OpenOptics: Enabling Open Research and Implementation of Optical Data Center Networks

Yiming Lei[1] Federico De Marchi[1] Jialong Li[2*] Raj Joshi[3] Shu-Ting Wang[4*] Xiaoqi Chen[5]
Balakrishnan Chandrasekaran[6] Yiting Xia[1]

[1] Max Planck Institute for Informatics [2] Shenzhen University of Advanced Technology
[3] Harvard University [4] UC San Diego [5] Purdue University [6] Vrije Universiteit Amsterdam

Abstract

Optical data center networks (DCNs) are emerging as a promising design for cloud infrastructure. However, existing optical DCN architectures operate as closed ecosystems, tying software solutions to specific optical hardware. We introduce OpenOptics, an open research and development framework that decouples software from hardware, allowing them to evolve independently. OpenOptics features: (1) a time-flow table abstraction as a common interface between optical hardware and software, (2) a unified workflow and user-friendly API for implementing various optical DCNs with simple Python scripts, and (3) a backend system that re-architects queue management to support the time-flow tables and provides rich infrastructure services for diverse applications. Built on programmable switches, OpenOptics achieves a record-breaking minimum optical circuit duration of 2 µs using commodity devices. We validate OpenOptics' generality by implementing six optical architectures and seven routing schemes on an optical testbed and conducting benchmarks on a 108-ToR setup, showcasing its efficiency. Additionally, case studies highlight novel research opportunities enabled by OpenOptics.

1 Introduction

In the post-Moore's law era for merchant silicon, the networking community has turned to optical circuit switch (OCS) technologies for their power, cost, and bandwidth advantages. Numerous optical data center network (DCN) architectures have been proposed to build high-performance DCN fabrics with different *OCS hardware* [11, 14, 16–18, 21, 24, 27, 33, 36–39, 42, 43, 48–50, 52, 53].

OCSes are, however, bufferless physical-layer devices that transmit only optical signals. They establish exclusive optical circuits between electrical communication endpoints, such as pods, Top-of-Rack switches (ToRs), and host NICs, and reconfigure the circuits to form varying network topologies (Fig. 1). Hence, to make an optical DCN functional, a *software system* must coordinate packet buffering and scheduling on these endpoints to align with the circuit availability.

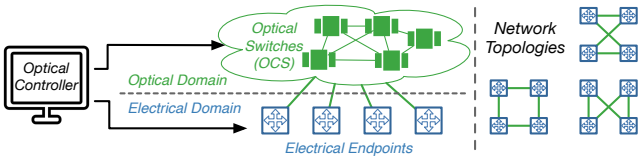

Figure 1: *An example optical DCN and its three topologies connected by different optical circuits over the OCSes.*

The *tight coupling between optical hardware and software systems* makes research and development in optical DCNs inherently siloed. Each optical DCN architecture forms a *closed ecosystem*, comprising specialized optical hardware and customized networking systems designed to support the hardware. As a result, system solutions are tied to the underlying hardware architecture, creating a high *barrier to entry* for system researchers and practitioners without optics expertise.

For this reason, research and implementation of optical DCNs are largely confined to hyperscalers (e.g., Google [43] and Microsoft [14]) and specialized startups [38], which can afford the substantial cross-disciplinary hardware-software co-development effort required to build an optical DCN from the ground up across the full network stack. Beyond the significant engineering investment, these efforts are strictly bound to a specific optical DCN architecture, making evolution to alternative designs extremely costly—often requiring rebuilding the entire system nearly from scratch.

Smaller industry players and research groups are fundamentally constrained by the lack of *experimental environments* and *prototyping tools* for optical DCNs. The few existing evaluation platforms emulate switches on hosts without the actual switch stack, achieving only modest scale (up to 8 ToRs) and capacity (10 Gbps) [41, 54]. Widely used network simulators [6, 7, 9, 23] lack native support for optical DCNs and, even if extended, are restricted to packet- or flow-level modeling, rendering them inadequate for the comprehensive ex-

*Work done while at Max Planck Institute for Informatics.

ploration required in this actively studied field. Consequently, system designs often rely on *ad-hoc* simulators and small-scale testbeds tailored to specific optical architectures, making it difficult to rigorously validate, compare, and iterate on competing proposals, thereby stalling the innovation cycle.

In this paper, we present OpenOptics, an open research and development framework that *decouples software systems from optical hardware, allowing them to evolve independently*. OpenOptics serves as a "narrow waist" for optical DCNs, providing a general system stack that enables seamless integration of diverse optical hardware and flexible exploration of network protocols. As a collaborative research sandbox and a first step toward low-effort end-to-end implementation, OpenOptics accelerates the identification and resolution of cross-layer issues, paving the way for the deployment of diverse optical DCN solutions in real-world settings.

The primary challenge towards this ambitious goal is defining a *common interface* between hardware and software across various optical architectures. Like the IP layer in traditional networks, we believe this "narrow waist" lies in routing, as the core function of optical DCNs is to direct traffic through the dynamic circuits. This interface must (1) capture the new requirement for routing packets across changing topologies; (2) resemble classical network abstractions to help system researchers adapt quickly to the new topic while hiding optical hardware details; (3) be expressive enough to encompass existing and future routing schemes in optical DCNs; and (4) maintain backward compatibility with classical abstractions to support traditional DCNs for performance comparisons.

We propose the *time-flow table* abstraction (§3) to unify diverse optical DCN architectures. This abstraction emphasizes the *notion of time*, reflecting the *circuit switching* nature of optical DCNs where circuits are discrete and persist for fixed time periods, despite variations in hardware and software designs. The time-flow table enables packets to exit a switch or NIC not immediately upon arrival, but at a scheduled time aligned with circuit availability. This abstraction can represent all routing primitives used in existing and potential future routing schemes for optical DCNs, and it reduces to a regular flow table for traditional DCNs when the time-related fields are set to wildcards.

We develop a series of technical mechanisms to support the time-flow table abstraction. The first addresses programmability, eliminating the need for manual low-level table specification and enabling high-level programming of optical DCNs. This is non-trivial because optical DCNs fall into two categories with distinct workflows: *traffic-aware* (*TA*), which dynamically reconfigure the topology based on traffic patterns [16–18,21,24,33–35,42,43,48,53], and *traffic-oblivious* (*TO*), which operate using a pre-determined circuit schedule independent of traffic conditions [11, 14, 37–39]. This separation creates a rigid boundary between *TA* and *TO* designs, limiting exploration of the broader design space.

To bridge this divide, we define a *programming model* that unifies the *TA* and *TO* workflows, breaking the boundary and opening up new (e.g., hybrid) designs. We further design a set of common *API functions* (§4.2) for *TA* and *TO* architectures that enable users to specify topologies and routing strategies using concise Python scripts. These APIs express high-level network behaviors, which are then compiled into low-level OCS configurations and time-flow table entries.

Moreover, rather than simply extending flow tables with additional fields, the time-flow table abstraction introduces a fundamental shift in packet processing. It requires time-based packet lookup, buffering, and scheduling, which go beyond traditional priority-based queue management and pose significant challenges for switches and NICs with limited buffers. Many proposals cannot be implemented end-to-end without such support [11,14,31–33,39], and the implemented ones [16,21,24,37,38,42,43,48,53] have to rely on legacy flow tables for static topologies, restricting routing to individual topology instances or a subset of continuously available circuits on the changing topologies.

We design a *backend system* that leverages P4 packet processing programmability to re-architect the *queue management system* (§5.1). With our nanosecond-precision time synchronization [30], the system performs time-based packet lookup and queue allocation based on circuit availability, pausing and resuming appropriate queues to buffer packets and release them precisely when circuits become available. The backend also provides rich *infrastructure services*, including common system functionalities shared across existing optical proposals, as well as optimization knobs to further improve the efficiency of such designs.

We built OpenOptics using Intel Tofino2 switches and the `libvma` userspace library on Mellanox NICs. The host-only version with Corundum FPGA-based NICs [22] is under development. OpenOptics provides a *full ecosystem* for users with different hardware availability. Users with real OCSes can plug them into OpenOptics for an end-to-end functional system, requiring no changes to the system stack when upgrading OCS hardware. For users without real OCSes, we provide an emulated optical network fabric using programmable switches (§5.3), emulating different types and structures of OCSes. Lastly, we offer a Mininet educational toolkit (§5.3) that runs OpenOptics in a virtual network for those without access to programmable switches.

We evaluate OpenOptics on a testbed with real and emulated OCSes running cloud applications. To demonstrate its generality, we implement six *TA* and *TO* architectures and seven routing schemes (§6). Though these prior work are either proprietary or only partially implemented, we validate OpenOptics' correctness by inferring performance trends from prior work and reproducing limited results with comparable settings. We also showcase new research use cases OpenOptics enables (§6) and conduct extensive benchmarking studies at scale (§7). In a 108-ToR setting with production DCN traces, all system components function efficiently, keep-

ing usage of switch buffer and other resources under 13.8%. OpenOptics supports a minimum circuit duration of 2 µs—the lowest ever achieved with commodity network devices, offering guidance for practical optical DCN deployment.

We have open-sourced OpenOptics[1] and provide comprehensive documentation, tutorials, and artifacts on its official website[2]. OpenOptics was well received at the SIGCOMM'24 system demo [29] and the SIGCOMM'25 tutorial [8]. By making OpenOptics publicly available and providing extensive supporting materials, we aim to foster broad adoption and community engagement, and to facilitate and democratize innovation in optical DCN research and engineering.

[This work does not raise any ethical issues.]

2 Background

We briefly introduce optical DCNs and their routing strategies to reveal key requirements for the OpenOptics design.

