RotorNet: A Scalable, Low-complexity, Optical Datacenter Network

William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich,
George Papen, Alex C. Snoeren, and George Porter
University of California, San Diego

ABSTRACT

The ever-increasing bandwidth requirements of modern datacenters have led researchers to propose networks based upon optical circuit switches, but these proposals face significant deployment challenges. In particular, previous proposals dynamically configure circuit switches in response to changes in workload, requiring network-wide demand estimation, centralized circuit assignment, and tight time synchronization between various network elements—resulting in a complex and unwieldy control plane. Moreover, limitations in the technologies underlying the individual circuit switches restrict both the rate at which they can be reconfigured and the scale of the network that can be constructed.

We propose RotorNet, a circuit-based network design that addresses these two challenges. While RotorNet dynamically reconfigures its constituent circuit switches, it decouples switch configuration from traffic patterns, obviating the need for demand collection and admitting a fully decentralized control plane. At the physical layer, RotorNet relaxes the requirements on the underlying circuit switches—in particular by not requiring individual switches to implement a full crossbar—enabling them to scale to 1000s of ports. We show that RotorNet outperforms comparably priced Fat Tree topologies under a variety of workload conditions, including traces taken from two commercial datacenters. We also demonstrate a small-scale RotorNet operating in practice on an eight-node testbed.

CCS CONCEPTS

• **Networks → Network architectures**;

KEYWORDS

Datacenter, optical switching

1 INTRODUCTION

Modern datacenter networks rely heavily on fiber-optic links to meet bandwidth demands [10, 29]. To avoid the expensive optical-electrical-optical (OEO) conversion necessary to connect such links with power-hungry electrical packet switches, researchers have proposed network architectures that switch much of a datacenter's traffic passively [1, 6, 13, 15, 17, 20, 26, 32] using optical circuit switches (OCSes). OCSes can support very high link bandwidths at low per-bit cost and power because they passively redirect light from one port to another, independent of data rate. Yet despite these alluring properties, optical circuit switching faces two major barriers to wide-scale adoption in the datacenter environment.

The first barrier to deployment is the attendant control plane. Existing proposals to employ OCSes in the datacenter reconfigure optical circuits in response to traffic demands. Such reconfiguration requires collecting network-wide demand information [20, 26, 32] in order to compute a schedule of switch configurations [4, 21], rate-limiting packet transmissions [4, 20, 21, 26], and synchronizing the OCSes with each other, the scheduler, and the endpoints [20]. This tight coupling among the various network components presents a significant challenge at scale. In this paper, we propose an OCS-based topology that has a fully decentralized control plane aimed at maximizing network throughput.

The second issue with employing commercial OCS devices in datacenters is their limited scalability, specifically their low port count (radix) and slow reconfiguration speed. Previous work [24] showed that MEMS-based optical switches can reconfigure quickly (e.g., in 10s of microseconds), or they can have a high port count (e.g., $O(100)$ ports), but not both. Fundamentally, this limitation is imposed by the need to implement a full crossbar abstraction; in other words, the requirement that a switch be able to forward traffic between any two ports. While previous proposals rely upon this full connectivity to construct network fabrics, we explore an alternative design point that circumvents the fundamental scaling limitations of MEMS-based crossbars.

We propose RotorNet, an OCS-based datacenter-wide network fabric that overcomes these challenges by departing from prior optical switching approaches in three distinct ways. First, RotorNet does not reconfigure the optical switches to match network traffic conditions. Instead, each switch independently rotates through a fixed, static set of configurations that provide uniform bandwidth between all endpoints. This design completely eliminates the need for a centralized control plane—because round-robin switch scheduling does not require demand estimation, schedule computation, or schedule distribution—yet delivers full bisection bandwidth for uniform traffic.

	Topology	CP Structure	CP Algorithm	CP Goal	CP Responds to Demand
c-Through [32]	$k\,(N_R \times N_R)$ xbars	Centralized	Edmonds [9]	Throughput maximization	Yes
Helios [13]	$k\,(N_R \times N_R)$ xbars	Centralized	Edmonds [9]	Throughput maximization	Yes
REACToR [20]	$k\,(N_R \times N_R)$ xbars	Centralized	Solstice [22]	Throughput maximization	Yes
Firefly [17]	$k\,(N_R \times N_R)$ xbars	Centralized	Blossom [9]	Throughput maximization	Yes
Mordia [26]	k rings	Centralized	BvN [3, 31]	Throughput maximization	Yes
MegaSwitch [8]	$N_R \times k$ rings	Centralized	Edge coloring variant [7]	Throughput maximization	Yes
ProjecToR [15]	$1\,(kN_R \times kN_R)$ xbar	Distributed	Stable-marriage variant [14]	Starvation-free low latency	Yes
RotorNet	N_{sw} **Rotor switches**	**Distributed**	**VLB variant [5, 30]**	**Throughput maximization**	**No**

Table 1: A comparison of the control planes (CP) of previously proposed OCS-based topologies.

Of course, datacenter traffic is known to be far from uniform [27], so RotorNet gives up at most a factor of two in link bandwidth—which is ample in an optically switched network—to support arbitrary traffic patterns. We rely on a form of Valiant load balancing (VLB) to send traffic through the fabric twice: first to an intermediate endpoint, and then once more to the ultimate destination. We show that classic VLB can be modified to regain a significant amount of bandwidth, meaning the factor-of-two reduction is a worst-case, not common-case, trade off. Our design provides strictly bounded delivery time and buffering while remaining robust to link and switch failures. Compared to a 3:1 Fat Tree of approximately equal cost, RotorNet delivers 1.6× the throughput under worst-case traffic, 2.3× the throughput for reported datacenter traffic, and up to 3× the throughput for uniform traffic.

Finally, RotorNet employs custom-designed OCSes that are tailored to our need, namely rotating through a small fixed set of configurations, which we call Rotor switches. Because of their simplified internal design, Rotor switches can scale to 1000s of ports with state-of-the-art reconfiguration speeds of $O(10)$ μs. A network of 128 Rotor switches can support 2,048 racks while remaining practical to build and deploy. At this scale, RotorNet serves traffic within $O(1)$ ms, and admits a hybrid optical/electronic architecture to extend support to traffic with lower latency requirements. We evaluate RotorNet experimentally on a small-scale testbed complete with a prototype Rotor switch and endpoints that implement our VLB-inspired indirection protocol.

The contributions of this paper include: 1) a new design direction for optically-switched networks that obviates closed-loop circuit scheduling, 2) open-loop Rotor switches and a datacenter-wide fabric, RotorNet, constructed from these switches, 3) an analysis of RotorNet showing that it supports uniform traffic at full bisection bandwidth, worst-case (permutation) traffic at half bisection bandwidth, and reported commercial datacenter traffic with 70–90% of the bandwidth of an ideal Fat Tree network at lower cost, and 4) a demonstration and evaluation of RotorNet's VLB-like routing algorithm on prototype hardware.

Additional information on RotorNet can be found at http://www.sysnet.ucsd.edu/circuit-switching.

2 BACKGROUND & RELATED WORK

RotorNet builds upon a line of research integrating optical circuit switching into datacenter networks. The physical-layer technologies employed by these approaches vary widely, with some using fiber-coupled optical crossbar switches, others using WDM-based
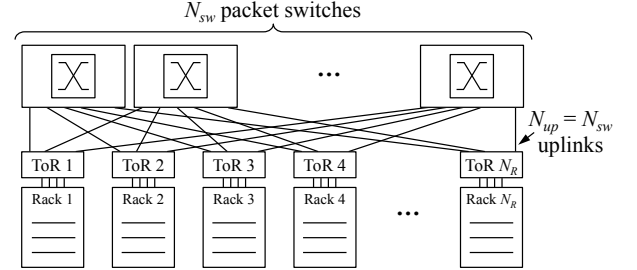

Figure 1: A traditional Clos-based network topology.

ring networks, and yet others using free-space optical transmitters and receivers. Table 1 abstracts away these implementation details and instead focuses on each proposal's topology and control plane. In all cases (save ProjecToR [15], which we discuss below), each design tries to match the configuration of the network to the current or future traffic demands to maximize overall network throughput, requiring a complex datacenter-wide control plane that is difficult to scale. To appreciate why such algorithms are required, we briefly review throughput maximization in the context of a single crossbar-based packet switch before describing previous OCS-based proposals.

2.1 Packet-switched folded-Clos networks

Figure 1 depicts a traditional Clos-based network topology with a single layer of core switches. Groups of servers are aggregated into N_R racks, and each server is connected to its local top-of-rack (ToR) switch. In this example, each ToR has $N_{up} = N_{sw}$ uplinks, each connected to one of N_{sw} core switches. In general, ToR switches are connected by a multi-stage network of homogeneous switches which, taken together, logically act as a single, large crossbar switch. To connect large numbers of racks with switches of moderate port count ($N_{up} \ll N_R$), published production networks employ at least three tiers of cascaded switches [10, 16, 29].

