Shale: A Practical, Scalable Oblivious Reconfigurable Network

Daniel Amir
Cornell University
Ithaca, New York, USA

Nitika Saran
Cornell University
Ithaca, New York, USA

Tegan Wilson
Cornell University
Ithaca, New York, USA

Robert Kleinberg
Cornell University
Ithaca, New York, USA

Vishal Shrivastav
Purdue University
West Lafayette, Indiana, USA

Hakim Weatherspoon
Cornell University
Ithaca, New York, USA

ABSTRACT

Circuit-switched technologies have long been proposed for handling high-throughput traffic in datacenter networks, but recent developments in nanosecond-scale reconfiguration have created the enticing possibility of handling low-latency traffic as well. The novel Oblivious Reconfigurable Network (ORN) design paradigm promises to deliver on this possibility. Prior work in ORN designs achieved latencies that scale linearly with system size, making them unsuitable for large-scale deployments. Recent theoretical work showed that ORNs can achieve far better latency scaling, proposing theoretical ORN designs that are Pareto optimal in latency and throughput.

In this work, we bridge multiple gaps between theory and practice to develop Shale, the first ORN capable of providing low-latency networking at datacenter scale while still guaranteeing high throughput. By interleaving multiple Pareto optimal schedules in parallel, both latency- and throughput-sensitive flows can achieve optimal performance. To achieve the theoretical low latencies in practice, we design a new congestion control mechanism which is best suited to the characteristics of Shale. In datacenter-scale packet simulations, our design compares favorably with both an in-network congestion mitigation strategy, modern receiver-driven protocols such as NDP, and an idealized analog for sender-driven protocols. We implement an FPGA-based prototype of Shale, achieving orders of magnitude better resource scaling than existing ORN proposals. Finally, we extend our congestion control solution to handle node and link failures.

CCS CONCEPTS

• **Networks** → **Data center networks**; **Network architectures**; **Network simulations**.

KEYWORDS

Optical Switches, Datacenter Networks, Nanosecond Switching

ACM Reference Format:
Daniel Amir, Nitika Saran, Tegan Wilson, Robert Kleinberg, Vishal Shrivastav, and Hakim Weatherspoon. 2024. Shale: A Practical, Scalable Oblivious Reconfigurable Network. In *ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3651890.3672248

ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0614-1/24/08
https://doi.org/10.1145/3651890.3672248

1 INTRODUCTION

Traditional datacenter networks have been designed primarily using packet switches. However, as network technologies and the desired characteristics of datacenter deployments continue to evolve, the limitations of packet switches are becoming more apparent. Due to the end of Moore's Law and Dennard Scaling, packet switches face increasing difficulty in scaling to meet network demands without consuming unnecessarily large amounts of power, both within high-density racks [39] and throughout the datacenter [2]. As a result, many emerging network designs have intentionally avoided using packet switches [9, 13, 18, 22, 24, 32, 40, 46]. Circuit switches present an exciting alternative to packet switches due to their reduced power consumption [2, 39], and potential to scale to arbitrary bandwidth (in the case of optical switches) [5, 24]. While historically, slow reconfiguration times limited the applicability of circuit switches to low-latency traffic, recent circuit switch designs have emerged that are capable of nanosecond-scale reconfiguration times, including both electrical [25] and optical [5, 8, 10] switches. Because circuit switches must be explicitly reconfigured to connect new source-destination pairs, using them imposes a design challenge. Until recently, circuit switched network designs have approached this problem by dynamically recomputing circuit switch configurations in response to new traffic [17, 18, 40]. Unfortunately, this dynamic reconfiguration process is impractical at the nanosecond timescales possible with recent switch designs, making them ill-suited to supporting low-latency traffic.

Oblivious Reconfigurable Networks (ORNs) are a new network design paradigm that can realize the potential of modern, rapid circuit switches. In ORNs, circuit switches are reconfigured oblivious to traffic, using a predetermined schedule, thus eliminating the latency inherent in classical designs. To support arbitrary traffic, ORNs use an oblivious routing scheme to route data indirectly to its destination. By co-designing the schedule and routing scheme, good performance can be achieved regardless of the traffic pattern, without dynamically adapting any part of the network. RotorNet [27], Shoal [39], and Sirius [5] are three network designs following the ORN concept that have already been demonstrated on physical test-beds. However, these designs all maximize throughput at the cost of poor latency scaling. As a result, they are not well suited to support modern datacenter traffic, that comprises a wide mixture of flows (multiple traffic classes) with different throughput and latency requirements [16].

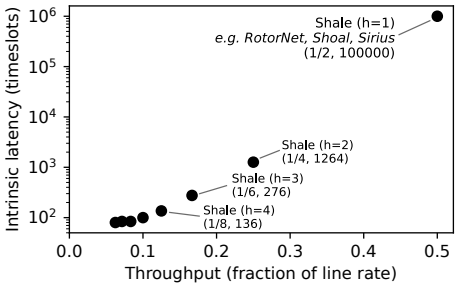

Figure 1: A comparison of the throughput and intrinsic latencies achievable by various tunings of Shale for a 100,000-node network. For our evaluation, timeslots begin every 5.632 ns.

We present Shale, an ORN that generalizes existing designs to achieve a *tunable* tradeoff between throughput and latency scaling. For datacenter-scale networks, Shale can be tuned to achieve an intrinsic latency multiple orders of magnitude lower than that of existing systems such as Shoal, Rotornet, and Sirius (which all reduce to a special tuning of Shale), as shown in Figure 1. Shale achieves this tunable tradeoff by carefully choosing a schedule from a collection of ORN schedules, each of which has been shown to achieve a Pareto optimal tradeoff between throughput and latency [4]. Shale additionally proposes a novel *interleaving* technique (Section 3.2) that carefully combines multiple ORN schedules. This allows multiple traffic classes, such as low latency and high throughput traffic, to simultaneously each be served on their ideal schedule.

Next, in order to achieve the Pareto optimal throughput-latency tradeoff, the traffic needs to be uniformly load balanced across all nodes in a Shale network (Section 3.1). This high degree of multi-pathing poses a challenge for existing congestion control algorithms, that fall short in maintaining low queuing in the network (Section 3.3). To overcome this, we design a novel congestion control algorithm which extensively modifies an existing ORN congestion control [39] to work in Shale's multi-hop context. Our algorithm achieves *bounded* queuing with minimal resource and processing overheads. Finally, we extend our congestion control mechanism to communicate node and link failures in real time, allowing traffic to be rerouted with minimal impact on throughput and latency (Section 3.4).

Using both an FPGA-based hardware prototype and large-scale packet simulations, we show that Shale's mechanisms achieve close to theoretical throughput and latency guarantees, while also achieving up to 13× better tail latency and 20× better tail buffer occupancy than state-of-the-art congestion control protocols such as NDP [19].

This work does not raise any ethical issues.

2 BACKGROUND

ORNs set up circuits based on a *traffic-oblivious schedule* which can be expressed in terms of timeslots. During each timeslot, the schedule dictates which nodes are to be connected to which other nodes, and switches remain in a fixed configuration, allowing nodes to send a single cell. This schedule is run synchronously, enabled by nanosecond-scale clock synchronization [5, 21, 38] and careful control of the propagation delay of each link. Timeslots are separated by *guard bands* to allow switches to reconfigure and absorb clock drift. To support arbitrary traffic patterns, *oblivious routing* is used to route data indirectly to its destination.

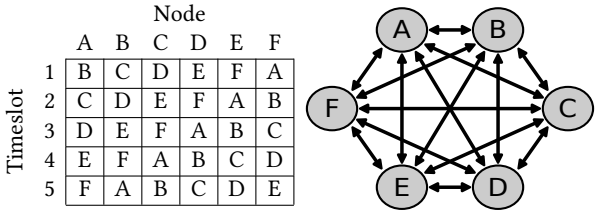

Figure 2: A single round-robin schedule for 6 nodes and connections formed over the course of the schedule.

Several designs, including Rotornet [27], Shoal [39], and Sirius [5], have been proposed following the ORN paradigm, all of which use similar, easy-to-express schedules and routing schemes. Here, we present an abstracted version of these designs which we refer to as the *Single Round-Robin Design* (SRRD). As the name implies, this design uses a schedule based on a single round-robin among all nodes in the system. We refer to one iteration of the schedule (one round-robin in this case) as an *epoch*.

To support arbitrary workloads, the SRRD uses a routing scheme based on Valiant Load Balancing (VLB) [45]. In VLB, when a node has a cell to send, rather than sending it directly to the destination, it first sends to a random intermediate node in the system[1]. The SRRD accomplishes this in one hop, which we refer to as the *spraying hop*, since it accomplishes packet spraying. Next, the cell is forwarded directly to the destination. This is also accomplished in one hop in the SRRD, which we refer to as a *direct hop*.

Since half of all hops in the SRRD are spraying hops, while half are direct hops that end at the destination node, a fully-congested SRRD network achieves destination throughput of half of the line rate. The use of VLB ensures that the network is evenly loaded across all workloads, preventing any bottleneck links that would further reduce throughput.

Ignoring queuing for now, this design achieves a latency of N timeslots in the worst case. The first hop to the intermediate node takes place over the first timeslot. However, in the worst case, the cell may need to wait an entire epoch to be able to forward the cell to the destination, for a total latency of N timeslots. As a result, this design has poor latency scalability; while ORNs have been proposed for systems with tens of thousands of nodes [5], systems based on the SRRD have thus far only been evaluated on hundreds of nodes.

3 SHALE

In this section, we present Shale, which is a generalization of existing SRRD ORN designs, providing tunable tradeoff between throughput and latency. We start by describing Shale's schedule and routing scheme, followed by how Shale combines its schedules to support multiple traffic classes. Finally, we describe Shale's congestion control.

[1]Both the SRRD and Shale deviate from this slightly to improve latency. When a node has a cell to send, it sends the cell to the first neighbor it can, rather than a random one. While long flows will sample all outgoing links, and flow arrival usually produces sufficient randomness for short flows, additional mechanisms (described in Section 3.3.2) ensure that intermediate nodes are well-distributed in all cases.

Timeslot	AA	BA	CA	⋯	BC	CC
1	BA	CA	AA	⋯	CC	AC
2	CA	AA	BA	⋯	AC	BC
3	AB	BB	CB	⋯	BA	CA
4	AC	BC	CC	⋯	BB	CB

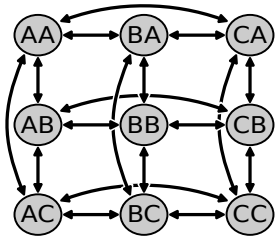

Figure 3: Shale's schedule with $h = 2$ for nine nodes. Here, each node is labeled with two letters, and participates in round robins with nodes differing by only one letter.