2.1 Optical DCNs

An optical DCN has an *optical network fabric* in the core, formed by interconnecting and configuring OCSes to realize a concerned DCN architecture (Fig. 1). OCSes transmit optical signals in the *physical layer*, acting as waveguides (similar to an optical fiber), but with the additional capability of circuit reconfiguration. An *optical controller*, e.g., a server or FPGA board, controls the OCSes to establish *optical circuits* between electrical communication endpoints, such as pods, ToRs, and host NICs, where a pair of them has exclusive access to the circuit. The controller reconfigures these circuits to create different network topologies (refer Fig. 1). As waveguides, OCSes are bufferless and cannot store or process packets, requiring the communication endpoints to be coordinated to send packets only when the optical circuits are available. This coordination is usually achieved using the optical controller, although endpoints can also self-organize (i.e., coordinate amongst themselves). Optical DCNs are broadly classified as either traffic-aware (*TA*) or traffic-oblivious (*TO*), based on how they reconfigure the circuits.

***TA* optical DCNs** establish circuits based on traffic demands, but the control loop, involving traffic collection and topology computation takes milliseconds to seconds, delaying latency-sensitive "mice" flows as they wait for circuits to be ready. One approach (*TA-1*), hence, uses a static network for mice flows, while sending elephant flows opportunistically over the optical circuits. The static network can either be a parallel electrical DCN, creating a so called electrical-optical hybrid DCN [21, 34, 48], or a default topology with some stable circuits [18, 24]. Another solution (*TA-2*) ensures every reconfigured topology is a connected graph, treating the optical DCN as a static network with occasional topology updates adapting to long-term traffic patterns [16, 17, 53].

[1] https://github.com/mpi-ncs/openoptics
[2] https://openoptics.mpi-inf.mpg.de

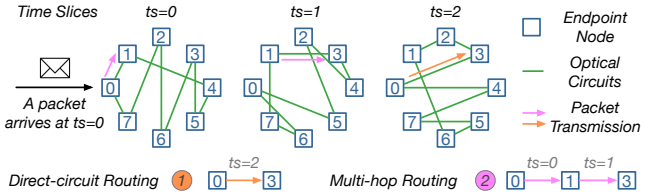

Figure 2: *A routing example in a TO optical DCN. The arrows represent the paths for a packet from endpoint node N_1 to N_3 arriving at time slice ts=0 under different routing schemes.*

Google's Jupiter and Lightwave fabrics follow this approach, with reconfiguration intervals of minutes to hours [35, 43]. This approach has demonstrated strong effectiveness and performance benefits for emerging machine learning training workloads [26, 49, 51].

***TO* optical DCNs**, in contrast, leverage advanced OCS technologies for rapid circuit reconfiguration, optimizing latency for mice flows [11, 14, 15, 37–39]. Each OCS configuration is held for a brief *time slice*, typically lasting only microseconds or even sub-microseconds, with a fixed *optical schedule* rotating through predefined topologies throughout time slices. These topologies maximize connectivity regardless of traffic patterns, often using expander graphs. The optical schedule repeats periodically, each *optical cycle* having tens to hundreds of time slices to fully diversify network connectivity over time.

2.2 Routing in Optical DCNs

Constrained to flow tables in the traditional SDN paradigm, routing in *TA* architectures treats each reconfigured topology independently, with packets routed within the topology instance as if on a static topology. In *TA-1* architectures, mice flows are routed on the static topology, while elephant flows switch between the static topology and sporadic optical circuits. The flow table on each electrical endpoint presents a default static route, and when a circuit becomes available, the optical controller updates it with a higher-priority route. *TA-2* architectures route traffic within each topology instance until it gets reconfigured, at which point the optical controller updates the flow table to shift traffic to the new paths.

Routing in *TO* architectures is more challenging due to the ultrashort time slices, where a packet in transit may cross multiple time slices, passing through different topologies. This requires routing paths to be pre-computed offline, based on the fixed optical schedule, and electrical endpoints must execute the routing plan accurately for each time slice.

In Fig. 2, for instance, a packet arriving at endpoint node N_0 at time slice $ts = 0$ and destined to N_3 can either wait until $ts = 2$ to take the *direct* circuit ① when circuit $N_0 \leftrightarrow N_3$ appears, or can go through a more readily available *multi-hop* path ② via the intermediate hop N_1. Multi-hop paths can happen in the same time slice [37] or across time slices [14, 31, 32, 39].

Node 0

Arrival Slice	Dst.	Egress Port	Departure Slice
0	3	2	2

(a) Direct path

Node 0

Arrival Slice	Dst.	Egress Port	Departure Slice
0	3	1	0

Node 1

Arrival Slice	Dst.	Egress Port	Departure Slice
0	3	2	1

(b) Multi-hop path

Node 0

Arrival Slice	Dst.	Egress Port	Departure Slice
*	3	2	*

(c) Static Topology

Node 0

Arrival Slice	Dst.	Source Routing Action Hops:<Egress Port, Departure Slice>
0	3	<1,0>,<2,1>

(d) Multi-hop Source Routing

Figure 3: *Time-flow table examples for:* ***(a)*** *direct path ① and* ***(b)*** *multi-hop path ② in Fig. 2,* ***(c)*** *a static path in an TA architecture or traditional DCN, and* ***(d)*** *source routing realization of path ②, i.e., equivalent of per-hop lookup in (b).*

In this example, path ② crosses two time slices: the packet reaches N_1 in $ts = 0$ over circuit $N_0 \leftrightarrow N_1$, then waits until $ts = 1$ for circuit $N_1 \leftrightarrow N_3$ to become available.

This time-based routing is fundamentally different from that of static networks, posing two additional requirements.

Req. 1: To determine the time slice in which a packet arrives and map it to the corresponding routing path.

Req. 2: To buffer the packet if it needs to be sent in a different time slice than the one in which it arrives.

As a general framework, OpenOptics must meet these new requirements of *TO* architectures, while being compatible with the traditional SDN paradigm used by *TA* architectures. This motivates us to propose the new time-flow table network abstraction, which, as shown in §4.2, not only supports both *TA* and *TO* architectures but also extends *TA* designs beyond the traditional SDN paradigm, enabling routing packets across different topologies.

3 Time-Flow Table

In this section, we introduce the key enabler of OpenOptics—the *time-flow table* abstraction, with the system support for it described in §5.1.

Optical devices are inherently diverse and exhibit a wide range of characteristics, such as bit error rate, insertion loss, and crosstalk. To provide an effective interface between optical hardware and software systems, we abstract the on-off of optical circuits that essentially influence network connectivity, under the premise that the underlying optical devices satisfy the necessary physical-layer requirements for correct operation.

Compared to the traditional flow table, this abstraction highlights *the notion of time* to meet the requirements of time-based routing presented above (§2.2). As shown in Fig. 3, for *Req. 1*, we define *arrival time slice* as a *match field*, allowing a switch or a host NIC to determine the time slice of a packet upon arrival and map it to the appropriate path; for *Req. 2*, we define *departure time slice* as an *action field* to enable routing protocols that may dispatch packets in future time slices [31,32,39]. The time-flow table is backward-compatible with regular flow tables, if we set arrival and departure time slices to be wildcards.

Next, we use examples from Fig. 3 to demonstrate the expressiveness of the time-flow table abstraction in presenting diverse routing schemes of *TA* and *TO* optical DCNs, specifically with routing paths ① and ② from Fig. 2.

Direct-circuit routing. Fig. 3 (a) shows the time-flow table of N_0 for the direct path ① from N_0 to N_3. A packet arriving at time slice $ts = 0$ must wait until $ts = 2$ for the direct circuit $N_0 \leftrightarrow N_3$, so the arrival time slice is set to 0, and the departure time slice is set to 2.

Multi-hop routing. For the multi-hop path ② via N_1, the packet is forwarded immediately from N_0 to N_1 at $ts = 0$, so both the arrival and departure time slices in N_0's time-flow table in Fig. 3 (b) are set to 0. At N_1, the packet arrives at $ts = 0$ (through circuit $N_0 \leftrightarrow N_1$) and waits until $ts = 1$ to be sent to N_3, so N_1's time-flow table has an arrival time slice of 0 and a departure time slice of 1.

Routing in *TA* architectures and static DCNs. In Fig. 3 (c), the time-flow table reduces to a standard flow table when the arrival and departure time slices are set to wildcards, allowing packets to be forwarded immediately upon arrival. Routing in *TA* architectures occurs in separate topology instances with static paths, similar to routing in a static irregular (non-Clos) topology, which can be presented with flow tables (refer §2.1). As the time-flow table reduces to a flow table, it supports *TA* architectures as well as static DCNs.

Source routing. We have shown examples of per-hop routing lookup so far. Some routing schemes, however, cannot break down the paths into per-hop lookups and have to be implemented with source routing [31, 37]. Our time-flow table supports source routing by including the entire path in the action field at the source endpoint node, as a sequence of <egress port, departure time slice> tuples for each hop. For example, Fig. 3 (d) is the source routing equivalent of the per-hop tables in Fig. 3 (b), where the hops <1,0> and <2,1> for N_0 followed by N_1 are to be written to the packet.

Multi-path routing. The time-flow table supports both per-packet and per-flow multi-path routing, as required by certain solutions [11, 14, 31, 39], through an optional path hashing field. For example, per-packet hashing can be based on timestamping or on the on-chip random number generator, while per-flow hashing using the five-tuple.

The above examples cover existing routing primitives in *TA* and *TO* optical DCNs and traditional static DCNs. Future routing solutions will likely combine these primitives and can therefore also be supported by the time-flow table.

