Network Load Balancing with In-network Reordering Support for RDMA

Cha Hwan Song, Xin Zhe Khooi, Raj Joshi, Inho Choi, Jialin Li, Mun Choon Chan
National University of Singapore
{songch,khooixz,rajjoshi,inhochoi,lijl,chanmc}@comp.nus.edu.sg

ABSTRACT

Remote Direct Memory Access (RDMA) is widely used in high-performance computing (HPC) and data center networks. In this paper, we first show that RDMA does not work well with existing load balancing algorithms because of its traffic flow characteristics and assumption of in-order packet delivery. We then propose ConWeave, a load balancing framework designed for RDMA. The key idea of ConWeave is that with the right design, it is possible to perform fine granularity rerouting and mask the effect of out-of-order packet arrivals transparently in the network datapath using a programmable switch. We have implemented ConWeave on a Tofino2 switch. Evaluations show that ConWeave can achieve up to 42.3% and 66.8% improvement for average and 99-percentile FCT, respectively compared to the state-of-the-art load balancing algorithms.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **In-network processing**; **Data path algorithms**; **Data center networks**.

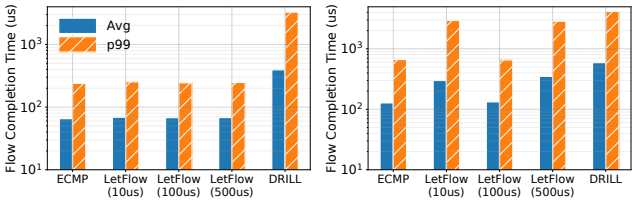
KEYWORDS

Network Load Balancing, Programmable Network, In-Network Packet Reordering, Programmable Switches, RDMA, P4

ACM Reference Format:
Cha Hwan Song, Xin Zhe Khooi, Raj Joshi, Inho Choi, Jialin Li, Mun Choon Chan. 2023. Network Load Balancing with In-network Reordering Support for RDMA. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3603269.3604849

1 INTRODUCTION

Remote Direct Memory Access (RDMA) allows end hosts to directly exchange data in the main memory while offloading network I/O responsibilities from the CPU onto RDMA-capable network interface cards (RNICs). Given the significant performance benefits that RDMA brings, it has been widely used in high-performance computing (HPC) settings deployed over proprietary Infiniband networks [51]. For its low-latency performance and to free up precious CPU cycles, nowadays, modern data centers are using RoCEv2 [14] to actively adopt RDMA technologies on Ethernet networks [17, 21, 25].



(a) Avg.Load 50% **(b) Avg.Load 80%**

Figure 1: RDMA FCTs with existing load balancing schemes on our hardware testbed (§4.2).

Data center network topologies (e.g., leaf-spine) are typically designed to scale while having sufficient redundancy in mind. Specifically, there are multiple end-to-end paths between any two server racks. Thus, to maximally utilize the available network capacity, load balancing is performed to spread network traffic across different paths. Equal Cost Multi-Path (ECMP), in particular, is widely used in today's data centers [38]. The next hop path is selected by hashing the packet fields and then taking the modulo of it over the number of available paths. Packets from a flow would always map onto the same path and thus these packets will always be delivered in the same order that it is sent.

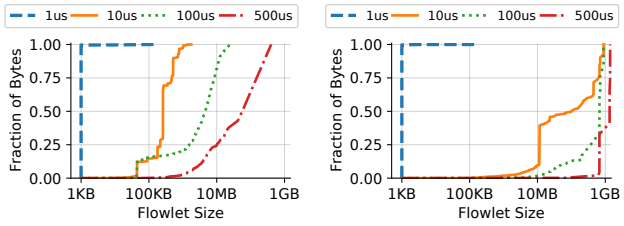
However, numerous studies have shown that ECMP is unable to distribute the load evenly over different paths [9, 27, 52] when the traffic is highly skewed. A plethora of works have been proposed to address the shortcomings of ECMP. For instance, some works use per-packet switching [18, 23] to achieve near-optimal load balance, but result in massive amounts of out-of-order packets. Other works [11, 35, 52, 59, 62] split a flow into chunks of packets based on inactive time gaps, so-called *flowlet* switching. Although flowlet switching provides a way to perform load balancing and avoids out-of-order packets, it is an opportunistic mechanism. Hence, the efficiency of flowlet-based approaches depends on the traffic characteristics i.e. whether there are flowlets available.

The motivation for this work comes from the observation that existing load balancing algorithms that improve upon ECMP are designed to run with TCP but not RDMA. In Fig. 1, we show how RDMA workloads perform using existing load balancing algorithms on our hardware testbed (see §4.2 for setup details). Regardless of the traffic load, existing approaches perform worse, if not on par, when compared to ECMP. We identify two causes for this performance degradation: *(i)* RDMA flow characteristics, and *(ii)* RDMA's response to packet out-of-order packets.

RDMA flow characteristics: Fig. 2 shows the flowlet sizes for TCP and RDMA traffic using different inactive time thresholds

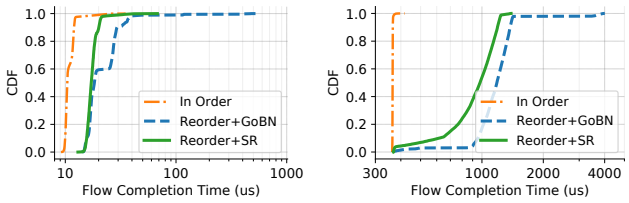
ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09.
https://doi.org/10.1145/3603269.3604849



(a) TCP (b) RDMA

Figure 2: Flowlet characteristics in TCP and RDMA using 8 concurrent connections performing bulk data transfer on 25Gbps link. We share similar findings with [42].



(a) 10KB message (b) 1MB message

Figure 3: The effect of 1 packet arriving out-of-order to the FCT on Go-Back-N and Selective-Repeat loss recovery mechanisms.

ranging from 1*us* to 500*us*. We see that for RDMA, flowlets are noticeably larger compared to TCP even for a small inactive time threshold of 10*us*. This implies that given an inactive time gap, there are significantly fewer chances to find flowlets in RDMA traffic compared to TCP. These observations are similar to those reported by Lu et al. [42]. This can be attributed to the fact that TCP transmits in bursts (e.g. TSO [34, 46]) and uses ACKs with batch optimization in order to achieve I/O and CPU efficiency which naturally creates time gaps between transmissions [20]. On the other hand, RDMA performs hardware-based packet pacing per connection (i.e., rate shaping) resulting in a continuous packet stream with small time gaps. Thus, due to the lack of sufficiently large flowlet gaps, flowlet switching-based approaches cannot work well with RDMA.

RDMA's response to out-of-order packets: RoCEv2[1] inherits many of the design assumptions of RDMA in Infiniband networks, one of which is that there is generally no loss in the network and therefore packets are delivered *in-order* [28]. As a result, when an RNIC receives a packet out-of-order, it treats it as an indication of packet loss (e.g., due to network congestion) and immediately initiates loss recovery which results in the sending RNIC decreasing its sending rate. On the contrary, TCP is more tolerant to out-of-order packets by buffering some out-of-order packets without immediate rate reductions or retransmissions (e.g., by waiting up to 3 dup-ACKs [13] or more [16]). Also, compared to TCP which is generally more programmable with kernel-level changes, RNICs are mostly fixed-function and typically have limited resources (e.g. for packet buffering) [60].

[1]In this paper, we use RoCEv2 and RDMA interchangeably.

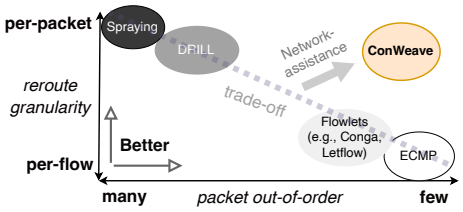


Figure 4: Trade-off between reroute granularity and packet out-of-order for in-network load balancing mechanisms [11, 18, 23, 59].

To quantify how out-of-order packets affect RDMA performance, we conduct two experiments consisting of one sender and one receiver connected to an Intel Tofino2 programmable switch [5]. The sender and receiver are both equipped with an NVIDIA Mellanox ConnectX-5 (CX5) [48] or ConnectX-6 (CX6) [49] RNICs that support different loss-recovery mechanisms, i.e., Go-Back-N (GBN) and Selective Repeat (SR), respectively. We artificially induce out-of-order packet arrivals by randomly selecting a packet from the RDMA flow and recirculating it in the switch before forwarding it.

Fig. 3 compares the FCTs[2] for short (10*KB*) and long (1*MB*) flows. We note that RDMA is highly sensitive to even a single out-of-order packet arrival. Generally, we observe that CX6 (using SR) exhibits better performance than CX5 (using GBN) due to fewer retransmissions. Nevertheless, in both cases, the performance is impacted by the reception of out-of-order packets which causes the sender to reduce its sending rate.

The reason for the rate reduction is that the RNIC interprets the detection of packet gap as packet loss, even though the cause can be attributed to either packet drops or out-of-order packet arrivals. However, if the cause is due to out-of-order packets, then the rate reduction is unnecessary, and together with spurious retransmission, it leads to lower network utilization.

In this paper, we pose the following question: Is it possible to support *fine-grained* rerouting for RDMA flows to spread and load balance the traffic among multiple paths without causing out-of-order packet arrivals? We answer this question in the affirmative and propose a solution called **ConWeave** (or *Congestion Weaving* [3]), a load balancing framework designed for RDMA in data centers.

ConWeave has 2 components, one running in the source and the other in the destination top-of-rack (ToR) switches. The component running in the source ToR switch continuously monitors the path delay for active flows and attempts to reroute whenever congestion is detected, instead of passively waiting for flowlet gaps. Such rerouting without sufficient packet time gaps can result in out-of-order packet arrivals at the destination ToR switch.

The key idea of ConWeave is to *mask* out-of-order packets from the RDMA end-hosts connected to the destination ToR switch. We do so by exploiting the state-of-the-art queue pausing/resuming features [39] on the Intel Tofino2 [5] programmable switch to put packets back in order, entirely in the data plane. To ensure that this

[2]We measure the flow completion time (FCT) based on *queue completion events* at the RDMA clients so as to understand the latency experienced by application-layer.

[3]In boxing, "weaving" refers to a defensive technique used to avoid attacks. The boxer shifts their body and weight from side to side in a weaving motion, making it difficult to land a clean punch.

can be done given the available hardware resources, ConWeave reroutes traffic in a principled manner such that out-of-order packets only arrive in a predictable pattern and these packets can be put back in the original order in the data plane efficiently. Notably, ConWeave is end-host agnostic. It is designed to be introduced in the network (i.e., at the ToRs) to work seamlessly with existing RNICs and applications without any modifications.

As shown in Figure 4, ConWeave presents a new paradigm in load balancer designs. With in-network reordering[4] capabilities, ConWeave can reroute traffic frequently while keeping out-of-order packets at bay. As a result, ConWeave is able to reach a more optimal operating point compared to existing schemes.