In a folded-Clos network fabric, end hosts are essentially decoupled from the packet switches forming the core of the network. Servers can send data to any destination with the expectation that the fabric will deliver that data without further communication with the sender. Each packet switch has internal buffering to absorb bursts, and a mechanism for conveying traffic through the switch at line rate. The only signals a sender may receive that a switch is under-provisioned are packet losses or ECN (Explicit Congestion
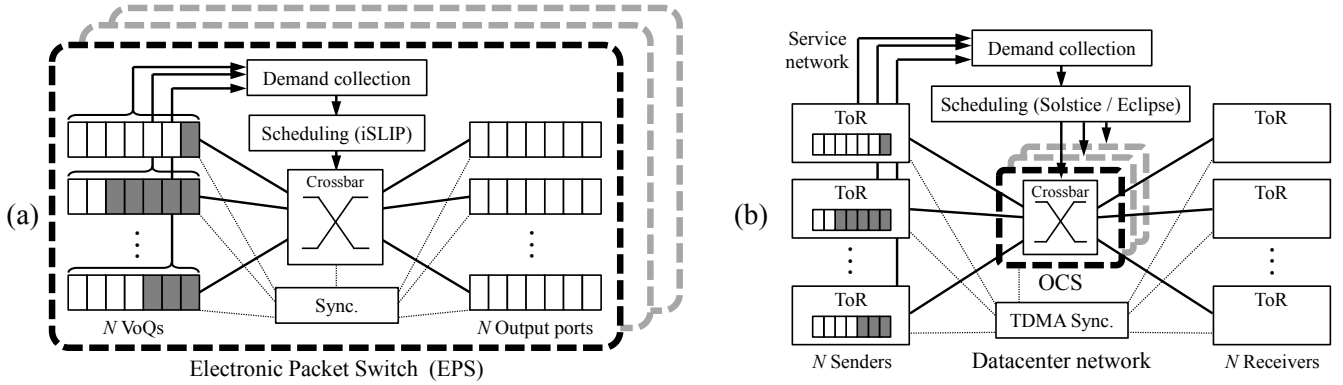
Figure 2: (a) An electronic packet switch encapsulates its control complexity within the switch itself. (b) Standard approaches to optical circuit switching expose the control complexity to the overall network.

Notification) markings when the offered load to the switch exceeds the capacity of one or more outgoing links; in such cases transport protocols (such as TCP) reduce the sending rates to converge to an admissible demand.

A textbook switch design model is a VoQ-scheduled crossbar, shown in Figure 2(a). Packets arrive at N input ports (left), where they are buffered in virtual output queues (VoQs). A scheduling algorithm, such as iSLIP [23] or one of its many variants, examines these queues and chooses the appropriate sequence of input-output port matchings needed to convey packets to the N output ports (right). Typically, the internal crossbar runs at some speed-up factor relative to the external line rate to hide any overhead involved in the internal processes. These processes, including demand collection (i.e., VoQ depth analysis), scheduling (e.g., iSLIP), reconfiguration (i.e., sequence of matchings), and synchronization (e.g., which VoQ is polled during each matching), together comprise a control plane that is implemented entirely within the switch, essentially hidden from the rest of the network and the sender.

2.2 Circuit-switched network fabrics

Prior OCS-based proposals fundamentally change the nature of the switch control plane. Proposals that dynamically reconfigure the network topology in response to observed or predicted traffic in order to maximize throughput must carry out the same tasks as the electronic packet switch shown above. Yet, while packet switches are able to hide their internal management processes inside discrete boxes, previous OCS-based topologies, by virtue of their inability to buffer and inspect packets, expose this control complexity to the entire network, effectively turning the network fabric into a giant, coupled crossbar. Figure 2(b) illustrates this distinction, depicting the network-wide demand collection to a central point [13, 20, 26], a centralized scheduling algorithm [4, 21], schedule distribution [20], and network-wide synchronization [13, 20, 26, 32].

A recent free-space optical proposal, ProjecToR [15], sidesteps this complexity by foregoing global throughput maximization, instead minimizing latency while avoiding starvation. Each ProjecToR switch has its own dedicated switching elements, allowing each ToR to independently provision free-space links based on only that ToR's local demand. It remains only to ensure that multiple ToRs

do not choose free-space paths that conflict at a single receiver, which would cause collisions. Thus, ProjecToR's control plane is inherently simpler than the other approaches in Table 1, and can be implemented in a distributed way. However, the resulting design does not necessarily maximize network throughput. Moreover, ProjecToR's reliance on free-space optics raises a myriad of other practical concerns (e.g., robustness to dust, vibration, etc.) that do not arise in wired fiber-optic networks.

3 DESIGN OVERVIEW

Similar to prior proposals based on optical circuit switches, Rotor-Net employs traditional, packet-switched ToR switches to connect end hosts to the fabric. The ToRs are connected optically to a set of custom OCSes, which we call Rotor switches, which collectively provide connectivity to each of the other ToRs in the network. If desired, ToRs can be further connected to an electrical packet-switched fabric as well to form a hybrid network, but we defer discussion for the time being to focus on the optical network. Importantly, the set of Rotor switches does not provide continuous connectivity between all pairs of ToRs; instead, they implement a schedule of connectivity patterns that, in total, provides a direct connection between any pair of ToRs within a specified time interval.

3.1 Open-loop switching

Unlike prior optically-switched network proposals, the configuration of the Rotor switches is not driven by network traffic conditions, either locally or globally. In RotorNet, the Rotor switches independently cycle through a fixed, pre-determined pattern of connectivity in a round-robin manner, irrespective of instantaneous traffic demands. We choose this time-sequence of Rotor switch configurations (each a one-to-one matching of input to output ports, or simply a "matching") so that each endpoint (i.e., ToR) is provided a direct connection to every other endpoint within one full cycle of matchings.

Figure 3 illustrates this approach, showing two full cycles of three matchings. We highlight the changing connectivity of the top port as it cycles through matchings connecting it to the 2nd, 3rd, and 4th ports across time. Because this approach decouples the
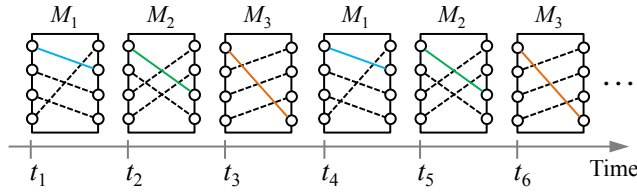
Figure 3: **Rotor switches move through a static, round-robin set of configurations, or matchings, spending an equal amount of time in each matching.**

Figure 4: **(a) A Rotor switch cycles through $(N_R - 1)$ matchings to provide full connectivity between racks. (b) Physically, these matchings are distributed among N_{sw} Rotor switches which, taken together, provide full connectivity.**

switch state from the traffic in the network, it requires no demand collection, no switch scheduling algorithm, and no network-wide synchronization. The switches simply run "open loop."

As a first cut, each N_R-port Rotor switch could repeatedly cycle through all $N_R!$ possible matchings, corresponding to the full set of potential matchings offered by a crossbar switch. However, it would take an infeasibly long time to complete this cycle for large N_R, and we could not guarantee that a given connection is implemented within any reasonable amount of time. Further, only $(N_R - 1)$ matchings are necessary to ensure connectivity between all N_R ToRs. An example set of these matchings is shown in Figure 4(a) for $N_R = 8$. Using these $(N_R - 1)$ matchings, we can guarantee that each ToR is connected to every other ToR within one matching cycle. This functionality is similar to a rotor device [2].

Still, for networks with many ToRs, cycling through even $(N_R - 1)$ matchings may still take too long. Instead, as shown in Figure 4(b), we distribute the $(N_R - 1)$ matchings among N_{sw} parallel Rotor switches, speeding up the matching cycle time by a factor of N_{sw}. We show in Section 4.2 that for a network of 10s of thousands of servers, we can cycle through as few as 16 matchings per switch with $N_{sw} = 128$. In this configuration, each Rotor switch only provides partial connectivity between the ToRs. Taken together, however, the complete set of switches restores full connectivity in the network. We discuss the implications of our design in Section 4, and explain how our design maintains connectivity even in the presence of switch failure in Section 5.3.

3.2 One-hop direct forwarding

Given the baseline round-robin connectivity provided by Rotor switches, each ToR must decide how to route traffic over the network. The simplest approach is for ToRs to send data only along one-hop, direct paths to each destination, resulting in equal bandwidth between each source-destination pair. For uniform traffic, this behavior results in throughput saturating the network's bisection bandwidth (minus the switch duty cycle), and is inherently starvation free. However, for skewed traffic patterns, this approach does not take advantage of slack network capacity when some ToRs are idle, wasting potentially significant amounts of bandwidth.