3.1 Schedule and Routing Scheme

Our recent theoretical work [4] has shown that within the ORN design space, many Pareto optimal tradeoffs are possible between worst-case throughput and *intrinsic latency* (latency resulting from the properties of the schedule and routing scheme, rather than queuing delay). Intuitively, this tradeoff exists because reducing intrinsic latency requires using paths with more hops, which have the flexibility to reach all nodes in the system using a shorter schedule. However, using more hops means each cell consumes more bandwidth over its entire path, reducing throughput. In our previous work [4], we proposed a family of theoretical ORN designs, called EBS, which forms the basis for Shale's schedule and routing scheme. EBS generalized the SRRD to use h spraying and h direct hops, rather than just one of each, achieving a maximum intrinsic latency of $O(h\sqrt[h]{N})$ and support throughput of $\frac{1}{2h}$. This configurable tradeoff between throughput and latency is always Pareto optimal for ORN designs, up to a constant factor in latency [4].

Shale directly uses the EBS schedule. In this schedule, rather than participating in a single round-robin among all nodes, nodes instead participate in h smaller round-robins, which we refer to as *phases*. Each node is assigned a unique set of h coordinates ranging from 1 to $\sqrt[h]{N}$. During a given phase, a node connects to each of the $\sqrt[h]{N} - 1$ nodes that match in all but a specific coordinate. An example of this schedule is shown in Figure 3.

Shale's routing scheme adapts the EBS routing scheme to work with fixed-size cells[2]. Paths are constructed using VLB, and are divided into *spraying* and *direct* hops. As in the SRRD, the first spraying hop is taken to the first available neighboring node. Following this, during the subsequent $h - 1$ phases, a random hop is taken. The first h hops thus have the overall effect of randomizing all h coordinates, sending the cell to a random intermediate node in the system. We refer to these hops as the *spraying semi-path*. Afterwards, during each of the following h phases, the cell is sent to the node which matches the destination in the corresponding coordinate. This takes up to h further hops, across up to h adjacent phases. We refer to the h direct hops as the *direct semi-path*.

For example, in $h = 2$ Shale, a cell could be sent from node AA to node CC via the path AA→BA→BB→CB→CC. In this path, the portion from AA to BB is the spraying semi-path, and from BB to CC is the direct semi-path.

Overall, paths in Shale are up to $2h$ hops long, and take place over up to $2h$ adjacent phases. This corresponds to an intrinsic latency equivalent to 2 epochs, or $2h(\sqrt[h]{N} - 1)$ timeslots. Because of the use of VLB, flow is well-distributed throughout the network on average for all workloads, leading to a worst-case throughput guarantee of $\frac{1}{2h}$ times line rate.

[2] EBS's design assumes fractional flow: each unit of flow is evenly divided between all possible intermediate nodes. In Shale, fixed-size cells cannot be subdivided, and must therefore each traverse a single path.

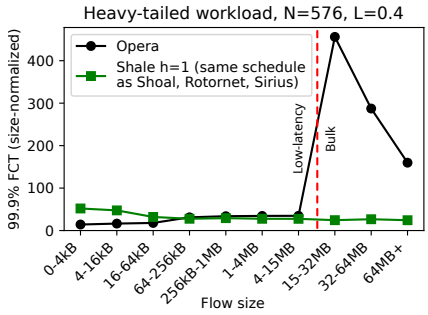

Figure 4: A comparison between similarly configured simulations of Opera and Shale with 576 nodes. Under Opera, long flows are penalized due to long timeslots (over 8000 ns) and use of RotorLB with a large number of nodes. Shale's far shorter timeslots (~5ns) leverage rapidly-reconfiguring switches to enable low latencies for short flows without these compromises.

3.2 Handling Multiple Traffic Classes

3.2.1 ***Existing approaches.*** Existing attempts to add support for multiple traffic classes to ORNs were motivated by the slow reconfiguration times of existing optical circuit-switched technology, which can penalize the completion times of short flows when used with ORNs. Opera [26] attempts to resolve this problem by running a single round-robin schedule in parallel using multiple transceivers at each node. Under Opera's schedule, at every point in time, nodes are connected in a random expander graph. Each configuration is held for several microseconds, allowing short flows to be routed using multiple hops in a single expander topology. Meanwhile, long flows are sent using the RotorLB transport protocol, which attempts to send traffic only when the source node is directly connected to the destination. For unbalanced workloads, RotorLB sends indirectly using VLB.

While this approach completely removes the effect of reconfiguration delay from short flow latencies (allowing it to outperform pure ORNs for the shortest flows), it also forces the use of long timeslots during which nothing is reconfigured. This is because each circuit configuration must be held at least as long as an end-to-end RTT, ideally longer to ensure most short-flow packets make it to their destination before a reconfiguration. Due to the speed of light, this duration can be orders of magnitude longer than the timeslot lengths that would otherwise be possible with new, rapidly-reconfiguring switches. When rapidly-reconfiguring switches are available, Opera has the effect of supporting low latencies at the expense of exacerbating certain scalability issues in the SRRD.

Figure 4 shows a comparison between Shale with $h = 1$ (using the same schedule as systems like Rotornet, Shoal, and Sirius), and Opera using similar 576-node configurations. We describe our Shale simulator setup, as well as the heavy-tailed workload used here, in Section 5, and use the publicly available Opera simulator [28]

with minor modifications to enable a similar simulation setup[3]. For Opera, we use the same 15MB bulk flow cutoff as was used in its original evaluation when testing this workload [26]. Both Shale and Opera are configured with a total aggregate bandwidth of 400 Gbps at each node and a propagation delay of 500 ns. While Shale is able to use a timeslot length of only 5.632 ns, Opera must hold each of its configurations for 8167 ns.

Due to RotorLB, bulk flows are heavily penalized by Opera, experiencing tail FCTs nearly 400 times slower than if they were sent directly to the destination at line rate. This is because in this system size, a node with a bulk flow to send will only be connected to the destination $\frac{1}{575}$ of the time, a problem which will only worsen with system size. This slowing of long flows imposes a severe scalability limitation and makes RotorLB impractical for very large systems.

At the same time, one of the advantages of RotorLB is that bulk traffic can be primarily buffered in the end-host's networking stack until a direct path is available. Abandoning this approach requires buffering traffic at intermediate nodes. However, as timeslot lengths increase, the amount of traffic that must be buffered similarly increases. This greatly increases the memory requirements to run such a system, compounding the high resource utilization imposed by scaling up SRRD-based systems (demonstrated in Figure 7).

Note that in Figure 4, the longest flows have a decreasing size-normalized FCT. This behavior also appeared in the original evaluation of Opera [26]. Despite this, for flows 128 MB and above, even the flow with the lowest size-normalized FCT in Opera was slower than the highest size-normalized FCT in Shale.

3.2.2 ***Interleaving.*** Shale's tuning parameter h provides multiple different schedules, each achieving a different tradeoff between throughput and latency for all traffic. This enables a new method of supporting multiple traffic classes, which we call *interleaving*, in which multiple sub-schedules are combined into a single schedule which combines their benefits. Each sub-schedule is used as-is with the routing unmodified, and each cell is routed on only one sub-schedule, ensuring that the properties of each schedule are maintained. In its simplest form, interleaving can be achieved by simply alternating between two different sub-schedules every other timeslot; for example, every even timeslot could be mapped to an entry of the higher-throughput $h = 2$ schedule, while every odd timeslot could be mapped to an entry of the lower-latency $h = 4$ schedule. This allows short flows to be sent on a low latency schedule while sending long flows on a high throughput schedule, maximizing the benefits of both. Depending on the workload and desired performance, different ratios of timeslots can be allocated to each schedule.

In the future, Shale could even be interleaved with demand-aware sub-schedules, which may be beneficial for mixed or partially known demands.

Performance impacts. An interleaved ORN sub-schedule has increased latency and reduced throughput proportional to the fraction of timeslots allocated to the sub-schedule. For example, a sub-schedule allocated half of the timeslots will take twice as long to complete each schedule iteration, doubling both the intrinsic latency and the sensitivity to queuing. However, propagation delay remains unaffected, and the degree of queuing in a low-latency schedule may be reduced if long flows are delegated to a separate, high-throughput sub-schedule. Similarly, a sub-schedule allocated half of the timeslots will only be able to support half as much throughput. However, the total throughput supported across interleaved high-throughput and low-latency sub-schedules will be greater than that supported by the low-latency schedule run in isolation. We evaluate multiple interleaved configurations of Shale in Section 5.2.

3.3 Congestion Control

Although Shale is based on an ORN design that achieves theoretically optimal intrinsic latency, in practice queuing can contribute a large fraction of total realized latencies. ORNs are particularly sensitive to queuing because their schedules empty at a rate of one cell per schedule iteration, rather than at line rate[4]. Congestion control is a core ingredient of Shale.

Shale differs from traditional packet-switched and circuit-switched networks in ways that make many existing congestion control mechanisms a poor fit. First, Shale relies heavily on multi-pathing, which is necessary to achieve good throughput on arbitrary workloads in ORNs [4]. These paths overlap in intricate ways, resulting in complex fate-sharing relationships that frustrate attempts to address congestion on an end-to-end basis. At the same time, the fixed size of cells makes it expensive to send short control messages. Space can be reserved in each timeslot for exclusive use by control messages, but if they are sent end-to-end, care must be taken to ensure they do not themselves get congested. Finally, congestion is likely to occur during all hops in a path - not just on the final hop due to incast. These differences make congestion control algorithms developed for packet-switched networks [12, 19, 30] potentially a poor fit for Shale.

3.3.1 ***Causes of congestion in Shale.*** We identify two primary forms of congestion in Shale: *egress* and *path-collision* congestion.

Egress congestion arises when cells with the same destination accumulate in queues leading to their destination. This can occur due to incast in which the total sending rate to a destination exceeds its available bandwidth. If this congestion is not addressed immediately, it quickly results in large amounts of queuing, as multiple senders can send cells more quickly than they can be delivered.

In Shale, egress congestion can also arise spontaneously due to the paths taken by cells, even when the destination's bandwidth is not exceeded. Randomized spraying ensures that traffic is evenly distributed between the final links to a given destination *on average*. However, during a given time period, some final links may receive more cells than others. Once a final link develops queuing, it remains until either fluctuations or congestion control result in less flow being routed along that link, or the total sending rate to the destination decreases. Egress congestion is thus likely to be

[3] Opera's simulator assumes end-hosts are connected to top-of-rack switches which actually participate in Opera. To compare to Shale which directly connects end-hosts, we simulate only one end-host connected to each TOR using a single 400 Gbps link, and configure each TOR with 8 x 50 Gbps uplinks.

[4] In Shale, increasing h reduces the length of the schedule, allowing queues to drain faster, but also increases the number of hops, and therefore the number of queues each cell traverses. Our evaluation shows that with effective congestion control, increasing h does indeed reduce latency even when congestion is considered.

sustained even in the absence of incast, particularly for heavy-tailed workloads.