The contributions of this paper are as follows:

- We present a lightweight design for resolving packet reordering on a commodity programmable switch. The system has been implemented using P4 running on the Intel Tofino2 [5] switch.
- We design ConWeave, a load balancer design that performs per-RTT latency monitoring for active flows, and path switching while masking the out-of-order packet arrivals using the above in-network packet reordering scheme.
- We evaluate ConWeave on both software simulations and hardware testbed. Our results show that ConWeave improves the average and 99-percentile FCT by up to 42.3% and 66.8%, respectively, compared to the state-of-the-art.

The paper is organized as follows: in §2, we discuss how can we perform reordering using programmable switches; then, we discuss the design and implementation in §3; later, the evaluation results of ConWeave are presented in §4; before wrapping up in §7, we offer additional discussions and outline future work in §5 and summarize related work in §6.

2 REORDERING OUT-OF-ORDER PACKETS IN THE NETWORK: A PRIMER

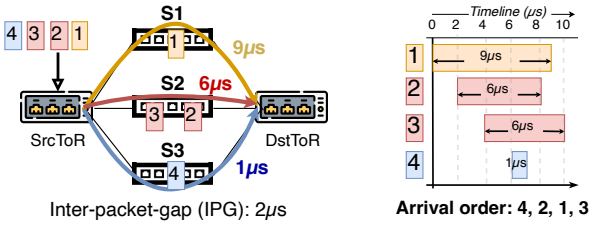
Generally, end-hosts assume the responsibility for putting the out-of-order packets back in order. With the emergence of commodity programmable switches, we believe that the network itself can, and should play an important role in reordering packets. Ideally, with data plane support, more fine-grained load-balancing mechanisms can be introduced if the network can mask these out-of-order behaviors from the end hosts. So, the question becomes, *how much reordering can commodity programmable switches, e.g., the Intel Tofino switches, support?*

Scenario: To make the discussion more concrete, we use the hypothetical scenario presented in Fig. 5. In the example, we assume packets are sent every 2*us*. Fig. 5a shows four packets of a single flow transmitted across paths with different delays, e.g., 9*us*, 6*us*, 1*us* for S1, S2, and S3, respectively. At the source-ToR, the packets are sent to paths S1, S2, and S3 to find a better path in an iterative way. Fig. 5b shows the packet in-flight times and the sequence of arrivals at the destination-ToR, i.e., 4, 2, 1, and 3.

2.1 Reordering on a Programmable Switch

In this paper, packet reordering refers specifically to the process of buffering and releasing the received packets *at the destination-ToR switch* so that the packets will be "restored" back to the same order as they were sent by the source-ToR switch.

[4]In this paper, we define "reordering" as the process of handling out-of-order packets by putting them back in order.



(a) Topology and path delays. (b) Packet arrival schedule.

Figure 5: Sample trace of a single flow and the order of packet arrivals at DstToR.

To answer the question of how much reordering can a commodity programmable switch perform, we first explore what features are needed to reorder packets in the data plane. For simplicity, we assume that there is no packet loss. Basically, we need to keep track of the *next expected packet sequence number* to ensure in-order delivery and hold out-of-order packets on a per-flow basis if needed. For example, if the expected sequence is 1 but a packet with sequence number 2 arrives, it has to be held until packet 1 arrives. To realize this logic, we need two key elements: (1) a primitive to hold/release packets (say, a *queue*), (2) stateful memory/operations that allow to update/check the state upon packet arrivals. The state operations include keeping track of the packet sequence number and the associated queue being used.

Required features: Fortunately, recent programmable switches have such capabilities. Using the Tofino2 [5] as an example:

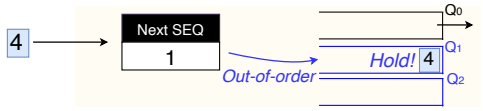
(1) Up to 128 First-In-First-Out (FIFO) queues per egress port with priority-based queue scheduling.
(2) Pause/Resume capability of an individual queue while maintaining line-rate packet processing for other queues.
(3) Tens of MBs stateful memory (e.g., register arrays) which can be updated by ALUs in the data plane.

Realizing mechanism: In Figure 6, we illustrate how the above primitives are used in the example scenario. In the beginning, we assign a queue Q_0 dedicated to forwarding in-order packets. When packet 4 arrives, we compare the packet sequence with the state *NEXT SEQ* and find that it is out-of-order. We hold the packet in the empty queue Q_1. Similarly, when packet 2 arrives which is out-of-order, we hold it in Q_2. Next, packet 1 arrives and is in order. Thus, we immediately forward it through Q_0. At this point, packet 2 is the next packet in sequence and is released from Q_2. The *NEXT SEQ* is increased to 3 accordingly. Lastly, for packet 3 arrival, we forward it, flush Q_1, and increment *NEXT SEQ* to 5 in a similar way.

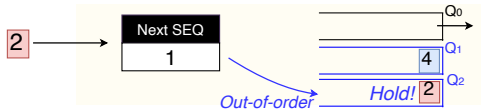
2.2 Practical Considerations

While the above sorting mechanism is logically simple, there are a couple of practical issues.

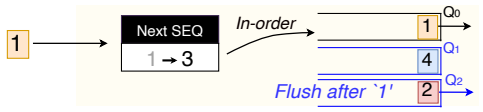
Limited queue resource: The idea of holding out-of-order packets in the data plane is bounded by the number of queues available. For instance, while our example needs only two queues to hold two out-of-order packets, holding N out-of-order packets would



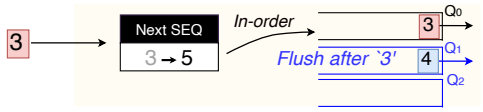
(a) Arrivals of packet 4, hold in Q_1.



(b) Arrival of packet 2, hold in Q_2.



(c) Arrival of packet 1, release 2.



(d) Arrival of packet 3, release 4.

Figure 6: An illustration of the packet reordering mechanism.

require N queues in the worst case (imagine N packets arrive in the reverse order). Unfortunately, there are only a limited number of queues that can be repurposed for sorting on hardware. Thus, this approach may not be amenable to cases where there can be *arbitrary out-of-orders arrivals*.

Lack of sorting primitive: One may consider using packet recirculation within the switch for sorting. However, such an implementation is inefficient. In fact, this is not a viable option as there are scenarios whereby the incoming out-of-order packets rate exceeds the speed of the in-order packet forwarding. In such cases, the recirculation port overflows resulting in packet drops. Another option is to use a sorted queue (e.g., PIFO [56]). Unfortunately, such data structures are not available on existing switch ASIC, and its approximated implementation (e.g., AIFO [61] and SP-PIFO [10]) falls short of providing strictly in-order delivery unless substantial hardware resources are dedicated for reordering.

In addition, to achieve high-speed processing (e.g., tens of terabits), switching ASICs allow only a limited set of stateful operations and imposes strict time constraints (e.g., per-stage in packet processing pipeline). Therefore, a restricted mechanism but yet fulfills the packet reordering requirement is needed.

Dealing with packet loss: Detecting out-of-order packet arrivals can be done by keeping track of the packet sequence number. However, a naive sequence tracing will stall when a packet loss occurs. The standard solution is to set a default waiting time and transmission proceeds to the next sequence after timeout. However, it is non-trivial to set a proper timeout value given that there can be substantial variability in path delay. Performing this task in the data plane with limited resources makes the task even more challenging.

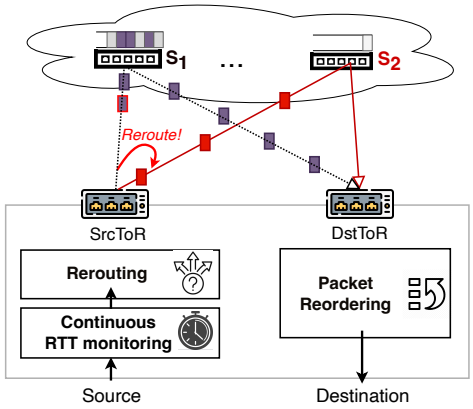


Figure 7: Overview of ConWeave. Only the ToR switches are required to be programmable.

3 CONWEAVE

The newfound capability to handle out-of-order packets in the network raises new opportunities for fine-grained load balancing mechanisms. Thus, we design ConWeave, a new load-balancing framework that tightly incorporates the network's reordering capability with "cautious" rerouting decisions.

Here, we describe the design and implementation of ConWeave. First, we provide an overview in §3.1, followed by discussions on the key components of ConWeave in §3.2 and §3.3. Finally, we discuss the implementation aspects of ConWeave in §3.4. We list the key parameters and packet types in Table 1 and Table 2, respectively.

3.1 Overview of ConWeave

The key idea of ConWeave is that we make decisions frequently (approximately every RTT) to determine if rerouting is advantageous based on network measurements. However, we need to take care that packets will be rerouted only when out-of-order packets can be efficiently sorted in the network prior to delivery to the end hosts.

Figure 7 depicts the overview of ConWeave:

- There are two components, one running on the source ToR switch and the other on the destination ToR switch. The ToR switches are connected through the data center network. We assume the use of some form of source routing so that the source ToR switch can "pin" a flow to a given path.
- The component on the source ToR performs the following functions: (1) latency monitoring to identify "bad" paths to avoid, (2) if congestion is detected, selects a new path and, (3) implements the mechanism which ensures that rerouting can be done "safely" without causing out-of-order arrival at the end hosts.
- The component at the destination ToR switch provides a packet reordering function that masks out-of-order delivery caused by rerouting.

To make further discussion of ConWeave more concrete, we refer to Fig. 8 when presenting the details using the example of a flow arrival and its rerouting. The example can be generalized to the case of many flows.

Parameter	Definition
θ_{reply}	RTT_REPLY cutoff time at SrcToR
$\theta_{\text{path\_busy}}$	Time period where a path is unavailable for rerouting
θ_{inactive}	The inactive time gap to force a new epoch

Table 1: ConWeave parameters (see §3.2)

Type	Definition
RTT_REQUEST	request pkt. sent from SrcToR to DstToR
RTT_REPLY	reply pkt. sent from DstToR to SrcToR
TAIL	Last packet along the OLD path
REROUTED	Packets sent through the new path after rerouting
CLEAR	Signals no more out-of-order pkts. in the epoch
NOTIFY	Signals a congested path to the SrcToR

Table 2: Packet types in ConWeave.

3.2 "Cautious" Rerouting Decisions

Ideally, we want fine-grained traffic rerouting, e.g., packet spraying, to maximize network utilization. However, this increases the number of packets that would arrive out-of-order in unpredictable patterns and would require multiple rounds of sorting at the receiving end. Thus, it is crucial for the rerouting design to produce *predictable* packet arrival patterns in order to exploit the hardware reordering capabilities efficiently. *How can this be done?*

We perform rerouting under the following three conditions: *(i)* the existing path is congested, *(ii)* there exists a viable path that is not congested, and *(iii)* any out-of-order packets caused by previous reroutes have been received at the destination ToR.

