusters. Philly [109], MLaaS [229], Helios [104], and Acme [105] consistently report heavy-tailed job sizes and durations, in which most jobs ask for a few accelerators and run briefly while a long tail of large ones dominates consumption, alongside low utilization, frequent queueing, and gang-scheduled training. Other traces fill gaps: Borg 2019 [212] lacks GPU information, ATLAS [11] is CPU-only, MIT Supercloud [132, 195] covers an academic GPU cluster, and the ALCF study [177] reports torus-based HPC systems whose jobs choose from pre-configured size buckets. None comes from a reconfigurable torus-based ML cluster, and most describe Clos fabrics where placement carries no bandwidth penalty. We therefore distill their size, shape, and load statistics into the synthetic workloads of §5.4.1 rather than replay any trace verbatim; §5.4.6 tests how sensitive our conclusions are to them.

5.6 Conclusion

This chapter addressed job placement in torus-based ML clusters, where a job’s parallelization strategy fixes the shape its allocation must take and a statically wired torus cannot always supply that shape. The penalty for getting this wrong is large and, worse, invisible to the scheduler at the moment it must decide: a job whose XPUs are scattered forwards its collective traffic through intermediate nodes and contends with its neighbors for link bandwidth, which we measured at up to $2.5 \times - 3 \times$ slower than the same job placed well (§5.2). No allocator confined to a fixed fabric can avoid paying it. Honoring a job’s shape strands idle accelerators until a matching region happens to appear, and scattering the job over whatever is free buys placement at the price of contention, so such an allocator can trade one cost against the other but cannot escape both. Our contribution is RFold, which escapes both by treating the topology as a parameter of the allocation rather than a constraint on it. RFold enumerates the legitimate shape variants of a job, folds each onto the cluster, reconfigures the OCS layer so that the folded shape is realized on the fabric, either as one solid block or as tiles glued together across cubes by optical circuits, and ranks the resulting plans. On a simulated 512-XPU reconfigurable torus, RFold delivers the lowest average job completion time at every load in the sweep and sustains the highest load before saturating, with its knee near 58% offered load under FIFO admission against 54% for the best shape-unaware policy and 44% for FirstFit. It also turns more of the cluster into finished work than any baseline, plateauing near 57% effective utilization against 45% for FirstFit and 52% for the best scatter policy.

More broadly, this chapter argues that job placement and topology cannot be designed in isolation once the fabric can be rewired. Every fixed-topology allocator faces that choice only because the network is taken as given: the job must bend to the fabric because the fabric cannot bend to the job. Letting both move dissolves the trade-off rather than rebalancing it. What makes that practical is that ML training announces its communication pattern before it starts, so the scheduler knows the shape it must build at the moment it has to decide, and never has to guess.

Seen against the thesis's arc, RFold is the redesign the thesis statement asks of job scheduling, and the last of the three. The transport of Chapter 3 rode a fixed schedule oblivious to demand, and the traffic engineering of Chapter 4 made a general-purpose fabric demand-aware; RFold goes even further, exploiting a workload whose communication pattern is known before the job starts. That foreknowledge is what lets it co-adapt the job and the topology at once, and so escape the fragmentation-versus-contention trade-off that forces every fixed-topology allocator to sacrifice one for the other. Reconfigurability, the same property that unsettled transport and traffic engineering, here becomes the resource that resolves the trade-off. That is the pattern the three redesigns share. Reconfigurability is what breaks the assumption each layer was built on, and it is also the resource each redesign spends to recover what was lost: transport had to be told when the fabric changed, traffic engineering had to account for what the hardware could actually install, and scheduling had to reshape the network to fit the job. In each case a network the layer had assumed to be fixed had quietly become a variable, and the redesign amounted to treating it as one.

Chapter 6

Conclusion

For many years, scaling the network meant building a larger packet-switched Clos fabric from faster switch chips, preserving the abstraction that the cluster behaved like one large, uniform, stable switch. That abstraction was extraordinarily productive: it let transports, TE systems, schedulers, and applications evolve while treating the network as a fixed substrate. But the forces that now dominate data center growth have made that substrate too expensive to scale by brute force alone: the bandwidth wall in switch silicon, the energy and cost of ever larger packet fabrics, and the explosive communication demands of distributed ML.

RDCNs offer a different way forward. By steering light directly through optical circuits, they deliver high bandwidth at far lower cost and energy than EPSes, and by rewiring those circuits at runtime, they spend network capacity where demand actually exists. The price of that efficiency is that the old abstraction breaks: the network is no longer uniformly provisioned between every pair of endpoints, and the topology is no longer stable over the lifetime of a flow, a TE epoch, or a job. This thesis has argued that those changes cannot be hidden entirely below the software stack. The performance promised by optical reconfigurability is recovered only when the software above it treats reconfiguration as a first-class design lever.

The three systems in this thesis make that argument concrete at three layers of the stack. At the transport layer, Chapter 3 showed that a conventional congestion-control loop loses efficiency when a connection's path alternates between TDNs with different bandwidths and latencies. Time-Division TCP maintains independent congestion-control state for each network condition and uses switch notifications to learn when the active condition changes, preserving a single connection abstraction across reconfigurations rather than rediscovering the network after each one. At the routing and TE layer, Chapter 4 showed that replacing a Clos spine with a spine-free reconfigurable fabric changes not only the topology but also the feasibility of implementing the control plane's decisions in real switch hardware. PreciseTE makes hardware realization part of the problem rather than an afterthought: a good fractional plan must still be installed through finite group tables and integer weights under heterogeneous hardware limits. It reduces group-table usage while preserving the intent of the TE solution. At the scheduling layer, Chapter 5 showed that in ML clusters, reconfigurability changes the boundary between compute placement and network topology. RFold treats the job shape and the topology as co-adaptable objects: it explores equivalent folded forms of a job's communication pattern while using optical reconfiguration to assemble a topology that can host that form without destructive contention.

Across all three, the same pattern repeats: the packet-switched assumption fails, a conventional layer loses performance, and the loss can be reclaimed by exposing the structure of reconfiguration to the software that makes decisions at that layer. This pattern also reframes the limitations of OCS, such as the lack of buffering, of per-packet forwarding, and of statistical multiplexing inside the switch, together with non-negligible reconfiguration delay. Those limitations are not merely obstacles to be papered over; they are design constraints that reveal where the software stack had been silently relying on uniformity and stability. The path to efficiency is not to make the optical fabric impersonate a packet fabric, but to let software exploit the properties that made the optical fabric attractive in the first place.

This thesis set out to confront the data center scaling crisis: the widening gap, charted in Chapter 1, between the bandwidth a data center demands and the bandwidth that ever faster packet switches can affordably supply. Its answer was not a faster switch but a different kind of network, one that reconfigures itself to place bandwidth where traffic actually flows, and its central claim was that such a network pays off only when the software above it is redesigned to treat reconfiguration as a first-class lever. The three systems make that claim concrete, and together they establish that reconfigurability is not a curiosity confined to one layer but a property the whole stack can be taught to exploit, from transport to traffic engineering to job placement.

Has the crisis, then, been solved? Not in the sense of being closed, for the demand it springs from keeps climbing and no single thesis could outrun it. What this thesis does settle is narrower and, we believe, more durable: reconfigurable optics are a workable substrate for real data centers, and the performance one might fear to lose in adopting them can be reclaimed in software. That result, though, is established under a particular hardware regime. Every design here works within the twin limits of today's OCS, slow switching and small radix, and leans on one of the two standard responses to them: placing the OCS above the ToR so that each optical port carries an aggregate rather than a single server, as in Chapters 4 and 5, or manufacturing all-to-all connectivity in time by cycling a rotor through a fixed schedule, as in Chapter 3. Both keep the optical layer slow and logically single-hop, and both were chosen precisely because the hardware permits nothing more.

That is also where the thesis stops and its remaining gap begins. The same two limits that shaped these designs also bound them: a fabric that rewires only every few minutes can chase only traffic that holds still that long, and a fabric confined to a single optical hop can serve only a slice of the data center's traffic while packet switches carry the rest. Relaxing those limits, driving reconfiguration three to six orders of magnitude faster and composing many switches into a fabric larger than any one of them, would open uses that slow, single-hop optics cannot reach. The remainder of this chapter looks into that gap. It turns from the *slow* reconfiguration the thesis has studied to *fast* reconfiguration at the nanosecond-to-microsecond scale, first asking what new use cases such speed would unlock (Section 6.1) and then examining the obstacles that stand in the way (Section 6.2): that faster reconfiguration is an end-to-end problem spanning hardware and software, and that an all-optical data center demands hierarchical, multi-hop optical switches with no buffers to hide imprecise timing.

6.1 New use cases enabled by faster reconfiguration

Each order of magnitude shaved off the reconfiguration time changes the granularity of traffic that an OCS fabric can profitably chase. A millisecond fabric can only track traffic that is stable over many milliseconds, such as an aggregated traffic matrix or a job-length communication pattern. A microsecond or nanosecond fabric can track traffic that shifts far more quickly, and that capability opens up use cases that slow reconfiguration cannot reach. We sketch four, each of which benefits from driving reconfiguration into the microsecond-to-nanosecond regime: adapting the topology to the collective a job is running at each instant, healing a job around a failed accelerator with minimal stall, making bandwidth fluid between a machine’s scale-up and scale-out domains, and, most ambitiously, carrying every flow of a data center across an all-optical fabric built from tiers of OCSes.

6.1.1 Adapting the topology to collective communication

Dense-model training communicates in deterministic, structured patterns: gradients are combined with a ring- or tree-based all-reduce whose participants and message sizes are fixed before the job begins, and it is this regularity that lets a scheduler choose one good topology and hold it for the job’s entire lifetime, as Chapter 5 does. Mixture-of-Experts (MoE) models [57] introduce a far less predictable pattern alongside it.

An MoE layer replaces a single feed-forward block with many parallel experts and a gating network that, for each token, selects a small subset of the experts to activate. Because the experts are sharded across GPUs, a token emitted by one layer may need to reach an expert that resides on any other GPU in the job, and this routing is data-dependent, recomputed every batch from the gating network’s output. The result is a dynamic all-to-all: a dispatch step scatters each token to its chosen experts and a combine step gathers the results back, and which GPU pairs exchange data, and how much, shifts from batch to batch and from layer to layer instead of being fixed in advance.

No single static topology can serve such a job well, because there is no one wiring that matches an all-to-all whose busy pairs are constantly moving; what the job wants instead is direct, high-bandwidth connectivity between any two GPUs on demand. Supplying that as a static full mesh is infeasible for the port-count reason that recurs throughout this chapter, since a single OCS cannot give every GPU a direct port to every other. The way out is the second of the two responses from the start of this chapter, building the mesh in time rather than in space: a fast OCS can cycle through a precomputed schedule of matchings [152] quickly enough that every GPU pair is directly connected in turn, manufacturing a time-division full mesh that carries the all-to-all. MixNet [139] already reconfigures an optical-electrical fabric at runtime to serve MoE’s shifting communication; driving the reconfiguration faster would let such a schedule track the traffic at the granularity of a single dispatch or combine, close to the pace at which the expert-communication pattern actually changes.