Path-collision congestion arises when cells with unrelated destinations happen to be enqueued at the same node to send on the same link. Due to Shale's use of many indirect paths (required to achieve good performance [4]), path collisions can occur between any two source-destination pairs. Unlike egress congestion, path-collision congestion occurs for all workloads with high utilization. However, because Shale uses many indirect paths for each flow, long-running flows do not create sustained load on queues distant from their destination. Path-collision congestion is both less workload dependent and more transient than egress congestion.

We address egress congestion with our **hop-by-hop** design. It extensively modifies an existing congestion control proposed for the SRRD [39] so that it can operate on Shale's schedule. We address path-collisions with our **spray-short** design, which allows intermediate nodes to slightly deviate from Shale's routing algorithm based on their local queue lengths. We refer to the combination of the two as **HBH+spray**.

3.3.2 ***Hop-by-hop congestion control.*** Existing ORN proposals based on the SRRD have made use of a hop-by-hop congestion control design [39]. When an intermediate node receives a cell to be forwarded, it counts how many cells are currently present in its send queue to that destination node. When it next connects to the original sender, it uses space reserved in the cell header to send the previously measured queue length. Because the queues are FIFO and empty at a constant rate of one cell per epoch, the queue length allows the sending node to predict exactly when the previously sent cell will be forwarded. The sending node then waits that many epochs before sending a subsequent cell.

This design maintains the following invariant: At each intermediate node and at every point in time, there is no more than one cell enqueued that was received from the same neighboring node and is intended for the same destination. In the context of the SRRD, this invariant limits the number of forwarded cells in each queue to the incast degree of the destination, bounding egress congestion. Additionally, due to the short path lengths used by the SRRD, path-collision congestion can only occur on the first hop. Thus, total congestion on each queue is limited to the sum of the outcast degree of the sender, and the incast degree of the destination.

Extending to Shale. To adapt this into our **hop-by-hop** design intended for use in Shale, we make several significant changes. The first change is necessary to support the longer multi-hop paths used in Shale. To maintain a similar invariant in this context, intermediate nodes may also need to wait to send particular cells on a given link. However, it is not always possible to predict in advance when a node will be able to forward a cell on its next hop. The first change is thus: rather than sending a queue length immediately when a cell is received, an intermediate node waits until it actually does forward the cell, and then sends a token back to the previous hop. This token gives the previous hop permission to send a new cell with the same destination via the given intermediate node; such cells are now *eligible* to be sent via the given link. Note that after a cell is sent on its final hop, there is no subsequent hop and thus no token needs to be generated; cells are always eligible to be sent on their final hop without regard to tokens.

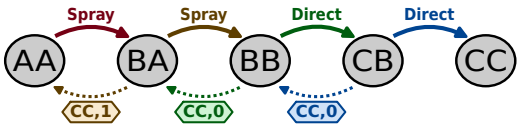

Figure 5: An example path which a cell might take in $h = 2$ Shale using hop-by-hop, showing both the hops taken by the cell and the returned tokens.

To prevent head-of-line blocking due to awaiting tokens, the second change is to transition away from strictly FIFO queues to PIEO (Push In Extract Out) queues [35]. These queues allow the first eligible cell to be extracted, even if it is not the first cell in the queue. PIEO queues can be efficiently implemented in hardware, as we demonstrate in our hardware prototype. If there are no eligible cells in its send queue, a node can instead generate a new cell from a flow it is sending, assuming that flow is itself eligible.

The third change prevents the possibility of deadlocks. Due to the arrangement of paths used by Shale, it is possible for a cycle of nodes to want to send each other cells with the same destination. If cells are only differentiated based on their destination, this would form a credit loop, causing a deadlock. To avoid this, **hop-by-hop** assigns cells to *buckets*, with each bucket corresponding to a given destination and an *index* representing the number of remaining spraying hops. A cell's eligibility is determined based on the bucket to which it will be assigned at the next hop, and tokens indicate which buckets have become eligible, rather than which destinations. This results in a slightly different invariant compared to hop-by-hop in the SRRD: At each intermediate node and at every point in time, there is no more than one cell present in each bucket that was received from the same neighboring node.

This design avoids deadlocks by eliminating cycles between buckets. In the spraying semi-path, each bucket leads to another bucket with a lower index, so cycles are not possible. In the direct semi-path, all buckets used have an index of 0. Cycles are prevented by the fact that each hop leads to a node that matches more coordinates of the destination.

For an example of how **hop-by-hop** works in practice, consider a cell sent from node AA to node CC via the path shown in Figure 5. When the cell arrives at node BA after its first hop, it is tagged with the previous hop AA, and because there is one remaining spraying hop, it is assigned to bucket $\langle$CC, 1$\rangle$. Once node BA forwards the cell on its next hop, it subsequently sends the token $\langle$CC, 1$\rangle$ to node AA at the first opportunity. Following this, node AA may send another cell to node BA which will be assigned to bucket $\langle$CC, 1$\rangle$.

One more change completes the design of **hop-by-hop**. Because a single node can generate multiple tokens intended for the same neighbor in a single epoch in our design, we reserve space in each cell header for two tokens, ensuring that any backlogs drain quickly.

Impacts. Because **hop-by-hop** separates cells with different destinations into different buckets, it primarily addresses egress congestion. The invariant maintained by **hop-by-hop** limits the number of cells enqueued at each node destined to each destination, bounding the queue lengths that can be created by egress congestion. In addition, **hop-by-hop** sends cells to uncongested destinations before to congested destinations, improving performance during incast.

The first negative impact of **hop-by-hop** is an increase in the size of cell headers. This increased overhead slightly decreases the achievable throughput for all workloads.

More significant is the potential for **hop-by-hop** to limit throughput when the propagation delay is too large relative to the epoch length. Once a node sends a cell, even if the next hop immediately forwards it, it still takes at least twice the propagation delay to receive a token back, which limits the sending rate of cells in the same bucket. This reduction in sending rate can be prevented by allowing nodes to send multiple cells before receiving back a token; we describe this in greater detail in Appendix D.

3.3.3 ***Spray via short queues.*** Our second design, **spray-short**, targets path-collision congestion. This design modifies Shale's routing algorithm slightly: when a node enqueues a cell on a spraying hop, instead of choosing a random queue in the next phase in which to enqueue the cell, it instead chooses the one with the fewest enqueued cells (with ties broken randomly).

Impacts. Enqueuing a cell on a spraying hop in the shortest possible send queue ensures that it collides with as few other cells as possible. This reduces the average number of path collisions in the network as a whole, in turn reducing path-collision congestion. **spray-short** may also ensure a more even load across the queues of a given node, lowering tail queue lengths and reducing total queuing at each node.

Because **spray-short** only uses locally available information at each node to decide which spraying hop to take, it adds no additional overhead to cell headers. However, **spray-short** modifies the routing algorithm to no longer be oblivious, departing from the theoretical model used in [4] and potentially breaking the throughput guarantee. When using **spray-short**, flow is not guaranteed to be evenly distributed among spraying hops, even on average. Depending on the workload, some links might be largely avoided as spraying hops due to heavy load by traffic on direct hops. When combined with **hop-by-hop**, **spray-short** might cause uneven fan-in near destination nodes, increasing **hop-by-hop**'s sensitivity to propagation delay (described in Section 3.3.2).

Despite these theoretical possibilities, we did not observe any throughput reduction due to **spray-short** in our experiments. Realizing a sustained reduction in throughput due to **spray-short** would require many flows to interact with each other such that for each flow, a small set of suboptimal spraying hops reliably have the shortest queues when cells from that flow arrive at intermediate nodes. This highly specific interaction appears to be extremely unlikely in practice, especially over a sustained period. We leave further theoretical investigation to future work.

3.4 Failures

In order to support arbitrary workloads with good performance, ORNs must route traffic via a large number of indirect paths, as discussed in Section 3. As a result, a single failure in Shale impacts all flows, as cells must avoid paths that traverse the failed node or link. To achieve this, failures must be both detected and communicated throughout the system. We accomplish this through an extension of **hop-by-hop**.

To detect failures in Shale, note that every epoch, each node both sends and receives a cell from each of its neighbors. If a node i does not receive a cell from a neighboring node j, it assumes that either the node or the link is failed. Once node i establishes that a failure has occurred, it immediately stops sending cells to node j, ensuring symmetric detection of link failures in case both nodes are still otherwise active.

To communicate information about failed links (and by extension, failed nodes), we introduce *invalidation tokens* and *re-validation tokens*. These tokens have the same format as the regular tokens used by **hop-by-hop**, and can be sent using the same portion of the header by adding two bits to differentiate them. Nodes send invalidation tokens to neighbors to indicate that they have no valid route for cells belonging to a given bucket. A re-validation token reverses the effect of a previously-sent invalidation token.

For buckets with zero remaining spraying hops, which correspond to direct hops, a node can send a single invalidation token to communicate all destinations that cannot be reached through direct semi-paths due to a single failed link. This is possible because of the fact that direct semi-paths are deterministic and form a tree. An invalidation token of the form $j, 0$ invalidates all direct semi-paths to node j itself, as well as to all child nodes of j in this tree.

Nodes use information gained from invalidation tokens to spray cells down paths that avoid failed links. We expand on this design in Appendix A.

Performance under failures. Routing around unusable paths can impact the throughput of remaining flows, especially those whose source or destination shares a phase group with a failed node. Fortunately, recent theoretical work [49] has shown that as long as failed nodes are well-distributed, with no more than h nodes missing from any given phase group, Shale's throughput guarantee is only reduced by a small factor. Performance can be maintained under arbitrary failures by occasionally exchanging the coordinates of failed nodes in order to ensure an even distribution. In Section 5.4, we show that even when up to 8% of nodes are failed, Shale maintains high throughput for the remaining nodes.

4 IMPLEMENTATION

We implement an FPGA-based prototype for Shale's end-host in Bluespec System Verilog [7], a high-level hardware description language that compiles to Verilog. Our prototype targets Terasic DE5-Net boards [44] comprising the Altera Stratix V FPGA [43], 234 K Adaptive Logic Modules (ALMs), 52 Mbits (6.5 MB) SRAM, and four 10 Gbps network ports.

4.1 End-host Design

One of the main challenges of implementing Shale's congestion control mechanism is that under **hop-by-hop**, nodes must send the first *eligible* enqueued cell, even if this cell is not at the head of the queue. We accomplish this with per-neighbor PIEO (Push In Extract Out) queues [35–37], which efficiently implement this capability. We store the contents of cells to be forwarded in per-phase, per-bucket FIFO queues which can be stored in DRAM, allowing us to store just the bucket IDs in the PIEO queues. To ensure efficient eligibility testing, we store these PIEO queues along with the per-bucket available token counts in on-chip memory. Finally, we store a per-neighbor FIFO queue of tokens to be returned. We illustrate these data structures in Figure 6.