Conditions *(i)* and *(ii)* are imposed to ensure that rerouting is needed and a good alternative path is available. Conditions *(iii)* is imposed to produce predictable arrival patterns in the sense that *any flow can only have in-flight packets in at most two paths at any instance of time.* The reason is the following. First, for rerouting to occur, condition (iii) is met. All the packets sent in the previous rerouting should have arrived at the destination ToR switch and all current in-flight packets are traveling on a single path. After rerouting, there can now be two active paths with inflight packets. Condition (iii) prohibits another rerouting to occur until the condition becomes true again.

Next, we describe the ConWeave's rerouting mechanism using Figure 8. In the initial state, a flow always begins with a new *epoch.*

3.2.1 Continuous RTT monitoring. For each individual active flow, ConWeave continuously monitors the RTT to detect congestion. To achieve that, ConWeave selects and marks one packet from the flow as ① RTT_REQUEST in every epoch at the SrcToR (Source ToR Switch). At the DstToR (Destination ToR Switch), whenever an RTT_REQUEST is received, it replies with an ② RTT_REPLY to the SrcToR. Here, the RTT_REPLY is always marked with the highest priority. Thus, the time taken for the SrcToR to receive it on the reverse path indicates the current path congestion status to the DstToR in the forward path. As long as the RTT_REPLY is received prior to the cutoff, the process is repeated for the next epoch. However, if an ③ RTT_REQUEST is sent and no reply was received before

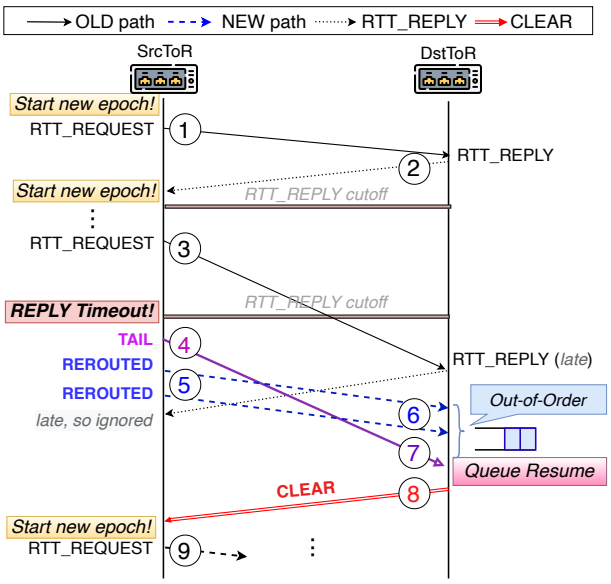


Figure 8: ConWeave in-action. Illustration for a single flow case but generalizes to many flows.

the cutoff time, we infer that the existing path is likely to experience congestion given the sudden increase in RTT. The rerouting mechanism is triggered.

Design intuition. The SrcToR is able to infer the path status, e.g., impending congestion if the RTT_REPLY does not reach back in time as opposed to waiting for the probes or piggyback packets [11, 62]. This ensures timely rerouting.

3.2.2 Path selection. Before rerouting, ConWeave needs to select a non-congested path to reroute. ConWeave keeps track of congested paths through in-band signaling between the ToR switches. Whenever a ConWeave DstToR receives packets with congestion indications (e.g., marked ECN bit [12, 63]), it triggers a NOTIFY packet carrying path-related information (e.g., path ID) that is sent to the SrcToR. Upon receiving a NOTIFY packet, the corresponding path will be marked as *unavailable* by the SrcToR for a time of $\theta_{\text{path\_busy}}$. We explain how $\theta_{\text{path\_busy}}$ is set in §4.1.

Whenever a new path is needed, ConWeave randomly selects a few sample paths (e.g., 2) (no active probing is performed) and checks if any of them are marked as *unavailable.* If a path is available, we select the path for rerouting. If all paths are *unavailable*, the network is considered highly congested and rerouting is aborted.

Design intuition. ConWeave does not actively probe for less loaded paths to minimize measurement overhead. Instead, it avoids congested paths by inferring the network load through in-band signaling. When the network load is high and all paths are congested, rerouting to another (congested) path exacerbates the congestion and is thus not performed.

3.2.3 Rerouting traffic. For rerouting purposes, ConWeave splits the packet stream into two "chunks" – OLD and NEW. Once a path is selected for rerouting, ConWeave sends one more packet over the OLD path by marking it as ④ TAIL. The remaining packets are

then rerouted to the NEW path and marked with the ⑤ REROUTED flag. The TAIL and REROUTED flags allow the DstToR to differentiate the two packet streams that are sent before, and after rerouting. Once rerouting is performed, the SrcToR expects a ⑧ CLEAR packet from the DstToR to indicate that all OLD packets have been received. DstToR knows that through the reception of the TAIL packet. With the CLEAR packet, the SrcToR can then proceed to start a new epoch ⑨.

Handling CLEAR packet loss. The CLEAR packet is critical in ensuring that ConWeave could progress into the next epoch. However, if the CLEAR packet is lost or not sent by the DstToR, ConWeave would not be able to resume the RTT monitoring mechanism. To overcome this, the SrcToR keeps track of an inactive period of connection, i.e., the gap between the last packet process time and the current time. If it exceeds θ_{inactive}, ConWeave automatically progresses to the next epoch without having to wait for the CLEAR packets that may never arrive. We set θ_{inactive} long enough such that it is unlikely to have out-of-order arrivals before starting a new epoch (i.e., sum of the *maximum* propagation time of the TAIL and CLEAR packets).

3.3 Masking Packet Reordering

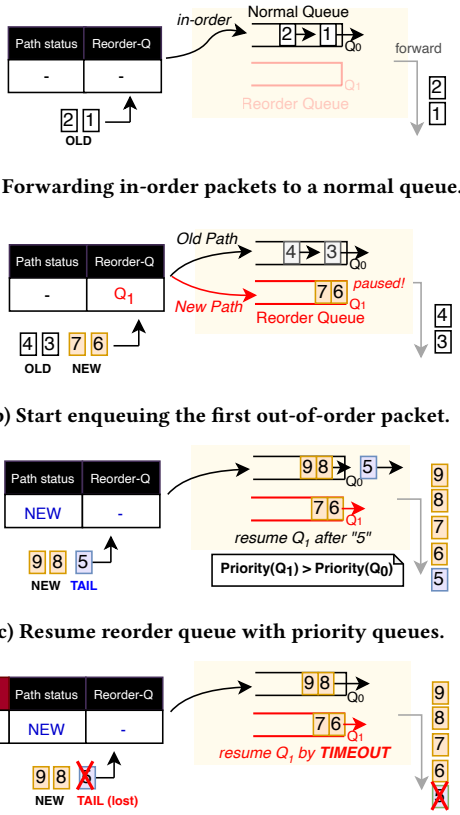
To ensure in-order packet delivery to the end hosts, we make use of the sorting primitives outlined in §2. Knowing that any flow can only have in-flight packets in at most two paths, sorting the packet streams is simple and can be done using only one queue to hold the out-of-order packets.

3.3.1 Reordering Packets. At the DstToR, the order of packets can be determined through the TAIL and REROUTED flags. Essentially, any REROUTED packets should only be forwarded to the destination after the TAIL is received and transmitted by the DstToR. For instance, if the ⑥ REROUTED packets arrived earlier than the ⑦ TAIL, they are held in a queue until the TAIL arrives. Once the TAIL arrives, the queue can then be resumed and flushed while a ⑧ CLEAR packet is sent to the SrcToR to signal that there are no out-of-order packets pending.

To explain the reordering process in detail, we illustrate this procedure in Fig. 9. Here, the default queue Q_0 has the lowest scheduling priority and is common for all flows. When the first REROUTED packets arrive before the TAIL (i.e., out-of-order), the DstToR assigns a queue (say Q_1) to this flow and the queue is paused to hold the REROUTED packets (see Fig. 9b). Upon reception of TAIL, ConWeave forwards the TAIL and then resumes Q_1. At the same time, REROUTED packets that arrive after TAIL are directed to Q_0 instead of Q_1. Packets in Q_0 will continue to be forwarded once Q_1 is completely flushed by the strict queue priority (see Fig. 9c).

Handling TAIL packet losses. In the event of losing the TAIL packet, the REROUTED out-of-order packets could remain in the queue indefinitely. To cope with the loss, we introduce a timer, T_{resume}. In the case of TAIL losses, the queue holding the packets would be resumed and flushed when T_{resume} expires. We depict this operation in Fig. 9d.

To determine T_{resume}, we estimate the expected arrival time of the TAIL. Note that the packet arrival time is its departure time plus the time in flight. We can approximate its expected arrival time at the DstToR using the OLD path's delay and the departure time of the TAIL at the SrcToR. Thus, all packets in ConWeave carry the TX_TSTAMP in the ConWeave header when transmitted by the SrcToR. In addition, the REROUTED packets include the TAIL_TX_TSTAMP to inform the DstToR on when the TAIL departed the SrcToR. Whenever the first REROUTED packet arrives, T_{resume} is initialized. Subsequently, the timer is updated by the packets arriving from the OLD path using the latest telemetry. We discuss the T_{resume} estimation algorithm in detail in Appendix §A.



(a) Forwarding in-order packets to a normal queue.

(b) Start enqueuing the first out-of-order packet.

(c) Resume reorder queue with priority queues.

(d) In case of TAIL loss, timer triggers resume.

Figure 9: ConWeave packet reordering. A single flow needs only one FIFO queue to reorder packets. ConWeave *DOES NOT* rely on the packet sequence numbers to reorder them. Instead, we differentiate the order of the packets through the REROUTED and TAIL flags.

3.4 Implementation

We implement ConWeave's data plane on an Intel Tofino2 programmable switch in ~2400 lines of P4_16 [15] code. The data plane consists of two key components, i.e., the rerouting module and the sorting module. The implementation is available under [4].