The benefit of faster reconfiguration is not confined to the irregular all-to-all of MoE. Even the deterministic collectives of ordinary training gain from it, because a single training step is not one communication pattern but a sequence of them: a job may perform an all-gather, a reduce-scatter, and an all-reduce in succession, and each of these has its own ideal topology. The scheduler of

Chapter 5 must commit a single topology for a job’s entire lifetime, a compromise across every collective that job will ever run.

Faster reconfiguration replaces that compromise with in-workload reconfiguration that adapts the real-time topology to the collective running at each instant. Because the communication pattern of a deterministic collective is fixed and known in advance, the fabric can be rewired to match it exactly as the step unfolds, devoting all of the available optical bandwidth to the subset of GPUs that are actually exchanging data in the current phase and so accelerating it, then reconfiguring for the next. Rather than adapting the topology to a job once, as in Chapter 5, the fabric adapts it continuously, collective by collective, as the job runs.

This sets a concrete speed requirement. The compute and communication phases of an ML training step alternate on the microsecond-to-millisecond timescale, so to have the right topology ready for an upcoming collective the fabric must reconfigure within that same window. Faster yet is better, because a reconfiguration blackout can be hidden underneath the preceding compute phase: a fabric quick enough to finish rewiring before the computation ends adds no delay to the collective at all, overlapping the cost of reconfiguration with useful work rather than stalling communication to pay it.

6.1.2 Pooling spare accelerators across scale-up domains

Faster reconfiguration helps a training job not only when its communication pattern shifts but when its hardware breaks. The rack-scale accelerator has turned the scale-up domain into a unit of manufacture. NVIDIA’s GB200 NVL72 [174] fuses 72 GPUs into a single NVLink domain over a copper backplane, wiring every GPU to every other through the rack’s NVLink switches so that the 72 behave as one large accelerator. That fusion is fixed when the rack is built: both the size of the domain and the identity of the GPUs in it are set by how the rack is cabled, and neither can change once the rack is on the floor.

A frozen scale-up domain is brittle in exactly the way a large training job cannot afford. The GPUs of a domain are gang-scheduled and run in tight lockstep, so the loss of any one of them degrades or stalls the whole domain: NVLink can route around a dead GPU, but only at reduced bandwidth, and a job that needs the full domain must instead fail the entire rack over to a spare or roll back to a checkpoint. At fleet scale these losses are not rare events but a steady background rate. Meta’s Llama 3 pre-training run on 16,384 H100 GPUs recorded an unexpected interruption roughly every three hours across 54 days, with GPU and HBM failures together accounting for nearly half of them [50]. Because a scale-up domain can neither shrink nor grow, operators buy availability by over-provisioning whole units, holding spare compute trays and even spare racks in reserve. But a spare pinned inside one rack cannot cover a failure in another, so every domain must independently carry enough reserve for its own worst case, and the fleet ends up holding far more idle silicon than its aggregate failure rate would require.

Reconfiguration dissolves that pinning. If the scale-up ports of many racks are brought out to a shared OCS rather than hard-cabled within each rack, the membership of a domain becomes something the network sets rather than something the rack fixes, and a single pool of spare GPUs sitting on the OCS can serve them all. When a GPU fails, the OCS splices a spare from the pool into the degraded domain, restoring its full shape without failing the whole rack over and without every rack hoarding its own reserve. The saving is the same statistical-multiplexing argument that

runs through this thesis: because failures in different racks are largely independent, one shared pool sized to the aggregate failure rate protects the entire fleet at a fraction of the spare count that per-rack provisioning demands, just as placing an OCS above the aggregate in Chapter 4 lets many blocks share capacity that each would otherwise strand.

This is not hypothetical. Google’s TPU supercomputers already use OCS to reconfigure their scale-up fabric around failures, composing a working machine out of healthy building blocks and leaving faulty ones aside, and it is this reconfigurability, as much as the switching speed itself, that lets a TPU pod sustain high availability and share spare capacity across the fabric rather than pinning it to each machine [112, 255]. Establishing the pool at all asks little of the fast regime, since failures themselves arrive only every few hours, and a slow, millisecond MEMS OCS is quick enough to serve it. But the speed of the reconfiguration governs how much each failure costs the job that hits it. A gang-scheduled domain cannot make progress while one of its GPUs is missing, so from the instant a GPU fails the whole job is blocked until the OCS has spliced in a spare, the new link has re-locked, and the failed GPU’s state has been reloaded. Every part of that recovery goes faster as the fabric does, so driving reconfiguration from milliseconds toward microseconds directly shrinks the stall, turning each failure from a long pause into a brief hiccup. The benefit compounds with the failure rate: the more often a large job is interrupted, the more of its total runtime rides on how quickly the network can heal. Fast reconfiguration is thus not what makes pooling possible, but what makes it cheap.

6.1.3 Unified scale-up and scale-out bandwidth

The same reconfigurability that reshapes which accelerators a job runs on can also reshape how much bandwidth reaches each of them. The bottleneck in AI infrastructure has shifted from compute to I/O. Adding raw compute is comparatively easy, but keeping that compute busy by feeding it data is hard, and the limiter is off-chip bandwidth. The difficulty is fundamental: an accelerator’s compute throughput scales with the area of its silicon, whereas the bandwidth it can escape off-chip scales only with the die’s perimeter, because the high-speed I/O lives on the chip’s edge, or “shoreline.” Doubling a die’s linear dimension roughly quadruples its compute but only doubles its I/O, so each larger and faster generation of accelerator carries proportionally less bandwidth per unit of compute, and data movement rather than arithmetic increasingly governs end-to-end performance.

Today’s AI systems make this scarce I/O even harder to use by partitioning it into two separate domains. A tightly coupled scale-up domain, such as NVLink within a node or rack, connects a small group of accelerators at very high bandwidth, while a scale-out domain of PCIe to a NIC and then Ethernet or InfiniBand connects nodes across the data center at far lower bandwidth. Because the scale-up domain carries the most bandwidth-hungry traffic, the great majority of an accelerator’s I/O is wired to it: an NVIDIA H100 [173] pairs 900 GB/s (7.2 Tb/s) of NVLink scale-up bandwidth with a 400 Gb/s network port, and a GB200 [174] pairs 1.8 TB/s (14.4 Tb/s) of NVLink with an 800 Gb/s port, an 18-to-1 split in each case.

This fixed split imposes two costs. First, the moment a GPU must exchange data with a GPU in another node it falls off a bandwidth cliff, throttled from the scale-up rate down to the far smaller scale-out rate, so applications are forced to be aware of the boundary and to map their parallelism so that the heaviest communication stays inside a scale-up domain, a persistent source

of complexity in distributed-training software. Second, the ratio between the two domains is frozen when the machine is built, so it cannot follow a workload whose demand shifts between them by phase: tensor- and expert-parallel communication stresses the scale-up domain while data-parallel all-reduce stresses the scale-out domain, and whichever domain a phase does not use sits idle while the other saturates.

Fast reconfiguration offers a way to dissolve the boundary. If an accelerator’s I/O, or with co-packaged optics [162, 222] its optical ports, is connected to an OCS inside the server or rack, the division between scale-up and scale-out need no longer be hard-wired. The OCS can steer most of the ports toward nearby accelerators during a scale-up-heavy phase and then re-steer them toward the rack’s uplinks when the workload turns scale-out-heavy, making bandwidth fluid between the two domains rather than statically partitioned. To the application the two networks then begin to behave as one unified pool [70, 120, 201], which both recovers the capacity that a fixed split strands and removes the boundary that training software must otherwise work around. The reconfiguration must be fast because the phases it chases turn over within a single training step, the regime that slow MEMS switching cannot reach.

The idea generalizes beyond AI accelerators. Any machine that today hard-partitions its internal bandwidth can gain from making that partition reconfigurable. A general-purpose server could attach its processors, memory, and storage to a central OCS and allocate bandwidth among them on demand, giving a memory-bound phase more processor-to-memory bandwidth and a communication-bound phase more processor-to-processor bandwidth, approaching resource disaggregation within a single machine and keeping a fixed I/O budget fully utilized.

6.1.4 Scaling an all-optical data center

Every use case so far has kept a packet fabric underneath the optical layer, using the OCS for one slice of the traffic while conventional switches carry the rest: the aggregated matrix in Chapter 4, the rotor mesh in Chapter 3, the scale-up domain in the sections above. The most ambitious use of faster reconfiguration is to remove that underlay entirely and carry every class of traffic optically, an all-optical data center in which OCSes, and not packet switches, form the whole network. The prize is that the fabric would inherit the properties that make optics attractive in the first place: an energy per bit far below that of a packet switch [120], and transparency to line rate, so that the same switch carries each faster generation of transceiver without itself being replaced.

The obstacle is radix. A single OCS reaches only hundreds to about a thousand ports, far short of the hundreds of thousands of endpoints in a data center, the same port-count limit that has recurred throughout this chapter. The way past it is the one packet networks already use to build large fabrics from finite-radix switches: compose many OCSes into a network of OCSes. Arranged into tiers, switches of radix R reach on the order of R^k endpoints at k tiers, so two or three tiers of thousand-port OCSes already span a data center, at the cost of an end-to-end path that now traverses several optical hops in succession.

Two familiar topologies carry over directly. A fat-tree, or folded-Clos, arrangement stacks tiers of OCSes so that any endpoint reaches any other through a fixed number of optical hops at full bisection bandwidth, mirroring the packet Clos fabrics that data centers are built from today. A dragonfly instead exploits locality to spend fewer long links: endpoints are gathered into groups, a regional OCS wires each group densely so that most traffic stays local and cheap, and a thinner

Table 6.1: The end-to-end latency of a reconfiguration is a sum of four terms spanning hardware and software. Reducing only the OCS device (the first row) leaves the total dominated by the remaining terms. The magnitudes are representative orders of magnitude, not measurements.

Component	Today (MEMS)	Fast-regime target	Chief dependency
OCS hardware switching	ms	ns– μ s	switching technology (Table 2.1)
Transceiver re-locking	μ s	ns	burst-mode receivers, shared clock
Control-plane decision & install	ms	sub- μ s	decentralized or precomputed schedules
End-host reaction	μ s	ns	kernel bypass, NIC offload, edge buffering

global OCS tier connects the groups to one another, so that only the minority of traffic that leaves its group pays for a global hop. Either way the reconfigurability compounds across tiers, giving a fabric that can rearrange not merely which endpoints a single switch connects but which regions of the data center are drawn close together.

Such a fabric leans harder on fast reconfiguration than anything else in this chapter. Where the thesis’s single-hop designs let a slow OCS track only stable, aggregated traffic, an all-optical fabric must carry every flow in the data center, including the bursty, fast-shifting traffic that a packet network absorbs today, so it must rewire at the pace that traffic moves. And because a path now crosses several OCSes, those hops must be reconfigured quickly and in concert for a continuous light path to exist at the instant a burst needs it. That coordination, together with the tight synchronization a bufferless multi-hop fabric demands, is what stands between this vision and a working system, and Section 6.2.2 takes it up as the second of the two challenges this chapter closes on.