While expressiveness is essential, a practical abstraction must also scale efficiently. Compared to a traditional flow table, the additional factor affecting scalability is the extra match key, the time slice. As discussed in §2.1, under *TA*

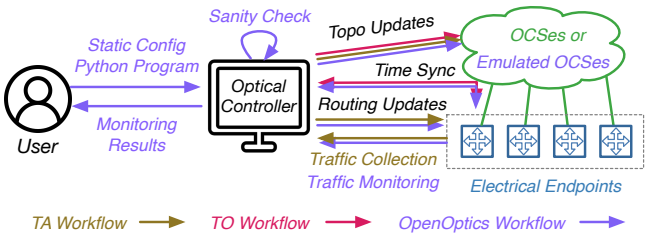

Figure 4: *OpenOptics system workflow.*

architectures the time-flow table falls back to a conventional flow table, and thus its size scales identically to flow tables. For *TO* architectures, the optical schedules are periodic (§2.1), with each cycle containing tens to hundreds of time slices [11, 14, 39], well within the capacity of modern switches (§7). In the data plane, electrical endpoints compute the current time slice by taking the modulo of the local timestamp on the time slice duration. Consequently, neither finer-grained timestamps nor longer time slices increase the number of table entries. Therefore, time-flow table preserves scalability comparable to that of conventional flow tables.

4 Programming Model

We introduced the time-flow table as a unified network abstraction that supports both *TA* and *TO* optical DCNs with diverse routing schemes (§3). A user-friendly system should, however, relieve users of the tedious and error-prone task of managing low-level time-flow table entries and optical circuits. We propose, hence, a unified programming model for OpenOptics to ease implementation of *TA* and *TO* architectures, and potential new ones that blur the *TA*-*TO* boundary.

4.1 System Workflow

A unified programming model necessitates a unified system workflow that consolidates the traditionally separate control paths of *TA* and *TO* architectures. In OpenOptics, users write a concise Python script to specify the desired optical DCN design with the user APIs (§4.2). The script describes the basic network setup (e.g., device count, port count, optical switch type, switching delay), the circuit schedule, routing strategies, and optionally, monitoring metrics. The framework translates these high-level specifications into low-level network configurations. In particular, circuit schedules are compiled into port-level configurations for optical switches, while routing strategies are instantiated as forwarding paths based on the given network setup and decomposed into time-flow table entries at each endpoint. Under this unified workflow, users implement either *TO* or *TA* architectures by specifying different network topology and routing policies.

***TA* workflow.** *TA* architectures (§2.1) start with a default topology. The electrical endpoints periodically report traffic statistics to the optical controller, using which the latter builds a global traffic matrix (TM). The controller then optimizes the topology for that TM and computes routing paths for the optimized topology. Next, it updates the flow tables (constrained by the legacy SDN paradigm, as established in §2.2) on the electrical endpoints. Once these routing changes are applied successfully, the controller reconfigures the topology by resetting the optical circuits.

***TO* workflow.** With a pre-determined optical schedule, *TO* architectures (§2.1) require minimal interaction between electrical endpoints and the optical controller. The controller reconfigures the OCSes per time slice to realize different topologies in the optical schedule. Routing paths for each topology are pre-computed offline and, according to the *TO* proposals, pre-loaded into the electrical endpoints, which synchronize with the optical controller to self-regulate the time slices and enforce the appropriate routing paths. Without time-flow table abstraction and the supporting queue management system, however, how to enforce routing changes per time slice remains largely unknown; it is one of the reasons why most *TO* proposals are not yet implemented.

OpenOptics supports *TO* architectures with the time-flow table (§3) and the corresponding queue management system (§5.1) for time-based packet scheduling. As detailed in §5.1, time-flow tables reside on electrical endpoints, such as switches and NICs with packet processing programmability, and the queue management system handles packet lookup, scheduling, buffering, and dispatching on a per-time-slice basis. Time synchronization solutions of existing *TO* proposals either rely on specific optical hardware [14] or achieve accuracy only sufficient for particular architectures [38]. We built a general hardware-independent synchronization protocol with nanosecond precision to support diverse architectures, which is described in a separate paper [30].

OpenOptics supports both traffic-based real-time topology updates (as in *TA*) and batch topology configurations (as in *TO*). Breaking the *TA*-*TO* boundary enables novel hybrid designs (§4.3), such as a periodically updated optical schedule that reflects traffic evolution, or *TA* and *TO* subnetworks in different hierarchies working in tandem.

OpenOptics enhances user interaction through API functions (§4.2). Users specify high-level network behavior via a static configuration (`json` file) for hardware setups (e.g., OCSes count and structure, optical uplinks per endpoint, and time slice duration), along with a Python program that invokes the API functions. The optical controller sanity checks the inputs, e.g., test the feasibility of circuits and routing paths, and compiles them into circuit connections and time-flow table entries. We extend the basic traffic collection functionality in the *TA* workflow to rich traffic monitoring APIs, providing network telemetry beyond traffic volume, e.g., buffer and link usage, to monitor network health.

Table 1: *OpenOptics user API functions, optional arguments in gray.*

Category	APIs	Description
Topology	`connect(Circuit<N1,port1,N2,port2,ts>)→bool`	Primitive function connecting `port1` `port2` of nodes `N1` `N2` in times slice `ts`.
	`topo(TM)→[Circuit]`	Abstract function to generate `Circuit`s, with building block `connect()`.
	`↳edmonds(TM),BvN(TM),jupiter(TM)`	Circuit scheduling algorithms materializing `topo()` for *TA* architectures.
	`↳round_robin(dimension,uplink)`	Optical schedule generation materializing `topo()` for *TO* architectures.
	`deploy_topo([Circuit])→bool`	Check feasibility and deploy the topology.
Routing	`add(Entry<arr_ts,src,dst,dep_ts>,node)→bool`	Add time-flow table `Entry` on `node`; `arr_ts`, `dep_ts` to `null` turns a flow table.
	`routing([Circuit])→[Path<src,dst,ts>]`	Abstract function to generate paths from `src` node to `dst` node at `ts`.
	`↳direct(), ecmp(), wcmp(), ksp()`	Routing algorithms materializing `routing()` for *TA* architectures.
	`↳vlb(), opera(), ucmp(), hoho()`	Routing algorithms materializing `routing()` for *TO* architectures.
	`neighbors(([Circuit],node,ts)→[node]`	Helper function returning all nodes having direct circuits to `node` in `ts`.
	`earliest_path([Circuit],src,dst,ts,max_hop)→[Path]`	Helper function returning paths from `src` to `dst` most recent to `ts` in `max_hop`.
	`deploy_routing([Path],LOOKUP,MULTIPATH)→bool`	Compile `Path`s into `Entry`s and deploy, with `LOOKUP` and `MULTIPATH` options.
Monitoring	`collect(interval)→TM`	Collect global traffic metric `TM` every `interval`.
	`buffer_usage(node,port,interval)→unsigned`	Query the buffer usage of `port` at `node` every `interval`.
	`bw_usage(node,port,interval)→unsigned`	Query the bandwidth usage of `port` at `node` every `interval`.

4.2 User API

A user first creates an OpenOptics network object and then calls the topology, routing, and monitoring APIs listed in Tab. 1. The API calls accept optional arguments for each architecture type to offer a unified workflow. For instance, *TA* architectures set the time slice `ts=null` to operate within each topology configuration, and *TO* architectures set the traffic matrix `TM=null` to disregard traffic.

Topology APIs. We define the primitive call `connect()` to establish a circuit between endpoint nodes.

The abstract function `topo()` builds on `connect()` to realize diverse topologies. We materialize it to implement *TA* circuit scheduling algorithms, such as the classic Edmonds' matching and Birkhoff-von-Neumann algorithms used in c-Through [48] and Mordia [42], and the gradual evolving approach used in Jupiter [43]. We also implement round-robin variants for *TO* optical schedules. For instance, Shale [11] uses a three-dimensional round-robin with a single optical uplink per node, and Opera [37] uses a single-dimensional round-robin with *N* uplinks per node. Users can define custom topologies by overriding `topo()`.

The action function `deploy_topo()` compiles the node-level circuits into OCS internal connections based on the OCS structure specified in the static configuration file. The optical controller verifies the feasibility of the physical circuits and deploys them onto the OCSes.

Routing APIs. The basic function `add()` allows users to add time-flow table entries directly, e.g., for debugging purposes. It supports flow table entries where the arrival and departure time slices are both set to `null` as wildcards (§3).

We define an abstract function `routing()` to implement diverse routing algorithms, which return the set of paths per source-destination node pair, and additionally per time slice for *TO* architectures. We materialize it for *TA* routing algorithms running within each topology configuration including direct-circuit routing [39], ECMP [10], WCMP [43], k-shortest path [53], and *TO* algorithms across topologies, including VLB [14, 39], Opera [37], UCMP [31], and HOHO [32].

Users can implement customized routing algorithms via overriding functions, and we provide helpers to simplify them. `neighbors()` retrieves, for instance, connected neighbors for a node in a time slice, useful for VLB, and *TA* routing algorithms on a topology instance when `ts=null`; similarly, `earliest_path()` finds the first path between a source and destination node since a given time slice, facilitating direct-circuit routing, UCMP, and HOHO, and shortest path routing is a special case of it with `ts=null` in one topology.

Finally, `deploy_routing()` compiles the paths into time-flow table entries and loads them onto each node. It supports `LOOKUP` types of per-hop and source routing by decomposing the path into next-hop nodes or retaining the entire path in the action field. It also supports packet- and flow-level `MULTIPATH` by hashing the ingress timestamp or five tuples.