ConWeave packet headers: We depict the layout of ConWeave's packet headers in Fig. 10. To minimize overhead, ConWeave repurposes the reserved fields (which are not included in the invariant CRC computation) in the RDMA BTH header [14]. We use 8 bits to

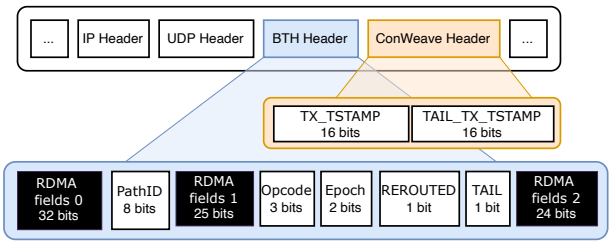


Figure 10: ConWeave header format (total 47 bits). We use the 15 reserved bits in the RDMA BTH header.

hold the `PathID` field which allows us to express up to 255 uplink paths in a 2-tier Clos topology in our prototype[5]. Next, the 3-bit `Opcode` field is used to differentiate between packets, i.e., normal, `RTT_REQUEST`, `RTT_REPLY`, `CLEAR`, and `NOTIFY`. The 2-bit `Epoch` field indicates the epoch of the packet [6]. Finally, the remaining 2 bits are allocated for the `REROUTED`, and `TAIL` flags, respectively. In addition, we include a separate header for ConWeave to carry the `TX_TSTAMP` – the time when the packet leaves the SrcToR, and `TAIL_TX_TSTAMP` – the time when the last `TAIL` has been sent by SrcToR, separately.

ConWeave packets: We refer to Table 2 for the ConWeave packets. ConWeave piggybacks information on existing packets to reduce overhead. More specifically, the DstToR mirrors the `RTT_REQUEST` received and modifies it before sending the modified packet back to the SrcToR as the `RTT_REPLY`. For `CLEAR`, we mirror and modify the `TAIL` or the T_{resume} timer packet at the DstToR. Finally, for `NOTIFY`, the DstToR mirrors and modifies the packet carrying the congestion signals (e.g., with the ECN bit marked) and then sends it to the SrcToR. Note that ConWeave control packets are always transmitted with the highest priority in the network and with payload truncation to ensure a low-latency feedback loop.

Timestamp resolution: Timestamps are used extensively in ConWeave. To minimize bandwidth and header overhead, we use only 16-bit timestamps in the ConWeave header (e.g., `TX_TSTAMP` and `TAIL_TX_TSTAMP`). With 16 bits, we can keep track of up to 32ms at 1us resolution. The most significant bit is used to keep track of potential wraparounds. We believe this is sufficient to handle the worst-case ToR-to-ToR path delay in data center networks.

3.4.1 Rerouting Module. To perform continuous RTT monitoring (§3.2.1), we make use of register arrays to store the timestamp when the last `RTT_REQUEST` was sent in the data plane. Every forwarded packet would check against the timestamp stored to determine whether the `RTT_REPLY` cutoff has been exceeded so that rerouting may be triggered. In addition, we maintain a set of states to keep track of rerouting status, e.g., timestamps to track connection status, current epoch, and whether the current path is rerouted or not.

For rerouting to happen (§3.2.2), there need to be available paths to select from. We keep track of the uplink path statuses using a 4-way associate hash table implemented using four register arrays, spanning across four pipeline stages. A packet would access all four registers to sample two paths and then decide whether to reroute or not.

3.4.2 Reordering Module. To reorder packets, we make use of the queue pause/resume feature on the Intel Tofino2 [5, 39] to hold the out-of-order packets, i.e., `REROUTED` packets that arrive before the `TAIL`. For each uplink (depending on the port link rate), we dedicate $N-1$ queues out of the N queues (e.g., 31 out of 32 queues for a 100G link). At any given time, reordering can be done for up to $M*(N-1)$ number of flows where M is the number of downlinks to servers. Later in §4.1.3, we show that only a fraction amount of the queues are needed.

To reorder packets, the flow first needs to be assigned an available queue to hold the out-of-order packets. Similarly, we make use of a 4-way associative hash table realized using 4 register arrays to perform a lookup for available queues. To deal with `TAIL` losses, we make use of individual resume timers for each queue. Since today's programmable switches lack timers, we realize the resume timer by mirroring the first out-of-order packet with payload truncation, then appending the specific connection information (e.g., connection ID, queue assigned) to the header before recirculating it. In every recirculation, it checks the associated timeout value.

Once the assigned queue is flushed, the entry in the hash table is then updated by the recirculated packet to mark the queue as available for other flows. We drop this recirculated packet whenever the respective queue is flushed. Note that Tofino2 supports 400Gbps recirculation bandwidth and one recirculation in ConWeave typically takes $\approx 1us$. Thus, there is no queuing delay in recirculation unless the total number of recirculated packets is over 1 BDP of recirculation loop (≈50KB) or the number of connections concurrently with reordering is over ~800, which is extremely rare.

3.4.3 Dataplane resource utilization. ConWeave uses stateful ALUs (SALUs) to maintain the connection states, e.g., timestamps, timers, path status, available queues and etc, in the data plane. In our prototype implemented on the Intel Tofino2, ConWeave requires ~22% of the total SRAM memory and uses ~44% of the available SALUs. The usage of other hardware resources (e.g., hash bits and VLIW instructions) is no more than ~15% of available resources on the switch. In the current prototype implementation, we did not integrate ConWeave with the reference data-center switch implementation, i.e., `switch.p4`. Instead, we applied our own L2/L3 switching and routing implementation to realize a multi-tier topology via network virtualization. Based on the current implementation, we believe that there is sufficient headroom for integration with other data-plane programs. In cases where there are more active connections than what ConWeave supports, ConWeave applies ECMP to the rest as a fallback while dynamically maintaining hardware states with hot and active connections [33, 57].

[5] In cases where an 8-bit field may not be sufficient to represent the available paths, source routing mechanisms (e.g., SRv6 [19]) should be used instead.

[6] Based on how ConWeave is designed, each connection typically has at most two epochs at any given time (i.e., at most 2 concurrent paths) whereas we assumed a failure-free case. However, in exceptional cases where packets arrive unexpectedly late (e.g., due to PFC deadlock [30]) or θ_{inactive} is set too small, a 2-bit epoch field may not be sufficient to handle epoch collisions resulting from bit wrap-around. Such occurrences are infrequent and do not break ConWeave. That said, this can be addressed by either increasing the number of bits in the epoch field or by comparing the departure timestamps of packets to allow older packets to bypass the ConWeave logic.

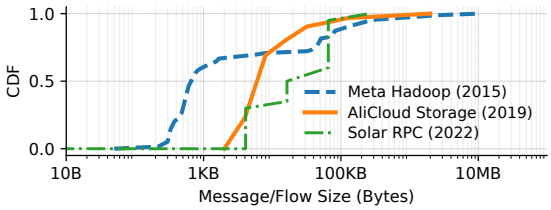


Figure 11: Traffic distribution from several applications in datacenters [40, 43, 53].

Parameter	Usage	Value
θ_{reply}	Timeout value for RTT_REPLY	8us
$\theta_{\text{path\_busy}}$	Duration to avoid using a congested path	8us
θ_{inactive}	Duration of flow inactivity to start a new epoch w/o out-of-order packets	300us

Table 3: Simulation parameters used for 2-tier topology (see §4.1).

4 EVALUATION

In this section, we perform software simulations using NS3 [6] and use a hardware testbed equipped with RNICs to evaluate the performance of ConWeave. Particularly, we seek to answer the following questions:

(1) How effective is the ConWeave's active congestion-evasive rerouting? We compare ConWeave's performance to existing state-of-the-art load balancing algorithms using both simulation and hardware testbed.
(2) What are the resource requirements of ConWeave in terms of buffer space and per-connection queues?
(3) What are the hardware resource and bandwidth consumption of ConWeave when implemented on a programmable switch such as the Tofino2 switch?

4.1 Software Simulations

We first present the setup for the simulation evaluation.

Network topologies: The topology in NS3 simulation is a Clos topology [8] with the over-subscription ratio of 2:1. By default, we use a leaf-spine topology which is common in data center clusters. The topology consists of 8×8 leaf-spine switches, and 128 servers (16 servers for each rack). All links are 100Gbps with $1us$ latency. For the switch model, we enable buffer sharing for flexible buffer allocation using a publicly available source code [41]. Each switch has a buffer size of 9MB.

Workloads: Fig. 11 shows several industry data center workloads available in the literature. We use the AliCloud storage [40] and Meta Hadoop [53] workloads in the simulation. The SolarRPC workload will be used in the hardware testbed evaluation. We schedule a flow by randomly selecting a pair of client and server and then select a flow size from the chosen flow size distribution. Inter-flow arrival times follow a Poisson distribution and the average flow arrival rate is used to control the overall traffic load intensity. Due to the space limitation, we will only show the results for AliCloud storage. The result for Meta Hadoop is shown in Appendix §B.

Transport: We use DCQCN [63] which is the standard congestion control scheme for commodity RDMA NICs [31, 49]. Since the recommendation in [63] does not fit into our setup due to a different scale, we find the parameters giving the low latency and high throughput, e.g., $(K_{min}, K_{max}, P_{max})$ = $(100KB, 400KB, 0.2)$ based on the observations in [40]. For the rest of the parameters, we follow the recommendations in the recent Mellanox driver/firmware [50].

Network flow controls: We implement two flow control mechanisms as follows:

- *Lossless RDMA* - Go-Back-N loss recovery and priority-based flow control (PFC).
- *IRN RDMA [44]* - Selective-Repeat for loss recovery and the end-to-end flow control that bounds the number of in-flight packets to 1 BDP (BDP-FC).

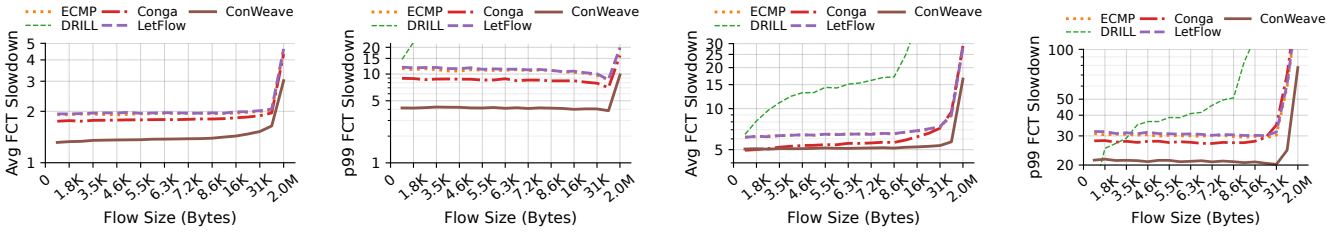
Schemes compared: We compare ConWeave with ECMP, Conga [11], Letflow [59], and DRILL [23]. For Conga and Letflow, we choose a flowlet time gap of 100 us. For DRILL, we use the recommended setting *DRILL(2,1)*, i.e., choosing a new output port with the smallest queue among 2 random samples and 1 current port. For ConWeave, the default parameters used are shown in Table 3. Specifically, θ_{reply} is a timeout value for RTT_REPLY. A smaller value allows more frequent rerouting, but too small a value may result in excessive rerouting. We present how we find the default value in appendix B.1. $\theta_{\text{path\_busy}}$ is the duration to avoid using the congested path after NOTIFY is received. The value of $\theta_{\text{path\_busy}}$ is chosen based on the ECN marking threshold. For instance, if the threshold is 100KB, then $\theta_{\text{path\_busy}}$ should correspond to the minimum time required to flush 100KB (e.g., $8us$ for a 100G link). Lastly, θ_{inactive} is the duration to start a new epoch based on an inactivity period. It must be long enough so that a new epoch starts without out-of-order packets. For instance, we used $300us$ for leaf-spine topology.