6.2 Challenges that must be addressed

The use cases above all presume something the field has not yet delivered: reconfiguration that is fast from end to end, and optical fabrics that stay all-optical across an entire data center. Two challenges stand between today’s slow, single-hop RDCNs and that future, and they are the natural continuation of the co-design theme this thesis has pursued.

6.2.1 Faster reconfiguration is an end-to-end problem

The reconfiguration time usually quoted for an OCS is only the switching element’s own settling time, but the latency that actually matters to an application is the interval from the decision to rewire to the moment traffic flows on the new topology. That interval is a sum of four terms that span both hardware and software, shown in Table 6.1, and progress on any one of them alone is wasted unless the others keep pace.

OCS hardware switching. The first term is the physical time for the switching element to settle into its new state: a MEMS mirror to tilt, a Mach-Zehnder interferometer to switch, or a laser to retune to a new wavelength. This is the millisecond figure for the MEMS switches in production today and the nanosecond-to-microsecond figure that silicon-photonics and metasurface devices target (Table 2.1), and it is the term that most optical-switching research works to reduce.

Transceiver re-locking. The second term is easy to overlook and can dominate the first. When connectivity changes, the receiver at the far end of a new circuit sees a signal from a different transmitter, with a different phase and a slightly different frequency, and its clock-and-data-recovery and adaptive-equalization circuitry must re-lock before a single bit can be decoded. At the high-order modulation used at modern line rates this retraining can itself take microseconds, imposing a guard time [18] on every reconfiguration during which the circuit is physically up but carries no usable data. Shrinking it requires physical-layer techniques such as burst-mode receivers that lock within a few symbols and a shared frequency reference that receivers need not re-acquire.

Control-plane software. The third term is the time for the control plane to collect the current demand, compute a new topology or matching, and disseminate it to the switches. A single round trip to a centralized controller in a large data center is already on the order of microseconds, so demand-aware centralized control cannot close this loop at fast cadence. This is why the fastest research fabrics are demand-oblivious [18, 152, 155], cycling through precomputed schedules rather than reacting to live demand; recovering demand-awareness at these speeds, through decentralized schedulers or matching engines embedded in the switches, is an open problem. It also inherits the forwarding-table constraint of Chapter 4: even when a good plan is computed, installing it takes time, so a fast fabric must select among preinstalled routes rather than rewrite tables on every reconfiguration.

End-host software stack. The fourth term is the time for the end host to notice the change and redirect its traffic onto the new circuit. The kernel network stack and interrupt path operate in the regime of “killer microseconds” [22], too coarse to react on a per-microsecond basis. Closing the gap requires pushing the reconfiguration-aware logic out of the general-purpose kernel and into hardware or specialized software: kernel-bypass data paths, NIC offload of the circuit-aware decision, and per-destination queues at the network edge [141, 201] that stand ready to burst the instant a circuit opens. It also requires transport that is proactive rather than reactive [90, 166], pacing transmissions against the known circuit schedule instead of probing for a path that may vanish before the probe returns, extending the reconfiguration-aware transport of Chapter 3 toward circuit lifetimes shorter than a round trip.

The crucial point is that these four terms compose additively (with overlap), so the end-to-end reconfiguration latency is governed by the slowest of them. Driving the OCS device from milliseconds to nanoseconds buys nothing if transceiver locking, control software, and the host stack remain at microseconds or milliseconds. Fast reconfiguration is therefore not a device milestone but a co-design problem that binds all four subsystems at once, and at a granularity

three orders of magnitude finer than the one this thesis targeted. Each of the thesis’s contributions attacked one of these terms in the slow regime, transport at the host in Chapter 3, control and TE software in Chapter 4, and scheduling in Chapter 5; the work that remains is to attack all four together in the fast regime.

6.2.2 An all-optical data center needs hierarchical, multi-hop switching

The second challenge follows from the port-count limit that recurs throughout this chapter. A single OCS cannot directly connect an entire data center, and the two responses this thesis relies on, aggregating the OCS above the ToR and time-division multiplexing a flat set of rotor switches, both keep the optical layer logically single-hop. The all-optical data center of Section 6.1.4 escapes this limit only by going hierarchical, composing OCSes into the tiers of a Clos or dragonfly so that an end-to-end path traverses several optical hops in succession. Multi-hop OCS raises two problems that single-hop designs are able to avoid.

Coordinated multi-hop reconfiguration. The OCSes along a path must be reconfigured in a coordinated way, so that a continuous end-to-end light path exists at the instant a burst needs to traverse it. It is not enough for each switch to be fast; the schedules of every tier must be aligned, so that the second hop of a path is established in time to receive what the first hop delivers. Computing and coordinating these interlocking schedules across a large multi-tier fabric, and doing so within the tight time budget of fast reconfiguration, is a control problem with no counterpart in the single-hop fabrics studied in this thesis.

Bufferless hops demand precise synchronization. The sharper problem is that an OCS is bufferless. A packet switch stores a packet at every hop and forwards it once the next link is free, absorbing small timing mismatches in its queues. An optical fabric has nowhere to hold data between hops. If a burst arrives at an intermediate OCS whose onward circuit is not yet established, or arrives during that switch’s reconfiguration guard time, it is not queued but simply lost. Multi-hop optical switching therefore requires that every hop along a path be time-aligned to within the reconfiguration guard window, and the tolerance tightens as hops are added and as the switching granularity shrinks toward nanoseconds. This makes fabric-wide clock synchronization a load-bearing requirement rather than a background convenience. Every OCS and every host must share a common notion of time precise enough that a burst launched from its source arrives at each successive hop exactly during that hop’s circuit window, which at nanosecond switching across several tiers can mean sub-nanosecond synchronization maintained across the whole data center. Datacenter time synchronization has advanced quickly, with network-assisted schemes such as DTP [131] and fault-tolerant designs such as Sundial [137] reaching sub-microsecond bounds at scale, and the physical-layer frequency synchronization that fast optical links already need can be made to serve double duty here. But synchronization tight enough to gate multi-hop optical transmission, maintained fabric-wide and robust to failures, remains unsolved. Compounding it, each optical hop attenuates the signal, so a multi-hop all-optical path must also budget for insertion loss and amplification along the way [120, 178], and failure recovery must proceed without the buffers that mask transient loss in a packet fabric. Hierarchical, synchronized, multi-

hop optical switching is thus the crux on which an all-optical data center turns, and it remains largely unexplored.

6.3 Concluding remarks

Both of these directions, the new use cases and the challenges they raise, extend the argument of this thesis rather than departing from it. Where the preceding chapters reclaimed the performance a slow, limited-radix RDCN loses when its broken assumptions of bandwidth uniformity and connectivity stability meet software that was never written for them, faster reconfiguration widens the same opening. It would let the fabric chase traffic it cannot follow today, from the dynamic all-to-all of MoE training and per-collective topology steering, to a unified pool of scale-up and scale-out bandwidth, to a shared optical fabric that pools spare accelerators across racks and splices them in to heal a job around the failures that punctuate large training runs, and ultimately to an all-optical data center built entirely from tiers of OCSes. Reaching that frontier asks the same thing the thesis has asked throughout, only under tighter timing: that reconfiguration speed be treated as an end-to-end property of hardware and software together, not a property of the switch alone, and that we learn to build and synchronize the hierarchical, bufferless, multi-hop optical fabrics such a data center would demand.

The conclusion of this thesis is therefore simple: the DCN is no longer a passive substrate. As the cost of uniform packet-switched capacity rises and the demand for communication continues to grow, the network must become more dynamic, more workload-aware, and more tightly coupled to the software that uses it. OCS provides the physical mechanism for that shift, but the mechanism alone is insufficient. To reshape DCNs with reconfigurability, whether at today’s millisecond timescales or the microsecond and nanosecond timescales to come, we must also reshape the transport protocols, TE systems, and schedulers that decide how that reconfigurability is used.

Bibliography

- [1] FabricEval TE Evaluation Framework, 2024. <https://github.com/shuoshuc/FabricEval>.
- [2] Anubhavnidhi Abhashkumar, Kausik Subramanian, Alexey Andreyev, Hyojeong Kim, Nanda Kishore Salem, Jingyi Yang, Petr Lapukhov, Aditya Akella, and Hongyi Zeng. Running bgp in data centers at scale. In *NSDI*, pages 65–81, 2021.
- [3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review*, 38(4):63–74, 2008. doi: 10.1145/1402958.1402967.
- [4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4): 63–74, 2008.
- [5] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. *Proceedings of NSDI 2010: 7th USENIX Symposium on Networked Systems Design and Implementation*, pages 281–295, 2010.
- [6] Carl Albing, Norm Troullier, Stephen Whalen, Ryan Olson, Joe Glenski, Howard Pritchard, and Hugo Mills. Scalable node allocation for improved performance in regular and anisotropic 3d torus supercomputers. In *Proceedings of the 18th European MPI Users’ Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI’11, page 61–70, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 9783642244483.
- [7] Aleliunas and Rosenberg. On embedding rectangular grids in square grids. *IEEE Transactions on Computers*, C-31(9):907–913, 1982. doi: 10.1109/TC.1982.1676109.
- [8] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM ’10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450302012. doi: 10.1145/1851182.1851192. URL <https://doi.org/10.1145/1851182.1851192>.
- [9] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 503–514, 2014.
- [10] Dario Amodei and Danny Hernandez. AI and compute. OpenAI blog, 2018. <https://openai.com/blog/ai-and-compute>.

[//openai.com/index/ai-and-compute/](https://openai.com/index/ai-and-compute/).

- [11] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 533–546, Boston, MA, July 2018. USENIX Association. ISBN 978-1-939133-01-4. URL <https://www.usenix.org/conference/atc18/presentation/amvrosiadis>.
- [12] Quentin Anthony, Benjamin Michalowicz, Jacob Hatef, Lang Xu, Mustafa Abduljabbar, Aamir Shafi, Hari Subramoni, and Dhabaleswar K. Panda. Demystifying the communication characteristics for distributed transformer models. In *2024 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 57–65, 2024. doi: 10.1109/HOTI63208.2024.00020.
- [13] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing router buffers. *SIGCOMM Comput. Commun. Rev.*, 34(4):281–292, aug 2004. ISSN 0146-4833. doi: 10.1145/1030194.1015499. URL <https://doi.org/10.1145/1030194.1015499>.
- [14] AWS. Amazon EC2 Trn1 Architecture, 2025. <https://awsdocs-neuron.readthedocs-hosted.com/en/v2.23.0/general/arch/neuron-hardware/trn1-arch.html>.
- [15] AWS. Amazon EC2 Trn2 Architecture, 2025. <https://awsdocs-neuron.readthedocs-hosted.com/en/v2.23.0/general/arch/neuron-hardware/trn2-arch.html>.
- [16] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL <https://www.usenix.org/conference/nsdi23/presentation/bai>.
- [17] Yan Baicheng, Yang Zhang, Xiao Limin, Zhou Yi, Wei Bing, and Song Yao. Lpms: A low-cost topology-aware process mapping method for large-scale parallel applications on shared hpc systems. In *Workshop Proceedings of the 48th International Conference on Parallel Processing, ICPP Workshops ’19*, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450371964. doi: 10.1145/3339186.3339208. URL

<https://doi.org/10.1145/3339186.3339208>.