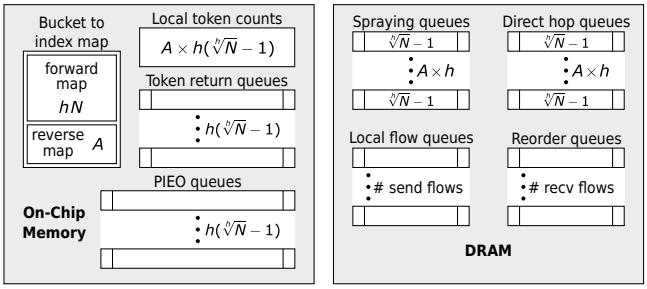

Figure 6: Memory layout of end-host implementation. Length of PIEO and token queues are tunable.

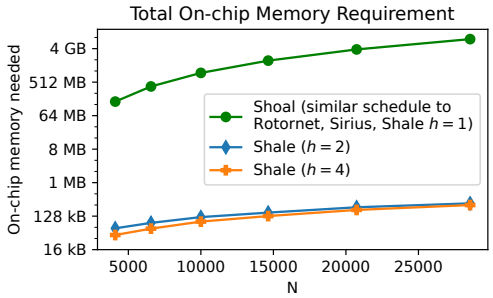

Figure 7: On-chip memory required by Shoal (representative of Rotornet and Sirius due to similar schedules and routing) and Shale for $h = 2$ and $h = 4$.

RX Path. When a node receives a cell, it first checks if it is the final destination of the cell. If it is, it places the cell into a reorder queue to be delivered to the application. Otherwise, the node determines whether the cell's next hop should be a spraying or a direct hop, and determines the next hop as described in Section 2. It decreases the remaining spraying hops in the header if needed, and enqueues the corresponding bucket ID in the PIEO queue for the next hop. It then writes the cell contents to the FIFO buffer associated with the cell's bucket and the phase of the next hop. In parallel, the node also updates its local token counts to reflect any tokens that were received in the cell header. These operations take 2 clock cycles in the critical path.

TX path. At the start of each timeslot, each node updates its current phase and neighbor, and attempts to dequeue an eligible bucket ID from the corresponding PIEO queue. If an eligible bucket exists in the queue, the first such bucket is returned, and the node reads a cell from the corresponding FIFO. If there is no eligible bucket, but there is an eligible local flow, it sends a cell from the local flow's queue. Otherwise, the node defaults to sending a dummy cell.

Once the cell to be sent is retrieved, the node adds up to 2 tokens enqueued for the current neighbor to the header, and starts sending. In total, the operations in the TX path take up to 7 clock cycles in the critical path. For a detailed breakdown of both the RX and TX paths, see Appendix C.

4.2 Optimizations

A naive version of our design would store the cells to be forwarded in per-neighbor, per-bucket FIFO queues. For each bucket and phase, **hop-by-hop** ensures that we only receive one cell from each neighbor in the phase. For spraying hops, because all $\sqrt[h]{N}-1$ of these cells could share the same bucket and next hop, each per-neighbor, per-bucket queue must individually be capable of holding all of them. This inflates the memory required by a factor of $\sqrt[h]{N}-1$, meaning each node must reserve space for a total of $h^2N(\sqrt[h]{N}-1)^2$ cells across all neighbors and buckets. We use two optimizations to dramatically reduce this memory requirement.

For our first optimization, we observe that for spraying hops, per-bucket queues can be shared across all neighbors in a given phase. This resolves the factor $\sqrt[h]{N}-1$ over-allocation of memory described in the previous paragraph. Note that because cells on their spraying hops can be sent via any neighbor in the same phase, this optimization does not affect the correctness of the routing algorithm. While this optimization does not apply to direct hops, all direct hops to the same destination received from the same phase will use the same next hop anyway. This optimization reduces the total memory across all buckets and phases to $h^2N(\sqrt[h]{N}-1)$.

Our second optimization uses the fact that empirically in our simulations, at any node and at any time, only a small fraction of the hN total possible buckets are active (with enqueued cells or outstanding tokens). Therefore, we only allocate cell buffers and token storage for a limited number of *active* buckets, set by the parameter A. To allocate indices, we use a freelist bitmap and a priority encoder. We store the mapping from bucket IDs to active bucket indices using a size hN array, and the reverse mapping using a size A array. This reduces the memory required for cell buffers to $2Ah(\sqrt[h]{N}-1)$ cells, and for local token counts to $Ah(\sqrt[h]{N}-1)$.

4.3 Hardware Resource Scaling

An important concern for Shale's scalability is its memory usage, both on-chip and in DRAM. Assuming each node has at most A active buckets at a time, at most Q_P bucket IDs enqueued in each PIEO queue, and at most Q_T tokens enqueued to return to neighbors, the total on-chip memory required by Shale is $O(h(\sqrt[h]{N}-1)(Q_P + Q_T + A) + hN)$. Additionally, as described in the previous section, Shale allocates space in DRAM to store $2Ah(\sqrt[h]{N}-1)$ cells to be forwarded to the next hop.

Shale's most significant compute requirements stem from its use of PIEO queues, which use priority encoders for eligibility testing and rank comparison. Because we only dequeue from one PIEO queue at a time, multiplexers can be used to share a single set of priority encoders across all PIEO queues at the same node, ensuring scalability on this front.

Figure 7 shows how Shale's memory requirements scale compared to existing designs, using Shoal [39] as a representative example. Shale's memory requirements are based on the maximum number of active buckets and PIEO queue length observed in our scalability experiments (Section 5.5), which are both doubled to account for other potential workloads. The disparity in memory requirements is due to Shale's reduced epoch length, which reduces the number of neighbors. Additionally, our memory optimizations (which in practice do not transfer well to existing schedules) allow us to reduce the amount of on-chip memory required per neighbor.

Takeaways. Shale with $h > 1$ uses orders of magnitude fewer hardware resources at datacenter scales than existing ORNs such as Rotornet, Sirius, and Shoal.

5 EVALUATION

We evaluate Shale using a custom packet-level simulator. In our simulation setup, end-hosts are equipped with a 400 Gbps network interface composed of eight 50 Gbps lanes, a configuration used by current 400 Gbps interfaces [31]. We assume a slot time of 45.056 ns, composed of 40.96 ns of usable slot time, followed by a 4.096 ns guard band. The usable slot time of 40.96 ns, combined with the per-lane bandwidth of 50 Gbps, results in a total cell size of 256 bytes. We reserve 12 bytes of each cell for the cell header, resulting in a payload of 244 bytes. Similar to [39], we take advantage of the eight lanes offered by each link to connect to neighbors in parallel, allowing a new timeslot to begin every 5.632 ns on average. The bandwidth of each lane, as well as size of the guard band, are achievable by an optical ORN based on tunable lasers, as demonstrated in [5]. To approximate a datacenter-scale network, we use a propagation delay of 0.5 μs, or 89 timeslots, and simulate 10,000 nodes except where otherwise specified.

We use synthetic workloads with flow sizes modeled after published datacenter traces. Flows arrive according to a Poisson process, and sources and destinations are chosen with uniform probability across all nodes. The first workload [6], which we call the *short flow workload*, uses the flow size distribution reported in a measurement study of production datacenters. The largest flow size in this workload is 3 MB, and it produces primarily path-collision congestion. We run its simulations for 2 million timeslots. The second workload [14], which we call the *heavy-tailed workload*, mimics a data mining workload. This workload features flow sizes of up to 1 GB, and thus produces significant levels of egress congestion. We run its simulation for 50 million timeslots.

To demonstrate performance under high utilization, we run our workloads at load factors near the theoretical throughput guarantee. The load factor L is defined as the average sending rate at each node, divided by the total available bandwidth at each node. On $h = 2$, we use a load factor of $L = 0.24$, while on $h = 4$ we use $L = 0.12$, approaching the throughput guarantee of each.

In order to evaluate performance across a wide range of flow sizes, we normalize each flow's completion time based on its size. For a flow with a length of F cells in a system with a propagation delay of P timeslots, the amount of time it would take to transmit the flow at line rate across one hop is $F+P$. We divide the actual flow completion time t by this value to compute the **size-normalized FCT**, $\frac{t}{F+P}$. After normalizing each individual flow's FCT, we divide flows into buckets based on their flow size and compute statistics independently for each bucket.

5.1 Hardware Prototype and Simulator

We begin by validating both our hardware prototype and our packet-level simulator by simulating identical permutation workloads on both, shown in Figure 8. To ensure that the hardware simulation is tractable, we simulate only 16 end hosts, using both $h = 2$ and $h = 4$ schedules. We simulate our hardware prototype using the ModelSim FPGA simulator [29]. Our simulation uses a clock speed of 156.25 MHz (resulting in a clock cycle of 6.4 ns) and 10 Gbps links, as found on the Terasic DE5-Net board equipped with Altera Stratix V FPGA [43]. To connect the nodes, we also implement a switch which connects these nodes according to Shale's connection

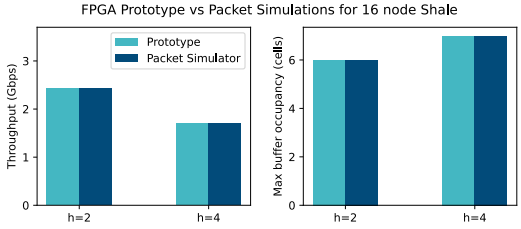

Figure 8: Throughput and queuing for the hardware prototype compared to packet simulations.

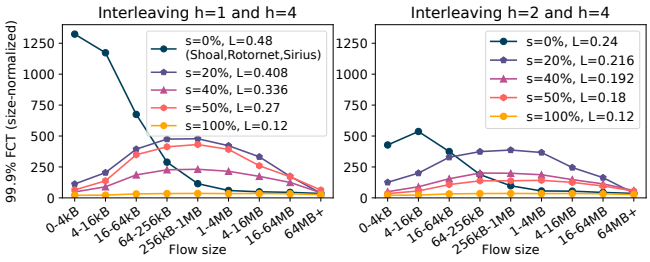

Figure 9: Size-normalized FCTs for the heavy-tailed workload under various interleaved schedules. s is the portion of timeslots allocated to the $h = 4$ subschedule, while L is the load factor used for each schedule.

schedule. We simulate a propagation delay of 2.5 μs and use 512B cells. All hosts are synchronized to the same clock. Accounting for overheads, we begin a new timeslot every 68 cycles, resulting in an available bandwidth of 9.412 Gbps.

Takeaways. Both our hardware prototype and packet simulator achieve almost exactly the same throughput and maximum queue lengths in these simulations (the differences are due to different randomization in spraying). For both $h = 2$ and $h = 4$, the throughput is above the theoretical guarantee of 2.353 Gbps and 1.176 Gbps, respectively. This helps confirm that both are correctly implemented, and that the hardware optimizations described in Section 4.1 do not impact correctness.

5.2 Interleaving

Figure 9 shows the results of simulating the heavy-tailed workload on various interleaved schedules with 10,000 nodes. For each schedule, the s parameter indicates what portion of the timeslots are used by the higher-h sub-schedule, which is used by short flows. Longer flows are sent on the lower-h sub-schedule which uses the remaining timeslots. The cutoff length is chosen to allow equivalent utilization of both for our given workload. Here, we run each schedule at almost the theoretical throughput limit, demonstrating increased throughput with interleaving. Note that when $s = 100\%$, the higher-h sub-schedule is used for all traffic, and when $s = 0\%$, only the lower-h sub-schedule is used. For $h = 1$, the schedule is identical to Rotornet, Shoal, and Sirius.