Monitoring APIs. We offer telemetry functions, such as `buffer_usage()` and `bw_usage()` for querying the buffer and bandwidth usage of a port on a node. The `collect()` function for traffic collection in *TA* architectures is considered a special case of monitoring. New monitoring functions can be easily added as needed.

4.3 Example Programs

Fig. 5 shows how the unified workflow (§4.1) and APIs (§4.2) simplify implementation of *TA* and *TO* architectures, and novel hybrid designs with a few lines of Python code.

RotorNet (*TO*). An OpenOptics network object is first created with a static configuration, using which the optical controller connects to the nodes, in this case hosts. RotorNet [39], being a *TO* architecture, uses a single-dimensional round-robin optical schedule with the number of optical uplinks per node specified in the configuration file, and it runs VLB routing on the optical schedule. The generated topologies and paths are deployed onto the OCSes and host NICs, where time-flow table entries are created for per-hop lookups, as opposed to source routing, enabling packet-level multipath

```
#config={"node":"host",
# "node_num":128,
# "uplink":2,
# "IPs":["10.0.0.1",..]}
net=openoptics.net(config)
circuits=round_robin(
  dimension=1,
  uplink=config.uplink)
paths=vlb(circuits)
net.deploy_topo(circuits)
net.deploy_routing(paths,
  LOOKUP="hop",
  MULTIPATH="packet")
```

(a) *RotorNet (TO).*

```
net=openoptics.net(config)
circuits=jupiter(
  TM=null)
paths=wcmp(circuits)
net.deploy_topo(circuits)
net.deploy_routing(paths)
while(
  TM=net.collect("24h")):
  circuits=jupiter(
    TM, circuits)
  paths=wcmp(circuits)
  net.deploy_routing(paths)
  net.deploy_topo(circuits)
```

(b) *Jupiter (TA).*

```
net=openoptics.net(config)
circuits=round_robin(
  1,config.uplink)
paths = vlb(circuits)
net.deploy_topo(circuits)
net.deploy_routing(paths)
while(
  TM=net.collect("10min")):
  #Custom optical schedule
  circuits=sorn(TM)
  paths=vlb(circuits)
  net.deploy_routing(paths)
  net.deploy_topo(circuits)
```

(c) *Semi-oblivious (TA+TO).*

```
#config={"node":"rack",
# "node_num":128,...};rack_conf={
# "node":"host","node_num":64,...}
net=openoptics.net(config)
for rack in net.nodes:
  r_cts=round_robin(
    1,rack_conf.uplink)
  rack.deploy_topo(r_cts)
  rack.deploy_routing(vlb(r_cts))
while(TM=net.collect("1h")):
  cts=BvN(TM)
  net.deploy_topo(cts)
  net.deploy_routing(wcmp(cts))
```

(d) *Hierarchical (TA+TO).*

Figure 5: *Code snippets of implementing various optical architectures using the OpenOptics APIs (§4.2).*

routing for packet spraying in VLB.

Jupiter (*TA*). The *TA* architecture Jupiter [43] starts with a uniform mesh topology, an empty TM, and WCMP routing. Once every 24 hours, a new TM is collected and the topology optimized. The computed routing changes are deployed as higher-priority routes atop existing ones before reconfiguring the topology, ensuring seamless traffic switches.

Semi-oblivious (*TA+TO*). OpenOptics enables new architectural designs beyond the traditional *TA-TO* boundary, thus naturally supports the recent semi-oblivious proposal that builds skewed round-robin optical schedules to reflect traffic patterns [46]. It starts with a single-dimensional round-robin optical schedule (Fig. 5c) and VLB routing as a regular *TO* network, and the topology and routing are updated every 10 minutes based on the new TM. Our topology API `topo()` can be easily extended to support the custom topology-building algorithm `sorn()`, which modifies `round_robin()` by creating dense connections between hotspot nodes and sparse ones elsewhere. This design is traffic-driven like *TA* architectures, but each reconfiguration loads an optical schedule with a batch of topologies like *TO* architectures.

Hierarchical (*TA+TO*). We can envision other types of *TA-TO* hybrid designs, such as the hierarchical network in Fig. 5d. For emerging ML workloads, GPU machines within a rack can be interconnected through a *TO* scale-up network, leveraging its reach connectivity, while ToRs can be further interconnected through a *TA* scale-out network to manage traffic locality across racks. We define separate static configuration files for the two network levels (Fig. 5d), and a network object is first created for the core inter-rack network. Then, for each node in the core network, an intra-rack *TO* network is created, similar to RotorNet (Fig. 5a). The inter-rack network then collects traffic and updates topology and routing like Jupiter (Fig. 5b).

5 OpenOptics System

In this section, we introduce the backend system for the time-flow table (§3) and programming model (§4), which are general to both switch-centric and host-centric optical DCNs with switches and hosts connected to the OCSes, respectively. Our

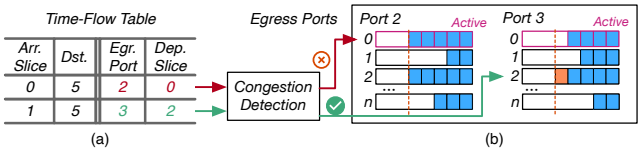

Figure 6: *Example of packet processing on a switch, assuming the active calendar queue is `q=0` for time slice `ts=0`.*

description is based on our current implementation for the mainstream switch-centric design for scalability, using Intel Tofino2 programmable switches and `libvma` userspace library [3] on Mellanox NICs. Our implementation applies to both ToR-based and pod-based switch-centric designs, where our switch system is deployed on ToRs or pod switches, and host system on end hosts. Other switches in a pod need no change. The host-centric version of OpenOptics with Corundum FPGA-based NICs [22] is under development.

The main system components include the customized queue management system to support the time-flow table (§5.1) and infra services that provide common features and optimization knobs for various optical architectures (§5.2). We also outline system peripherals, including the emulated optical fabric and Mininet education toolkit (§5.3).

5.1 Time-Based Queue Management

The time-flow table requires time-based packet scheduling to dispatch packets at the planned departure time slice, and buffering is needed if the departure time slice is later than the arrival time slice, both go beyond priority-based packet scheduling in traditional queue management systems. We leverage the *queue pausing* feature of programmable switches [28] and FPGA-based NICs [22] to rearchitect the queue management system, enabling controlled pausing and resuming of queues. Our design, however, is not tied to native queue-pausing capabilities; similar functionality can be approximated using priority queues [47]. Based on the target topology and routing policy, the framework assigns departure time slices to egress queues and automatically manages their pausing and resuming. Packets are sent to queues accord-

ing to their assigned departure time slices and buffered until their scheduled release time, during which the corresponding queues remain paused.

We implement this design inspired by the calendar queues framework [47]. A calendar queue is associated with a "calendar day", and packets can be enqueued at a "rank" for a future day. A *calendar day* is *a time slice* in our case. We create a set of calendar queues per egress port, assigning each time slice a queue sequentially until queue exhaustion to wrap around. The *rank* of an ingress packet is the difference between its *departure time slice* and *arrival time slice*.

Queue pausing and resuming can be triggered by any ingress packet in the data plane. We control this process using the on-chip packet generator available in programmable switches [25], which reliably sends a packet into the ingress pipeline at a specified time interval. We configure this interval to match the *time slice duration*. Recall OpenOptics synchronizes electrical endpoints, switches and host NICs included, with the optical controller at nanosecond precision [30], so the on-switch packet generator can send a packet at the start of each time slice to initiate *queue rotation* across all egress ports. This rotation pauses the currently active queue and resumes the queue for the upcoming time slice. Each switch keeps track of the *active queue* for the current time slice, which is consistent across all egress ports.

Fig. 6 illustrates two sets of calendar queues at egress ports `p=2` and `p=3`. At time slice `ts=0`, the active queue is `q=0` for all ports. An incoming packet matching the first entry in Fig. 6 (a) is mapped to `q=0` of `p=2` in Fig. 6 (b), because its departure time slice equals the arrival time slice, meaning it should be enqueued in the active queue for immediate transmission. When the time slice advances to `ts=1`, queue rotation occurs, making `q=1` the active queue. Another packet arriving at this time matches the second entry and should be enqueued in `q=2` of `p=3`, to be sent out one time slice later.

Queue pausing is supported on FPGA-based NICs [22], making our queue management system easily portable to host-centric optical DCNs. For *TA* architectures and static DCNs that rely on traditional flow tables, we can simply disable calendar queues and revert to the default queue settings.

5.2 Infrastructure Services

We abstract common functionalities and optimization options from existing optical architectures as infra services, allowing users to focus on exploring upper-layer network protocols instead of reinventing the wheel. Most of these features are proposed but not implemented, with a few implemented in ways specific to particular architectures. Our infra services enable complete and straightforward implementation of these proposals, and future ones.

Congestion detection. Many optical architectures adopt congestion control (CC) mechanisms [19, 31, 32, 37, 41], yet congestion detection in calendar queues (§5.1) is non-trivial. An optical circuit transmits a fixed amount of data per time slice. During congestion, packets may miss their scheduled time slice and remain paused in the calendar queue for a full cycle, leading to excessive delays. This occurs when a calendar queue accumulates more data than it can transmit within a time slice. Under short time slices, this threshold of accumulated data—determined by the remaining time in the time slice—may fall below the congestion threshold typically used in CC protocols for regular queues, introducing an additional condition for congestion detection.