- [18] 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 ACM SIGCOMM 2020 Conference*, SIGCOMM '20, pages 782–797, 2020. doi: 10.1145/3387514.3406221.
- [19] 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. *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 782–797, 2020. doi: 10.1145/3387514.3406221. URL <https://dl.acm.org/doi/10.1145/3387514.3406221>.
- [20] 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. ISBN 9781450379557. doi: 10.1145/3387514.3406221. URL <https://doi.org/10.1145/3387514.3406221>.
- [21] Nikhil Bansal and Mor Harchol-Balter. Analysis of SRPT scheduling: investigating unfairness. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 279–290, 2001. doi: 10.1145/378420.378792.
- [22] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017. doi: 10.1145/3015146.
- [23] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, page 267–280, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450304832. doi: 10.1145/1879141.1879175. URL <https://doi.org/10.1145/1879141.1879175>.
- [24] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the seventh conference on emerging networking experiments and technologies*, pages 1–12, 2011.
- [25] Maciej Besta and Torsten Hoefler. Slim fly: A cost effective low-diameter network topology. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 348–359, 2014. doi: 10.1109/SC.2014.34.
- [26] Abhinav Bhatele and Laxmikant V. Kale. Application-specific topology-aware mapping for three dimensional topologies. In *2008 IEEE International Symposium on Parallel and*

Distributed Processing, pages 1–8, 2008. doi: 10.1109/IPDPS.2008.4536348.

- [27] Abhinav Bhatele, Kathryn Mohror, Steven H. Langer, and Katherine E. Isaacs. There goes the neighborhood: performance degradation due to nearby jobs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.
- [28] Abhinav Bhatele, Nikhil Jain, Katherine E. Isaacs, Ronak Buch, Todd Gamblin, Steven H. Langer, and Laxmikant V. Kale. Optimizing the performance of parallel applications on a 5d torus via task mapping. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, 2014. doi: 10.1109/HiPC.2014.7116706.
- [29] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, jul 2014. ISSN 0146-4833. doi: 10.1145/2656877.2656890. URL <https://doi.org/10.1145/2656877.2656890>.
- [30] Lawrence S. Brakmo, Sean W. O’Malley, and Larry L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications, SIGCOMM ’94*, page 24–35, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 0897916824. doi: 10.1145/190314.190317. URL <https://doi.org/10.1145/190314.190317>.
- [31] Broadcom Inc. Broadcom ships tomahawk 5, industry’s highest bandwidth switch chip to accelerate AI/ML workloads. Press release, 2022. <https://investors.broadcom.com/news-releases/news-release-details/broadcom-ships-tomahawk-5-industrys-highest-bandwidth-switch>.
- [32] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Commun. ACM*, 59(5):50–57, April 2016. ISSN 0001-0782. doi: 10.1145/2890784. URL <https://doi.org/10.1145/2890784>.
- [33] Jiamin Cao, Yu Guan, Kun Qian, Jiaqi Gao, Wencong Xiao, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. Crux: Gpu-efficient communication scheduling for deep learning training. In *Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM ’24*, page 1–15, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706141. doi: 10.1145/3651890.3672239. URL <https://doi.org/10.1145/3651890.3672239>.
- [34] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *ACM Queue*, 14, September-October: 20 – 53, 2016. URL <http://queue.acm.org/detail.cfm?id=3022184>.
- [35] Kai Chen, Ankit Singla, Atul Singh, Kishore Ramachandran, Lei Xu, Yueping Zhang, Xitao Wen, and Yan Chen. Osa: An optical switching architecture for data center networks with unprecedented flexibility. *IEEE/ACM Transactions on Networking*, 22(2):498–511, 2014. doi: 10.1109/TNET.2013.2253120.
- [36] Shawn Chen. Public ml traces, January 2025. URL <https://github.com/shuoshuc/public-ml-traces>.

- [37] Shawn Shuoshuo Chen, Weiyang Wang, Christopher Canel, Srinivasan Seshan, Alex C. Snoeren, and Peter Steenkiste. Time-division tcp for reconfigurable data center networks. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 19–35, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394208. doi: 10.1145/3544216.3544254. URL <https://doi.org/10.1145/3544216.3544254>.
- [38] Shawn Shuoshuo Chen, Keqiang He, Rui Wang, Srinivasan Seshan, and Peter Steenkiste. Precise data center traffic engineering with constrained hardware resources. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 669–690, Santa Clara, CA, 2024. USENIX Association. URL <https://www.usenix.org/conference/nsdi24/presentation/chen-shawn>.
- [39] Shawn Shuoshuo Chen, Daiyaan Arfeen, Minlan Yu, Peter Steenkiste, and Srinivasan Seshan. Toward co-adapting machine learning job shape and cluster topology, 2025. URL <https://arxiv.org/abs/2510.03891>.
- [40] Yuchung Cheng, Neal Cardwell, Nandita Dukkkipati, and Priyaranjan Jha. The RACK-TLP Loss Detection Algorithm for TCP. RFC 8985, February 2021. URL <https://rfc-editor.org/rfc/rfc8985.txt>.
- [41] Hyunseung Choo, Seong-Moo Yoo, and Hee Yong Youn. Processor scheduling and allocation for 3d torus multicomputer systems. *IEEE Trans. Parallel Distrib. Syst.*, 11(5):475–484, May 2000. ISSN 1045-9219. doi: 10.1109/71.852400. URL <https://doi.org/10.1109/71.852400>.
- [42] Cisco. Cisco application centric infrastructure fundamentals, 2015. https://www.cisco.com/c/en/us/td/docs/switches/datacenter/aci/apic/sw/1-x/aci-fundamentals/b_ACI-Fundamentals/m_fundamentals.html.
- [43] Cisco. Cisco nexus 9000 series nx-os verified scalability guide, release 9.3(12), 2023. <https://www.cisco.com/c/en/us/td/docs/switches/datacenter/nexus9000/sw/93x/scalability/guide-9312/cisco-nexus-9000-series-nx-os-verified-scalability-guide-9312.html>.
- [44] Daniele De Sensi, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoefer. An in-depth analysis of the slingshot interconnect. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. ISBN 9781728199986.
- [45] Daniele De Sensi, Tiziano De Matteis, Konstantin Taranov, Salvatore Di Girolamo, Tobias Rahn, and Torsten Hoefer. Noise in the clouds: Influence of network performance variability on application scalability. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, 6(3):1–27, 2022. doi: 10.1145/3570609.
- [46] Mehmet Deveci, Karen D. Devine, Kevin Pedretti, Mark A. Taylor, Sivasankaran Rajamanickam, and Ümit V. Çatalyürek. Geometric mapping of tasks to processors on parallel computers with mesh or torus networks. *IEEE Transactions on Parallel and Distributed*

Systems, 30(9):2018–2032, 2019. doi: 10.1109/TPDS.2019.2900043.

- [47] Eric Ding and Rachee Singh. Pipswitch: A circuit switch using programmable integrated photonics. In *Optical Fiber Communication Conference (OFC) 2025*, page W2A.41. Optica Publishing Group, 2025. doi: 10.1364/OFC.2025.W2A.41. URL <https://opg.optica.org/abstract.cfm?URI=OFC-2025-W2A.41>.
- [48] Advait Dixit, Pawan Prakash, Y. Charlie Hu, and Ramana Rao Kompella. On the impact of packet spraying in data center networks. In *2013 Proceedings IEEE INFOCOM*, pages 2130–2138, 2013. doi: 10.1109/INFOCOM.2013.6567015.
- [49] György Dósa, Rongheng Li, Xin Han, and Zsolt Tuza. Tight absolute bound for first fit decreasing bin-packing: $FFD(L) \leq 11/9 OPT(L) + 6/9$. *Theoretical Computer Science*, 510:13–61, 2013. ISSN 0304-3975. doi: <https://doi.org/10.1016/j.tcs.2013.09.007>. URL <https://www.sciencedirect.com/science/article/pii/S0304397513006774>.
- [50] Abhimanyu Dubey et al. The Llama 3 herd of models, 2024. <https://arxiv.org/abs/2407.21783>.
- [51] Mike Duckett, Jerome Moisan, Tom Anschutz, Diamantis Kourkouvelis, and Peter Arberg. Accommodating a Maximum Transit Unit/Maximum Receive Unit (MTU/MRU) Greater Than 1492 in the Point-to-Point Protocol over Ethernet (PPPoE). RFC 4638, September 2006. URL <https://www.rfc-editor.org/info/rfc4638>.
- [52] Leah Epstein. Two-dimensional online bin packing with rotation. *Theoretical Computer Science*, 411(31):2899–2911, 2010. ISSN 0304-3975. doi: <https://doi.org/10.1016/j.tcs.2010.04.021>. URL <https://www.sciencedirect.com/science/article/pii/S0304397510002161>.
- [53] Weibei Fan, Fu Xiao, Mengjie Lv, Lei Han, and Shui Yu. Efficient fault-tolerant path embedding for 3d torus network using locally faulty blocks. *IEEE Transactions on Computers*, 73(9):2305–2319, 2024. doi: 10.1109/TC.2024.3416695.
- [54] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. *ACM SIGCOMM Computer Communication Review*, 40(4):339–350, 2010. doi: 10.1145/1851275.1851223.
- [55] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. *SIGCOMM Comput. Commun. Rev.*, 40(4):339–350, aug 2010. ISSN 0146-4833. doi: 10.1145/1851275.1851223. URL <https://doi.org/10.1145/1851275.1851223>.
- [56] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference, SIGCOMM ’10*, page 339–350, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450302012. doi: 10.1145/1851182.1851223. URL <https://doi.org/10.1145/1851182.1851223>.