3.3 Two-hop indirect forwarding

To improve throughput in skewed traffic conditions, we rely on the classic and well-studied technique of indirection. Like Valiant's routing method [30], we allow traffic to pass through intermediate
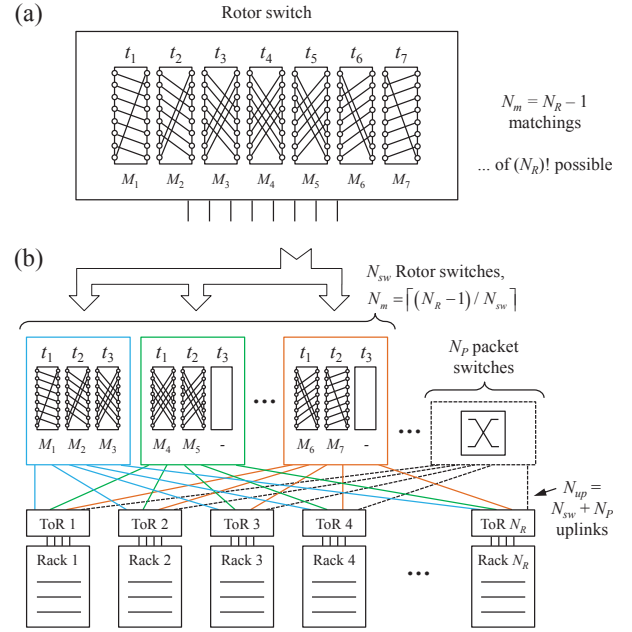
endpoints, which subsequently forward traffic to the final destination. Chang et al. showed that Valiant's method, when coupled with two stages of round-robin switches, yields 100% throughput for arbitrary input traffic[1] [5]. Shrivastav et al. are investigating an approach similar to Chang's applied to rack-scale interconnects [28]. RotorNet is a datacenter-wide fabric, and we leverage the large number of ToR switch uplinks to extend Chang's approach, parallelizing it across a number of Rotor switches. For large networks, such as the example network in Section 4.2, this modification reduces the matching cycle time, and thus the delivery time of traffic, by more than 100× compared to sequentially cycling through all matching patterns. RotorNet routes traffic through the same single-stage fabric twice, and a straightforward implementation would reduce throughput by at most a factor of two (as half the network bandwidth would now be consumed by indirect traffic), yielding half bisection bandwidth for arbitrary input traffic. We argue that this trade-off is justified by the fact that raw network bandwidth is plentiful in optical networks. Moreover, through careful extensions to the basic Valiant load balancing approach (described in Section 5 and evaluated in Section 7), we are able to recapture a significant amount of the theoretical throughput loss in practice, meaning the factor-of-two reduction in throughput is a worst-case, not common-case, trade off. Indirection requires buffering traffic within the network, but outside of the optical Rotor switches themselves, since they cannot buffer light. Indirect traffic is buffered on a per-rack basis, either at the ToR switch or in end-host memory

---

[1]Subject to the minor technical condition that input traffic can be modeled as a stationary and weakly mixing stochastic process.

accessible to the ToR through RDMA. Buffering data on end hosts, similar to Liu et al. [20], is likely the simplest implementation as end hosts are already required to buffer direct traffic while waiting for the appropriate matching to come up in the schedule. We quantify the additional buffering requirements imposed by indirection in Section 7.5.

4 PRACTICAL CONSIDERATIONS

In this section, we explain how the design of RotorNet allows it to scale to modern datacenter sizes of tens of thousands of servers and thousands of ToRs cost-effectively. We provide a comparison to existing Fat Tree topologies, and address manufacturing and deployment concerns.

4.1 Scalability

4.1.1 Achieving high port count. The key to RotorNet's scalability is the use of Rotor switches, which are fundamentally more scalable than traditional optical circuit switches. In particular, a traditional N-port OCS implements a crossbar, meaning it can be configured to any of $N!$ matching patterns. This flexibility limits the switching speed and radix of a MEMS-based OCS because the physical requirements of each micro-mirror switching element are coupled to the switch radix [24]. As a result, commercially available OCSes have radices on the order of 100 ports and reconfiguration times of 10s of milliseconds.

Connecting thousands of ToRs together with conventional OCSes requires those OCSes be cascaded in a multi-stage optical topology, which introduces significant signal attenuation. Higher signal attenuation, in turn, requires higher sensitivity optical transceivers or optical amplification, which would almost certainly negate any cost savings gained by using OCSes. Similarly, designs that employ $O(100)$-port OCSes to connect pods (instead of racks) replace less of the electronic network, limiting their cost effectiveness. Finally, even if the OCS radix could be increased or signal loss reduced, commercial crossbar OCSes still reconfigure much too slowly to support datacenter traffic dynamics.

In Rotor switches, switching elements need only differentiate between the number of *matchings*—as opposed to the number of *ports* in the crossbar—making them fundamentally more scalable. For example, in a 2,048-rack RotorNet, the number of matchings in each Rotor switch is two-orders-of-magnitude smaller than the switch radix. We show in Section 6.1 that Rotor switches with 2,048 ports can achieve a reconfiguration time of 20 μs using existing technology. Thus, Rotor switches can connect 1000s of racks in the datacenter, and achieve a response time on the same order as other state-of-the-art approaches [12, 15].

4.1.2 Reducing cycle time through sharding. In a circuit-switched network, connecting the endpoints is only half the issue; the other half is ensuring timely service. This involves managing the delay due to circuit reconfigurations, and for RotorNet, waiting until connectivity is established through one of the Rotor matchings.

The time it takes to cycle through all matchings in RotorNet is a critical metric that gates how much time passes before all endpoints have an opportunity to communicate with each other. To speed up the rate at which we cycle through matchings, we employ a different subset of matching patterns in each Rotor switch, as

Number Rotor switches ($N_{sw} = N_{up}$):	512	256	128
Number matchings / switch (N_m):	4	8	16
Cycle time (μs) at duty cycle = 0.9:	800	1,600	3,200
Cycle time (μs) at duty cycle = 0.75:	320	640	1,280

Table 2: Trade-offs between the main parameters in a 2,048-rack RotorNet, assuming a 20-μs reconfiguration delay. For reasonable values of N_{sw} and duty cycles, the entire network-wide cycle time can remain on the order of 1 ms.

shown in Figure 4(b). The number of matchings implemented by each switch is $N_m = \lceil (N_R - 1)/N_{sw} \rceil$.

Table 2 shows the cycle times for a 2,048-rack RotorNet using various numbers of Rotor switches and duty cycles. The duty cycle is the fraction of time traffic can be sent over the network, accounting for the time the OCS spends reconfiguring. For a given reconfiguration speed and number of matchings, a higher duty cycle leads to a longer matching cycle. For the configurations shown in Table 2—including the particular realization described below—cycle times on the order of 1 ms are possible. Such delays are shorter than disk access times, and would thus serve disk-based data transfers well.

4.1.3 Supporting low-latency traffic. For flows with latency requirements smaller than the cycle time (≈1 ms), we rely on previously demonstrated hybrid approaches [13, 20, 32], where low-latency data is sent over a heavily over-subscribed packet-switched network, and all other traffic traverses RotorNet. To achieve a hybrid architecture, our design simply faces a fraction of the upward-facing ToR ports toward a packet-switched network, as shown in Figure 4(b). Applications are required to choose which traffic is sent over the packet-switched network, setting QoS bits in the packet header which then trigger match-action rules in the ToR. This design supports a certain percentage of low-latency bandwidth, for example 10–20%. The remaining 80–90% transits RotorNet.

We note that RotorNet's control plane is self-contained—no part of it relies on the existence or operation of a packet-switched network. In particular, no control messages or configurations are sent over the packet-switched fabric. Thus, RotorNet management is entirely separate from any packet-switched network used to support a hybrid design.

4.2 An example network

As a concrete comparison point, consider a hypothetical 65,536-end-host network with 400-Gb/s links to each end host (a 26-Pb/s network). We design the network using $k = 64$-port electronic switches, each with an aggregate bandwidth of 25.6 Tb/s. Such switches are projected to be available in the next few years. While a Fat Tree treats each ToR port as a single 400-Gb/s link, in RotorNet we split the bandwidth of each ToR port, creating 128 100-Gb/s links which connect to a set of 128 Rotor switches. Commercial ToR switches today offer this same functionality—a single 100-Gb/s port can be broken into four logical 25-Gb/s links, providing a larger number of lower-bandwidth ports. While the upward-facing ToR ports in RotorNet are logically distinct, we package these ports into 32 400-Gb/s transceivers, where each transceiver has 4 transmit fibers and 4 receive fibers in a ribbon cable, with each fiber carrying

Network architecture	# EPS [# ports]	#TRX	# Rotors [# ports]	Agg. BW
1:1 Fat Tree	5.1 k [328 k]	262 k	0 [0]	100%
3:1 Fat Tree	2.6 k [168 k]	103 k	0 [0]	33%
RotorNet, 10% EPS	2.3 k [149 k]	84 k	128 [262 k]	50-100%
RotorNet, 20% EPS	2.5 k [162 k]	96 k	128 [262 k]	50-100%

Table 3: Components and relative bandwidths of 65,536-end-host networks built with Fat Tree and hybrid RotorNet architectures, assuming k = 64 port packet switches (and ToRs). RotorNet is cost-comparable with a 3:1 Fat Tree, but as we show in Section 7, delivers higher throughput.

a logically separated 100-Gb/s channel. We emphasize that because the optical signal loss of a Rotor switch is low (2 dB), RotorNet can use the same or similar (cost) optical transceivers used in a Fat Tree network. The fibers from each ToR are routed to a central location and broken out to connect to the N_{sw} = 128 Rotor switches. Each Rotor switch provides 2,048 ports, one for each rack in the network, and implements N_m = 16 matchings.

An explicit cost comparison between RotorNet (or any optical network proposal) and a Fat Tree depends on volume pricing of packet switches and transceivers, installation expenses, and the manufacturing cost of OCSes. These cost numbers are not publicly available, so we use the number of components as a proxy for cost when comparing networks. Table 3 shows the component counts—including electrical packet switches (EPS), transceivers (TRX), and Rotor switches—for this 65,536-end-host network built with fully provisioned (1:1) and over-subscribed (3:1) Fat Tree topologies, as well as hybrid RotorNet topologies with 10% and 20% packet switching bandwidth. RotorNet requires fewer packet switches and transceivers than both the Fat Trees, but does require optical switching hardware that the electronic networks do not. Because Rotor switches can be mass-produced as described below, the per-port cost of a Rotor switch can be less than that of a packet switch. Considering that optical transceivers are the dominant cost in today's datacenter network fabrics [29], we estimate that for between 10–20% packet switching, a hybrid RotorNet will be cost-competitive with a 3:1 over-subscribed Fat Tree.