Performance metrics: As the primary metric, we use *FCT slowdown*, i.e., a flow's actual FCT normalized by the base FCT when the network has no other traffic. To measure the overhead and effectiveness of ConWeave, we record the usage of the number of reorder queues per egress port and the reorder queue memory usage per switch by sampling every $10us$ from all nodes.

4.1.1 Reduction in FCT. We run the simulations with 50% and 80% average traffic loads which represent a moderately and highly loaded network, respectively. In Fig. 12 and Fig. 13, we show the average and tail FCT slowdowns. In some instances, DRILL's performance figures are not included because the FCTs are too large to be included without substantial change in the scale.

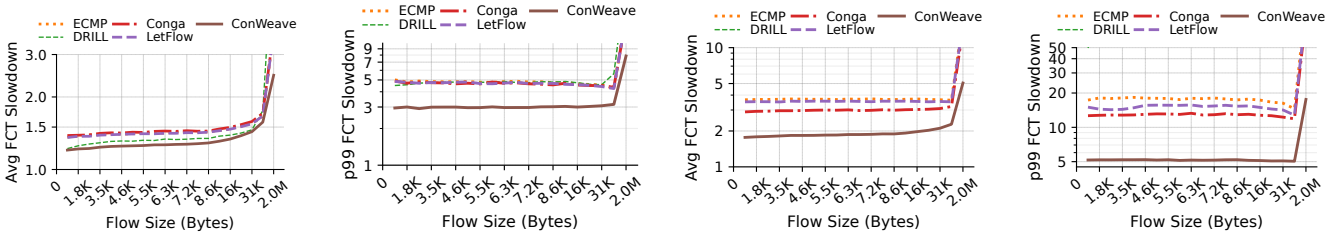
For moderate traffic loads (i.e., 50%), ConWeave improves the average and 99-percentile FCT slowdowns for overall flow sizes by at least 23.3%, 45.8% in lossless RDMA, and 12.7%, 46.2% in IRN RDMA when compared to others. On the other hand, in a highly loaded network (e.g., 80%), the average and tail FCTs improvement are at least 17.6%, 35.8% in lossless RDMA, and 42.3%, 66.8% in IRN RDMA. Our results show that ConWeave is effective in rerouting flows away from congested links and provides significant improvements over the baseline algorithms.

4.1.2 Load balancing efficiency. In this evaluation, we investigate ConWeave's load balancing efficiency. Fig. 14 shows the CDF of *throughput imbalance* [11] across the 8 uplinks for each ToR switch



(a) 50% Avg.Load (avg) (b) 50% Avg.Load (99%-ile) (c) 80% Avg.Load (avg) (d) 80% Avg.Load (99%-ile)

Figure 12: Avg. and tail FCT slowdown for *AliStorage* in *Lossless RDMA* (50% and 80% Avg.Load).



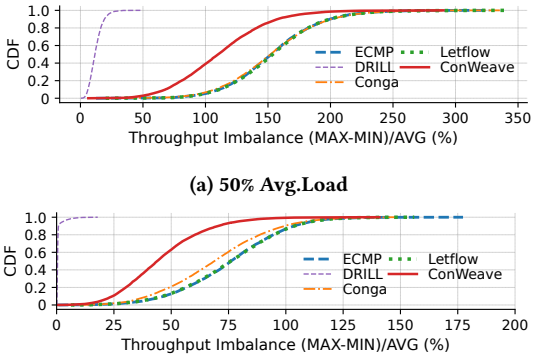
(a) 50% Avg.Load (avg) (b) 50% Avg.Load (99%-ile) (c) 80% Avg.Load (avg) (d) 80% Avg.Load (99%-ile)

Figure 13: Avg. and tail FCT slowdown for *AliStorage* in *IRN RDMA* (50% and 80% Avg.Load).

using 50% and 80% average load. The throughput imbalance is defined as the maximum throughput minus the minimum throughput divided by the average (among the uplinks). We calculate it using snapshots sampled every 100*us* from all nodes. From the result, we observe that except for DRILL, ConWeave is the most effective in terms of spreading the load to various links. Recall that DRILL performs per-packet switching resulting in a large amount of out-of-order packets. Hence, while it achieves good load balancing among the links, it has poor application performance over RDMA.

4.1.3 Hardware resource consumption. Fig. 15 shows the number of queues used per switch egress port. Most of the time, we observe that ConWeave only needs to support less than 10 queues for reordering regardless of the network loads. In the worst case, the number of queues needed does not exceed 15. Given that the number of queues available per egress port found on commodity programmable switches ranges from 32 up to 128 [24], this shows that ConWeave requires only a fraction of the queues for reordering.

Fig. 16 shows the total buffer memory usage per switch for packet reordering. In general, ConWeave in lossless RDMA consumes more buffer memory than IRN RDMA. This is because while the flow control (BDP-FC) in IRN limits the in-flight packets to one BDP, lossless RDMA can keep sending packets of a flow during its packet reordering process and thus consuming more buffer memory. Specifically, in lossless RDMA with 80% network load, the 99.9-percentile and maximum queue overhead is 1.5MB and 2.4MB, respectively. Even so, these numbers correspond to only a fraction of available buffer space on commodity switching ASICs which typically have tens of MBs [1, 5] of them. We discuss ConWeave's scalability and its alternative design options in §5.



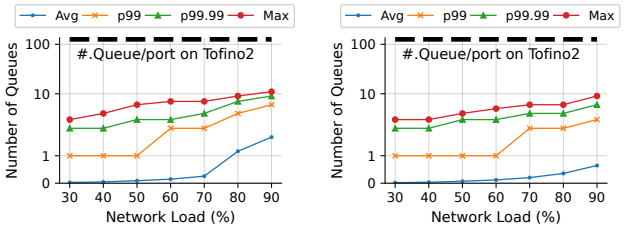
(a) 50% Avg.Load

(b) 80% Avg.Load

Figure 14: Load imbalance between throughput of ToR uplinks for 50% and 80% Avg.Load in *IRN RDMA*.

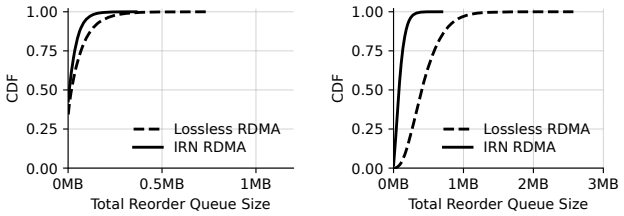
4.1.4 Three-tier Topology. So far, the evaluations have been performed with a two-tier (Clos) topology. In this section, we evaluate ConWeave on a three-tier topology (i.e., more hops). A 3-tier topology introduces more hops and thus potentially longer response time and more cross-traffic variation. We use a fat-tree topology [8] with its parameter $k = 8$ and the over-subscription ratio of 2:1, which involves 256 servers in total (8 servers for each rack), and the average network load is 60%. In lossless RDMA, we use 8*us* for θ_{reply}, 16*us* for $\theta_{\text{path\_busy}}$, and 1*ms* for θ_{inactive}.

We depict the results in Fig. 17. We find that across the lossless and IRN RDMA, ConWeave improves the average and 99-percentile FCT slowdowns by at least 21.4%, 40.8% for short (<1BDP) flows,



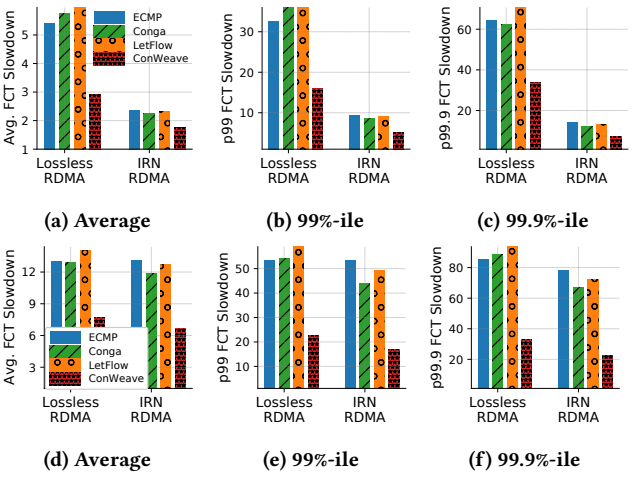
(a) *Lossless RDMA* (b) *IRN RDMA*

Figure 15: Number of queues usage per egress port. Note that the y-axis is in log scale while the dashed line on the top refers to the number of queues available per port.



(a) 50% Avg.Load (b) 80% Avg.Load

Figure 16: Total queue memory overhead per switch.



(a) Average (b) 99%-ile (c) 99.9%-ile

(d) Average (e) 99%-ile (f) 99.9%-ile

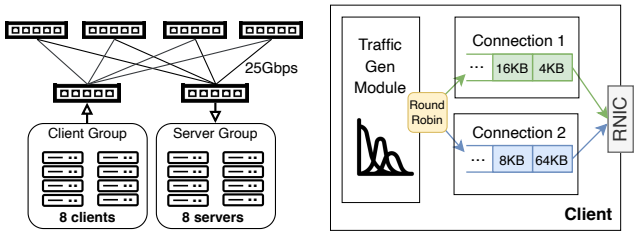
Figure 17: FCT slowdowns for short (a-c) and long (d-f) messages for fat-tree topology.

and 40.1%, 57.8% for long (>1BDP) flows, respectively. Overall, ConWeave outperforms the baseline load balancing mechanisms on the 3-tier topology.

4.2 Hardware Testbed Evaluations

Next, we evaluate our prototype for ConWeave on a hardware testbed consisting of one Tofino1 and Tofino2 switch, respectively.

Network topologies: Our testbed topology comprises of two leaves and four spine switches (see Fig. 18a). We realize the topology by virtualizing the Tofino1 switch as four spine switches while



(a) Testbed topology. (b) Traffic generation.

Figure 18: Leaf-spine topology in the testbed and a traffic generator at clients for each client-server pair.

the Tofino2 switch serves as the two leaf switches running ConWeave. All links operate at 25Gbps and the topology has an oversubscription ratio of 2:1. Each leaf switch is connected to a client/server group, where each group consists of 8 nodes that are connected with a single 25Gbps Mellanox RNIC (both CX5 and CX6 [48, 49]) port.

Workload: Each client-server pair maintains 2 persistent connections and sends `RDMA WRITE` following the SolarRPC workload (see Fig. 11).

Transport: We use DCQCN with the same parameters in NS3 simulation (§4.1).