- [57] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022. URL <http://jmlr.org/papers/v23/21-0998.html>.
- [58] Guangnan Feng, Dezun Dong, Shizhen Zhao, and Yutong Lu. GRAP: Group-level resource allocation policy for reconfigurable dragonfly network in HPC. In *Proceedings of the 37th ACM International Conference on Supercomputing (ICS)*, 2023. doi: 10.1145/3577193.3593732.
- [59] Yinxiao Feng and Kaisheng Ma. Switch-less dragonfly on wafers: A scalable interconnection architecture based on wafer-scale integration. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '24*. IEEE Press, 2024. ISBN 9798350352917. doi: 10.1109/SC41406.2024.00102. URL <https://doi.org/10.1109/SC41406.2024.00102>.
- [60] Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. Orion: Google’s software-defined networking control plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.
- [61] Sally Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649, December 2003. URL <https://www.rfc-editor.org/info/rfc3649>.
- [62] Sally Floyd, Jamshid Mahdavi, Matt Mathis, and Dr. Allyn Romanow. TCP Selective Acknowledgment Options. RFC 2018, October 1996. URL <https://rfc-editor.org/rfc/rfc2018.txt>.
- [63] Sally Floyd, Dr. K. K. Ramakrishnan, and David L. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, September 2001. URL <https://rfc-editor.org/rfc/rfc3168.txt>.
- [64] Linux Foundation. Data plane development kit (DPDK), 2022. URL <http://www.dpdk.org>.
- [65] Cheng Peng Fu and S.C. Liew. Tcp veno: Tcp enhancement for transmission over wireless access networks. *IEEE Journal on Selected Areas in Communications*, 21(2):216–228, 2003. doi: 10.1109/JSAC.2002.807336.
- [66] Xinwei Fu, Zhen Zhang, Haozheng Fan, Guangtai Huang, Mohammad El-Shabani, Randy Huang, Rahul Solanki, Fei Wu, Ron Diamant, and Yida Wang. Distributed training of large language models on aws trainium. In *Proceedings of the 2024 ACM Symposium on Cloud Computing, SoCC '24*, page 961–976, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712869. doi: 10.1145/3698038.3698535. URL <https://doi.org/10.1145/3698038.3698535>.
- [67] Satoshi Fujita and Takeshi Hada. Two-dimensional on-line bin packing problem with rotatable items. *Theor. Comput. Sci.*, 289(2):939–952, October 2002. ISSN 0304-3975. doi: 10.1016/S0304-3975(01)00410-8. URL [https://doi.org/10.1016/S0304-3975\(01\)00410-8](https://doi.org/10.1016/S0304-3975(01)00410-8).

- [68] Juan J. Galvez, Nikhil Jain, and Laxmikant V. Kale. Automatic topology mapping of diverse large-scale parallel applications. In *Proceedings of the International Conference on Supercomputing, ICS '17*, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350204. doi: 10.1145/3079079.3079104. URL <https://doi.org/10.1145/3079079.3079104>.
- [69] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. RDMA over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference, SIGCOMM '24*, 2024. doi: 10.1145/3651890.3672233.
- [70] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 249–264, 2016.
- [71] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533. USENIX Association, April 2021. ISBN 978-1-939133-21-2. URL <https://www.usenix.org/conference/nsdi21/presentation/gao>.
- [72] Yiannis Georgiou, Emmanuel Jeannot, Guillaume Mercier, and Adèle Villiermet. Topology-aware resource management for hpc applications. In *Proceedings of the 18th International Conference on Distributed Computing and Networking, ICDCN '17*, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450348393. doi: 10.1145/3007748.3007768. URL <https://doi.org/10.1145/3007748.3007768>.
- [73] 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*, pages 216–229, 2016. doi: 10.1145/2934872.2934911.
- [74] 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. ISBN 9781450341936. doi: 10.1145/2934872.2934911. URL <https://doi.org/10.1145/2934872.2934911>.
- [75] Soudeh Ghorbani, Zibin Yang, P. Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17*, page 225–238, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346535. doi: 10.1145/3098822.3098839. URL

<https://doi.org/10.1145/3098822.3098839>.

- [76] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M G Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M K Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, and Amin Vahdat. Aquila: A unified, low-latency fabric for datacenter networks. pages 1249–1266. USENIX Association, 4 2022. ISBN 978-1-939133-27-4. URL <https://www.usenix.org/conference/nsdi22/presentation/gibson>.
- [77] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, and Amin Vahdat. Aquila: A unified, low-latency fabric for datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1249–1266, Renton, WA, April 2022. USENIX Association. ISBN 978-1-939133-27-4. URL <https://www.usenix.org/conference/nsdi22/presentation/gibson>.
- [78] Graphcore. Bow Pod64 Reference Design Datasheet, 2024. <https://docs.graphcore.ai/projects/bow-pod64-datasheet/en/2.1.0/product-description.html>.
- [79] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VI2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM ’09, pages 51–62, 2009. doi: 10.1145/1592568.1592576.
- [80] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. VI2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, 2009.
- [81] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VI2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM ’09, page 51–62, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585949. doi: 10.1145/1592568.1592576. URL <https://doi.org/10.1145/1592568.1592576>.
- [82] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: a gpu cluster manager for distributed deep learning. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI’19, page 485–500, USA, 2019. USENIX Association. ISBN 9781931971492.

- [83] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16*, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341936. doi: 10.1145/2934872.2934908. URL <https://doi.org/10.1145/2934872.2934908>.
- [84] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To infinity and beyond: Time-Warped network emulation. In *3rd Symposium on Networked Systems Design & Implementation (NSDI 06)*, San Jose, CA, May 2006. USENIX Association. URL <https://www.usenix.org/conference/nsdi-06/infinity-and-beyond-time-warped-network-emulation>.
- [85] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL <https://www.gurobi.com>.
- [86] Andrei Gurtov, Tom Henderson, Sally Floyd, and Yoshifumi Nishida. The NewReno Modification to TCP’s Fast Recovery Algorithm. RFC 6582, April 2012. URL <https://www.rfc-editor.org/info/rfc6582>.
- [87] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: A new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, jul 2008. ISSN 0163-5980. doi: 10.1145/1400097.1400105. URL <https://doi.org/10.1145/1400097.1400105>.
- [88] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. Augmenting data center networks with multi-gigabit wireless links. In *Proceedings of the ACM SIGCOMM 2011 Conference, SIGCOMM '11*, page 38–49, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450307970. doi: 10.1145/2018436.2018442. URL <https://doi.org/10.1145/2018436.2018442>.
- [89] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R. Das, Jon P. Longtin, Himanshu Shah, and Ashish Tanwer. Firefly: A reconfigurable wireless data center fabric using free-space optics. In *Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM '14*, page 319–330, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328364. doi: 10.1145/2619239.2626328. URL <https://doi.org/10.1145/2619239.2626328>.
- [90] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the 2017 ACM SIGCOMM Conference, SIGCOMM '17*, pages 29–42, 2017. doi: 10.1145/3098822.3098825.
- [91] Vipul Harsh, Sangeetha Abdu Jyothi, and P. Brighten Godfrey. Spineless data centers. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks, HotNets '20*, page 67–73, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381451. doi: 10.1145/3422604.3425945. URL <https://doi.org/10.1145/3422604.3425945>.
- [92] Yannick Hauri, Debopam Bhattacharjee, Manuel Grossmann, and Ankit Singla. ”internet from space” without inter-satellite links. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks, HotNets '20*, page 205–211, New York, NY, USA, 2020. Association

- for Computing Machinery. ISBN 9781450381451. doi: 10.1145/3422604.3425938. URL <https://doi.org/10.1145/3422604.3425938>.
- [93] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based load balancing for fast datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15*, page 465–478, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335423. doi: 10.1145/2785956.2787507. URL <https://doi.org/10.1145/2785956.2787507>.
 - [94] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. Masq: Rdma for virtual private cloud. 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 1–14, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379557. doi: 10.1145/3387514.3405849. URL <https://doi.org/10.1145/3387514.3405849>.
 - [95] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019. doi: 10.1145/3282307. URL <https://doi.org/10.1145/3282307>.
 - [96] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for Fine-Grained resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association. URL <https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center>.
 - [97] Torsten Hoefer and Marc Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proceedings of the International Conference on Supercomputing, ICS '11*, page 75–84, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450301022. doi: 10.1145/1995896.1995909. URL <https://doi.org/10.1145/1995896.1995909>.
 - [98] Torsten Hoefer, 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. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '22*. IEEE Press, 2022. ISBN 9784665454445.
 - [99] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):15–26, aug 2013. ISSN 0146-4833. doi: 10.1145/2534169.2486012. URL <https://doi.org/10.1145/2534169.2486012>.
 - [100] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn,

- Jonathan Zolla, Joon Ong, and Amin Vahdat. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in google’s software-defined wan. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM ’18*, page 74–87, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355674. doi: 10.1145/3230543.3230545. URL <https://doi.org/10.1145/3230543.3230545>.
- [101] C. Hopps. Rfc2992: Analysis of an equal-cost multi-path algorithm, 2000.
- [102] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Contra: A programmable system for performance-aware routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 701–721, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL <https://www.usenix.org/conference/nsdi20/presentation/hsu>.
- [103] Kuo-Feng Hsu, Praveen Tammanna, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Adaptive weighted traffic splitting in programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 103–109, 2020.
- [104] 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*, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3476223. URL <https://doi.org/10.1145/3458817.3476223>.
- [105] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, and Tianwei Zhang. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 709–729, Santa Clara, CA, April 2024. USENIX Association. ISBN 978-1-939133-39-7. URL <https://www.usenix.org/conference/nsdi24/presentation/hu>.
- [106] Markus Ihmsen, Nadir Akinci, Markus Becker, and Matthias Teschner. A parallel sph implementation on multi-core cpus. *Computer Graphics Forum*, 30(1):99–112, 2011. doi: <https://doi.org/10.1111/j.1467-8659.2010.01832.x>. URL <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2010.01832.x>.
- [107] Ankush Jain, Charles D. Cranor, Qing Zheng, Dominic Manno, George Amvrosiadis, and Gary A. Grider. Lessons from profiling and optimizing placement in amr codes. In *2025 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–14, 2025. doi: 10.1109/CLUSTER59342.2025.11186462.
- [108] Emmanuel Jeannot. Process mapping on any topology with TopoMatch. *Journal of Parallel and Distributed Computing*, 170:39–52, 2022. ISSN 0743-7315. doi: 10.1016/j.jpdc.2022.08.002. URL <https://doi.org/10.1016/j.jpdc.2022.08.002>.
- [109] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages

- 947–960, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL <https://www.usenix.org/conference/atc19/presentation/jeon>.
- [110] Prithish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas Quinn. Massively parallel cosmological simulations with changa. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, 2008. doi: 10.1109/IPDPS.2008.4536319.
 - [111] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13, 2019. URL https://proceedings.mlsys.org/paper_files/paper/2019/file/b422680f3db0986ddd7f8f126baaf0fa-Paper.pdf.
 - [112] 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*, 2023. doi: 10.1145/3579371.3589350.
 - [113] 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. ISBN 9798400700958. doi: 10.1145/3579371.3589350. URL <https://doi.org/10.1145/3579371.3589350>.
 - [114] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A domain-specific supercomputer for training deep neural networks. *Commun. ACM*, 63(7):67–78, June 2020. ISSN 0001-0782. doi: 10.1145/3360307. URL <https://doi-org.cmu.idm.oclc.org/10.1145/3360307>.
 - [115] Abdul Kabbani, Balajee Vamanan, Jahangir Hasan, and Fabien Duchene. Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies, CoNEXT '14*, page 149–160, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450332798. doi: 10.1145/2674005.2674985. URL <https://doi.org/10.1145/2674005.2674985>.
 - [116] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. Dynamic load balancing without packet reordering. *SIGCOMM Comput. Commun. Rev.*, 37(2):51–62, mar 2007. ISSN 0146-4833. doi: 10.1145/1232919.1232925. URL <https://doi.org/10.1145/1232919.1232925>.
 - [117] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: Measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement, IMC '09*, page 202–208, New York, NY,