4.3 Manufacturing and deployment

While the matching patterns in Figure 4 provide complete connectivity, for pragmatic reasons we choose a different set of $(N_R - 1)$ matchings which consist of only bidirectional connections (i.e. their adjacency matrix is symmetric).

To reduce manufacturing cost, instead of configuring each Rotor switch with unique matching patterns, we can instead build all Rotor switches with the same set of internal bidirectional matchings and simply permute the input wiring pattern to each switch in order to realize disjoint matchings among Rotor switches. This approach dramatically reduces cost because it requires only one unique optical element to perform the matchings internal to each switch, rather than having to build N_{sw} unique switches. Space does not permit a full discussion, but the main limitation of such an approach is that the number of Rotor switches must be a power

of two. We leave to future work the investigation of other methods for arranging the internal connectivity of Rotor switches.

RotorNet also offers a path for incremental deployment to reduce the upfront cost. Consider an eventual deployment that will support N_R racks with N_{sw} Rotor switches. By choosing the appropriate input port wirings to Rotor switches (or equivalently the matching patterns contained within each switch), N_R/X racks can be deployed with each ToR switch having $1/X$ of its upward-facing ports populated, connecting to N_{sw}/X Rotor switches. This configuration provides a network with $1/X^2$ the bisection bandwidth. Similar to the matching patterns described above, the only constraint on this approach is that X be a power of two.

5 DISTRIBUTED INDIRECTION

Top-of-rack switches in RotorNet implement RotorLB (RotorNet Load Balancing), a lossless, fully distributed protocol based on the principle of Valiant load balancing [30]. When indirecting traffic, RotorLB injects traffic into the network fabric exactly two times: traffic is first sent to an intermediate rack, where it is temporarily stored, and then forwarded to its final destination. RotorLB stitches together two-hop paths over time; i.e., the source is connected to an intermediate rack during one matching, but the intermediate rack is connected to the destination in a subsequent matching.

Unlike traditional VLB, which always sends traffic over random two-hop paths, RotorLB (1) prioritizes sending traffic to the destination directly (over one-hop paths) when possible, and (2) only injects new indirect traffic when that traffic will not subsequently interfere with the intermediate rack's ability to send traffic directly. These two policies improve network throughput by up to 2× (for uniform traffic) compared to traditional VLB.

5.1 ToR and end-host responsibilities

In RotorLB, each ToR switch is responsible for keeping an up-to-date picture of the demand of each end host within the rack and for exchanging in-band control information with other ToRs. There are two types of traffic the ToR must track: *local* traffic generated by hosts within the rack, and *non-local* traffic that is being indirected through the rack. Each ToR is responsible for pulling traffic from end hosts at the appropriate times and managing the storage of non-local traffic. Indirect traffic can be stored in off-chip memory at the ToR or in DRAM at the end hosts. We analyze the amount of memory required for buffering indirect traffic later in Section 7.5. RDMA can achieve microsecond latencies when pulling/pushing data directly from/to end-host memory.

Each end host, in turn, must send to the ToR a message when it has traffic to send, including the quantity and destination of the traffic. This information can be generated by inspecting transmit queues or relaying application send calls. Following prior work on inter-rack datacenter networks [13, 15, 17], we abstract away intra-host and intra-rack bottlenecks (which also exist in packet-switched networks) and focus on RotorNet's ability to handle inter-rack traffic.

5.2 RotorLB algorithm and example

The RotorLB algorithm runs on each ToR switch. At the start of each *matching slot* (the time period for which a Rotor switch implements

---

Algorithm 1 RotorLB Algorithm

> **function** PHASE 1(Enqueued data, slot length)
> 　　$alloc \leftarrow$ maximum possible direct data
> 　　$capacity \leftarrow$ slot length minus $alloc$
> 　　$offer \leftarrow$ remaining local data
> 　　send $offer$, $capacity$ to connected nodes　　　　　▷ offer
> 　　send allocated direct data
> 　　$remain \leftarrow$ size of unallocated direct data
> 　　**return** $remain$
> **function** PHASE 2($remain$, LB length)
> 　　recv $offer$ and $capacity$ from connected nodes
> 　　$indir \leftarrow$ no allocated data
> 　　$avail \leftarrow$ LB length minus $remain$
> 　　$offer_i \leftarrow offer_i$ if $avail_i \neq 0$
> 　　$offerscl \leftarrow$ fairshare of $capacity$ over $offer$
> 　　**while** $offerscl$ has nonzero columns **do**
> 　　　　**for all** nonzero columns i in $offerscl$ **do**
> 　　　　　　$tmpfs \leftarrow$ fairshare of $avail_i$ over $offerscl_i$
> 　　　　　　$avail_i \leftarrow avail_i - sum(tmpfs)$
> 　　　　　　$indir \leftarrow tmpfs$
> 　　　　$offerscl \leftarrow offer - indir$
> 　　　　$tmplc \leftarrow capacity - sum(indir)$
> 　　　　$offerscl \leftarrow$ fairshare of $tmplc$ over $offerscl$
> 　　send $indir$ to connected nodes　　　　　　　　▷ accept
> **function** PHASE 3(Enqueued local data)
> 　　recv $indir$ from connected nodes
> 　　$local_i \leftarrow$ enqueued local data for host i
> 　　$indir_i \leftarrow min(indir_i, local_i)$
> 　　send $indir_i$ indirect local data for host i

---

each matching), RotorLB determines the quantities of direct and indirect traffic to send during that slot. Before sending new indirect traffic into the network, RotorLB prioritizes the delivery of stored non-local traffic (which is on its second hop) as well as local traffic that can be sent directly to the destination in the current slot. With any remaining link capacity in that slot, RotorLB indirects traffic on a per-destination granularity. To limit buffering and bound delivery time of indirected traffic, RotorLB only indirects as much traffic as can be delivered within the next matching cycle (the full cycle of matching slots). We use a pairwise "offer/accept" protocol between ToRs to exchange current traffic conditions, and then adaptively determine the amount of indirect traffic to be sent based on those conditions. The fraction of link bandwidth used to support indirect traffic varies between 0% (if the rack's locally-generated traffic can saturate the link) and 100% (if the rack has no locally-generated traffic to send during a matching slot, and would otherwise be idle).

To most effectively balance load, we allow traffic from the same flow to be sent over RotorNet's single one-hop path and also to be indirected over multiple two-hop paths. This multipathing can lead to out-of-order delivery at the receiver. Ordered delivery can be ensured using a reorder buffer at the receiver, and we evaluate this approach in more detail in Section 7.3.

Below, we describe the basic operation of the RotorLB algorithm, moving through a simple example from the perspective of a single
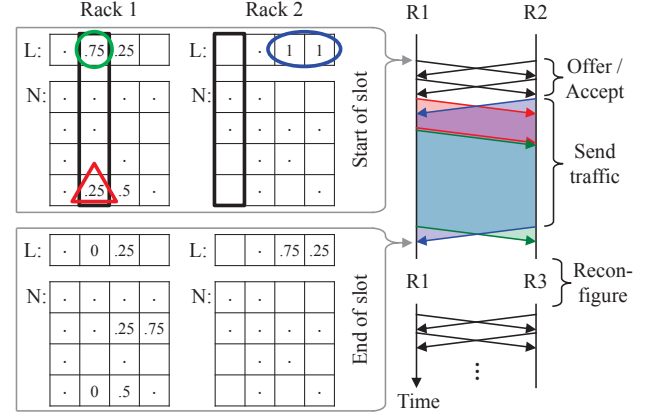

Figure 5: RotorLB example. Matrix rows represent sources and columns represent destinations; L and N represent local and non-local traffic queues, respectively; matrix elements show normalized traffic demand. In the current matching between racks 1 and 2, traffic which can be sent directly is bounded by black rectangles, stored indirect traffic is marked by a red triangle, one-hop direct traffic is marked by a green circle, and new indirect traffic is indicated by a blue oval.

Rotor switch connection over the course of one matching slot. The RotorLB algorithm is outlined in Algorithm 1.

Consider the ToRs of two racks, R1 and R2, which have current demand information for the hosts within each rack stored in non-local (N) and local (L) queues, as shown in Figure 5. In this example, demands are normalized so that one unit of demand can be sent over the ToR uplink in one matching slot. Note that, as described below, there is no central collection of demand–each host simply shares its demand with its ToR switch, and ToR switches share aggregated demand information in a pairwise fashion.

Phase 1: Send stored non-local and local traffic directly. Connectivity in RotorNet is predictable, and each ToR switch anticipates the start of the upcoming matching slot as well as to which rack it will be connected. After taking a snapshot of the N and L queues, the ToR computes the amount of traffic destined for the upcoming rack. Delivery of stored non-local traffic on its second (and final) hop is prioritized to ensure data is not queued at the intermediate rack for long periods of time. Delivery of local traffic has the next priority level. In Figure 5, R1 has 0.25 units of stored non-local traffic (red triangle) and 0.75 units of local traffic (green circle) destined for R2, so it allocates the entire ToR uplink capacity for the matching slot duration to send this traffic. R2 has no stored non-local or local traffic for R1, so no allocation is made.

The ToR then forms a RotorLB protocol *offer* packet which contains the amount of local traffic and the ToR uplink capacity which will remain after the allocated data is sent directly. The smaller of the two quantities constitutes the amount of indirect traffic the ToR can offer to other racks. Once the matching slot starts, the ToR sends the offer packet to the connected rack. As an optimization, rather than waiting for the entire offer/accept process to complete,

the ToR can also begin sending the stored non-local and local traffic which was been allocated for direct delivery to the destination.