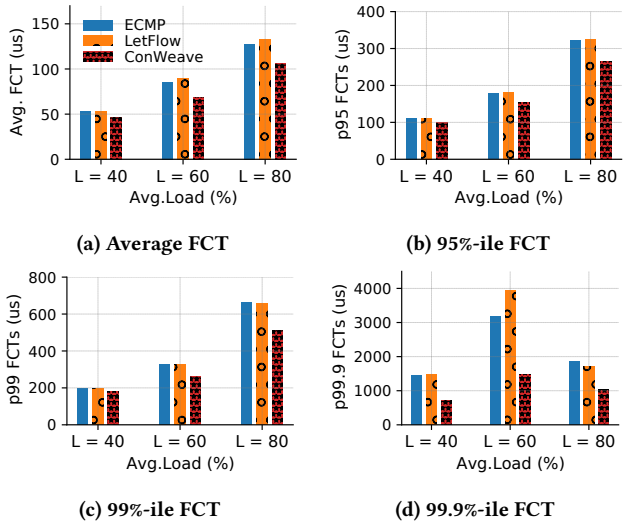
Network flow control: We configure our testbed for lossless RDMA. The switches and RNICs are configured with PFC enabled. Loss-recovery-wise, we only use Go-Back-N on the RNICs and configure the DCQCN ECN marking scheme using the parameters as per the simulations (§4.1). Selective Repeat is not used as it is not compatible with the CX5 RNICs.

Schemes compared: We compare ConWeave with ECMP and Letflow (100us). Given the lack of available P4 implementations for Conga and LetFlow, we only reimplement the more recent LetFlow in P4 on the Intel Tofino2 given the similar performance trends of Conga and LetFlow. We do not evaluate DRILL given its known poor performance on RDMA (§4.1). For ConWeave, we accordingly use $12us$ for θ_{reply}, $32us$ for $\theta_{\text{path\_busy}}$ (100KB flush time with 25G link), and $10ms$ for θ_{inactive} as we run on lossless RDMA.

Performance metrics: We evaluate the performance using the absolute FCTs measured in microseconds (*us*).

4.2.1 Application performance. In Fig. 19, we evaluate the impact of ConWeave on RDMA application performance by measuring their average and tail FCTs. We observe that ConWeave completes the flows at least 11%~23% faster than other schemes for diverse network loads between 40% to 80%. Specifically, we notice a significant improvement of the 99.9-percentile FCTs, i.e., from 39.67% up to 52.96%, for ConWeave when compared to its counterparts. Thus, the results prove that ConWeave effectively reroutes traffic by evading congested links to improve RDMA application performance.

4.2.2 ConWeave bandwidth usage. In the forward direction, ConWeave adds a 4-byte header to each RDMA data packet (see §3.4). In the reverse direction, the ConWeave DstToR sends control packets (e.g., `REPLY`, `FIN`, `FEEDBACK`) to the SrcToR. In Table 4, we present the bandwidth overhead of ConWeave in the reverse direction and



(a) Average FCT (b) 95%-ile FCT

(c) 99%-ile FCT (d) 99.9%-ile FCT

Figure 19: FCTs comparison for *Solar* workload in *Lossless RDMA* measured in the testbed.

Avg. Load	Average Bandwidth			
	RDMA DATA	RTT_REPLY	CLEAR	NOTIFY
20	22.01 Gbps	0.12 Gbps	0.01 Gbps	≈ 0.0 Gbps
50	55.44 Gbps	0.26 Gbps	0.04 Gbps	0.02 Gbps
80	84.67 Gbps	0.48 Gbps	0.16 Gbps	0.24 Gbps

Table 4: Bandwidth overhead of control packets compared to RDMA data bandwidth (RDMA DATA).

compare it with the RDMA forwarding bandwidth (DATA field in Table 4) measured at SrcToR uplinks of the client group. We observe the additional bandwidth overhead of ConWeave is a small fraction of the available bandwidth.

5 DISCUSSION AND FUTURE WORK

ConWeave on SmartNICs: While ConWeave is designed to exploit the capabilities of commodity programmable switches, our key insights can also be applied to recent SmartNICs. For instance, the switch logic of ConWeave can be implemented on the Nvidia BlueField DPUs [47] or the Intel E2000 IPUs [32], albeit with different trade-offs in resource usage, deployment cost, and complexity. Nevertheless, running ConWeave on ToR switches has the following advantages. First, it incurs less redundancy in path state maintenance as the ToR switch serves as a natural aggregation point for path monitoring, selection, and switching. Second, since the resources used for packet reordering on the ToR switch can be shared by all servers in the rack, this can lead to reduced resource usage due to statistical multiplexing.

Scaling to larger network: Switch hardware resource limitation presents a scalability challenge for ConWeave when the network expands to millions of servers. We note that hardware resource usage, such as the number of queues and the amount of on-chip memory, is proportional to the number of active flows that necessitate packet reordering at the destination ToR. The number of such flows, in turn, depends only on the number of servers per rack and applications per server. Neither number grows significantly with increasing network size. For example, in a fat-tree topology [8], the size of the network increases *cubic* to the number of servers per rack.

In the rare cases where switch hardware resources are exhausted, unresolved out-of-order packets can lead to performance degradation. To reduce the likelihood of resource exhaustion, one can consider either using *external switch memory* [36] such as host DRAM and SmartNICs to buffer packets temporarily, or applying *admission control* so that destination ToRs permit source ToRs to do rerouting only when there are spare resources. We leave these investigations for future work.

Interaction with congestion control: The primary focus of this work revolves around DCQCN, the *de-facto* standard transport protocol in commodity RNICs. In ConWeave design, DCQCN's ECN-based congestion marking offers two distinct advantages. Firstly, it ensures that the delay resulting from packet reordering is not erroneously attributed to network congestion. Secondly, the ECN threshold provides valuable insights into the minimal time required to alleviate congestion within the queue. On the other hand, ConWeave is also compatible with delay-based protocols such as Swift [37]. However, it is essential that any delay incurred due to packet reordering at the destination ToR switch should not be interpreted as a congestion signal in these cases.

Integrating with rate control: In its current prototype, while ConWeave takes the approach of avoiding congested paths by rapid and frequent rerouting, it does not take into account the effect on rate control. For instance, after it reroutes from a congested to an idle path, the congestion feedback from the previous path can still unnecessarily reduce the rate. It will be interesting to investigate how a predictable and scheduled load balancing mechanism like ConWeave can be co-designed with a rate control mechanism.

Incremental deployment: ConWeave's design allows operators to incrementally deploy ConWeave in data centers running alongside non-ConWeave ToR switches. For inter-rack communications that involve non-ConWeave ToR switches, the default ECMP is applied. The optimum partial deployment strategy for maximum benefits remains an area for further investigation in future research.

6 RELATED WORK

RDMA load balancing in data centers: There has been plenty of literature on addressing data center network load balancing in various granularities from per-flow to per-packet. Apart from the conventional Equal Cost Multi-Path (ECMP), existing works predominantly leverage *flowlets* [11, 35, 52, 59], a set of sub-streams of a flow stream divided by the inactive time gap, to proactively avoid out-of-order delivery. However, as highlighted previously, such an opportunistic design turns out to be not effective on RDMA given the fewer chances to reroute [42]. Per-packet rerouting (e.g., spraying [18] or DRILL [23]) may provide near-optimal load balancing if out-of-order delivery does not matter, but it incurs an enormous performance impact on RDMA. On the contrary, ConWeave masks the out-of-orders in the network while operating a load balancing at a fine granularity. Table 5 summarizes the existing literature and ConWeave in the context of RDMA.

Schemes	Method of congestion sensing	Minimum reroute granularity	Rerouting frequency in RDMA	Restoring packet orders before the receiving ends	Adverse effects by out-of-order pkts	Compatible with RNICs
ECMP [29]	Oblivious	Flow	Low	No need	Low	Yes
Presto [27]	Oblivious	Flowcell	High (for 64KB flowcells)	Reordering buffer at end-hosts	Low	No
Conga [11]	Global path utilization	Flowlet	Low (to ensure less out-of-order)	No	Low	Yes
Hula [35]	Global path utilization					
Letflow [59]	Obllivious					
DRILL [23]	Local queue utilization	Packet	High	No	High	Yes
Hermes [62]	Global congestion signaling	Packet	Low (too cautious)	No	Low	No
ConWeave	**Global congestion signaling**	**RTT**	**High**	**In-network reordering**	**Low**	**Yes**

Table 5: Summary of prior work for network load balancing.

Some works consider a multi-path transport design with end-host modifications. MP-RDMA [42] proposes a multi-path RDMA transport through custom-designed RNICs, or purely software-based implementation [58]. Moreover, it may not be compatible with the legacy RNICs and thus it cannot be easily deployed in data centers. In contrast, ConWeave is complementary to existing routing protocols and operates on current commodity programmable switches and RNICs.

Packet reordering on programmable switches: With the emergence of data plane programmability, efforts have been made to fully/partially offload functions at the end hosts to the switching hardware for performance acceleration. The packet reordering (or, sorting) function in the programmable switch has been explored in the context of packet scheduling. For instance, many queue abstraction designs have been proposed to flexibly express a variety of scheduling algorithms and to be efficiently implemented on programmable switches [10, 54–56, 61]. However, their primitives are substantially more expensive to support as their requirements to reorder packets for a per-flow basis on hardware are also more complex. ConWeave's requirement is simpler and is designed to satisfy the packet reordering need in the context of load balancing.

Offloading packet reordering to application/transport layer: Instead of avoiding packet out-of-order in the network, some works [22, 26, 27, 45] implement a dedicated reordering buffer on the transport or application layer. However, these approaches are either too complex to be implemented on commodity RNIC hardware or incur a significant overhead on the CPU negating the benefits of using RDMA. Furthermore, they predominantly rely on congestion-oblivious rerouting (i.e., packet spraying) whose susceptibility to network asymmetry is well-known by previous studies [59, 62].

Load balancing in HPC: The emergence of AI/ML applications has led to a strong emphasis within the high-performance computing (HPC) community on achieving optimal (RDMA) network load balancing. Concurrent with ConWeave, leading companies in the field of HPC, including Cisco [3], NVIDIA [7], and Broadcom [2], have developed proprietary systems that incorporate per-packet rerouting and packet reordering capabilities in their proprietary switches and/or DPUs integrated into SmartNICs. ConWeave shares the same goal on RDMA load balancing with these systems and closely aligns itself with this emerging industrial trend. ConWeave's design and its publicly available implementation can serve as a possible benchmark for further research on RDMA load-balancing using commodity programmable hardware.