- USA, 2009. Association for Computing Machinery. ISBN 9781605587714. doi: 10.1145/1644893.1644918. URL <https://doi.org/10.1145/1644893.1644918>.
- [118] Nanxi Kang, Monia Ghobadi, John Reumann, Alexander Shraer, and Jennifer Rexford. Efficient traffic splitting on commodity switches. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–13, 2015.
 - [119] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. In *Proceedings of the ACM Workshop on Frontiers in Computer Communications Technology*, SIGCOMM ’87, page 2–7, New York, NY, USA, 1987. Association for Computing Machinery. ISBN 0897912454. doi: 10.1145/55482.55484. URL <https://doi.org/10.1145/55482.55484>.
 - [120] Iwan Kartawira, Michael DeMerchant, Swapna Nannapaneni, Spencer Giacalone, Shawn Chen, and James Burch. Optical circuit switching for AI and hyperscale datacenters: Benefits, technologies, current and future deployments. White paper, Open Compute Project (OCP), April 2026.
 - [121] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.
 - [122] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. Clove: Congestion-aware load balancing at the virtual edge. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT ’17, page 323–335, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450354226. doi: 10.1145/3143361.3143401. URL <https://doi.org/10.1145/3143361.3143401>.
 - [123] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. FreeFlow: Software-based virtual RDMA networking for containerized clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 113–126, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL <https://www.usenix.org/conference/nsdi19/presentation/kim>.
 - [124] John Kim, William J. Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. *SIGARCH Comput. Archit. News*, 36(3):77–88, jun 2008. ISSN 0163-5964. doi: 10.1145/1394608.1382129. URL <https://doi.org/10.1145/1394608.1382129>.
 - [125] John Kim, William J. Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. *Proceedings - International Symposium on Computer Architecture*, pages 77–88, 2008. ISSN 10636897. doi: 10.1109/ISCA.2008.19.
 - [126] Benjamin Klenk and Holger Fröning. An overview of mpi characteristics of exascale proxy applications. In *High Performance Computing: 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, June 18–22, 2017, Proceedings*, page 217–236, Berlin, Heidelberg, 2017. Springer-Verlag. ISBN 978-3-319-58666-3. doi: 10.1007/978-3-319-58667-0_12. URL <https://doi.org/10.1007/>

978-3-319-58667-0_12.

- [127] 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. ISSN 0734-2071. doi: 10.1145/354871.354874. URL <https://doi.org/10.1145/354871.354874>.
- [128] Abhishek Vijaya Kumar, Eric Ding, Arjun Devraj, Darius Bunandar, and Rachee Singh. Reconfigurable torus fabrics for multi-tenant ML. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '26, pages 1547–1565, New York, NY, USA, 2026. Association for Computing Machinery. doi: 10.1145/3779212.3790238. URL <https://doi.org/10.1145/3779212.3790238>.
- [129] Sameer Kumar and Norm Jouppi. Highly available data parallel ml training on mesh networks. In *NeurIPS 2020 Workshop on Machine Learning for Systems*, 2020. URL https://mlforsystems.org/assets/papers/neurips2020/highly_kumar_2020.pdf.
- [130] Oh-Kyoung Kwon, Ji-hoon Kang, Seungchul Lee, Wonjung Kim, and Junehwa Song. Efficient task-mapping of parallel applications using a space-filling curve. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '22, page 384–397, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450398688. doi: 10.1145/3559009.3569657. URL <https://doi.org/10.1145/3559009.3569657>.
- [131] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 454–467, 2016. doi: 10.1145/2934872.2934885.
- [132] Baolin Li, Rohin Arora, Siddharth Samsi, Tirthak Patel, William Arcand, David Bestor, Chansup Byun, Rohan Basu Roy, Bill Bergeron, John Holodnak, Michael Houle, Matthew Hubbell, Michael Jones, Jeremy Kepner, Anna Klein, Peter Michaleas, Joseph McDonald, Lauren Milechin, Julie Mullen, Andrew Prout, Benjamin Price, Albert Reuther, Antonio Rosa, Matthew Weiss, Charles Yee, Daniel Edelman, Allan Vanterpool, Anson Cheng, Vijay Gadepally, and Devesh Tiwari. Ai-enabling workloads on large-scale gpu-accelerated system: Characterization, opportunities, and implications. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1224–1237, 2022. doi: 10.1109/HPCA53966.2022.00093.
- [133] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Lyra: Elastic Scheduling for Deep Learning Clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 835–850. Association for Computing Machinery, 2023. ISBN 978-1-4503-9487-1. doi: 10.1145/3552326.3587445. URL <https://doi-org.cmu.idm.oclc.org/10.1145/3552326.3587445>.
- [134] 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 *2017 16th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 34–43, 2017. doi: 10.1109/ISPDC.2017.11.

- [135] Wenxue Li, Xiangzhou Liu, Yuxuan Li, Yilun Jin, Han Tian, Zhizhen Zhong, Guyue Liu, Ying Zhang, and Kai Chen. Understanding communication characteristics of distributed training. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*, APNet '24, page 1–8, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400717581. doi: 10.1145/3663408.3663409. URL <https://doi.org/10.1145/3663408.3663409>.
- [136] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM '19*, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359566. doi: 10.1145/3341302.3342085. URL <https://doi.org/10.1145/3341302.3342085>.
- [137] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkkipati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1171–1186, 2020.
- [138] Heng Liao, Bingyang Liu, Xianping Chen, Zhigang Guo, Chuanning Cheng, Jianbing Wang, Xiangyu Chen, Peng Dong, Rui Meng, Wenjie Liu, Zhe Zhou, Ziyang Zhang, Yuhang Gai, Cunle Qian, Yi Xiong, Zhongwu Cheng, Jing Xia, Yuli Ma, Xi Chen, Wenhua Du, Shizhong Xiao, Chungang Li, Yong Qin, Liudong Xiong, Zhou Yu, Lv Chen, Lei Chen, Buyun Wang, Pei Wu, Junen Gao, Xiaochu Li, Jian He, Shizhuan Yan, and Bill McColl. Ub-mesh: a hierarchically localized nd-fullmesh datacenter network architecture, 2025. URL <https://arxiv.org/abs/2503.20377>.
- [139] Xudong Liao, Yijun Sun, et al. MixNet: A runtime reconfigurable optical-electrical fabric for distributed mixture-of-experts training. In *Proceedings of the ACM SIGCOMM 2025 Conference, SIGCOMM '25*, 2025. doi: 10.1145/3718958.3750465.
- [140] Sean Lie. Wafer-scale ai: Gpu impossible performance. In *2024 IEEE Hot Chips 36 Symposium (HCS)*, pages 1–71, 2024. doi: 10.1109/HCS61935.2024.10664673.
- [141] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit switching under the radar with REACToR. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 1–15, 2014.
- [142] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit switching under the radar with reactor. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI'14*, page 1–15, USA, 2014. USENIX Association. ISBN 9781931971096.
- [143] He Liu, Matthew K. Mukerjee, Conglong Li, Nicolas Feltman, George Papen, Stefan Savage, Srinivasan Seshan, Geoffrey M. Voelker, David G. Andersen, Michael Kaminsky, George Porter, and Alex C. Snoeren. Scheduling techniques for hybrid circuit/packet networks. In *Proceedings of the 11th ACM Conference on Emerging Networking Ex-*

- periments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334129. doi: 10.1145/2716281.2836126. URL <https://doi.org/10.1145/2716281.2836126>.
- [144] He Liu, Matthew K. Mukerjee, Conglong Li, Nicolas Feltman, George Papen, Stefan Savage, Srinivasan Seshan, Geoffrey M. Voelker, David G. Andersen, Michael Kaminsky, George Porter, and Alex C. Snoeren. Scheduling techniques for hybrid circuit/packet networks. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334129. doi: 10.1145/2716281.2836126. URL <https://doi.org/10.1145/2716281.2836126>.
 - [145] Hong Liu, Ryohei Urata, Kevin Yasumura, Xiang Zhou, Roy Bannon, Jill Berger, Pedram Dashti, et al. Lightwave fabrics: At-scale optical circuit switching for datacenter and machine learning systems. In *Proceedings of the ACM SIGCOMM 2023 Conference*, SIGCOMM '23, pages 499–515, 2023. doi: 10.1145/3603269.3604836.
 - [146] V. Lo, K.J. Windisch, Wanqian Liu, and B. Nitzberg. Noncontiguous processor allocation algorithms for mesh-connected multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(7):712–726, 1997. doi: 10.1109/71.598346.
 - [147] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-Path transport for RDMA in datacenters. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 357–371, Renton, WA, April 2018. USENIX Association. ISBN 978-1-939133-01-4. URL <https://www.usenix.org/conference/nsdi18/presentation/lu>.
 - [148] Jens Mache, Virginia Lo, and Kurt Windisch. Minimizing message-passing contention in fragmentation-free processor allocation. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing Systems (PDCS)*, pages 120–124, 1997.
 - [149] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL <https://www.usenix.org/conference/nsdi20/presentation/mahajan>.
 - [150] Saverio Mascolo, Claudio Casetti, Mario Gerla, M. Y. Sanadidi, and Ren Wang. Tcp westwood: Bandwidth estimation for enhanced transport over wireless links. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, MobiCom '01, page 287–297, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581134223. doi: 10.1145/381677.381704. URL <https://doi.org/10.1145/381677.381704>.
 - [151] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, mar 2008. ISSN

0146-4833. doi: 10.1145/1355734.1355746. URL <https://doi.org/10.1145/1355734.1355746>.