Phase 2: Allocate buffer space for new non-local traffic. Shortly after the start of the slot, the ToR switch receives the protocol packet containing the remote rack's offer of indirect traffic. At this point, it computes how much non-local traffic it can accept from the remote rack. To do this, the ToR examines how much local and non-local traffic remain from Phase 1. The amount of non-local traffic it can accept per destination is equal to the difference between amount of traffic that can be sent during one matching slot and the total queued local and non-local traffic. Because the amount of accepted indirect traffic is limited to the amount that can be delivered in the next matching slot (accounting for any previously-enqueued traffic), the maximum delivery time of indirect traffic is bounded to $N_m + 1$ matching slots, or approximately one matching cycle (see Section 7.3). The algorithm handles multiple simultaneous connections by fair-sharing capacity across them.

In Figure 5, R1 sees via the offer packet that R2 would like to forward 1 unit of traffic destined for each R3 and R4 (blue oval), and that R2 has a full-capacity link to forward that data. R1 already has 0.25 units of local traffic for R3 and 0.5 units of stored non-local traffic for R3. Therefore, it allocates space to receive $1 - 0.75 = 0.25$ units destined for R3 and 0.75 units for R4 from R2, which fully utilizes the remaining link capacity from R2 and ensures that all queued traffic at R1 will be admissible.

Once the allocation is made to receive non-local traffic, the ToR switch responds with a protocol *accept* packet informing the remote rack how much traffic it can forward on a per-destination basis.

Phase 3: Forward local traffic indirectly. Finally, the ToR switch receives the protocol accept packet from the remote rack. After it finishes sending direct traffic determined in Phase 1, it forwards new non-local traffic to the remote rack per the allocation specified by the accept packet.

In Figure 5, R2 receives an accept packet informing it that 0.25 units of traffic destined for R3 and 0.75 units destined for R4 may be sent. It forwards this traffic, which is stored as non-local traffic at R1. Finally, the Rotor switch reconfigures and establishes a new connection, and the RotorLB algorithm runs again.

5.3 Fault tolerance

A simple extension to RotorLB, called RotorLB-FT, ensures that the network is tolerant to failures. The key idea is to rely on indirect paths to "route around" any such failures. Failures of a link, ToR, or Rotor switch are discovered at the beginning of a particular matching slot, because a rack will not receive RotorLB protocol messages over the link corresponding to the failure.

The primary modification to the algorithm is to give traffic which would traverse a failed network element priority over all other traffic, so that it is ensured indirect bandwidth regardless of the traffic pattern. Each ToR switch first determines the number of candidate fault flows. This is the number of locally generated flows which both see a fault on a one-hop path out of the local rack and do not see a fault on a one-hop path out of the currently connected rack (i.e. a clear two-hop path exists). The ToR switch then allocates at most $1/(2(N_R - 1) - 1)$ of the uplink bandwidth to each candidate
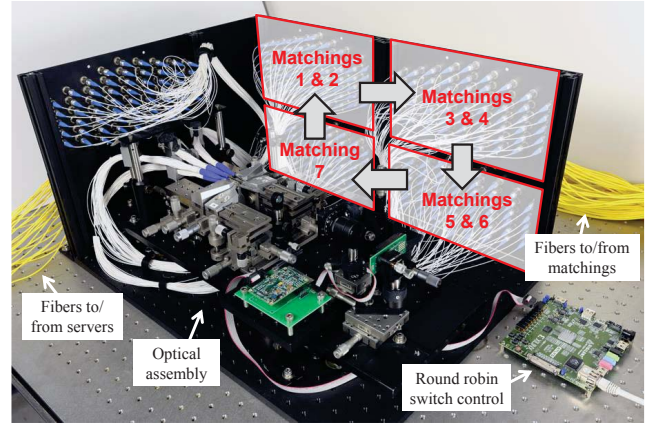

Figure 6: We installed 7 Rotor matchings into a prototype OCS to create two parallel Rotor switches. An FPGA sets the switches to cycle through the matchings in open loop.

fault flow. This factor is chosen conservatively so that if there is only one remaining uplink out of a given rack, and every other rack sends fault traffic to that rack, the sum of the fault traffic and an equal share of local traffic will be admissible to leave the rack. In other words, the overall algorithm is still guaranteed to be starvation free with a bounded delivery time of $N_m + 1$ matching slots. The performance of RotorLB-FT is evaluated in Section 7.7.

6 ROTORNET PROTOTYPE

We evaluate the feasibility of RotorNet and demonstrate Rotor switching in operation by constructing a small-scale prototype network that runs RotorLB to communicate data between endpoints. We also use the prototype results to validate our model of RotorNet, which we use to simulate the behavior of RotorNet at large scale later in Section 7.

6.1 Prototype architecture

To construct our prototype, we need an optical Rotor switch to instantiate the matching patterns specified by the RotorNet architecture. However, no commercial optical switches support the partial connectivity necessary to demonstrate the advantages of Rotor switching. Rather than constructing a Rotor switch from scratch, we modified a research device called a Selector switch [25] which is particularly well-suited for extension to a Rotor switch. Specifically, this prototype Selector switch is a gang switch that allows an array of 61 single-mode fiber input ports to be switched, as a group, between four output arrays of 61 fibers with a reconfiguration speed of 150 μs.

We constructed a Rotor switch from this Selector switch, using an array of fiber optics to hard-wire the Rotor matchings into the switch. More precisely, to support 8 endpoints, we divided the monolithic Selector switch into two Rotor switches, and, following Section 4, we distributed a full set of 7 Rotor matchings across the two switches, with each Rotor switch implementing either 3 or 4 matching patterns. Next, we programmed an FPGA to send control signals to the Rotor switches so they rotated through the

Rotor matchings in an open loop. This setup is summarized in Figure 6. The end result is two Rotor switches which support up to 8 endpoints, reconfigure 100–1,000× faster than commercial OCSes, and are compatible with commercial optical transceivers without requiring optical amplification.

While the MEMS device used in the prototype is an off-the-shelf product and not optimized for speed, we conducted detailed optical analysis [25] to show that Selector switches can scale to at least 2,048 ports with 20-μs reconfiguration speed and 2-dB insertion loss. Neither of these quantities are fundamental scaling limits, but rather specific design points at which we assessed the switch's performance. These scaling results also apply to Rotor switches, as they are essentially Selector switches with Rotor matching patterns installed. In this section, we report data gathered with the 150-μs prototype switch, but assume a reconfiguration speed of 20 μs in the analysis in Section 7.

RotorLB on endpoints. Using the prototype Rotor switches, we built an eight-ToR RotorNet ($N_R = 8$), using commodity end hosts to emulate ToRs. Each end host was equipped with a dual-port 10-Gb/s Myricom network interface controller (NIC) and two optical transceivers, establishing one 10-Gb/s optical link to each Rotor switch.[2] We implemented RotorLB on the end hosts as a user-level process, using the Myricom Sniffer API to directly inject and retrieve packets from the NIC. The only requirement to run RotorLB in practice is that endpoints be made aware of the Rotor switches' states. In a real implementation using ToR switches, each ToR could monitor the status of its optical links to determine when one matching ends (i.e., the link goes down), and the next matching begins (i.e., the link comes back up again). The commercial NICs we used did not have a built-in low-latency way to detect link up/down events, so we used an out-of-band channel to notify end hosts of the switch reconfiguration events.

6.2　Prototype evaluation of RotorLB

We emulated a RotorLB ToR switch on each server using distinct user-level threads to generate, send, and receive UDP traffic, with an additional thread to process state changes of the Rotor switches. To analyze performance under a variety of traffic conditions, we generated traffic patterns with different numbers of "heavy" connections. Each heavy connection attempts to send data at line rate. We define the *traffic density* as the fraction of heavy connections out of all possible connections (56 in our 8-endpoint prototype). For each traffic density, we repeated the experiments with 32 randomly-generated traffic matrices representing the inter-rack demand.

As a baseline, we first sent data through the network using only one-hop forwarding. Next, we repeated the experiments with RotorLB running on the endpoints. Figure 7 shows the relative network throughput under RotorLB normalized to that of one-hop forwarding. We see that RotorLB significantly improves throughput for sparse traffic patterns. For a single active heavy connection, RotorLB improves throughput by the expected factor of $N_R - 1$ (7 in this case), as traffic can now take advantage of all paths through the network. Further, we see that our implementation of RotorLB
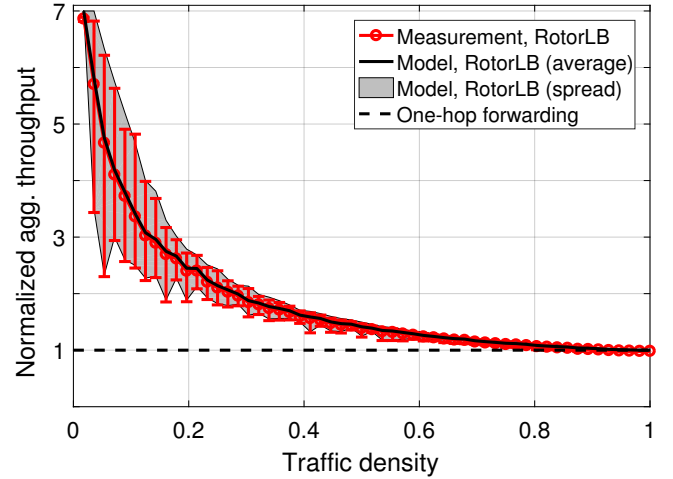
---

[2]10 Gb/s was chosen for convenience; the Rotor switch is rate agnostic.