Interleaving allows the lower short-flow latencies of higher-h schedules to be achieved while supporting much higher throughput. As fewer timeslots are allocated to the higher-h subschedule, the time taken for each subschedule iteration increases, usually resulting in longer latencies. At the same time, the reduced flow-size

cutoff means that more flows are allocated to the low-h subschedule instead. This can occasionally improve latency in the higher-h subschedule, as with the $s = 40\%$ schedule outperforming the $s = 50\%$ schedule when interleaving $h = 1$ and $h = 4$ on this workload.

Takeaways. Interleaving allows Shale to simultaneously and scalably achieve high throughput and low latency. Depending on the specific requirements, network operators can choose an interleaving that balances latency and throughput.

5.3 Congestion Control Mechanism

We first evaluate our congestion control mechanisms, demonstrating the effectiveness of the combined **HBH+spray**.

Baselines. We use four alternative mechanisms as baselines.

(1) **none** provides a baseline of Shale with no additional congestion control. Shale implicitly prioritizes forwarded cells over newly originated cells, providing a primitive admission control (which remains active in other designs).
(2) **priority** provides a baseline of Shale with in-network scheduling. Prior works [3] have shown that sending packets from shortest flow first results in near-optimal mean flow completion times. In **priority**, when a cell arrives at a node, it is assigned a priority value $p = t + \ell * E$ based on its arrival time t, the overall size ℓ of the flow to which it belongs (preventing starvation), and the epoch length E. Cells with lower priorities are sent first.
(3) **ISD** (Idealized Sender-Driven) provides a very optimistic upper bound for the performance achievable by an end-to-end sender-driven congestion control mechanism which aims to achieve fair sharing between senders, such as the TCP suite of algorithms. In this design, nodes have clairvoyant, up-to-date knowledge of how many flows are currently being sent to each destination in the network. Whenever a node starts or finishes sending a flow, the global view of flows is immediately updated for all nodes. Nodes limit their sending rate to ensure that the total amount of traffic being sent to each receiver does not exceed a maximum bandwidth R. We use an R of $\frac{1.25}{2h}$, a low value that still allows the throughput bound to be reached.
(4) **RD** and **NDP** are both based on the receiver-driven transport protocol used by NDP [19], but simplified to only use PULL messages. **NDP** also uses packet trimming, providing a close approximation to NDP within the context of Shale. As with **hop-by-hop**, this protocol reserves a portion of each timeslot to send small control messages. However, here control messages are sent end-to-end using the same routing as regular cells. To reduce the queuing encountered by these control messages, only one PULL message is sent for every 20 cells received from each sender.

NDP has good performance when used with packet spraying on packet-switched networks. In this environment, queuing is almost exclusively due to incast, occurring in the queue between the destination TOR and end-host [19]. This greatly differs from Shale, where queuing can occur throughout the path. For our **NDP**, we use a maximum queue length of 100 to prevent too many cells from being dropped and reducing throughput due to retransmissions. Despite this conservative cutoff, when running the heavy-tailed workload under $h = 4$, over 3% of packets are dropped.

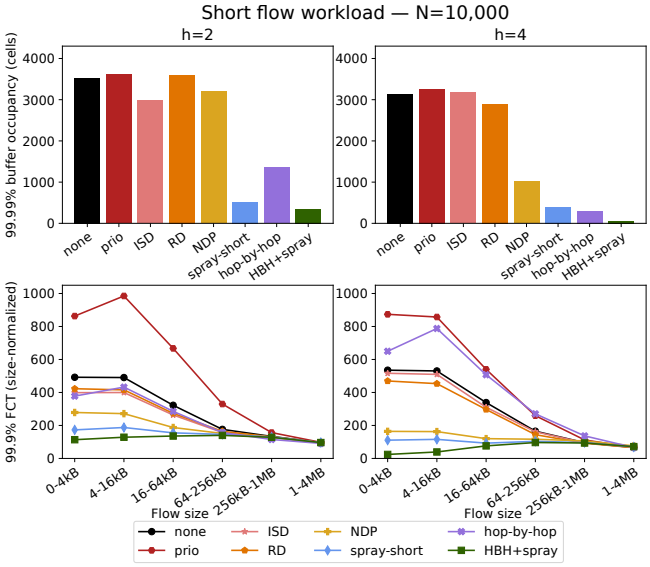

Figure 10: Buffer occupancies and normalized flow completion times for the short flow workload.

Short flow workload. We begin by examining the performance of our various congestion control mechanisms and baselines using the short flow workload, which primarily creates path collision congestion.

The lower graphs in Figure 10 show the FCTs achieved by various congestion control mechanisms for the short flow workload. As expected, **spray-short** is quite effective at improving tail flow completion times for all flow size categories. While **hop-by-hop** on its own is not well-suited for this workload, as it mainly addresses egress congestion, it still limits queuing, especially for $h = 4$. The best queuing and FCTs are achieved by the combined **HBH+spray**.

priority's scheduling policy optimizes mean flow completion time [3] (For mean FCT graphs, see Appendix B.1); here, this comes at the expense of tail latency. **ISD** and **RD** are clearly mismatched with this workload, having only marginal differences compared to **none**. This demonstrates the need to address path-collision congestion using in-network interventions, rather than end-to-end rate limiting. While **NDP** performs decently in this test, it is not quite as effective as **spray-short** and **HBH+spray**. Even though **NDP** had a lower maximum queue length than **spray-short** for both $h = 2$ and $h = 4$ (for queue length graphs, see Appendix B.2), many more queues reach near this maximum length. This made cells more likely to encounter long queuing delays.

In some of the lines in these graphs, the normalized FCT for 4-16 kB flows is longer than for 0-4 kB flows. For both of these flow sizes, the FCT is dominated by the latency of the most-delayed cell. When the cell latency distribution has an especially long tail, 4-16 kB flows are more likely to sample deep into this tail, resulting in worse tail FCTs.

The top two graphs in Figure 10 show the tail buffer occupancies, or total number of enqueued cells, at each node over the course of the simulation. As with flow completion times, **spray-short** is highly effective at reducing buffer occupancy for this workload.

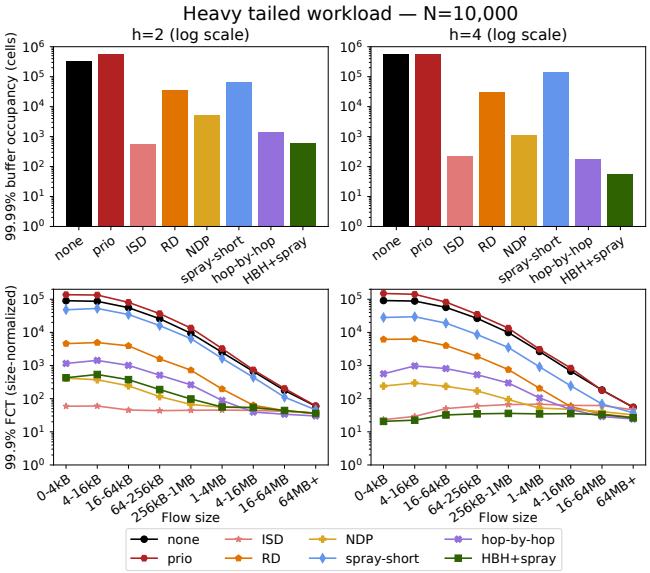

Figure 11: Buffer occupancies and normalized flow completion times for the heavy-tailed workload.

Combining with **hop-by-hop** results in even lower buffer occupancy, particularly for $h = 4$.

Finally, we measured throughput over the course of the workload. For this workload, following an initial ramp-up period, all mechanisms achieved average throughput within 2.5% of L, the target load. In particular, **spray-short** does not negatively affect the throughput of this workload.

Heavy-tailed workload. This workload produces considerable egress congestion, leading to FCTs and queuing differing by multiple orders of magnitude between different congestion control mechanisms.

The lower graphs in Figure 11 show tail FCTs. For this workload, **hop-by-hop** is effective at reducing tail latencies. In particular, short flows have their FCTs reduced by 2-3 orders of magnitude compared to **none**. **HBH+spray** brings an additional improvement. In particular, for $h = 4$, **HBH+spray** achieves tail normalized FCTs of under 24 for short flows. Because of $h = 4$'s use of 8-hop paths, this is within 3x of the theoretical limit without queuing. For $h = 2$, while **HBH+spray** limits queuing, tail latencies are slightly higher than for **NDP** and significantly higher than the idealized **ISD**. This is due to cases where short flows are incasted with long flows. **hop-by-hop** treats all cells to the same destination identically, so cells from short flows experience the same egress congestion as cells from incasted long flows; we discuss this further in Appendix B.3.

The top graphs in Figure 11 show the observed tail buffer occupancy. While **spray-short** is able to slightly reduce tail buffer occupancies, by maintaining its queuing invariant, **hop-by-hop** is able to reduce them by several orders of magnitude, outperforming both **RD** and **NDP**. **HBH+spray** achieves even better performance: for $h = 4$, nodes at the 99.99th percentile had fewer than 100 cells enqueued in total.

Takeaways. Shale's congestion control, **HBH+spray**, outperforms all of our baselines on both the short flow workload, and on the heavy-tailed workload when $h = 4$. In the latter case, **HBH+spray** achieves over an order of magnitude better short flow FCTs and buffer occupancy than **NDP**. For the heavy-tailed workload on $h = 2$, it is only significantly outperformed by the idealized **ISD** baseline.

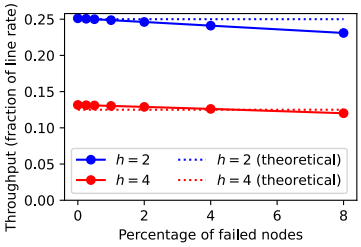

Figure 12: Throughput achieved with 10K nodes under failures, and the lower bound without failures.

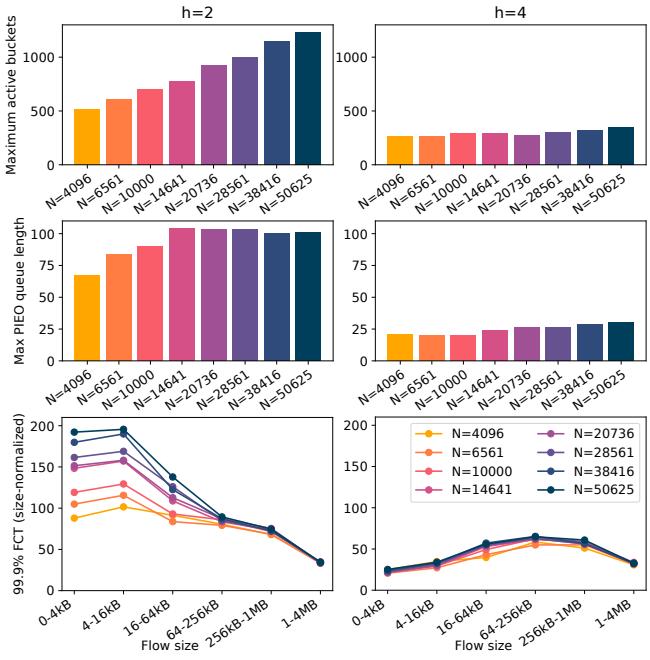

Figure 13: Active buckets, queue lengths, and FCTs for the short flow workload as system size scales.