7 CONCLUSION

In this paper, we show that existing load-balancing schemes do not work for RDMA traffic because of the lack of sufficiently large flowlet gaps and RDMA's performance degradation in the face of out-of-order packets. To tackle RDMA's intolerance for out-of-order packets, we first design an in-network packet reordering scheme that resolves out-of-order packets before delivering them to an RDMA receiver. We then present ConWeave, a load balancer design that performs fine-grained load-balancing of RDMA flows such that the out-of-order packets could be reordered by the in-network reordering mechanism. Through software simulations and hardware testbed evaluations, we show that ConWeave consistently outperforms existing designs. By also highlighting the need for developing load balancing algorithms specifically designed for RDMA traffic, we believe that ConWeave opens up a new chapter on load balancing for RDMA in datacenter networks.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers and our shepherd Yu Hua for their invaluable feedback. This research is supported by the Singapore Ministry of Education Academic Research Fund Tier 2 (Grant Number: MOE2019-T2-2-134).

Ethics statement: This work does not raise any ethical issues.

REFERENCES

[1] 2017. Broadcom Trident 3. https://packetpushers.net/broadcom-trident3-programmable-varied-volume/.
[2] 2023. Broadcom Jericho3-AI. https://www.broadcom.com/company/news/product-releases/61156 [Accessed: May 2023].
[3] 2023. Cisco Silicon One. https://blogs.cisco.com/sp/building-ai-ml-networks-with-cisco-silicon-one [Accessed: May 2023].
[4] 2023. ConWeave repository. https://github.com/conweave-project.
[5] 2023. Intel Tofino 2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.
[6] 2023. Network Simulator 3 (NS-3). https://www.nsnam.org/.
[7] 2023. NVIDIA Spectrum X. https://nvdam.widen.net/s/6lmkmc8lqg/nvidia-spectrum-x-whitepaper [Accessed: May 2023].
[8] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. *ACM SIGCOMM CCR* (2008).
[9] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. 2010. Hedera: dynamic flow scheduling for data center networks.. In *Proceedings of NSDI*.
[10] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. 2020. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues. In *Proceedings of NSDI*.
[11] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. 2014. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proceedings of SIGCOMM*.
[12] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center Tcp (DCTCP). In *Proceedings of SIGCOMM*.

[13] Mark Allman, Hari Balakrishnan, and Sally Floyd. 2001. *Enhancing TCP's loss recovery using limited transmit.* Technical Report.

[14] InfiniBand Trade Association. 2020. InfiniBand Architecture Specification Release 1.4 Annex A17: RoCEv2.

[15] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR* (2014).

[16] Yuchung Cheng, Neal Cardwell, Nandita Dukkipati, and Priyaranjan Jha. 2021. The RACK-TLP Loss Detection Algorithm for TCP. *RFC 8985* (2021).

[17] Alibaba Cloud. 2023. Shared Memory Communications over RDMA (SMC-R). https://www.alibabacloud.com/help/en/elastic-compute-service/latest/smc-r.

[18] Advait Dixit, Pawan Prakash, Y Charlie Hu, and Ramana Rao Kompella. 2013. On the impact of packet spraying in data center networks. In *Proceedings of INFOCOM.*

[19] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Bruno Decraene, Stephane Litkowski, and Rob Shakir. 2018. *Segment routing architecture.* Technical Report.

[20] Doug Freimuth, Elbert Hu, Jason LaVoie, Ronald Mraz, Erich Nahum, and John Tracey. 2006. *Evaluating Batching for TCP Offload.* Technical Report. Technical report, IBM, IBM TJ Watson Research Center.

[21] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. 2021. When Cloud Storage Meets RDMA. In *Proceedings of NSDI.*

[22] Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, and Mohammad Alizadeh. 2016. Juggler: a practical reordering resilient network stack for datacenters. In *Proceedings of EuroSys.*

[23] Soudeh Ghorbani, Zibin Yang, P Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. 2017. DRILL: Micro load balancing for low-latency data center networks. In *Proceedings of SIGCOMM.*

[24] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E Anderson. 2022. Backpressure Flow Control. In *Proceedings of NSDI.*

[25] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *Proceedings of SIGCOMM.*

[26] 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 *Proceedings of SIGCOMM.*

[27] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. 2015. Presto: Edge-based load balancing for fast datacenter networks. *ACM SIGCOMM CCR* (2015).

[28] Torsten Hoefler, Duncan Roweth, Keith Underwood, Robert Alverson, Mark Griswold, Vahid Tabatabaee, Mohan Kalkunte, Surendra Anubolu, Siyuan Shen, Moray McLaren, et al. 2023. Data Center Ethernet and Remote Direct Memory Access: Issues at Hyperscale. *Computer* (2023).

[29] Christian Hopps. 2000. *Analysis of an equal-cost multi-path algorithm.* Technical Report.

[30] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. 2016. Deadlocks in datacenter networks: Why do they form, and how to avoid them. In *Proceedings of HotNets.*

[31] Intel. 2023. Intel Ethernet 800 Series. https://www.intel.sg/content/www/xa/en/design/products-and-solutions/networking-and-io/ethernet-800-series/data-transfer-with-rdma-video.html.

[32] Intel. 2023. Intel® Infrastructure Processing Unit (Intel® IPU) ASIC E2000. https://www.intel.sg/content/www/xa/en/products/details/network-io/ipu/e2000-asic.html.

[33] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of SOSP.*

[34] Rishi Kapoor, Alex C Snoeren, Geoffrey M Voelker, and George Porter. 2013. Bullet trains: a study of NIC burst behavior at microsecond timescales. In *Proceedings of CoNEXT.*

[35] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. Hula: Scalable load balancing using programmable data planes. In *Proceedings of SOSR.*

[36] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. 2018. Generic external memory for switch data planes. In *Proceedings of HotNets.*

[37] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. 2020. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of SIGCOMM.*

[38] Petr Lapukhov, Ariff Premji, and Jon Mitchell. 2016. *Use of BGP for routing in large-scale data centers (RFC7938).* Technical Report.

[39] Jeongkeun Lee. 2020. Advanced Congestion & Flow Control with Programmable Switches. https://opennetworking.org/wp-content/uploads/2020/04/JK-Lee-Slide-Deck.pdf.

[40] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: High precision congestion control. In *Proceedings of SIGCOMM.*

[41] Hwijoon Lim, Wei Bai, Yibo Zhu, Youngmok Jung, and Dongsu Han. 2021. Towards timeout-less transport in commodity datacenter networks. In *Proceedings of EuroSys.*

[42] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. 2018. Multi-Path Transport for RDMA in Datacenters. In *Proceedings of NSDI.*

[43] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. 2022. From luna to solar: the evolutions of the compute-to-storage networks in Alibaba cloud. In *Proceedings of SIGCOMM.*

[44] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting network support for RDMA. In *Proceedings of SIGCOMM.*

[45] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of SIGCOMM.*

[46] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading.. In *Proceedings of NSDI.*

[47] NVIDIA. 2023. Mellanox BlueField2 DPU SmartNICs. https://store.mellanox.com/categories/dpu.html [Accessed: Jan 2023].

[48] NVIDIA. 2023. Mellanox Connect X-5. https://www.nvidia.com/en-sg/networking/ethernet/connectx-5.

[49] NVIDIA. 2023. Mellanox Connect X-6. https://www.nvidia.com/en-sg/networking/ethernet/connectx-6.

[50] NVIDIA. 2023. Mellanox Firmware XX.35.2000. https://network.nvidia.com/support/firmware/mlxup-mft/.

[51] Gregory F Pfister. 2001. An introduction to the infiniband architecture. *High performance mass storage and parallel I/O* (2001).

[52] Mubashir Adnan Qureshi, Yuchung Cheng, Qianwen Yin, Qiaobin Fu, Gautam Kumar, Masoud Moshref, Junhua Yan, Van Jacobson, David Wetherall, and Abdul Kabbani. 2022. PLB: congestion signals are simple and effective for network load balancing. In *Proceedings of NSDI.*

[53] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. 2015. Inside the social network's datacenter network. In *Proceedings of SIGCOMM.*

[54] Naveen Kr Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating fair queueing on reconfigurable switches. In *Proceedings of NSDI.*

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

[56] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable packet scheduling at line rate. In *Proceedings of SIGCOMM.*

[57] Cha Hwan Song, Xin Zhe Khooi, Dinil Mon Divakaran, and Mun Choon Chan. 2023. DySO: Enhancing application offload efficiency on programmable switches. *Computer Networks* (2023).

[58] Feng Tian, Yang Zhang, Wei Ye, Cheng Jin, Ziyan Wu, and Zhi-Li Zhang. 2021. Accelerating Distributed Deep Learning using Multi-Path RDMA in Data Center Networks. In *Proceedings of SOSR.*

[59] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *Proceedings of NSDI.*

[60] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, et al. 2023. SRNIC: A Scalable Architecture for RDMA NICs. In *Proceedings of NSDI.*

[61] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. 2021. Programmable packet scheduling with a single queue. In *Proceedings of SIGCOMM.*

[62] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. 2017. Resilient datacenter load balancing in the wild. In *Proceedings of SIGCOMM.*

[63] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. *ACM SIGCOMM CCR* (2015).

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

A QUEUE RESUME TIMER, T_{resume}

Recall that the TAIL demarcates the packets that are *before* and *after* rerouting at the SrcToR (see §3). A paused queue is used to hold out-of-order packets and the arrival of the TAIL packet flushes it. However, if the TAIL is lost, the out-of-order packets would be stuck indefinitely. To solve this, we employ a timer, T_{resume}, that flushes the paused queue in the event of TAIL losses.

Intuitively, T_{resume} should be set as the *expected arrival time* of the TAIL which can be inferred from the OLD path's delay, i.e., the departure time of TAIL plus the estimated path delay. Unfortunately, exactly predicting the path delay is challenging given the ever-changing network conditions, e.g., due to congestion. To address this challenge, we continuously update the estimation of T_{resume} using up-to-date telemetry (i.e., path delay) extracted from every packets (sent before the TAIL) arriving from the OLD path. The fields, TX_TSTAMP and TAIL_TX_TSTAMP (see §3.4), carried by the packets are used to perform the estimation of T_{resume}.

Using Fig. 20, we illustrate how T_{resume} is derived in the event of packets arriving out-of-order. Here, ① packet A and ② packet B are sent through a congested path. As no RTT_REPLY arrives back at the SrcToR within the cutoff time, the rerouting mechanism is thus triggered. A ③ TAIL packet is sent through the OLD path before rerouting the ④ subsequent packets (marked as REROUTED) through the NEW path. Note that our method requires no time-synchronization between the ToR switches.