It is challenging to determine if an incoming packet can fit into the intended calendar queue, as commercial switches cannot access the occupancy of calendar queues (egress queues) in the ingress pipeline before enqueuing.[3] We thus implement a *queue occupancy estimation* method using a register array in the ingress pipeline to track each calendar queue's occupancy. Since registers can only be updated by ingress packets, we increase the occupancy when packets are enqueued and use a packet generator to create an ingress packet every *update interval* to reduce the occupancy, assuming line-rate packet dequeuing. Evaluation in Fig. 12 shows 50 ns update interval results in less than one packet estimation error with minimal switch pipeline processing overhead. More details of this design are explained in Appx. A. It is worth noting that queue occupancy updates are executed entirely in the switch data plane and does not involve per-update interaction with the control plane or the user scripts (§4.1).

A calendar queue is full if its occupancy exceeds the admissible data amount for the elapsed time of the time slice (bandwidth $\times$ time). Congestion occurs if (1) the calendar queue is full or (2) the congestion threshold is reached, whichever happens first. This service detects congestion while giving users flexibility in how to respond. For example, in Fig.6, an incoming packet in `ts=0` that matches the first table entry finds the intended calendar queue `q=0` on port `p=2` full. The architecture can then trigger its own CC mechanism, such as packet dropping, packet trimming [37], or deferring the packet to a later time slice [31, 32]. In contrast, a packet arriving at `ts=1` that matches the second entry can be enqueued safely in `q=2` of `p=3`, which is not full yet.

Traffic push-back. While we give users flexibility in choosing CC solutions, we offer optional traffic push-back as a last-resort protection if CC fails. When a packet detects its designated calendar queue is full, it and all subsequent packets to that queue should be rejected. If the push-back service is enabled, a rejected packet triggers a *push-back* message containing the time slice of the queue. This message is broadcast to the sender ToR or pod switch, which forwards it to all connected hosts, preventing them from sending traffic to the destination switch during that time slice. This broadcast increases push-back coverage to hosts that might attempt to send to the same queue. Traffic is blocked on the hosts using

[3] Tofino2's "ghost thread" feature claims to support this functionality [28], but our tests show queue occupancy readings are outdated by milliseconds, insufficient for microsecond-scale time slices in *TO* architectures.

the flow pausing service, which will be introduced next.

Flow pausing. Many *TA* architectures pause elephant flows at the source until they can be sent over direct circuits [16,17,21,24,42,48]. In *TO* architectures, while calendar queues (§5.1) provide buffering, elephant flows still place a significant burden on switch buffers. It is desirable to also route elephant flows over direct circuits as an optimization. We implement flow pausing on hosts to support this.

We utilize *flow aging* [13] to identify elephant flows without explicit flow size information. ToRs or pod switches broadcast *signal messages* to connected hosts, notifying them of upcoming circuit connections. We implement flow pausing with the user-space `libvma` library to achieve high performance and transparency to TCP/UDP applications [3]. `libvma` links sockets to the userspace `lwIP` stack, where we intercept socket send calls to suspend data transmission from the segment queue until the circuit is ready. Suspending and resuming applications require no additional memory beyond the segment queue, as applications are naturally backpressured by the socket interface when the segment queue reaches its capacity. Fig. 9 demonstrates high throughput of our implementation.

Traffic collection. A key feature of *TA* architectures is traffic collection. Architectures that reconfigure topology at seconds to minutes frequency establish circuits to serve elephant flows [16, 17, 21, 24, 48], so we leverage flow pausing on hosts, with packets buffered in separate queues inside `vma` based on the destination switch. Hosts periodically report traffic volume per destination switch to their connected switches, which then aggregate and relay the data to the optical controller. For infrequent topology reconfigurations on an hourly scale, as in Jupiter [43], the network operates on a static topology most of the time. Switches simply track the total traffic sent to each destination switch and report the statistics to the optical controller.

Buffer offloading. Some multi-hop routing schemes consume excessive buffer space at intermediate switches, such as VLB where packets may wait an entire optical cycle before being forwarded. Some architectures suggest offloading packet buffering from switches to the connected hosts [37,39]. Although this proposal has not progressed beyond simulation, OpenOptics, as a general framework, implements it to enable full realization of these architectures and support more demanding buffering needs for future designs.

Each switch only keeps N calendar queues per egress port for the immediate future and stores the rest for later time slices onto hosts under it. As the time slices corresponding to the host-resident calendar queues approach, the packets are sent back to the switch in advance, guided by circuit notification messages. To keep the logic on switches lightweight, they randomly select hosts to balance the load and delegate bookkeeping to hosts, which initiate returning of offloaded packets. We implement buffer offloading also with `libvma`, dedicating an application per host to isolate it from the main data path and ensuring low latency. The offloaded packets are stored similar to paused flows as described above.

(a) *Testbed photo.* (b) *Testbed diagram.*

Figure 7: *Testbed setup (servers in 7a are omitted from 7b).*

5.3 Emulated Optical Fabric and Mininet

Emulated optical fabric. To extend OpenOptics to users without real OCSes, we provide an emulated optical fabric using P4 programmable devices. It abstracts diverse structures of OCSes (Fig. 1) as a single logical OCS, maximizing connectivity with limited device ports. To enhance realism, we enable the "cut-through" mode on commercial devices to minimize processing delays, closely approximating physical OCS performance.

The emulated optical fabric is synchronized with the optical controller, and circuit on-offs are emulated using a lookup table, indexed by the time slice. Available circuits for each time slice are realized by direct-through entries between the connected ports, while packets over disconnected circuits fail to match any entries and are dropped. The circuit reconfiguration period is modeled as a special time slice whose duration equals the reconfiguration delay. During this interval, all packets traversing the affected circuits are dropped.

The lookup table is configured through the topology APIs shown in Table 1. For *TO* architectures, the table is static during execution and does not require runtime updates. For *TA* architectures, the circuits update typically occur on the order of milliseconds, seconds, or even longer time scales [16, 26, 49], which are well supported by modern network devices.

Mininet toolkit. We offer a fully emulated version of OpenOptics on Mininet for users without physical network equipment. While OpenOptics' time-flow table and system implementation are portable to various P4-capable environments, its novel time-based packet scheduling (§5.1) is not natively supported by the Mininet switch stack. Thus, we implement queue pausing on the `BMv2` open-source P4 software switch [1], by enabling processes to pull packets from egress queues only during specified time periods, and migrate the OpenOptics queue management system. This emulated version includes all OpenOptics components in Fig. 4, with the optical controller as a Mininet application, and the electrical endpoints and emulated optical fabric running on `BMv2`. Users interact with the optical controller using the same static configurations and APIs as in the physical system.

6 Research Use Cases

Below, we discuss OpenOptics' ability to enable novel, previously unachievable, research use cases.

Testbed. To demonstrate the capabilities of OpenOptics, we build a testbed with both real and emulated optical fabrics, as well as a traditional electrical fabric. As shown in Fig. 7, the setup includes a Polatis Series 6000 MEMS OCS, four EdgeCore DCS810 Intel Tofino2 switches, and four servers equipped with Mellanox ConnectX-5 100 Gbps dual-port NICs. The MEMS OCS functions as an optical network fabric. Two Tofino2 switches are each divided into four logical ToRs (Fig. 7b), while the third Tofino2 switch emulates another optical fabric with flexible structures of emulated OCSes, and the fourth serves as an electrical fabric. Each of the eight ToRs connects to the OCS and the electrical fabric with a 400 Gbps link and to the emulated optical fabric with four 100 Gbps links to support flexible emulation. The servers' dual-port NICs provide eight 100 Gbps links, each connected to a ToR to work as eight individual hosts.

Traffic. We run both latency-sensitive and throughput-intensive applications on the testbed to examine mice and elephant flows. We use the *Memcached* [4] key-value storage for the latency-sensitive application, running one Memcached server and seven Memslap [5] benchmarking clients each on a host. The clients perform SET operations at milliseconds intervals, writing 4.2 KB of data to the server. As for the throughput-intensive application, we run *ring allreduce* on the hosts using the *Gloo* collective communication library [2], with varying data sizes from 800 KB to 20 MB.

Case I: realistic comparison of architectures. We implement representative optical architectures on OpenOptics to validate system correctness and demonstrate side-by-side performance comparisons enabled by OpenOptics. Particularly, we implemented *c-Through* [48], *Jupiter* [43], and *Mordia* [42] from the *TA* class, and *RotorNet* [39] and *Opera* [37] from the *TO* class. We also realized the traditional *Clos* [10] network for baseline comparison.

Clos relies on the electrical network fabric, while the MEMS OCS, with tens of milliseconds long reconfiguration delays, supports Jupiter and c-Through. As a hybrid architecture, c-Through also connects to the electrical fabric, rate-limited to 10 Gbps for consistency with the original design. Mordia, RotorNet, and Opera, requiring finer-grained circuit reconfiguration, use the emulated optical fabric. We apply the native routing schemes and optical fabric settings for these architectures and also run *UCMP* [31] routing on top of RotorNet to show the latest performance of *TO* architectures.

The distribution of flow completion times (FCTs) we obtain using OpenOptics for mice and elephant flows (Fig. 8) match the FCT trends reported in the concerned prior work (that are either proprietary or only partially realized), which confirms the correctness of our implementation. We augment this validation later (§7) by reproducing published results.

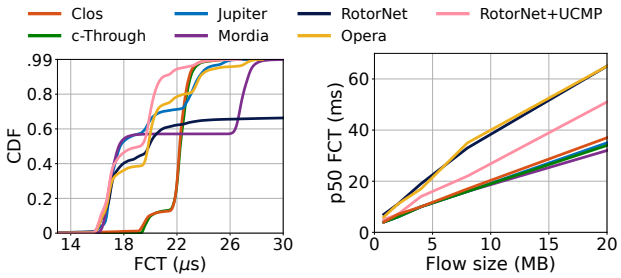

Figure 8: *FCTs of (a) Memcached and (b) Gloo allreduce.*

In Fig. 8a, c-Through shows similar mice flow FCTs to Clos. As a hybrid architecture, it uses the optical fabric only for elephant flows, while sending mice flows over electrical network, which has minimal impact on their FCTs. Jupiter improves FCTs by offering an optimized topology with fewer hops tailored to the traffic. Mordia, also a *TA* design, establishes circuits on demand. This results in low FCTs for flows with immediately available circuits, but a long tail otherwise.