Figure 7: Measured and modeled throughput under RotorLB relative to that using one-hop forwarding. Circles represent the average througput over 32 random traffic patterns; error bars show the maximum and mimimum.

adaptively converges to the throughput of one-hop forwarding for uniform traffic, as intended.

Figure 7 also overlays the modeled (Section 7) relative throughput of RotorLB to one-hop forwarding for the same set of traffic conditions used in the measurements. The close agreement between model and measurement demonstrates that our RotorLB implementation operates as designed, and also validates our model's ability to accurately predict RotorLB performance in practice. We use this model to explore the performance of RotorNet at scale.

7　EVALUATION

In this section, we evaluate the behavior and performance of RotorNet at a larger scale than was feasible to prototype. In particular, we employ three distinct simulators at varying levels of fidelity and validate the results of our packet-level simulation against a Mininet-based emulation.

7.1　Methodology

Depending on the simulator, we consider two different types of traffic models, fluid flow and flow level. We describe each below.

Fluid-flow model: We start with a network model that treats traffic as a continuous fluid flow, allowing us to measure the bandwidth, delivery time, and buffering requirements for stable and dynamic traffic patterns using Matlab. This model maximizes the amount of data sent through RotorNet. For purposes of comparison, we model a Fat Tree as a single, ideal packet switch. In particular, the packet switch is ideal in that it connects all racks with a single non-blocking crosspoint fabric and its per-port bandwidth is assigned by a linear program solver configured to maximize throughput. This idealized packet switch upper bounds the performance of a Fat Tree network, which in practice has lower performance due to transport protocol and hashing effects.

Flow-level simulator: We also use a custom-written flow-level simulator that relies on discrete flow events to track the arrival and completion times of flows in RotorNet. While we do not simulate packets in this method, the flow-level results approximate the performance of RotorNet with a lossless transport protocol running on top of it, such as RDMA over Converged Ethernet (RoCE).

Packet-level simulator: We also model a baseline 3-level Fat Tree using the ns-3 packet-level simulator. This allows us to model the effects of TCP under flow hashing and port contention. Under moderate to heavy network load, these effects lead to a long tail in flow completion times in Fat Trees, as previously observed [33].

Mininet emulator: In addition to ns-3, we model the baseline 3-level Fat Tree using Mininet [18] with 100-Mb/s links. This allows us to corroborate our ns-3 findings using a real TCP stack, though we still require ns-3 to validate our work at multi-gigabit scale.

7.1.1 Traffic models. Our throughput analyses employ two distinct traffic models, described below. We defer discussion of the traffic characteristics used to evaluate flow completion time until Section 7.4.

Synthetic stochastic: We model inter-rack traffic as a random process, and vary the total number of heavy connections (varying the *traffic density* as defined in Section 6.2). For the fluid-flow network models, we represent traffic as a random binary matrix with elements that are either 1, representing an elephant flow(s) between a given rack pair, or 0, representing no traffic exchanged between that rack pair. In a hybrid deployment, low-latency and mice flows would be "swept" to the packet switch, resulting in a 0 entry in the demand matrix. For each fixed traffic density, we individually evaluate 100 randomly-generated traffic matrices, showing the average and error bars across runs in the plots.

Commercial traces: In addition to randomly-generated binary demand matrices, we also model traffic patterns reported in production datacenters. Following the approach of Ghobadi et al. [15], we use published traffic-matrix heatmaps collected over 5-minute intervals to define the probability of inter-rack communication. We then use a Poisson generation process to determine flow arrival times and drew flow sizes from reported distributions. We use traces reported from a Microsoft datacenter by Ghobadi et al. [15] and from two different Facebook cluster types, Hadoop and front-end web-server [11, 27]. Given RotorNet's cycle time of approximately 1 ms, we model flows arriving within 1-ms windows using the Poisson process. Finally, to emulate a hybrid RotorNet with 10% packet-switched bandwidth, we retain the largest flows that contributed 90% of the total bytes and input that to the RotorNet model.

7.2 Throughput analysis

We compare the throughput of RotorNet to that of an idealized packet switch using the fluid-flow model described in Section 7.1, considering basic one-hop forwarding, RotorLB, and a computationally intensive linear program that provides an upper bound on potential throughput. For reference, we include the throughput of a 3:1 over-subscribed Fat Tree, which is estimated in Section 4.2 to be cost-comparable with RotorNet.
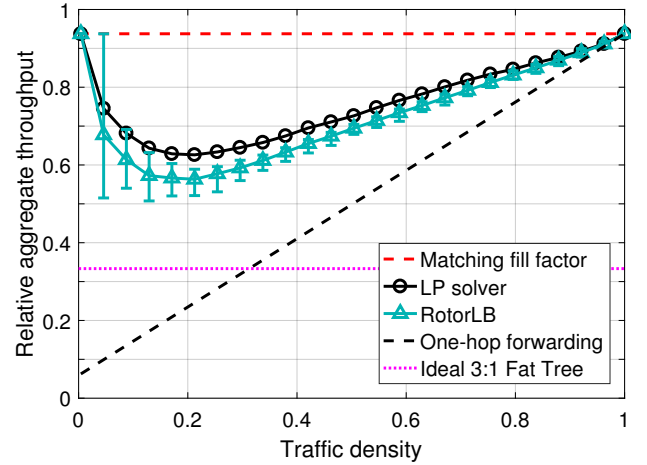
(a) $N_R = 16$, $N_m = 4$, **LP result shown for reference.**
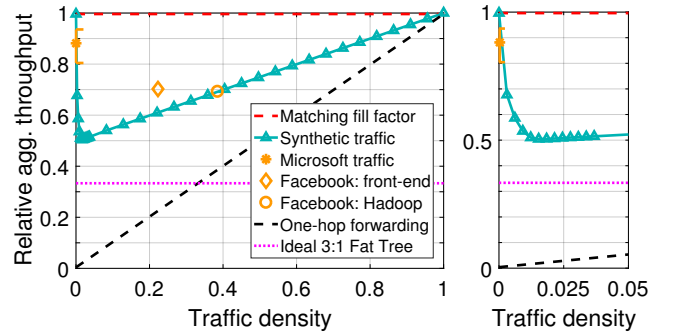
(b) $N_R = 256$, $N_m = 8$ **(c)** **Detail of (b)**

Figure 8: RotorLB throughput (relative to an ideal packet switch) vs. traffic density. (b) and (c) also indicate throughput under commercial datacenter traffic patterns.

Figure 8 shows the throughput of RotorLB normalized to that of the ideal, fully-provisioned electrical network. The matching fill factor, which is how perfectly the Rotor matchings are distributed into Rotor switches ($(N_R - 1)/(N_m N_{sw})$), upper bounds the throughput. While the fill factor is 0.9375 with only $N_R = 16$ racks, Figure 8b shows that the matching fill factor approaches 1 by $N_R = 256$ racks.

As expected, after accounting for the matching fill factor, one-hop forwarding results in 100% throughput for uniform traffic, but throughput is linearly reduced for skewed traffic. Before assessing RotorLB, we use a linear program (LP) solver to calculate the upper-bound throughput for RotorNet under ideal forwarding; that is, if we had perfect knowledge of the network-wide traffic demand, and indirect traffic was not restricted to a maximum of two hops. The result is plotted in Figure 8a for $N_R = 16$ (it is not feasible to run the LP solver at larger scales).

Next, we calculate the throughput for the RotorLB protocol described in Section 5. These calculations assume constant traffic patterns; we explore the responsiveness of RotorLB to changing traffic patterns in Section 7.6. Figure 8a shows that RotorLB achieves close to the ideal (LP-solver) throughput over all traffic densities

despite its fully distributed implementation. Figure 8b shows RotorLB's throughput at scale for 256 racks. Figure 8c highlights that, at scale, RotorLB's throughput is within a factor of two of an ideal fully-provisioned packet network independent of traffic conditions. Recall from Section 3 that this worst-case factor-of-two reduction in bandwidth is expected from RotorLB's two-hop routing (i.e. up to half of RotorNet's bandwidth is used by traffic on its second hop). We find that, out of all traffic patterns studied, permutation traffic yields the worst-case relative throughput for RotorNet compared to an ideal packet network—a worst-case of 50% relative throughput. We observe this factor-of-two lower-bound holds for larger-scale networks with $N_R > 256$.

Finally, following Section 7.1, we model the throughput of RotorLB in a 256-rack RotorNet under a number of production datacenter traffic patterns reported in the literature. The Microsoft traffic pattern is very sparse, with only a small number of rack pairs communicating in each time interval. The pattern varies between millisecond time intervals, leading to the slight variance in throughput indicated in Figure 8b. The Facebook Hadoop cluster traffic pattern exhibits denser connections, yielding a throughput nearly equivalent to random traffic. The Facebook front-end pattern displays heavy multicast and incast characteristics, setting its throughput apart from that of random traffic.

For the datacenter traffic patterns considered, RotorNet provides 70–95% the throughput of an ideal fully-provisioned electrical network, despite being significantly less expensive, as discussed in Section 4.2. Compared to a 3:1 over-subscribed Fat Tree of approximately equal cost, RotorNet delivers 1.6× the throughput under worst-case (permutation) traffic, 2.3× the throughput for reported datacenter traffic patterns, and up to 3× the throughput for uniform traffic.