- [152] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. RotorNet: A scalable, low-complexity, optical datacenter network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17*, pages 267–280, 2017. doi: 10.1145/3098822.3098838.
- [153] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. Rotornet: A scalable, low-complexity, optical datacenter network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17*, page 267–280, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346535. doi: 10.1145/3098822.3098838. URL <https://doi.org/10.1145/3098822.3098838>.
- [154] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. Rotornet: A scalable, low-complexity, optical datacenter network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17*, page 267–280, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346535. doi: 10.1145/3098822.3098838. URL <https://doi.org/10.1145/3098822.3098838>.
- [155] 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, 2020. URL <https://www.usenix.org/conference/nsdi20/presentation/mellette>.
- [156] 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. ISBN 978-1-939133-13-7. URL <https://www.usenix.org/conference/nsdi20/presentation/mellette>.
- [157] 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. ISBN 978-1-939133-13-7. URL <https://www.usenix.org/conference/nsdi20/presentation/mellette>.
- [158] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), mar 2014. ISSN 1075-3583.
- [159] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From luna to solar: The evolutions of the compute-to-storage networks in alibaba cloud. In *Proceedings of the ACM SIGCOMM*

- 2022 Conference, SIGCOMM '22, page 753–766, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394208. doi: 10.1145/3544216.3544238. URL <https://doi.org/10.1145/3544216.3544238>.
- [160] George Michelogiannakis, Yiwen Shen, Min Yee Teh, Xiang Meng, Benjamin Aivazi, Taylor Groves, John Shalf, Madeleine Glick, Manya Ghobadi, Larry Dennison, and Keren Bergman. Bandwidth steering in HPC using silicon nanophotonics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2019. doi: 10.1145/3295500.3356145.
 - [161] Henry Kwok Mikel Jimenez Fernandez. Building express backbone: Facebook’s new long-haul network, 2017. URL <https://engineering.fb.com/2017/05/01/data-center-engineering/building-express-backbone-facebook-s-new-long-haul-network/>.
 - [162] Cyriel Minkenberg, Rajagopal Krishnaswamy, Aaron Zilkie, and David Nelson. Co-packaged datacenter optics: Opportunities and challenges. *IET Optoelectronics*, 15(2): 77–91, 2021. doi: 10.1049/ote2.12020.
 - [163] Jeffrey C. Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with modeling network topologies at multiple levels of abstraction. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI’20, page 403–418, USA, 2020. USENIX Association. ISBN 9781939133137.
 - [164] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond GPUs for DNN scheduling on Multi-Tenant clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 579–596, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL <https://www.usenix.org/conference/osdi22/presentation/mohan>.
 - [165] Jose Miguel Montanana, Michihiro Koibuchi, Hiroki Matsutani, and Hideharu Amano. Balanced dimension-order routing for k-ary n-cubes. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops, ICPPW '09*, page 499–506, USA, 2009. IEEE Computer Society. ISBN 9780769538037. doi: 10.1109/ICPPW.2009.64. URL <https://doi.org/10.1109/ICPPW.2009.64>.
 - [166] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 ACM SIGCOMM Conference, SIGCOMM '18*, pages 221–235, 2018. doi: 10.1145/3230543.3230564.
 - [167] A.W. Mu’alem and D.G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001. doi: 10.1109/71.932708.
 - [168] Matthew K. Mukerjee, Christopher Canel, Weiyang Wang, Daehyeok Kim, Srinivasan Seshan, and Alex C. Snoeren. Adapting TCP for reconfigurable datacenter networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 651–666, Santa Clara, CA, February 2020. USENIX Association. ISBN

- 978-1-939133-13-7. URL <https://www.usenix.org/conference/nsdi20/presentation/mukerjee>.
- [169] Matthew K. Mukerjee, Christopher Canel, Weiyang Wang, Daehyeok Kim, Srinivasan Seshan, and Alex C. Snoeren. Adapting tcp for reconfigurable datacenter networks. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 651–666, USA, 2020. USENIX Association. ISBN 9781939133137.
 - [170] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL <https://www.usenix.org/conference/osdi20/presentation/narayanan-deepak>.
 - [171] NS-3 Consortium. ns-3 network simulator, 2023. <https://www.nsnam.org/>.
 - [172] NVIDIA. Nvidia collective communication library (nccl) documentation, June 2025. URL https://docs.nvidia.com/deeplearning/nccl/archives/nccl_2273/user-guide/docs/index.html.
 - [173] NVIDIA Corporation. NVIDIA H100 Tensor Core GPU Architecture. Whitepaper, NVIDIA, 2023. <https://resources.nvidia.com/en-us-hopper-architecture>.
 - [174] NVIDIA Corporation. NVIDIA Blackwell Architecture Technical Brief. Technical brief, NVIDIA, 2024. <https://resources.nvidia.com/en-us-blackwell-architecture>.
 - [175] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 69–84, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi: 10.1145/2517349.2522716. URL <https://doi.org/10.1145/2517349.2522716>.
 - [176] Herbert Owen, Oriol Lehmkuhl, Pasqua D'Ambra, Fabio Durastante, and Salvatore Filippone. Alya toward exascale: algorithmic scalability using psctoolkit. *J. Supercomput.*, 80(10):13533–13556, March 2024. ISSN 0920-8542. doi: 10.1007/s11227-024-05989-y. URL <https://doi.org/10.1007/s11227-024-05989-y>.
 - [177] Tirthak Patel, Zhengchun Liu, Raj Kettimuthu, Paul Rich, William Allcock, and Devesh Tiwari. Job characteristics on large-scale systems: Long-term analysis, quantification, and implications. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17, 2020. doi: 10.1109/SC41405.2020.00088.
 - [178] Giannis Patronas et al. Optical switching for data centers and advanced computing systems. *Journal of Optical Communications and Networking*, 17(1):A87–A95, 2024. doi: 10.1364/JOCN.534317.
 - [179] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the*

Thirteenth EuroSys Conference, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355841. doi: 10.1145/3190508.3190517. URL <https://doi.org/10.1145/3190508.3190517>.

- [180] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiah Fainman, George Papen, and Amin Vahdat. Integrating microsecond circuit switching into the data center. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 447–458, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320566. doi: 10.1145/2486001.2486007. URL <https://doi.org/10.1145/2486001.2486007>.
- [181] Jon Postel. Internet Control Message Protocol. RFC 792, September 1981. URL <https://rfc-editor.org/rfc/rfc792.txt>.
- [182] 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, pages 66–85, 2022. doi: 10.1145/3544216.3544265.
- [183] 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. ISBN 9781450394208. doi: 10.1145/3544216.3544265. URL <https://doi.org/10.1145/3544216.3544265>.
- [184] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. Alibaba HPN: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, SIGCOMM '24, 2024. doi: 10.1145/3651890.3672265.
- [185] Mubashir Adnan Qureshi, Yuchung Cheng, Qianwen Yin, Qiaobin Fu, Gautam Kumar, Masoud Moshref, Junhua Yan, Van Jacobson, David Wetherall, and Abdul Kabbani. Plb: Congestion signals are simple and effective for network load balancing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 207–218, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394208. doi: 10.1145/3544216.3544226. URL <https://doi.org/10.1145/3544216.3544226>.
- [186] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 266–277, New York, NY, USA, 2011. Association for Computing Machinery. ISBN

9781450307970. doi: 10.1145/2018436.2018467. URL <https://doi.org/10.1145/2018436.2018467>.
- [187] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. *SIGCOMM Comput. Commun. Rev.*, 41(4):266–277, aug 2011. ISSN 0146-4833. doi: 10.1145/2043164.2018467. URL <https://doi.org/10.1145/2043164.2018467>.
 - [188] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath TCP. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 399–412, San Jose, CA, April 2012. USENIX Association. URL <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/raiciu>.
 - [189] 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. ISBN 978-1-939133-39-7. URL <https://www.usenix.org/conference/nsdi24/presentation/rajasekaran>.
 - [190] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. ASTRA-SIM: Enabling SW/HW co-design exploration for distributed DL training platforms. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020.
 - [191] Juan A. Rico-Gallego, Juan C. Díaz-Martín, Ravi Reddy Manumachu, and Alexey L. Lastovetsky. A survey of communication performance models for high-performance computing. *ACM Comput. Surv.*, 51(6), January 2019. ISSN 0360-0300. doi: 10.1145/3284358. URL <https://doi.org/10.1145/3284358>.
 - [192] Matthew Roughan. Simplifying the synthesis of internet traffic matrices. *SIGCOMM Comput. Commun. Rev.*, 35:93–96, 10 2005. ISSN 0146-4833. doi: 10.1145/1096536.1096551. URL <https://doi.org/10.1145/1096536.1096551>.
 - [193] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network’s (datacenter) network. *SIGCOMM Comput. Commun. Rev.*, 45(4): 123–137, aug 2015. ISSN 0146-4833. doi: 10.1145/2829988.2787472. URL <https://doi.org/10.1145/2829988.2787472>.
 - [194] Subhajit Sahu, Kishore Kothapalli, and Dip Sankar Banerjee. Shared-memory parallel dynamic louvain algorithm for community detection. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1204–1205, 2024. doi: 10.1109/IPDPSW63119.2024.00207.
 - [195] Siddharth Samsi, Matthew L Weiss, David Bestor, Baolin Li, Michael Jones, Albert Reuther, Daniel Edelman, William Arcand, Chansup Byun, John Holodnack, Matthew Hubbell, Jeremy Kepner, Anna Klein, Joseph McDonald, Adam Michaleas, Peter Michaleas, Lauren Milechin, Julia Mullen, Charles Yee, Benjamin Price, Andrew Prout, Antonio Rosa, Allan Vanterpool, Lindsey McEvoy, Anson Cheng, Devesh Tiwari, and Vijay Gadepally. The

- mit supercloud dataset. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8, 2021. doi: 10.1109/HPEC49654.2021.9622850.
- [196] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V. Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17*, page 418–431, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346535. doi: 10.1145/3098822.3098853. URL <https://doi.org/10.1145/3098822.3098853>.
 - [197] Daniele De Sensi, Tommaso Bonato, David Saam, and Torsten Hoefer. 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. ISBN 978-1-939133-39-7. URL <https://www.usenix.org/conference/nsdi24/presentation/de-sensi>.
 - [198] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. Compute trends across three eras of machine learning. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2022. doi: 10.1109/IJCNN55064.2022.9891914. URL <https://arxiv.org/abs/2202.05924>.
 - [199] Hua Shao, Xiaoliang Wang, Yuanwei Lu, Yanbo Yu, Shengli Zheng, and Youjian Zhao. Accessing cloud with disaggregated Software-Defined router. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 1–14. USENIX Association, April 2021. ISBN 978-1-939133-21-2. URL <https://www.usenix.org/conference/nsdi21/presentation/shao>.
 - [200] Alexander Shpiner, Zachy Haramaty, Saar Eliad, Vladimir Zdornov, Barak Gafni, and Eitan Zahavi. Dragonfly+: Low cost topology for scaling datacenters. In *2017 IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*, pages 1–8, 2017. doi: 10.1109/HiPINEB.2017.11.
 - [201] 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, 2019.
 - [202] 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. *ACM SIGCOMM Computer Communication Review*, 45(4):183–197, 2015. doi: 10.1145/2829988.2787508.
 - [203] 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. *SIGCOMM Comput. Commun. Rev.*, 45(4):183–197, aug 2015. ISSN 0146-4833. doi: 10.1145/2829988.2787508. URL <https://doi.org/10.1145/2829988.2787508>.