TO architectures such as RotorNet and Opera are more sensitive to routing. RotorNet employs VLB, which introduces significant circuit-waiting delays by waiting at intermediate hops, resulting in long tail FCT. Opera, in contrast, has low FCTs utilizing longer but always-available paths. UCMP lowers FCTs further by reducing path length.

For elephant flows in Fig. 8b, *TA* architectures c-Through, Jupiter, and Mordia exhibit similar FCTs as Clos. They establish a ring topology using optical circuits that matches the traffic perfectly. *TO* architectures RotorNet and Opera, however, double the FCTs as the circuits are available only half the time. UCMP improves throughput with more efficient routes, leading to reduced FCTs.

Case II: investigation of the transport layer. Transport performance in optical DCNs is a new research focus due to challenges with dynamic paths, such as reTCP [41] and TDTCP [19] that address bandwidth disparity in hybrid electrical-optical networks. Both were evaluated on the Etalon emulator [41] for hybrid architectures only, limiting research in other architectural settings. This case study demonstrates transport performance analysis on *TO* architectures using OpenOptics' multi-architecture support.

We measure the throughput of long-lasting `iperf3` flows between hosts on Clos, RotorNet with VLB and direct-circuit routing, and finally a hybrid version of RotorNet with 100 Gbps bandwidth through the optical fabric and 10 Gbps bandwidth through the electrical fabric, as in TDTCP.

In Fig. 9(a), the 40 Gbps throughput in Clos is the upper bound for `iperf3` on our testbed because it is CPU-bound. As expected, for direct-circuit routing, our host system with flow pausing (§5.2) achieves roughly half that throughput due to the circuit being available 50% of the times. In contrast, VLB exhibits significantly lower throughput compared to the baseline. Surprisingly, hybrid RotorNet also lags behind

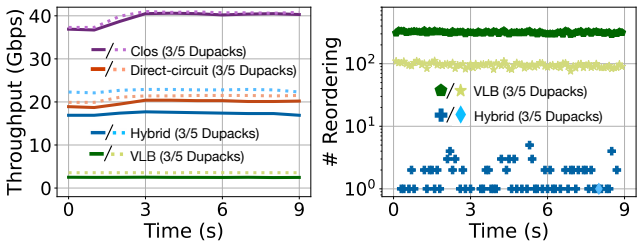

Figure 9: *(a) TCP throughput and (b) number of packet reordering events with iperf traffic.*

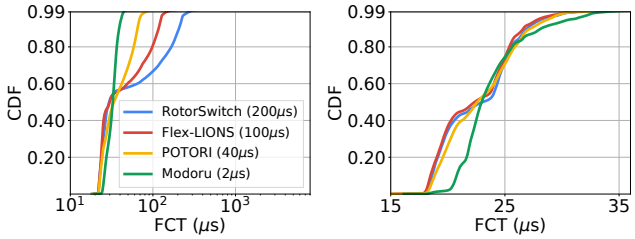

Figure 10: *Mice flow FCTs on RotorNet with OCSes of different time slice durations, under (a)VLB and (b)UCMP routing.*

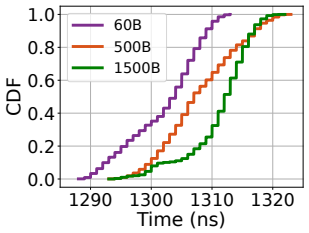

Figure 11: *switch-to-switch delay with diff. packet sizes.*

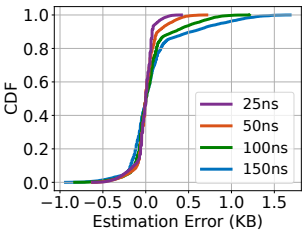

Figure 12: *EQO error under diff. update intervals.*

direct-circuit routing. We attribute this gap to the TCP performance under packet reordering, as measured in Fig. 9(b). To verify our observation, we increase the TCP dupack threshold from the default 3 to 5, which almost eliminates reordering events in hybrid RotorNet and pushes the throughput back to the expected value, i.e., near 25 Gbps with 50% of data through the electrical fabric and 50% of data through the optical fabric. While slightly improved, VLB throughput is still low due to excessive reordering.

This case study demonstrated the process of troubleshooting transport performance, and it shows how researchers can tune transport protocol parameters and evaluate newly designed protocols for optical DCNs with OpenOptics.

Case III: choice of optical hardware. The optics community has developed numerous OCSes, focusing on device-level characteristics like port count, reconfiguration delay, loss, and cost. Network architects traditionally select devices based on a general understanding of trade-offs between these factors, without insight into how these attributes impact network performance. In this case study, we demonstrate how OpenOptics facilitates more informed decisions on optical hardware selection through its emulation capabilities.

We sample four recently proposed OCS technologies and emulate the RotorNet architecture with them by inputting their physical characteristics and OCS structures into the static configuration file. Fig.10 shows the FCTs for the Memcached application (as in Fig. 8a) relative to the supported time slice duration of each OCS device.

In Fig. 10a, RotorNet's native VLB routing exhibits long tail FCTs proportional to the time slice duration, as packets are randomly routed through intermediate ToRs. In the worst case, a packet waits an entire optical cycle at the intermediate ToR for a direct circuit to the destination. This result suggests the shorter the time slice, the merrier, but OCS costs rise substantially with shorter time slices. Fortunately, Fig. 10b presents a different trade-off with UCMP routing, which strategically selects intermediate ToRs, reducing FCTs and making performance less sensitive to time slice duration. UCMP is less effective under very short time slices, such as 2 µs, due to higher risks of missing the planned time slice and being deferred to later slices. The best performance occurs at 100 µs time slices, with little difference at 200 µs, allowing good performance with more cost-effective OCS devices.

We have demonstrated finding the performance-cost sweet spot for OCS devices with a specific workload. OpenOptics enables deeper emulations for researchers and network architects to explore the interplay between optical hardware, network architectures, routing, and higher-layer protocols.

7 Performance Benchmarks

We now evaluate the system components at scale and reproduce results from existing architecture implementations. We create a large-scale setting with the testbed equipment in Fig. 7 and run DCN traffic traces. OpenOptics users can adopt this approach for scalable analysis with limited hardware.

Experimental setup. We emulate the 108-ToR topology from the Opera paper [37] because its rigid ToR and OCS structures make it compatible with other optical architectures. In Opera, each ToR has six 100 Gbps uplinks to the optical network fabric and six 100 Gbps downlinks to hosts. We implement a single representative ToR on a Tofino2 switch, referred to as the *observed ToR*, and connect it to another Tofino2 switch through six 100 Gbps links, which emulates the optical network fabric. We populate the full time-flow table on the observed ToR with entries for the 108-ToR network, and the emulated optical fabric operates at full network scale. Three servers with dual-port NICs are connected to the observed ToR, acting as six hosts with one 100 Gbps link each. We replay the widely-used RPC [40], Hadoop [44], and KV store [12] DCN traces on the hosts and scale the load to reach 40% core link utilization as in production DCNs [45].

Efficiency of queue management system. Rotation of calendar queues (§5.1) must be aligned precisely with the

time slices, but delays between switches, including the switch pipeline processing, packet serialization, and on-wire propagation delays, interfere with the accuracy of time calculation. We measure these delays from the observed ToR back to itself through the MEMS OCS to use the same clock for accurate timestamping. This is done by continuously sending packets at line rate using the on-chip packet generator on the switch. The ToR-to-ToR delay is from when queue rotation is triggered on the sender side to when each packet arrives at the Rx MAC of the receiver side. Fig. 11 plots the delays with different packet sizes. The minimum delay is 1287 ns, which we can offset by starting queue rotation earlier to ensure that the least delayed packet catches the upcoming circuit. The maximum delay is 1324 ns, indicating that the most delayed packet arrives $1324 - 1287 = 34$ ns after the circuit is established. No packet should be sent in this 34 ns to avoid packet loss. As will be discussed soon, this only wastes $34\,\text{ns}/2\,\mu\text{s} = 1.7\%$ of the 2 µs minimum time slice duration.

Queue occupancy estimation accuracy. The error of queue occupancy estimation (§5.2) is the difference between our estimated queue occupancy (EQO) from the ingress pipeline and the ground-truth queue occupancy read by an egress packet. We measure it with the above setting (in Fig. 11) but combine line-rate traffic and bursty traffic to fill and drain the queue periodically. The estimation accuracy increases with the update interval (Fig. 12). We find 50 ns draws a good balance between the estimation accuracy and the consumption of pipeline processing resources on the ToR. The estimation error is within 725 B, less than half MTU-size packet, and the packet generator sends one packet every 50 ns (or at 20 Mpps), which creates only 1.3% pipeline forward overhead on Tofino2 with 1.5 Bpps processing capacity.

Minimum time slice duration. We derive the minimum achievable time slice duration in OpenOptics to evaluate its generality to support various optical proposals. To ensure a duty cycle above 90%, the slice duration is typically set to at least 10× the guardband, which covers the maximum of circuit reconfiguration and unavoidable system delays, preventing data loss. This analysis focuses on system limits, assuming OCS hardware is not a limiting factor, and asks what time slice duration OpenOptics can achieve in that case.