5.4 Performance under Failures

We evaluate the effect of node failures on throughput for 10K nodes with Shale $h = 2, 4$. We use a synthetic workload consisting of 10 overlaid permutation traffic matrices to specifically benchmark the effect on throughput. Our permutations do not include the failed nodes, ensuring that we properly measure the throughput achievable by the remaining nodes. We run this simulation for 2 million timeslots and report the average destination throughput over the course of the run. Figure 12 shows the results of our experiment.

Takeaways. Under node failures, throughput is reduced roughly in proportion to the number of failed nodes. As long as most nodes are active, good throughput is maintained.

5.5 Scalability

To demonstrate the scalability of Shale, we consider two metrics: (i) the hardware resource requirements to buffer packets and (ii) the tail flow completion times. We simulate the short flow workload with systems of various sizes between 4,096 nodes and 50,625 nodes, as shown in Figure 13. The top graphs show the maximum number of active buckets and maximum PIEO queue length observed, two important values for determining the hardware resources required (as discussed in Section 4.3). The bottom graphs show the 99.9% size-normalized FCTs.

Even when scaling system size by over an order of magnitude, $h = 2$ Shale only uses 2.5 times more active buckets, and PIEO queue lengths appear to plateau. $h = 4$ Shale is even more resource efficient, with the number of active buckets remaining nearly flat, and queue lengths increasing by only around 50%. For $h = 2$, there is at most a 2x increase in short-flow FCTs, while for $h = 4$, Shale maintains almost exactly the same short flow latencies.

Takeaways. Shale has exceptional scalability. Both latencies and hardware resource requirements are similar even when scaling the system size by over an order of magnitude, especially for $h = 4$.

6 RELATED WORK

Existing ORN designs. RotorNet [27] proposes a datacenter-wide ORN based on Rotor switches, a form of optical circuit switch that requires committing to a fixed schedule. Prototype Rotor switches were demonstrated with a reconfiguration time of 150 μs, but it was argued that 10 μs should be possible for a production switch. Sirius [5] evolves on RotorNet by using tunable lasers and diffraction gratings to reconfigure the network far more rapidly, achieving a guard band of only 3.84 ns. Shoal [39] proposes an electronic ORN within a resource-disaggregated rack. By using circuit switches, Shoal reduces power consumption compared to packet switches, supporting 100s of nodes in a single rack.

While there has been clear progress in improving the hardware capabilities and reach of ORNs, these designs all use a schedule and routing scheme similar to the one described in Section 2. As a result, all of these systems have poor latency scalability. Shale represents a major step forward for ORNs by introducing a tunable tradeoff between throughput and latency. Because it is agnostic to the specific hardware implementation, Shale can extend all three of these designs, enabling them to scale to far larger environments.

Opera [26] uses greatly expanded timeslots, configuring the network as a new expander graph during each timeslot, allowing latency-sensitive traffic to be routed via multiple hops on a single configuration. We discuss the consequences of this design and compare it to interleaving in Shale in Section 3.2.1. Cerberus [15] uses a derivative of the Rotornet ORN design as one component of an optical datacenter network, along with demand-aware reconfiguration and static graphs. This demonstrates the potential of using ORNs as a building block of demand-aware networks, as we propose to do with interleaving. MARS [1] analyzes ORNs through the properties of the time-collapsed connection graph, and uses this to derive a tradeoff between throughput and buffering of forwarded cells. However, as with [4] it does not consider the effects of queuing at intermediate nodes.

Non-oblivious reconfigurable networks. Hybrid networks such as Helios [11] and c-Through [47] combine packet switches and MEMS-based optical circuit switches to build low cost, high bandwidth datacenter networks. These works rely on a central controller to periodically reconfigure the optical circuit switch based on the current traffic demand. Jupiter Evolving [33] augments a traditional datacenter network by connecting machine aggregation blocks with optical switches. These switches reconfigure infrequently, and use traffic engineering to ensure efficient use of the resulting direct-connect topology. TopoOpt [48] uses optical switches to create an optimized direct-connect topology for DNN model training, and customizes the all-reduce algorithm to best match the resulting topology. The recent Lightwave Fabrics paper [24] combines both an in-chip-interconnection and a datacenter-wide optical network to create optimized fixed topologies for machine learning jobs. These works demonstrate the incredible potential of optical circuit switches, but require extensive knowledge or even engineering of workloads, reducing their flexibility compared to ORNs.

Ethernet flow control. Ethernet flow control attempts to prevent packet loss, creating a lossless link layer. Ethernet flow control attempts to prevent packet loss, creating a lossless link layer. PAUSE frames [41] allow an overloaded node to request that its neighbors pause sending. This pause affects all traffic, potentially creating new congestion at previous hops which can cascade throughout the network [34]. Priority Flow Control, or PFC [42], allows separate pausing of 8 traffic classes, but otherwise suffers from similar weaknesses. Thanks to Shale's highly regular schedule and routing, we show that it is practical to avoid head-of-line blocking in **hop-by-hop**, avoiding these issues.

Asynchronous Transfer Mode (ATM). ATM networks operate using virtual circuits, in which end hosts must negotiate a path before sending traffic. Under credit-based congestion control [20], ATM switches implemented flow control on a per-virtual-circuit basis, with nodes exchanging credits to indicate free buffer space, similar to tokens in **hop-by-hop**. By maintaining credit per-destination, **hop-by-hop** is able to achieve similar isolation capabilities to credit-based congestion control while being coordination free.

7 CONCLUSION

In this paper, we introduced Shale, a practical, scalable ORN design. Shale generalizes existing ORN designs to achieve a tunable tradeoff between throughput and latency scaling that is Pareto optimal among all ORN designs (up to a constant factor). We show that by interleaving multiple tunings in parallel, both latency- and throughput-sensitive flow can be routed on the most advantageous schedule. In order to deliver low latency, we developed **hop-by-hop** and **spray-short**, two congestion control mechanisms that synergize with each other to provide a complete solution. We implemented an FPGA-based prototype end-host, showing that Shale can scale to datacenter-sized networks while using orders of magnitude fewer hardware resources than existing ORN designs. We also extended Shale's congestion control to communicate node and link failures, and showed that Shale continues to operate well even under failures. Shale's design is agnostic to the specific switching technology, and thus enables ORNs in all domains to be scaled to larger network sizes than would be practical with existing designs.

AVAILABILITY

Our packet-level Shale simulator and FPGA end-host prototype are available under an open-source license at https://reconfigurable-networks.github.io/.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Hitesh Ballani, and our anonymous reviewers for their extensive and constructive feedback. This work was supported in part by NSF grants CHS-1955125, DBI-2019674, CNS-2331111, CAREER-2239829, CCF-2402851, and CCF-2402852, a Microsoft Investigator Fellowship, and research awards from Google and Cisco (23089533).

REFERENCES

[1] Vamsi Addanki, Chen Avin, and Stefan Schmid. 2023. Mars: Near-optimal throughput with shallow buffers in reconfigurable datacenter networks. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7, 1 (2023), 1–43.

[2] Slavisa Aleksic. 2010. Electrical Power Consumption of Large Electronic and Optical Switching Fabrics. 95 – 96. https://doi.org/10.1109/PHOTWTM.2010.5421958

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. PFabric: Minimal near-Optimal Datacenter Transport. In *SIGCOMM*.

[4] Daniel Amir, Tegan Wilson, Vishal Shrivastav, Hakim Weatherspoon, Robert Kleinberg, and Rachit Agarwal. 2022. Optimal Oblivious Reconfigurable Networks. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing (STOC 2022)*. Association for Computing Machinery, New York, NY, USA, 1339–1352. https://doi.org/10.1145/3519935.3520020

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

[6] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC '10)*. Association for Computing Machinery, New York, NY, USA, 267–280. https://doi.org/10.1145/1879141.1879175

[7] Bluespec [n. d.]. Bluespec SystemVerilog. http://wiki.bluespec.com/bluespec-systemverilog-and-compiler. ([n. d.]).

[8] Q. Cheng, A. Wonfor, J. L. Wei, R. V. Penty, and I. H. White. 2014. Demonstration of the feasibility of large-port-count optical switching using a hybrid Mach-Zehnder interferometer-semiconductor optical amplifier switch module in a recirculating loop. *Opt. Lett.* 39, 18 (Sep 2014), 5244–5247. https://doi.org/10.1364/OL.39.005244

[9] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. 2015. R2C2: A Network Stack for Rack-scale Computers. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 551–564. https://doi.org/10.1145/2785956.2787492

[10] M. Ding, A. Wonfor, Q. Cheng, R. V. Penty, and I. H. White. 2017. Scalable, low-power-penalty nanosecond reconfigurable hybrid optical switches for data centre networks. In *2017 Conference on Lasers and Electro-Optics (CLEO)*. 1–2.

[11] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. 2010. Helios: a hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM '10)*. Association for Computing Machinery, New York, NY, USA, 339–350. https://doi.org/10.1145/1851182.1851223

[12] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2015. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. 1–12.

[13] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. 2016. ProjecToR: Agile Reconfigurable Data Center Interconnect. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 216–229. https://doi.org/10.1145/2934872.2934911

[14] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*. 51–62.

[15] Chen Griner, Johannes Zerwas, Andreas Blenk, Manya Ghobadi, Stefan Schmid, and Chen Avin. 2021. Cerberus: The power of choices in datacenter topology design-a throughput perspective. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 5, 3 (2021), 1–33.

[16] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. 2015. Queues Don't Matter When You Can JUMP Them!. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 1–14. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/grosvenor

[17] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. 2011. Augmenting data center networks with multi-gigabit wireless links. In *Proceedings of the ACM SIGCOMM 2011 conference*. 38–49.

[18] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R. Das, Jon P. Longtin, Himanshu Shah, and Ashish Tanwer. 2014. FireFly: A Reconfigurable Wireless Data Center Fabric Using Free-Space Optics. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 319–330. https://doi.org/10.1145/2619239.2626328

[19] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *SIGCOMM*.

[20] Raj Jain. 1996. Congestion control and traffic management in ATM networks: Recent advances and a survey. *Computer Networks and ISDN systems* 28, 13 (1996), 1723–1738.