- [204] 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. ISBN 9781450335423. doi: 10.1145/2785956.2787508. URL <https://doi.org/10.1145/2785956.2787508>.
- [205] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 225–238, San Jose, CA, April 2012. USENIX Association. ISBN 978-931971-92-8. URL <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/singla>.
- [206] Ankit Singla, Chi Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. *Proceedings of NSDI 2012: 9th USENIX Symposium on Networked Systems Design and Implementation*, pages 225–238, 2012.
- [207] Cha Hwan Song, Xin Zhe Khooi, Raj Joshi, Inho Choi, Jialin Li, and Mun Choon Chan. Network load balancing with in-network reordering support for rdma. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM ’23, page 816–831, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702365. doi: 10.1145/3603269.3604849. URL <https://doi.org/10.1145/3603269.3604849>.
- [208] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 642–657. Association for Computing Machinery, 2023. ISBN 9798400702297. doi: 10.1145/3600006.3613175. URL <https://doi.org/10.1145/3600006.3613175>.
- [209] Kaushik Kandadi Suresh, Bharath Ramesh, Goutham Kalikrishna Reddy Kuncham, Hari Subramoni, and Dhabaleswar K. DK Panda. Towards dynamic message passing protocols for stencil-based communication patterns. In *2025 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12, 2025. doi: 10.1109/CLUSTER59342.2025.11186470.
- [210] Emil Talpes, Douglas Williams, and Debjit Das Sarma. Dojo: The microarchitecture of tesla’s exa-scale computer. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–28, 2022. doi: 10.1109/HCS55958.2022.9895534.
- [211] Wei Tang, Zhiling Lan, Narayan Desai, Daniel Buettner, and Yongen Yu. Reducing fragmentation on torus-connected supercomputers. In *2011 IEEE International Symposium on Parallel and Distributed Processing*, pages 828–839, 2011. doi: 10.1109/IPDPS.2011.82.

- [212] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3387517. URL <https://doi.org/10.1145/3342195.3387517>.
- [213] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342407. doi: 10.1145/2901318.2901355. URL <https://doi.org/10.1145/2901318.2901355>.
- [214] Ozan Tuncer, Vitus J. Leung, and Ayse K. Coskun. Pacmap: Topology mapping of unstructured communication patterns onto non-contiguous allocations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, page 37–46, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335591. doi: 10.1145/2751205.2751225. URL <https://doi.org/10.1145/2751205.2751225>.
- [215] Paul Tune and Matthew Roughan. Spatiotemporal traffic matrix synthesis. *SIGCOMM Comput. Commun. Rev.*, 45:579–592, 8 2015. ISSN 0146-4833. doi: 10.1145/2785956.2787471. URL <https://doi.org/10.1145/2829988.2787471><https://doi.org/10.1145/2785956.2787471>.
- [216] 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. URL <https://arxiv.org/abs/2208.10041>.
- [217] 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. URL <https://arxiv.org/abs/2208.10041>.
- [218] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 407–420, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/vanini>.
- [219] Jasmina Vasiljevic and Davor Capalija. Blackhole & tt-metalium: The standalone ai computer and its programming model. In *2024 IEEE Hot Chips 36 Symposium (HCS)*, pages 1–30, 2024. doi: 10.1109/HCS61935.2024.10664810.
- [220] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Balde-schwieler. Apache hadoop yarn: yet another resource negotiator. In *Proceedings of the 4th*

- Annual Symposium on Cloud Computing, SOCC '13*, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324281. doi: 10.1145/2523616.2523633. URL <https://doi.org/10.1145/2523616.2523633>.
- [221] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450332385. doi: 10.1145/2741948.2741964. URL <https://doi.org/10.1145/2741948.2741964>.
 - [222] Mark Wade, Erik Anderson, Shahab Ardalan, Pavan Bhargava, Sidney Buchbinder, Michael L. Davenport, John Fini, Haiwei Lu, Chen Li, Roy Meade, Chandru Ramamurthy, Michael Rust, Forrest Sedgwick, Vladimir Stojanovic, Derek Van Orden, Chong Zhang, Chen Sun, Sergey Y. Shumarayev, Conor O’Keeffe, Tim T. Hoang, David Kehlet, Ravi V. Mahajan, Matthew T. Guzy, Allen Chan, and Tina Tran. TeraPHY: A chiplet technology for low-power, high-bandwidth in-package optical I/O. *IEEE Micro*, 40(2): 63–71, 2020. doi: 10.1109/MM.2020.2976067.
 - [223] Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T.S. Eugene Ng, Michael Kozuch, and Michael Ryan. C-through: Part-time optics in data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 327–338, 2010. doi: 10.1145/1851182.1851222.
 - [224] Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T.S. Eugene Ng, Michael Kozuch, and Michael Ryan. C-through: Part-time optics in data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 327–338, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450302012. doi: 10.1145/1851182.1851222. URL <https://doi.org/10.1145/1851182.1851222>.
 - [225] Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T.S. Eugene Ng, Michael Kozuch, and Michael Ryan. C-through: Part-time optics in data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 327–338, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450302012. doi: 10.1145/1851182.1851222. URL <https://doi.org/10.1145/1851182.1851222>.
 - [226] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362818. doi: 10.1145/3302424.3303953. URL <https://doi.org/10.1145/3302424.3303953>.
 - [227] 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. ISBN 978-1-939133-33-5. URL <https://www.usenix.org/conference/nsdi23/presentation/wang-weiyang>.

- [228] 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. ISBN 978-1-939133-33-5. URL <https://www.usenix.org/conference/nsdi23/presentation/wang-weiyang>.
- [229] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, Renton, WA, April 2022. USENIX Association. ISBN 978-1-939133-27-4. URL <https://www.usenix.org/conference/nsdi22/presentation/weng>.
- [230] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 459–471, Lombard, IL, April 2013. USENIX Association. ISBN 978-1-931971-00-3. URL <https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/winstein>.
- [231] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI’11*, page 99–112, USA, 2011. USENIX Association.
- [232] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. ASTRA-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023.
- [233] William Won, Saeed Rashidi, Sudarshan Srinivasan, and Tushar Krishna. LIBRA: Enabling workload-aware multi-dimensional network topology optimization for distributed training of large AI models. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2024.
- [234] Zhenguo Wu, Benjamin Klenk, Larry Dennison, and Keren Bergman. ACTINA: Adapting circuit-switching techniques for AI networking architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, SC ’25, 2025. doi: 10.1145/3712285.3759842.
- [235] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL <https://www.usenix.org/conference/osdi18/presentation/xiao>.

- [236] Xu Yang, Zhou Zhou, Wei Tang, et al. Balancing job performance with system performance via locality-aware scheduling on torus-connected systems. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, 2014.
- [237] Rio Yokota, L.A. Barba, and Matthew G. Knepley. Petrbbf — a parallel $O(n)$ algorithm for radial basis function interpolation with gaussians. *Computer Methods in Applied Mechanics and Engineering*, 199(25):1793–1804, 2010. ISSN 0045-7825. doi: <https://doi.org/10.1016/j.cma.2010.02.008>. URL <https://www.sciencedirect.com/science/article/pii/S0045782510000666>.
- [238] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-39727-4.
- [239] Hao Yu, I-hsin Chung, and Jose Moreira. Topology mapping for blue gene/l supercomputer. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 52–52, 2006. doi: 10.1109/SC.2006.63.
- [240] Jiahua Yu and Qian-Ping Gu. A practical algorithm for embedding graphs on torus. In *2015 Third International Symposium on Computing and Networking (CANDAR)*, pages 50–57, 2015. doi: 10.1109/CANDAR.2015.10.
- [241] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. Resilient data-center load balancing in the wild. In *Proceedings of the 2017 SIGCOMM, Los Angeles, CA, USA*, August 2017. URL <https://www.microsoft.com/en-us/research/publication/resilient-datacenter-load-balancing-wild/>.
- [242] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference, IMC '17*, page 78–85, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351188. doi: 10.1145/3131365.3131375. URL <https://doi.org/10.1145/3131365.3131375>.
- [243] Yin Zhang, Matthew Roughan, Nick Duffield, and Albert Greenberg. Fast accurate computation of large-scale ip traffic matrices from link loads. *SIGMETRICS Perform. Eval. Rev.*, 31:206–217, 6 2003. ISSN 0163-5999. doi: 10.1145/885651.781053. URL <https://doi-org.cmu.idm.oclc.org/10.1145/885651.781053>.
- [244] Zhehui Zhang, Haiyang Zheng, Jiayao Hu, Xiangning Yu, Chenchen Qi, Xuemei Shi, and Guohui Wang. Hashing linearity enables relative path control in data centers. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 855–862. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL <https://www.usenix.org/conference/atc21/presentation/zhang-zhehui>.
- [245] Rui Zhang-Shen and Nick McKeown. Designing a predictable internet backbone with valiant load-balancing. In *Proceedings of the 13th International Conference on Quality of Service, IWQoS'05*, page 178–192, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3540262946. doi: 10.1007/11499169_15. URL https://doi.org/10.1007/11499169_15.

- [246] Ran Zhao, Chao Li, Xiaowei Guo, Yi Liu, Sifan Long, Sen Zhang, Yanlong Qiu, and Canqun Yang. An improved parallel overset grid method for fluid simulation with moving boundary. In *Proceedings of the 52nd International Conference on Parallel Processing, ICPP '23*, page 818–827, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400708435. doi: 10.1145/3605573.3605634. URL <https://doi.org/10.1145/3605573.3605634>.
- [247] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM SIGCOMM 2022 Conference, SIGCOMM '22*, pages 428–440, New York, NY, USA, August 2022. Association for Computing Machinery. ISBN 978-1-4503-9420-8. doi: 10.1145/3544216.3544224. URL <https://dl.acm.org/doi/10.1145/3544216.3544224>.
- [248] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL <https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin>.
- [249] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. Wcmp: Weighted cost multipathing for improved fairness in data centers. *Proceedings of the Ninth European Conference on Computer Systems*, 2014. doi: 10.1145/2592798.2592803. URL <http://dx.doi.org/10.1145/2592798.2592803>.
- [250] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15*, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335423. doi: 10.1145/2785956.2787484. URL <https://doi.org/10.1145/2785956.2787484>.
- [251] Zhanda Zhu, Christina Giannoula, Muralidhar Andoorvedu, Qidong Su, Karttikeya Mangalam, Bojian Zheng, and Gennady Pekhimenko. Mist: Efficient distributed training of large language models via memory-parallelism co-optimization. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys '25*, page 1298–1316, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi: 10.1145/3689031.3717461. URL <https://doi.org/10.1145/3689031.3717461>.
- [252] Christopher Zimmer, Saurabh Gupta, Scott Atchley, Sudharshan S. Vazhkudai, and Carl Albing. A multi-faceted approach to job placement for improved performance on extreme-scale systems. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1015–1025, 2016. doi: 10.1109/SC.2016.86.
- [253] Alexander Zimmermann, Arnd Hannemann, and Tim Kosse. Flowgrind - a new performance measurement tool. In *2010 IEEE Global Telecommunications Conference GLOBE-*

COM 2010, pages 1–6, 2010. doi: 10.1109/GLOCOM.2010.5684167.

- [254] 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 *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 761–774, Santa Clara, CA, April 2024. USENIX Association. ISBN 978-1-939133-39-7. URL <https://www.usenix.org/conference/nsdi24/presentation/zu>.
- [255] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steve Hendry, Zachary Daykin, et al. Resiliency at scale: Managing Google’s TPUv4 machine learning supercomputer. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024.