The 34 ns queue rotation variance (Fig. 11) should be covered by the guardband. In addition, the queue occupancy estimation error of 725 B (Fig. 12) translates to 58 ns delay under 100 Gbps bandwidth, meaning packets may get impacted by false negatives for 58 ns, where a full queue is mistakenly seen as not full. Besides, our synchronization work shows up to 15 ns sync errors in a 108-ToR optical DCN [30], which requires a guardband of $15 \times 2 = 30$ ns for clock discrepancies above and below the actual clock. Therefore, the total guardband is $34 + 58 + 30 = 122$ ns, and with added headroom for runtime variations, we set the guardband to 200 ns. OpenOptics thus supports a minimum time slice duration of $200\,\text{ns} \times 10 = 2\,\mu\text{s}$, the shortest time slice achievable with commodity devices. We observe no packet loss in all the experiments with this guardband value.

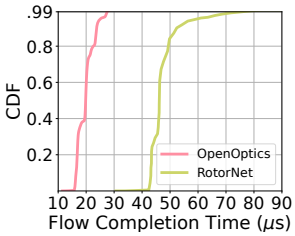

Figure 13: *UDP latency in OpenOptics vs. in RotorNet (Fig.15 blue curve in [38]).*

Table 2: *Resource usage of OpenOptics on Tofino2.*

Resource	***Usage***
SRAM	3.8%
TCAM	2.3%
Stateful ALU	9.4%
Ternary Xbar	13.8%
VLIW Actions	5.6%
Exact Xbar	7.8%

Emulation accuracy. We verified OpenOptics' correctness in Fig. 8 on both real and emulated fabrics. We now augment this validation by reproducing experimental results from "Realizing RotorNet" [38], the only *TO* architecture implemented so far, with actual OCSes. We replicate the UDP RTT latency experiment by continuously sending UDP packets from one host to another and measuring the RTT for each packet. Per Fig. 13, OpenOptics' results on the emulated optical fabric and RotorNet's on real OCSes exhibit similar latency distribution patterns, with stepped RTT increases corresponding to additional routing hops. The similar curve shapes indicate comparable routing behaviors between our emulated fabric and the actual OCS fabric, confirming correctness of OpenOptics. OpenOptics achieves lower RTTs and eliminates the long tail because our switch system and `libvma` stack produce less delay and variance than RotorNet's FPGA NICs and the kernel UDP implementation.

Switch resource usage. As listed in Table 2, the resource usage of an OpenOptics-enabled ToR in the 108-ToR DCN is a small percentage. The low usage of SRAM, VLIW Actions, and TCAM indicates the efficient implementation of registers and lookup tables. Stateful ALU and Ternary Xbar have higher usage, due to the arithmetic calculations and branching operations for slice-miss detection. All the resources are under 13.8% of an Intel Tofino2 switch's capacity, leaving sufficient room for OpenOptics to scale up to larger DCNs.

Optimization efficiency. We evaluate OpenOptics' optimization options in Appx. B using architectures and routing schemes that require them. Buffer offloading maintains switch buffer usage well below Tofino2's limit, even for buffer-intensive VLB routing, with offloaded packets reloaded to switches quickly and predictably. Congestion detection, paired with traffic pushback, facilitates and enhances existing congestion reactions, ensuring high throughput and zero loss. We also break down the performance gains of these two features to demonstrate their individual effectiveness.

8 Related Work

In §2, we presented an overview of DCN architectures [11, 14, 16–18, 21, 24, 27, 33, 36, 37, 39, 42, 43, 48–50, 52, 53], and implemented six of them over the 2 µs minimum time slice duration (§6), showing generality of OpenOptics. Our time-flow table abstraction applies to all architectures, regardless of constraints from today's commodity devices.

UCMP [31] and HOHO [32] are general routing algorithms for *TO* architectures. We realized them on OpenOptics and found out UCMP could reduce RotorNet's performance sensitivity to optical hardware, potentially allowing for the use of more cost-effective OCSes (§6). Recent transport proposals of reTCP [41], TDTCP [19], and Flare [20] highlight the need for system designs on top of optical architectures, which validates the design purpose of OpenOptics. Our TCP throughput case study (§6) confirms packet reordering, which is addressed in these works.

Etalon [41] and ExReC [54] are the closest evaluation platforms for optical DCNs, but they emulate switches on hosts without a full switch stack, also achieving limited scale (8 ToRs) and capacity (10 Gbps) [41, 54]. In contrast, OpenOptics is more realistic with the full switch and host implementations. We have demonstrated feasibility of OpenOptics in a 108-ToR setting at 100 Gbps (§7), and it can support higher capacity with better equipment. Additionally, OpenOptics offers a comprehensive solution, including real OCS support, emulated OCSes, and Mininet.

9 Conclusion

We presented the design and realization of OpenOptics, an open research and development framework that enables systematic system innovation and practical implementation for optical DCNs. While much prior work on optical DCNs has focused primarily on optical hardware, relegating the systems dimension to high-level simulations, OpenOptics re-centers the problem on end-to-end system realization. By decoupling software systems from the underlying optical architectures, OpenOptics transforms optical DCNs from closed designs into programmable and extensible systems.

At the core of OpenOptics are carefully designed abstractions (e.g., the time-flow table), a unified and expressive programming model, and a set of modular infrastructure services that serve as foundational building blocks for complex optical DCN designs. Together, these contributions establish a principled "narrow waist" that bridges hardware diversity and system innovation, substantially lowering the barrier to exploration, prototyping, and deployment.

OpenOptics enables researchers and practitioners to systematically transfer decades of experience and tooling from traditional DCNs into the optical designs, fostering reproducibility, comparability, and rapid iteration. By open-sourcing OpenOptics, we aim to catalyze a vibrant research and engineering community, accelerate innovation, and ultimately pave the way toward the practical and widespread deployment of optical DCNs.

Acknowledgments

We thank our shepherd, Keqiang He, and the anonymous reviewers for their insightful comments. We also thank Shizhen Zhao, Paolo Costa, Zhizhen Zhong, Alex Forencich, George Papen, George Porter, Kai Chen, and Bobby Bhattacharjee for their early feedback on this work. We are grateful to the participants of the OpenOptics demo at SIGCOMM'24 and the tutorial at SIGCOMM'25 for their early adoption and valuable feedback.

References

[1] Behavioral Model. https://github.com/p4lang/behavioral-model.

[2] Gloo. https://github.com/facebookincubator/gloo.

[3] Mellanox Messaging Accelerator. https://github.com/Mellanox/libvma/blob/master/README.

[4] Memcached. https://memcached.org/.

[5] Memslap. http://docs.libmemcached.org/bin/memslap.html.

[6] ns-3. https://www.nsnam.org/.

[7] OMNeT++. https://omnetpp.org/.

[8] OpenOptics Tutorial. https://openoptics.mpi-inf.mpg.de/tutorials/tutorial_index.html.

[9] OPNET. https://opnetprojects.com/.

[10] 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.

[11] Daniel Amir, Nitika Saran, Tegan Wilson, Robert Kleinberg, Vishal Shrivastav, and Hakim Weatherspoon. Shale: A practical, scalable oblivious reconfigurable network. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 449–464, 2024.

[12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[13] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Pias: Practical information-agnostic flow scheduling for commodity data centers. *IEEE/ACM Transactions on Networking*, 25(4):1954–1967, 2017.

[14] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, et al. 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*, pages 782–797, 2020.

[15] Joshua L Benjamin, Thomas Gerard, Domaniç Lavery, Polina Bayvel, and Georgios Zervas. Pulse: optical circuit switched data center architecture operating at nanosecond timescales. *Journal of Lightwave Technology*, 38(18):4906–4921, 2020.

[16] 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, 2013.

[17] 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. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1903–1911. IEEE, 2015.

[18] Li Chen, Kai Chen, Zhonghua Zhu, Minlan Yu, George Porter, Chunming Qiao, and Shan Zhong. Enabling {Wide-Spread} communications on optical fabric with {MegaSwitch}. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 577–593, 2017.

[19] 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*, pages 19–35, 2022.

[20] Federico De Marchi, Jialong Li, Ying Zhang, Wei Bai, and Yiting Xia. Unlocking superior performance in reconfigurable data center networks with credit-based transport. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 842–860, 2025.

[21] 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*, pages 339–350, 2010.

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

[23] Kaihui Gao, Li Chen, Dan Li, Vincent Liu, Xizheng Wang, Ran Zhang, and Lu Lu. Dons: Fast and affordable discrete event network simulation with automatic parallelization. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 167–181, 2023.

[24] 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*, pages 216–229, 2016.

[25] Raj Joshi, Ben Leong, and Mun Choon Chan. Timertasks: Towards time-driven execution in programmable dataplanes. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, pages 69–71, 2019.

[26] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th annual international symposium on computer architecture*, pages 1–14, 2023.

[27] Mehrdad Khani, Manya Ghobadi, Mohammad Alizadeh, Ziyi Zhu, Madeleine Glick, Keren Bergman, Amin Vahdat, Benjamin Klenk, and Eiman Ebrahimi. Sip-ml: high-bandwidth optical network interconnects for machine learning training. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 657–675, 2021.

[28] Jeongkeun Lee. Advanced congestion & flow control with programmable switches. In *P4 Expert Roundtable Series*, 2020.

[29] Yiming Lei, Federico De Marchi, Raj Joshi, Jialong Li, Balakrishnan Chandrasekaran, and Yiting Xia. An open research framework for optical data center networks. In *Proceedings of the ACM SIGCOMM 2024 Conference: Posters and Demos*, pages 86–88, 2024.

[30] Yiming Lei, Jialong Li, Zhengqing Liu, Raj Joshi, and Yiting Xia. Syncwise: Error-aware time synchronization for reconfigurable data center networks. In *23rd USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2025.