[21] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. 2016. Globally Synchronized Time via Datacenter Networks. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 454–467. https://doi.org/10.1145/2934872.2934885

[22] Sergey Legtchenko, Nicholas Chen, Daniel Cletheroe, Antony Rowstron, Hugh Williams, and Xiaohan Zhao. 2016. XFabric: A Reconfigurable In-Rack Network for Rack-Scale Computers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 15–29. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/legtchenko

[23] Jason Lei and Vishal Shrivastav. 2024. Seer: Enabling Future-Aware Online Caching in Networked Systems. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 635–649. https://www.usenix.org/conference/nsdi24/presentation/lei

[24] Hong Liu, Ryohei Urata, Kevin Yasumura, Xiang Zhou, Roy Bannon, Jill Berger, Pedram Dashti, Norm Jouppi, Cedric Lam, Sheng Li, et al. 2023. Lightwave Fabrics: At-Scale Optical Circuit Switching for Datacenter and Machine Learning Systems. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 499–515.

[25] Macom M21605 Crosspoint Switch [n. d.]. Macom M21605 Crosspoint Switch. https://www.macom.com/products/product-detail/M21605/. ([n. d.]).

[26] William M. Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C. Snoeren, and George Porter. 2020. Expanding across time to deliver bandwidth efficiency and low latency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 1–18. https://www.usenix.org/conference/nsdi20/presentation/mellette

[27] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. 2017. RotorNet: A Scalable, Low-complexity, Optical Datacenter Network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 267–280. https://doi.org/10.1145/3098822.3098838

[28] William M. Mellette and George Porter. 2020. opera-sim. https://github.com/TritonNetworking/opera-sim. (2020). https://github.com/TritonNetworking/opera-sim

[29] Modelsim [n. d.]. ModelSim-Intel® FPGAs Standard Edition Software. https://www.intel.com/content/www/us/en/software-kit/750637/modelsim-intel-fpgas-standard-edition-software-version-20-1.html. ([n. d.]). https://www.intel.com/content/www/us/en/software-kit/750637/modelsim-intel-fpgas-standard-edition-software-version-20-1.html

[30] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *SIGCOMM*.

[31] QSFP-DD MSA. 2020. QSFP-DD Hardware Specification for QSFP DOUBLE DENSITY 8X PLUGGABLE TRANSCEIVER. http://www.qsfp-dd.com/wp-content/uploads/2020/08/QSFP-DD-Hardware-rev5.1.pdf. (2020).

[32] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. 2013. Integrating Microsecond Circuit Switching into the Data Center. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 447–458. https://doi.org/10.1145/2486001.2486007

[33] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, et al. 2022. Jupiter evolving: transforming google's datacenter network via optical

circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 66–85.

[34] S-A Reinemo, Tor Skeie, Thomas Sodring, Olav Lysne, and O Trudbakken. 2006. An overview of QoS capabilities in InfiniBand, advanced switching interconnect, and ethernet. *IEEE Communications Magazine* 44, 7 (2006), 32–38.

[35] Vishal Shrivastav. 2019. Fast, Scalable, and Programmable Packet Scheduler in Hardware. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 367–379. https://doi.org/10.1145/3341302.3342090

[36] Vishal Shrivastav. 2022. Programmable Multi-Dimensional Table Filters for Line Rate Network Functions. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 649–662. https://doi.org/10.1145/3544216.3544266

[37] Vishal Shrivastav. 2022. Stateful Multi-Pipelined Programmable Switches. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 663–676. https://doi.org/10.1145/3544216.3544269

[38] Vishal Shrivastav, Ki Suh Lee, Han Wang, and Hakim Weatherspoon. 2019. Globally Synchronized Time via Datacenter Networks. *IEEE/ACM Transactions on Networking* 27, 4 (2019), 1401–1416. https://doi.org/10.1109/TNET.2019.2918782

[39] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A Network Architecture for Disaggregated Racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi19/presentation/shrivastav

[40] Ankit Singla, Atul Singh, and Yan Chen. 2012. OSA: An Optical Switching Architecture for Data Center Networks with Unprecedented Flexibility. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, San Jose, CA, 239–252. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/chen_kai

[41] IEEE Computer Society. 1997. IEEE Standards for Local and Metropolitan Area Networks: Specification for 802.3 Full Duplex Operation. *IEEE Std 802.3x-1997 and IEEE Std 802.3y-1997 (Supplement to ISO/IEC 8802-3: 1996/ANSI/IEEE Std 802.3, 1996 Edition)* (1997).

[42] IEEE Computer Society. 2011. IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 17: Priority-based Flow Control. *IEEE Std 802.1Qbb-2011 (Amendment to IEEE Std 802.1Q-2011 as amended by IEEE Std 802.1Qbe-2011 and IEEE Std 802.1Qbc-2011)* (2011), 1–40. https://doi.org/10.1109/IEEESTD.2011.6032693

[43] Stratix V [n. d.]. Intel® Stratix® Series FPGAs and SoCs. https://www.intel.com/content/www/us/en/products/details/fpga/stratix.html. ([n. d.]). https://www.intel.com/content/www/us/en/products/details/fpga/stratix.html

[44] Terasic [n. d.]. DE5-Net FPGA development kit. http://de5-net.terasic.com.tw. ([n. d.]).

[45] Leslie G Valiant and Gordon J Brebner. 1981. Universal schemes for parallel communication. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*. 263–277.

[46] Meg Walraed-Sullivan, Jitendra Padhye, and David A. Maltz. 2014. Theia: Simple and Cheap Networking for Ultra-Dense Data Centers. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets-XIII)*. ACM, New York, NY, USA, Article 26, 7 pages. https://doi.org/10.1145/2670518.2673885

[47] Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T.S. Eugene Ng, Michael Kozuch, and Michael Ryan. 2010. c-Through: Part-time Optics in Data Centers. In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM '10)*. Association for Computing Machinery, New York, NY, USA, 327–338. https://doi.org/10.1145/1851182.1851222

[48] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. 2023. {TopoOpt}: Co-optimizing Network Topology and Parallelization Strategy for Distributed Training Jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 739–767.

[49] Tegan Wilson, Daniel Amir, Vishal Shrivastav, Hakim Weatherspoon, and Robert Kleinberg. 2022. Extending Optimal Oblivious Reconfigurable Networks to All N *(APOCS 2022)*.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A EXPANDING ON NODE FAILURES

A single node or link failure in Shale impacts all flows in the system, as cells must now be sprayed so as to avoid paths that traverse the failed node or link. Achieving this requires three ingredients: detecting failures, propagating information about failures, and reacting to information about failures. We implement these ingredients through an extension of **hop-by-hop**.

To detect failures in Shale, note that nodes both send and receive a cell from each of their neighbors once per epoch, even if there is no traffic to send. If a node i does not receive a cell from a neighboring node j, it can conservatively assume that either the node or the link is failed. Once node i establishes that a failure has occurred, it immediately stops sending cells to node j, ensuring symmetric detection of link failures in case both nodes are still otherwise active.

To propagate information about failed links (and by extension, failed nodes), we introduce *invalidation tokens*, which indicate which paths must be invalidated due to a failed link. Invalidation tokens have a similar format as the regular tokens used in **hop-by-hop**, and can similarly be sent using the portion of headers allocated to returning tokens. We add a single bit to the header in order to differentiate a regular token from an invalidation token. Invalidation tokens are of the form $\{j, n\}$, and indicate that a node has no valid way to route cells with n spraying hops remaining toward destination j.

Tokens with $n = 0$ communicate that the last link on the direct path leading to node j is no longer usable. Because a link failure invalidates many direct semi-paths at once, a single invalidation token with index 0 may indicate that cells at node i can no longer reach multiple destinations via direct semi-paths. Therefore, invalidation tokens with index 0 do not correspond to a single bucket (recall, a bucket indicates a specific destination and number of spraying hops remaining) in the same way as regular tokens. Meanwhile, tokens with $n > 0$ do correspond to a single bucket. They indicate that a node has no valid way to route toward destination j on a path with n spraying hops remaining.

We first address direct hops. When node i determines that its link in phase p to neighboring node j has failed, it immediately drops all cells awaiting their last hop to node j. Cells being sent on their direct semi-path via node j to another destination are reset to their first spraying hop and re-enqueued, while cells on spraying hops via node j are simply re-enqueued to a different neighboring node in the same phase. Node i then sends the invalidation token $\{j, 0\}$ to all of its neighbors.

When a node k receives an invalidation token of the form $\{j, 0\}$ from neighbor k' in phase q, it learns that the final link in the direct path from k' to j has failed. Since direct paths are deterministic and form a tree, it can compute the final link (i, j) and the phase p that this final link occurs in by considering the direct path the invalidation token must have traveled on over the past few phases. Node k reacts by first dropping all cells enqueued to be sent on a direct semi-path to node j via node k'. It then finds all cells enqueued on direct semi-paths via the failed link to destinations other than j, and resets them to their first spraying hop. Finally, node k forwards the invalidation token $\{j, 0\}$ to all neighboring nodes it connects with in phases p through $q - 1$, inclusive. These are the neighbors that could themselves forward a cell to node k whose direct path would then traverse the failed link.

We now address spraying hops. Once a node k receives an invalidation token from a neighbor k' of the form $\{j, n\}$, it from then on avoids sending cells to destination j on their $(h - n)$th spraying hop via k'. If it has any such cells currently enqueued, it instead attempts to spray them via a different spraying hop in the same phase which has not yet been invalidated. If k has already received identical invalidation tokens from all other neighbors in the same phase p, this means that node k cannot reach destination j on paths with n spraying hops remaining via any of its neighbors in phase p. Instead of re-spraying the cells, it drops them. Additionally, it sends an invalidation token of the form $\{j, n+1\}$ to all neighboring nodes in phase $p - 1$.

To account for failed links coming back online, we introduce *re-validation tokens*. Re-validation tokens have the same format as invalidation tokens and are propagated in the same way, but have the reverse effect. To differentiate invalidation and re-validation tokens, we add an additional bit in the header.

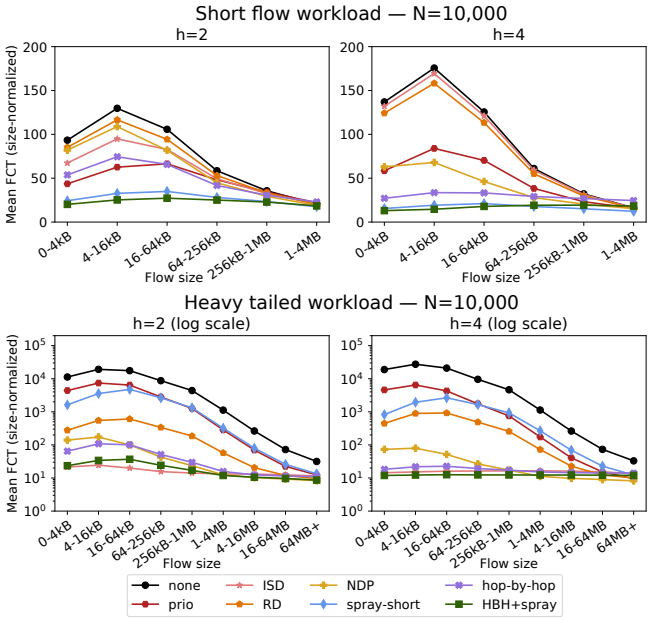

Figure 14: Mean slowdowns for experiments in Section 5.3.