## 7.3 Bounded delivery time and reordering

By design, RotorNet offers strictly bounded delivery time for one-hop traffic because it cyclically provisions bandwidth between all network endpoints. However, under RotorLB, two-hop traffic is delivered later than one-hop traffic. Because data from the same flow may be sent over both one-hop and two-hop paths (as discussed in Section 5), care must be taken to ensure data is delivered in order to the receiver. Here, we evaluate the delivery characteristics and requirements for ordered delivery of traffic under RotorLB.

Adopting terminology from MPTCP, we refer to the fractions of a single flow that are sent over different one-hop and two-hop paths as *subflows*. As discussed in Section 5, a given rack will only accept as much two-hop traffic to each destination as it can deliver in one matching slot. Therefore, we expect the worst-case delivery time of a two-hop subflow to be equal to $N_m + 1$ matching slots (i.e. one matching *cycle* + one matching *slot*). This worst case occurs when traffic is forwarded to an intermediate rack and that intermediate rack is currently connected to the destination. This can happen because RotorLB does not forward indirect traffic during the same matching slot in which it is received.

To test this behavior, we track the start time and tail delivery time of all subflows over a variety of traffic patterns in a RotorNet with 256 racks and $N_m = 8$ matchings per switch. The results are shown in Figure 9. Figure 9a shows that, indeed, no subflows exceed
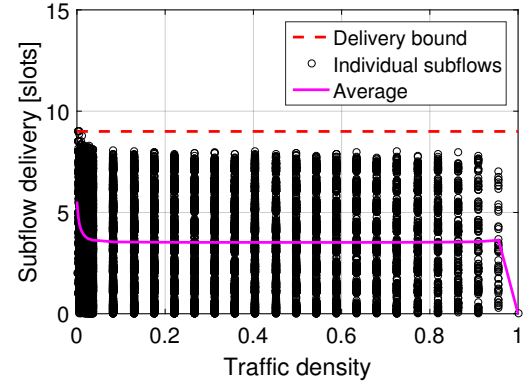
(a) Subflow delivery times for all traffic densities.
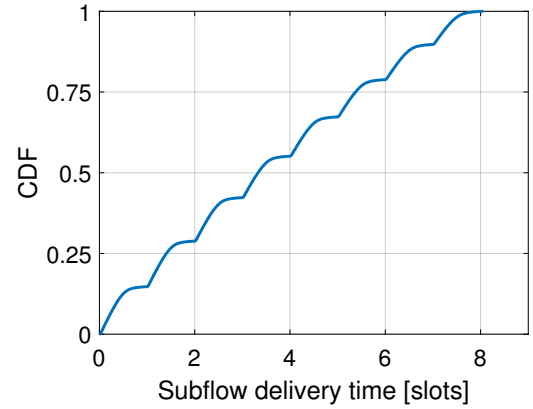
(b) CDF for traffic density = 0.2 from (a).

Figure 9: The delivery time of two-hop subflows is bounded (shown for $N_R = 256$, $N_m = 8$).

the designed delivery bound of $N_m + 1 = 9$ matching slots for any traffic density. Note that although only a subset of subflows are plotted, the scatter plot is representative of the total distribution.

Looking in more detail at the distribution of subflow delivery times for a given traffic density in Figure 9b, we see that subflows are approximately uniformly distributed in time with two notable exceptions. First, because delivery of stored two-hop traffic is prioritized, two-hop subflows are more likely to be delivered at the start of a matching slot, accounting for the slight non-uniformity in the CDF. Second, for traffic densities above approximately 0.05, the worst-case delivery time is rare, and most subflows are delivered within one matching cycle (rather than one cycle + one slot).

Subflows injected into RotorNet will be sequentially ordered between matching slots, but during a single matching slot subflows from the same flow may take both one-hop and two-hop paths, depending on the quantity and the time that traffic is committed to the network by the application. If data sent over a two-hop subflow logically precedes data sent over a one-hop flow, that data will arrive out of order at the receiver. We assign the responsibility of ensuring ordered delivery of data to a reordering process at each end host. Fortunately, because all injected traffic is guaranteed to be delivered in $N_m + 1$ matching slots, the receiver only needs to store
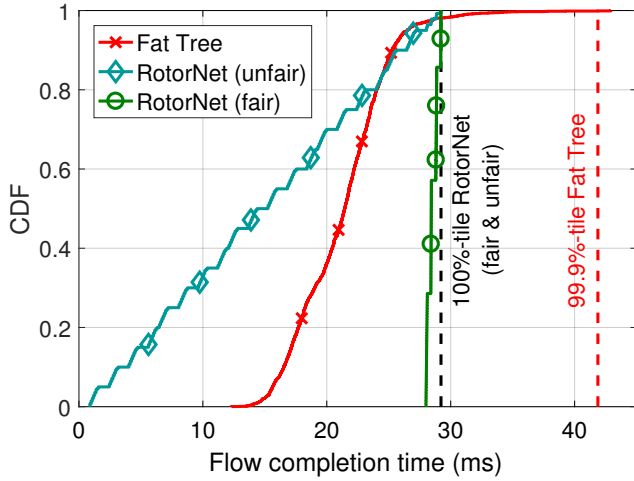
Figure 10: Flow completion times in RotorNet and a three-level Fat Tree, both with k = 4-port ToRs, 10-Gb/s links, and 140 200-kB flows per end host. Two examples of bandwidth allocation in RotorNet are shown: fair sharing bandwidth across flows and greedily allocating bandwidth to flows one-by-one.

data received within a sliding window equal to that delivery bound. For a 100-Gb/s end host link, 200-μs slot time, and N_m = 8, each host would need a receive buffer of 23 MB to ensure ordered delivery of all incoming flows. This is a modest memory requirement given the specifications of modern servers.

7.4 Flow completion times

Flow completion time (FCT) remains an important metric for network performance, especially considering the prevalence of services that are impacted by the presence of a long tail in the FCT distribution [33]. While contemporary multi-rooted trees coupled with TCP can provide sufficient bandwidth on paper for applications, FCT in such networks is subject to considerable variance. Here, RotorNet improves on the state of the art by providing a choice between either low FCT variance or significantly reduced mean FCT compared to Fat Trees, while in both cases eliminating the long tail present in Fat Trees.

To demonstrate this capability, we simulated a 10-Gb/s k = 4 three-level Fat Tree in ns-3, and an equivalently sized RotorNet using our flow-level simulator. Each host communicates with cross-rack hosts in an all-to-all pattern with 10 flows of size 200 KB each, for a total of 2,240 flows overall. Figure 10 depicts the FCT in milliseconds for the Fat Tree, and for greedy flow completion and fair-shared bandwidth strategies for RotorNet. Note that the p100 for both RotorNet strategies is identical and a full 30% smaller than p99.9 for the Fat Tree. In addition, p50 for greedy flow completion RotorNet is 33% smaller than the Fat Tree. For the fair-shared bandwidth strategy, the variance (normalized to the mean) is just 0.6%, compared to 61% for the Fat Tree.

In order to reduce FCT variance for a Fat Tree network, a network operator might consider operating the network at lower utilization
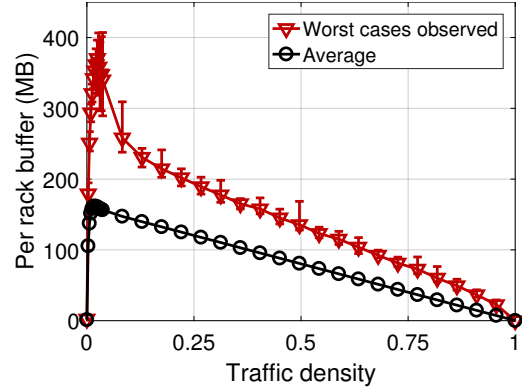
(a) Per-rack buffering for all traffic densities.
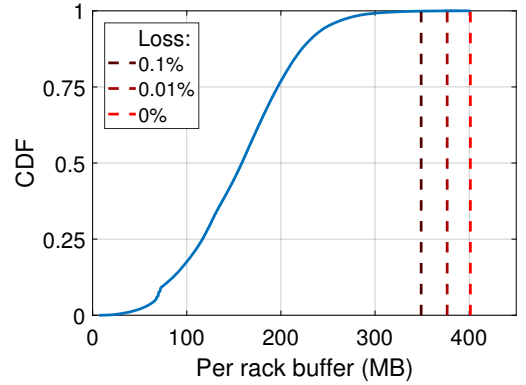
(b) CDF for traffic density = 0.02 from (a).

Figure 11: Per-rack buffering required for a 100-Gb/s deployment, 20-μs switching, 90% duty cycle, and N_R = 256.

levels. To characterize how much lower, we deploy a 100-Mb/s k = 4 Fat Tree on Mininet[18] and test the same traffic pattern. To emulate lower network utilization, we cap the flow size and rate to 10% and 20% of the nominal values. While reducing the utilization to 10% brings mean-normalized variance down to 22% from 70%, we note that the value is still well above that for fair-shared RotorNet. Thus, the low utilizations typical in real-world datacenter networks can be considered as an implicit duty-cycle cost levied to keep variations in performance low.

7.5 Buffering

Here we use the fluid-flow model from Section 7.1 to determine the amount of buffering required at each rack to support two-hop traffic under various traffic patterns.