[31] Jialong Li, Haotian Gong, Federico De Marchi, Aoyu Gong, Yiming Lei, Wei Bai, and Yiting Xia. Uniform-cost multi-path routing for reconfigurable data center networks. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 433–448, 2024.

[32] Jialong Li, Yiming Lei, Federico De Marchi, Raj Joshi, Balakrishnan Chandrasekaran, and Yiting Xia. Hop-on hop-off routing: A fast tour across the optical data center network for latency-sensitive flows. In *Proceedings of the 6th Asia-Pacific Workshop on Networking*, pages 63–69, 2022.

[33] Cong Liang, Xiangli Song, Jing Cheng, Mowei Wang, Yashe Liu, Zhenhua Liu, Shizhen Zhao, and Yong Cui. Negotiator: Towards a simple yet effective on-demand reconfigurable datacenter network. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 415–432, 2024.

[34] 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.

[35] Hong Liu, Ryohei Urata, Kevin Yasumura, Xiang Zhou, Roy Bannon, Jill Berger, Pedram Dashti, Norm Jouppi, Cedric Lam, Sheng Li, et al. Lightwave fabrics: at-scale optical circuit switching for datacenter and machine learning systems. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 499–515, 2023.

[36] Yunpeng James Liu, Peter Xiang Gao, Bernard Wong, and Srinivasan Keshav. Quartz: a new design element for low-latency dcns. *ACM SIGCOMM Computer Communication Review*, 44(4):283–294, 2014.

[37] 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.

[38] William M Mellette, Alex Forencich, Rukshani Athapathu, Alex C Snoeren, George Papen, and George Porter. Realizing rotornet: Toward practical microsecond scale optical networking. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 392–414, 2024.

[39] 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*, pages 267–280, 2017.

[40] 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 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

[41] 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.

[42] 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. *ACM SIGCOMM Computer Communication Review*, 43(4):447–458, 2013.

[43] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, et al. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 66–85, 2022.

[44] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

[45] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

[46] Nitika Saran, Daniel Amir, Tegan Wilson, Robert Kleinberg, Vishal Shrivastav, and Hakim Weatherspoon. Semi-oblivious reconfigurable datacenter networks. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, pages 150–158, 2024.

[47] Naveen Kr Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *Proceedings of NSDI*, 2020.

[48] Guohui Wang, David G Andersen, Michael Kaminsky, Konstantina Papagiannaki, TS Eugene Ng, Michael Kozuch, and Michael Ryan. c-through: Part-time optics

in data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 327–338, 2010.

[49] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Zhijao Jia, Dheevatsa Mudigere, Ying Zhang, Anthony Kewitsch, and Manya Ghobadi. Topoopt: Optimizing the network topology for distributed dnn training. *arXiv preprint arXiv:2202.00433*, 2022.

[50] Dingming Wu, Yiting Xia, Xiaoye Steven Sun, Xin Sunny Huang, Simbarashe Dzinamarira, and TS Eugene Ng. Masking failures from application performance in data center networks with shareable backup. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 176–190, 2018.

[51] 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*, pages 1211–1222, 2025.

[52] Yiting Xia, Mike Schlansker, TS Eugene Ng, and Jean Tourrilhes. Enabling topological flexibility for data centers using omniswitch. In *HotCloud*, 2015.

[53] Yiting Xia, Xiaoye Steven Sun, Simbarashe Dzinamarira, Dingming Wu, Xin Sunny Huang, and TS Eugene Ng. A tale of two topologies: Exploring convertible data center network architectures with flat-tree. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 295–308, 2017.

[54] Johannes Zerwas, Chen Avin, Stefan Schmid, and Andreas Blenk. Exrec: Experimental framework for reconfigurable networks based on off-the-shelf hardware. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, pages 66–72, 2021.

A Queue Occupancy Estimation

We implement queue occupancy estimation using a register array in the ingress pipeline to track each calendar queue's occupancy. Ideally, these occupancy registers should be updated whenever packets are enqueued and dequeued. However, since registers at the ingress pipeline can only be updated by ingress packets, we increment the queue's occupancy by the packet size upon enqueueing, while for dequeuing, we estimate occupancy reduction through periodic updates.

We use packet generator to create an ingress packet at every *update interval* to trigger updates. As dequeuing happens at line rate, the packet triggers the occupancy register of the active queue to decrease by the product of the *link bandwidth* and the *update interval*, or sets it to zero if the result is negative, indicating the queue has emptied. As shown in Fig. 12, an update interval of 50 ns results in less than one packet estimation error at minimal pipeline processing overhead.

We make queue occupancy estimation a common module for slice-miss prevention in *TO* architectures. For example, Sirius and Shale query the occupancy of candidate intermediate nodes to send packets to the least occupied ones [11, 14]; Opera caps queue occupancy and drops packets exceeding this limit [37]; UCMP and HOHO check queue occupancy to defer packets that cannot fit within the intended time slice to a later one [31, 32]. Besides supporting these mechanisms, we also use push-back messages (§5.2) to halt flows at their origin, preventing overwhelming the circuits.

Buffer offloading efficiency. We have shown good performance of the host system in Fig. 9: flow pausing (§5.2) ensures expected throughput (50% of maximum throughput with 50% circuit availability) for direct-circuit routing without packet reordering. Here, we evaluate the stability of RTTs between switches and hosts, which is critical for buffer offloading.

We send 1500 B packets from the observed ToR to a connected host at 100 µs intervals. The host returns them upon receipt to simulate offloading and retrieval. Fig. 14 shows our `libvma`-based implementation ensures stable RTTs: 95% of RTTs exhibit a small variance of 0.75 µs, and their deviation from the expected 100 µs intervals remains within ± 0.25 µs. The performance is significantly better compared to a kernel module baseline, confirming the efficiency of `libvma`. In practice, we return offloaded packets to the switch early to offset the base delay and variance. Packets arriving marginally earlier are buffered on the switch consuming minimal memory.

Switch buffer usage. VLB, HOHO, and UCMP are the only routing solutions that have packets wait at intermediate nodes, so we evaluate switch buffer usage with those. We run 300 µs time slices, considered long for *TO* architectures. As shown in Table 3, HOHO and UCMP maintain low buffer usage across traces, as they prioritize latency by directing packets to the nearest time slices. In contrast, VLB causes packets to wait for extended periods at intermediate nodes. Nonetheless, the maximum buffer usage of 12.78 MB remains well below the 64 MB limit of Tofino2 switches, and buffer offloading to hosts (§5.2) reduces the buffer load to a minimal level. These results indicate that the calendar queues (§5.1) in our switch system are sustainable for higher bandwidth in the future.

Table 3: *99.9%-ile buffer usage under different traces with* 300 µs *slice duration. Total buffer on Tofino2 is* 64 MB.

Routings	***VLB (offloaded)***	***HOHO***	***UCMP***
KV Store	9.48 MB (1.26 MB)	2.40 MB	2.44 MB
RPC	9.96 MB (1.38 MB)	3.06 MB	4.15 MB
Hadoop	12.78 MB (1.56 MB)	3.90 MB	6.54 MB

B Extra Benchmarking Results

Effectiveness of congestion detection and traffic push-back. Among the routing schemes, HOHO is most vulnerable to congestion. It minimizes latency by always sending packets over the earliest available time slices, which can lead to overshooting and overwhelming the slice. When congestion is detected, HOHO defers packets that cannot fit into the scheduled time slice to a later one, but it does not implement congestion control to stop or slow down incoming traffic. In this case, traffic push-back can be enabled as a last line of defense when congestion persists. Therefore, we evaluate the effectiveness of our congestion detection and traffic push-back mechanisms (§5.2), specifically with HOHO.

We stress-test the calendar queues by increasing traffic load to 70% core link utilization, triggering the congestion detection and traffic push-back mechanisms. As shown in Table 4, without congestion detection or push-back (column 1), packets are always enqueued to the most desirable calendar queue, even when it exceeds the slice capacity. The queue pauses after the time slice ends, holding buffered packets until the next cycle(s), leading to long delays and packet loss. With congestion detection alone (column 2), loss rates and average delay decrease slightly as packets are deferred to later time slices when the primary queue is full, but queues eventually fill up, causing losses. When both mechanisms are combined (column 3), push-back engages once the primary

Table 4: *Effectiveness of congestion detection and traffic push-back in HOHO with Hadoop/RPC/KV-store traces under* 300 µs *time slices.*

Congestion Detection	✗	✓	✓
Traffic Pushback	✗	✗	✓
Throughput (Gbps)	67/69/64	67/69/64	50/54/49
Loss Rate	1.1%/1.0%/2.1%	1.1%/1.0%/2.0%	0%/0%/0%
Average Delay	160/150/98 µs	125/144/91 µs	6/5/8 µs
95%-ile Delay	2.2/2.2/2.1 ms	2.1/2.2/1.9 ms	85/90/83 µs

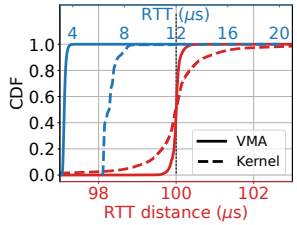

Figure 14: *RTT delay (blue) and RTT distance to the* 100 µs *interval (red).*

queue is full, and slice-miss detection handles in-flight traffic before senders react. This eliminates packet loss, and low queuing delay indicates most packets are enqueued in the primary calendar queue. These measures have similar effects on Opera and UCMP, reducing packet loss rate from 2.5% to 0% in Opera, also eliminating loss and reducing tail latency from 1.4 ms to 115 µs in UCMP.