B ADDITIONAL GRAPHS

B.1 Mean FCT Slowdowns

Figure 14 shows the mean flow completion times that were observed for the experiments shown in Figure 10 and Figure 11. **HBH+spray** has very low mean queuing times, and is usually either the best or nearly the best out of all evaluated congestion control mechanisms. These graphs show that **priority** indeed improves the mean flow completion time compared to **none**. However, because **priority** does not actually reduce queue lengths, **HBH+spray** outperforms it even on this measure.

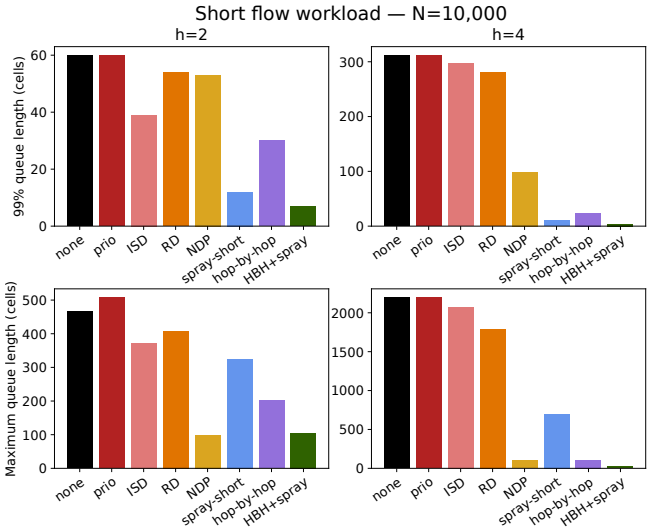

Figure 15: Queue lengths observed during the short flow workload.

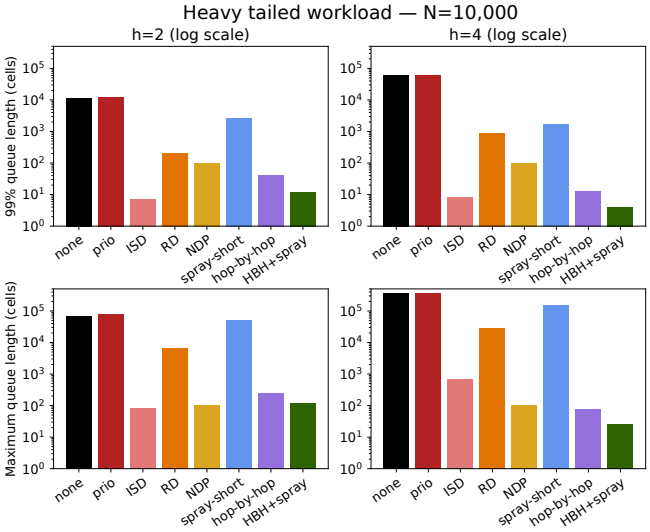

Figure 16: Queue lengths observed during the heavy-tailed workload.

B.2 Queue Lengths

Figure 15 shows the queue lengths observed during the short flow workload. Although **NDP** and **HBH+spray** have similar maximum queue lengths, **NDP** has a far higher 99th percentile queue length. This helps to explain why **NDP** experiences worse buffering.

Figure 16 shows the queue lengths observed during the heavy-tailed workload. Note that for both $h = 2$ and $h = 4$ **NDP**, over 1% of queues are at or near the point where packets would be dropped.

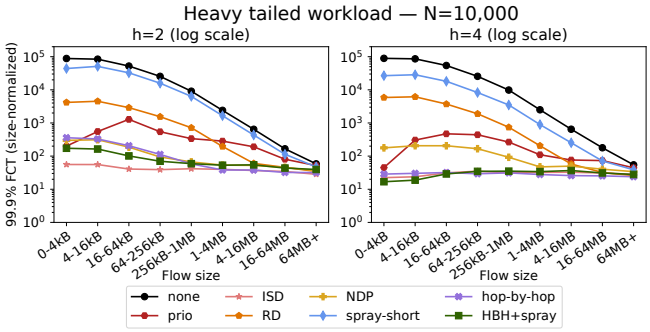

Figure 17: Tail latencies for flows that do not experience incast with very long (>256 MB) flows.

B.3 Tail FCT slowdowns for non-incasted flows

Because **hop-by-hop** does not differentiate between cells bound for the same destination, if a short flow is sent to the same destination as an ongoing long flow experiencing egress congestion, the short flow will experience the same egress congestion. For $h = 4$, because each node only has a small number of neighbors at our system scale, **hop-by-hop** is able to strongly limit the queue lengths in this type of situation. However, at our system size, nodes in $h = 2$ have an order of magnitude more neighbors, allowing longer per-bucket queues to build up at each node, increasing latency for incasted flows. Additionally, the epoch length for $h = 2$ is an order of magnitude longer than for $h = 4$ at our system size, making latency more sensitive to this effect. For this reason, **HBH+spray** experiences somewhat more tail latency than the **ISD** baseline for the heavy-tailed workload on $h = 2$, but not on $h = 4$.

In order to evaluate the degree of this effect, Figure 17 shows the tail latencies only for flows that are not being incast to the same destination as ongoing very long flows (size > 256 MB). When those flows are excluded, **HBH+spray** with $h = 2$ performs more similarly to **ISD**.

C ADDITIONAL IMPLEMENTATION DETAILS

Figure 18 visualizes the RX and TX paths described in Section 4.1, along with the clock cycles needed in the critical path for each step. Each node in Shale sends and receives one cell per timeslot, requiring both the RX and TX path to be run once per timeslot. However, they can be run in parallel.

Note that the TX path takes multiple clock cycles to complete. Dequeuing from a PIEO queue takes 4 cycles [35], only 3 of which are a part of our critical path. This delays transmission as the end-host dequeues the next eligible bucket from the relevant PIEO queue. Adding to this is the overhead from DRAM read latencies, which can delay the final steps of the TX path by many clock cycles. We therefore precompute which cells to send, allowing us to use the entire timeslot for transmission.

For our evaluation in Section 5.1, timeslots are long enough that the entire TX and RX paths complete within a single timeslot. However, both our TX and RX paths can be pipelined to enable much faster timeslot periods. The main limit is PIEO operations, which occupy the PIEO hardware module for four cycles. Our design can easily support four-cycle timeslots by using a dedicated PIEO

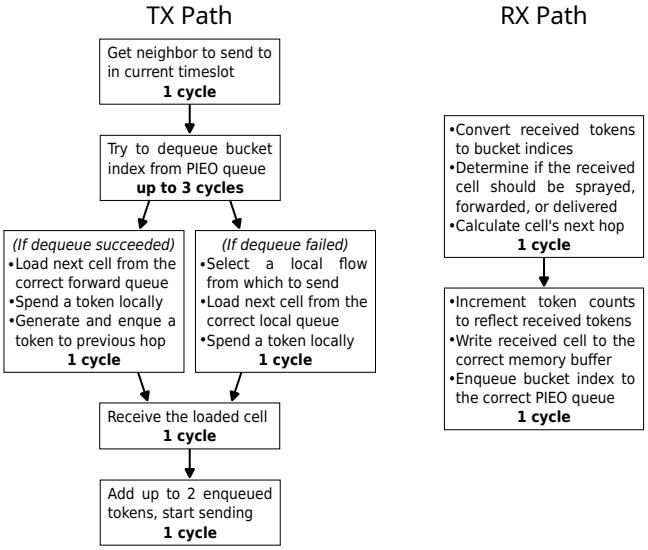

Figure 18: Step-by-step breakdown of receive and send paths in Shale's hardware implementation.

source id	15 bits	destination id	15 bits
remaining sprays	2 bits	sequence number	22 bits
token 1	17 bits	token 2	17 bits
CRC checksum	8 bits		

Figure 19: 12-byte header format valid for up to 32,768 nodes.

module for both the RX and TX paths. Our packet simulations in Sections 5.2 to 5.5 begin a new timeslot every 5.632 ns. Today's networking ASICs frequently have a clock rate of around 1 GHz [23], which is more than sufficient to support this timeslot period.

In order to minimize the overhead of reconfiguration, we set the timeslot period based on the length of the guard band (i.e., the reconfiguration delay) and the port count at each node. When these two parameters are kept constant, the same frequency ASIC is sufficient to support our routing and congestion control even at higher line rates. If needed, timeslots could potentially be started more frequently by either using a higher-frequency PIEO module, or using multiple PIEO modules in parallel.

In our evaluation of Shale, we assume 256-byte cells with a 12-byte header and 244-byte payload. Figure 19 shows a header structure that is valid for Shale with $h \leq 4$ and up to 32,768 nodes. This header supports **hop-by-hop**, which we combine with **spray-short** (which does not require support from the header).

D THE TOKEN BUDGET PARAMETER

There is significant potential for **hop-by-hop** to limit throughput when the propagation delay is too large relative to the epoch length. Once a node sends a cell, even if the next hop immediately forwards it, it still takes at least twice the propagation delay to receive a token back. This limits the sending rate of cells in the same bucket. If propagation delay is double the epoch length, then the round-trip to return tokens takes 4 epochs, and only $\frac{1}{4}$ the bandwidth of each link is available to a given bucket.

To mitigate this, we introduce a parameter T, the *token budget*. Rather than only sending one cell from the same bucket on a given link before receiving a token back, nodes may send up to T such cells. Increasing T increases the maximum sending rate of individual flows, at the expense of decreasing **hop-by-hop**'s effectiveness.

Increasing T is the most useful on the first hop. This is because the most constrained parts of the path between any two nodes are the first and final hop. While frames can always be sent on the final hop without waiting for tokens, this is not the case for the first hop. We therefore introduce an additional parameter T_F, the *first-hop token budget*. This parameter operates the same as T but only affects first hops, achieving most of the benefits of increasing T while reducing its negative effects.

For permutation traffic, Shale's throughput guarantee can be met as long as the propagation delay is no greater than hT_FE, where E is the epoch length. Increasing T_F beyond this point allows senders to take advantage of periods of reduced overall network activity to send above Shale's throughput guarantee. Shale's use of VLB ensures that there is high fan-out on subsequent hops in the spraying semi-path, and high fan-in along the direct semi-path. The degree of fan-in / fan-out is $\sqrt[h]{N} - 1$, so Shale's throughput guarantee can be met as long as $\frac{1}{2h(\sqrt[h]{N}-1)}$ of the bandwidth of the penultimate link is available. This is true when the propagation delay is no greater than $hT(\sqrt[h]{N} - 1)E$. For systems with longer propagation delay, the T parameter can be increased to mitigate this effect.