Figure 11a shows the observed worst cases and average amount of buffering needed at each rack to support various traffic densities, with 100 randomly generated traffic patterns per density. In general, buffering can be expressed as the product of upward-facing ToR bandwidth and matching slot time, however to make the results concrete we show absolute storage in bytes, based on 100-Gb/s links, a reconfiguration delay of 20 μs, and duty cycle of 90%. We see that the largest amount of required buffering occurs for a traffic density of approximately 0.02, resulting in each rack requiring 400
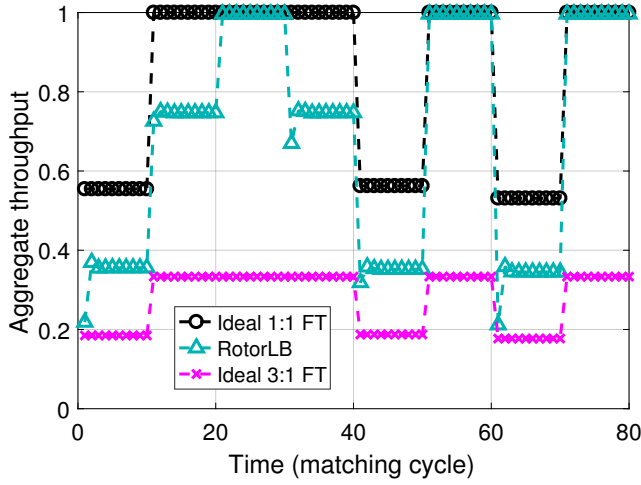
Figure 12: Responsiveness of RotorLB and ideal 1:1 and 3:1 Fat Trees (FT) to changing traffic patterns, N_R = 256. RotorLB converges within two matching cycles.

Failure type:	1 link	25% Rotor sw.	50% Rotor sw.
RotorLB:	0.97 (0)	0.73 (0)	0.53 (0)
RotorLB-FT:	0.99 (0.38)	0.89 (0.3)	0.70 (0.07)

Table 4: Jain's fairness index across flows (higher is better) and (in parenthesis, higher is better) ratio of minimum to maximum bandwidth of flows under various failure scenarios. RotorLB-FT promotes fair allocation of bandwidth under failures and, critically, ensures all flows get non-zero bandwidth.

MB of memory for two-hop traffic. If this traffic were stored on end hosts, rather than at the ToR switch, 12.5 MB per end host would be required assuming 32 end hosts per rack.

Next, we consider the amount of memory which could be saved by permitting a small amount of loss in the network (as opposed to the strictly lossless fabric we have discussed so far). Figure 11b shows a CDF of the per-rack buffer requirements for the worst-case traffic density observed in Figure 11a, which occurs at a traffic density of approximately 0.02. We see that increasing the loss rate to 0.01% only reduces the memory requirement by about 6%. This small achievable reduction in buffering gained by permitting loss supports our decision to design RotorNet as a lossless fabric.

7.6 Responsiveness

Here, we assess how quickly RotorLB responds to changing traffic patterns. We use the fluid-flow model from Section 7.1, and abruptly switch between different traffic patterns as RotorLB runs continuously. Figure 12 shows a typical time series of the aggregate throughput per matching cycle under RotorLB as we vary the traffic pattern. The throughputs of a fully-provisioned and 3:1 over-subscribed ideal packet-switched network (as described in Section 7.2) are shown for reference. Upon changing the traffic pattern, RotorLB converges to the new sustained throughput within two matching cycles. This fast response is because, as described in Section 7.3, two-hop traffic is drained from intermediate queues after one matching cycle, allowing RotorLB to adapt to the new traffic pattern within two matching cycles.

7.7 Fault tolerance

Recall from Section 3 that each Rotor switch implements only a fraction of RotorNet's connectivity, meaning that the failure of one or more Rotor switches could lead to a significant reduction in overall connectivity. However, we find that using the RotorLB-FT protocol described in Section 5.3, network connectivity is retained

even under extensive failure conditions. Table 4 shows fairness metrics for RotorLB and RotorLB-FT subject to various failures under all-to-all traffic. We consider a single link failure, a case in which one quarter of the Rotor switches fail, and a case where half of the Rotor switches fail. RotorLB-FT scores higher on Jain's fairness index [19] than RotorLB by 20% and 30% for the cases that one-quarter and one-half of the Rotor switches fail, respectively. Critically, RotorLB-FT provides non-zero bandwidth to all flows in all failure scenarios considered, whereas RotorLB delivers zero bandwidth to some flows even in the case of a single link failure.

8 CONCLUSIONS

Optical switching holds the promise of overcoming impending limitations in electrical packet switching, yet has seen resistance to industrial adoption due to practical barriers. In this paper, we describe a network fabric based on Rotor switches which decouples the control of OCSes from the rest of the network, greatly simplifying network control and deployment. Additionally, because Rotor switching has such a simple implementation, we can design new OCSes that leverage this new-found simplicity for greater scalability. The combination of these two capabilities has the potential to meet next-generation datacenter bandwidth demands in a manner much simpler than existing approaches.

REFERENCES

[1] Dan Alistarh, Hitesh Ballani, Paolo Costa, Adam Funnell, Joshua Benjamin, Philip Watts, and Benn Thomsen. A High-Radix, Low-Latency Optical Switch for Data Centers *(SIGCOMM '15)*.
[2] Thomas Beth and Volker Hatz. 1991. A Restricted Crossbar Implementation and Its Applications. *SIGARCH Comput. Archit. News* 19, 6 (Dec. 1991).
[3] Garrett Birkhoff. 1946. Tres Observaciones Sobre el Algebra Lineal. *Univ. Nac. Tucumán Rev. Ser. A* 5 (1946).
[4] Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, and Pramod Viswanath. Costly Circuits, Submodular Schedules and Approximate Carathéodory Theorems *(SIGMETRICS '16)*.
[5] Cheng-Shang Chang, Duan-Shin Lee, and Yi-Shean Jou. 2002. Load Balanced Birkhoff–von Neumann Switches, Part I: One-stage Buffering. *Computer Communications* 25, 6 (2002), 611–622.
[6] Kai Chen, Ankit Singla, Atul Singh, Kishore Ramachandran, Lei Xu, Yueping Zhang, and Xitao Wen. OSA: An Optical Switching Architecture for Data Center Networks and Unprecedented Flexibility *(NSDI '12)*.

[7] Kai Chen, Xitao Wen, Xingyu Ma, Yan Chen, Yong Xia, Chengchen Hu, and Qunfeng Dong. WaveCube: A Scalable, Fault-tolerant, High-performance Optical Data Center Architecture *(IEEE INFOCOM '15)*.

[8] Li Chen, Kai Chen, Joshua Zhu, Minlan Yu, George Porter, Chunming Qiao, and Shan Zhong. Enabling Wide-Spread Communications on Optical Fabric with MegaSwitch *(NSDI '17)*.

[9] Jack Edmonds. 1965. Paths, trees, and flowers. *Canad. J. Math.* 17 (1965).

[10] Facebook. 2014. Introducing data center fabric, the next-generation Facebook data center network. https://goo.gl/mvder2. (2014).

[11] Facebook. 2015. FBFlow Dataset. https://www.facebook.com/network-analytics. (2015).

[12] Nathan Farrington, Alex Forencich, George Porter, P-C Sun, Joseph E Ford, Yeshaiahu Fainman, George C Papen, and Amin Vahdat. 2013. A Multiport Microsecond Optical Circuit Switch for Data Center Networking. *IEEE Photonics Technology Letters* 25, 16 (2013).

[13] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers *(SIGCOMM '10)*.

[14] David Gale and Lloyd S Shapley. 1962. College admissions and the stability of marriage. *The American Mathematical Monthly* 69, 1 (1962).

[15] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. ProjecToR: Agile Reconfigurable Data Center Interconnect *(SIGCOMM '16)*.

[16] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network *(SIGCOMM '09)*.

[17] 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 *(SIGCOMM '14)*.

[18] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible Network Experiments Using Container-based Emulation *(CoNEXT '12)*.

[19] Raj Jain. 1991. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling*.

[20] 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 *(NSDI '14)*.

[21] 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 *(CoNEXT '15)*.

[22] 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 *(CoNEXT '15)*.

[23] Nick McKeown. 1999. The iSLIP Scheduling Algorithm for Input-Queued Switches. *IEEE/ACM Transactions on Networking* 7, 2 (1999).

[24] William M. Mellette and Joseph Ford. 2015. Scaling Limits of MEMS Beam-Steering Switches for Data Center Networks. *Journal of Lightwave Technology* 33, 15 (Aug. 2015).

[25] William M. Mellette, Glenn M. Schuster, George Porter, George Papen, and Joseph Ford. 2017. A scalable, partially configurable optical switch for data center networks. *Journal of Lightwave Technology* 35, 2 (Jan. 2017).

[26] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang-Chen Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Integrating Microsecond Circuit Switching into the Data Center *(SIGCOMM '13)*.

[27] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network *(SIGCOMM '15)*.

[28] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2017. Shoal: a lossless newtork for high-density and disaggregated racks. *Cornell Technical Report* (2017).

[29] 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 '15)*.

[30] Leslie G. Valiant. 1982. A Scheme for Fast Parallel Communication. *SIAM J. Comput.* 11, 2 (1982).

[31] John Von Neumann. 1953. A certain zero-sum two-person game equivalent to the optimal assignment problem. *Contributions to the Theory of Games* 2 (1953).

[32] 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 *(SIGCOMM '10)*.

[33] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks *(SIGCOMM '12)*.