Initializing T_{resume} when the first REROUTED packet arrives: The DstToR keeps track of the most recent packet's departure time from the SrcToR and the arrival time at the DstToR. For the case of ⑤ packet A, we denote the time of transmission at SrcToR as t_A^{TX} and the arrival time at SrcToR as t_A^{RX}. This information will be used in the estimation of T_{resume}. If ⑥ REROUTED packets arrive earlier than TAIL, they are out-of-order and a paused queue is allocated to buffer the packets. For the first out-of-order packet, we initialize the queue resume timer T_{resume}. The estimation of T_{resume} comes in two steps. First, we calculate the *time difference*, t_{DIFF}, between when packet A and the TAIL is transmitted, i.e., $t_{DIFF} = t_{TAIL}^{TX} - t_A^{TX}$. Assuming both packets A and TAIL experienced the same path delay, TAIL should arrive in t_{DIFF} after t_A^{RX}. Thus, T_{resume} can be estimated using the packet before REROUTED, in this case packet A's t_A^{RX}, as reference, in the following manner:

$$T_{\text{resume}} \approx \underbrace{t_A^{RX}}_{\text{pkt A's arrival time}} + \underbrace{(t_{TAIL}^{TX} - t_A^{TX})}_{t_{DIFF} \text{ between TAIL and pkt A}}$$

In the case where there are no prior packets that arrive before REROUTED, we initialize T_{resume} as follows:

$$T_{\text{resume}} \approx \underbrace{t_{RR}^{RX}}_{\text{first REROUTED's arrival time}} + \theta_{\text{resume\_default}}$$

Note that $\theta_{\text{resume\_default}}$ must be long enough to wait for the following packets through the old path (e.g., ~ 200us for leaf-spine topology with IRN RDMA and ~ 600us for PFC-enabled fat-tree topology in §4.1).

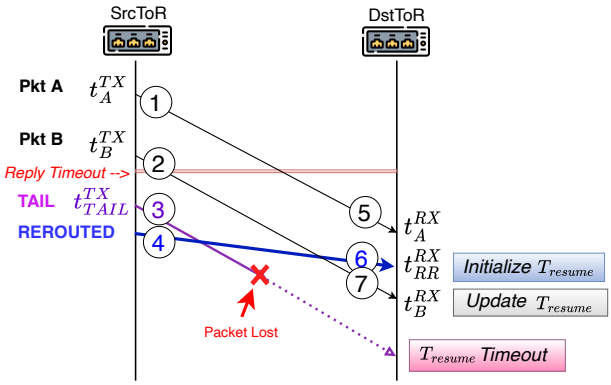


Figure 20: Estimation of queue resume time T_{resume}. We highlight no need for time-synchronization between switches.

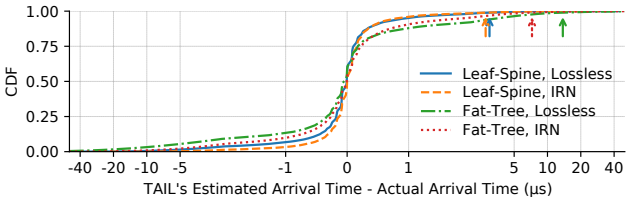


Figure 21: CDF of T_{resume} estimation error before introducing $\theta_{\text{resume\_extra}}$ for different topologies and network flow controls. We use 60% average network load. The plus value implies a haste queue flush. The arrows indicate 99-percentile.

Updating T_{resume} estimations using packets arriving from the OLD path: When ⑦ subsequent packets from the OLD path arrive at DstToR after the first REROUTED packet, T_{resume} is re-estimated. We denote packet B's transmission time at SrcToR and arrival time at DstToR as t_B^{TX} and t_B^{RX}, respectively. Similar to initialization, we update T_{resume} using the latest information of packet B. For instance, when packet B arrives at t_B^{RX}, T_{resume} is updated as follows:

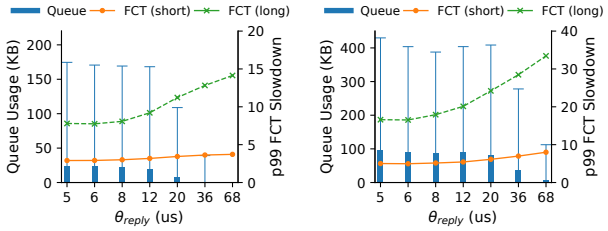
$$T_{\text{resume}} \approx t_B^{RX} + (t_{TAIL}^{TX} - t_B^{TX})$$

Note that this procedure applies to packets from the OLD path that arrives after the first REROUTED packet until TAIL arrives, or when T_{resume} expires.

Extra term to deal with network uncertainty: However, given the dynamic network conditions due to the wild variation of network uncertainty by congestion or PFC at small timescales, the path delay experienced by packet B and TAIL can hardly be exactly the same. This can potentially lead to pre-mature queue flushes prior to the arrival of the TAIL. Therefore, we add a small extra term, $\theta_{\text{resume\_extra}}$, to T_{resume}. Finally, the revised T_{resume} estimation at t_B^{RX} can be denoted as:

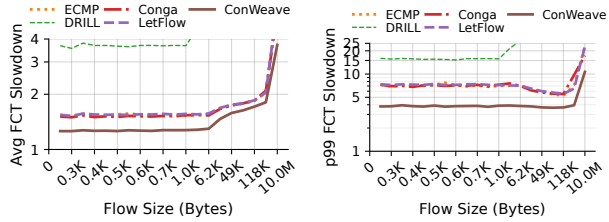
$$T_{\text{resume}} \approx t_B^{RX} + (t_{TAIL}^{TX} - t_B^{TX}) + \theta_{\text{resume\_extra}}$$

Setting the "right" value for $\theta_{\text{resume\_extra}}$: Through simulations, we study how T_{resume} (without $\theta_{\text{resume\_extra}}$) differs from the actual arrival time of the TAIL. Empirically, we find that setting $\theta_{\text{resume\_extra}}$ to 3.0us Leaf-Spine (Lossless), 2.7us Leaf-Spine

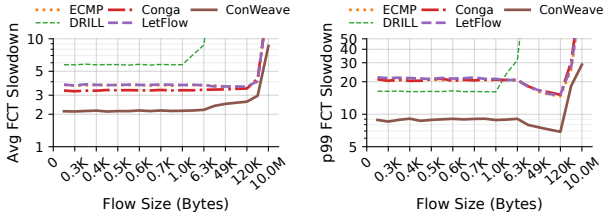


(a) 50% Avg.Load (b) 80% Avg.Load

Figure 22: Avg/99%-ile per-switch reorder queue usage and 99%-ile FCT slowdowns for diverse θ_{reply} in *IRN RDMA*. The smaller parameter makes a finer-grained rerouting.



(a) 50% Avg.Load (avg) (b) 50% Avg.Load (99%-ile)



(c) 80% Avg.Load (avg) (d) 80% Avg.Load (99%-ile)

Figure 23: Avg. and 99% tail FCT slowdown for *Meta Hadoop* in *Lossless RDMA* (50% and 80% avg.load).

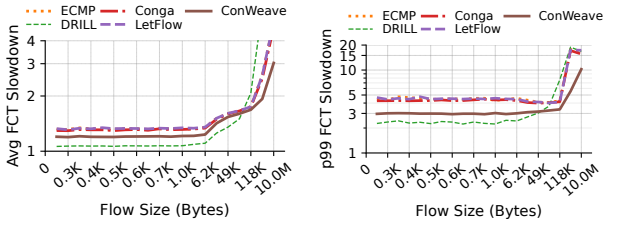
(IRN), 13.7*us* Fat-Tree (Lossless), 7.2*us* Fat-Tree (IRN) is sufficient for 99% of the time. In our evaluation, we used a small value (e.g., 16*us*) in IRN RDMA for a fast loss recovery. Since networks become dynamic by PFC pausing in Lossless RDMA, a large value is used (e.g., 64*us*) owing to extremely rare packet loss.

B SUPPLEMENTARY OF EVALUATIONS

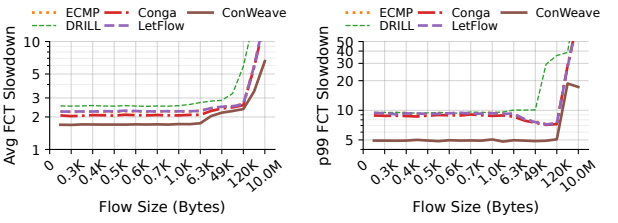
In this section, we supplement the evaluation results. In §B.1, we present how we tune the parameter θ_{reply}. After that, we present the simulation results using *Meta Hadoop* workload in §B.2. We use the same evaluation setup as in §4.1.

B.1 ConWeave parameter tuning

ConWeave's rerouting granularity is proportional to the parameter θ_{reply}, i.e., a smaller value produces more frequent rerouting. A finer-grained rerouting would improve the performance, but it also increases the reordering overhead such as queue consumption. In Figure 22, we show the average/99-percentile queue memory usage for reordering and 99-percentile FCT slowdown by varying θ_{reply} from 5*us* to 68*us*, where the ToR-to-ToR basis propagation delay is 4*us* (4 hops and 1*us* delay per link). We observe that a smaller

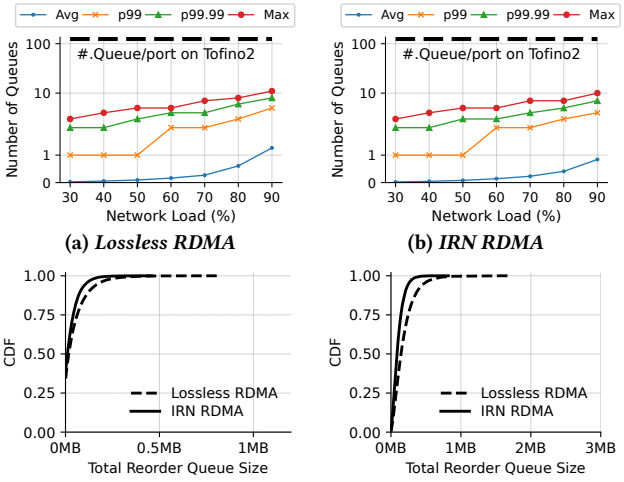


(a) 50% Avg.Load (avg) (b) 50% Avg.Load (99%-ile)



(c) 80% Avg.Load (avg) (d) 80% Avg.Load (99%-ile)

Figure 24: Avg. and 99% tail FCT slowdown for *Meta Hadoop* in *IRN RDMA* (50% and 80% avg.load).



(a) *Lossless RDMA* (b) *IRN RDMA*

(c) 50% Avg.Load (d) 80% Avg.Load

Figure 25: Per-port number of queues (a-b) and per-switch queue memory usage (c-d) for *Meta Hadoop* workload.

θ_{reply} provides a lower FCT slowdown with more queue memory usage for reordering, but the performance starts to decrease when θ_{reply} is over 8*us*. This explains our choice of θ_{reply}, 8*us*.

B.2 FCT slowdowns using *Meta Hadoop* workload

From Figure 23 to 24, we present the NS3 simulation results for Meta Hadoop workload. Similar to the Alibaba cloud workload, ConWeave significantly improves the FCT slowdowns. Specifically, when the traffic load is 80% on average, ConWeave achieves 40.7% and 59.4% improvement in lossless RDMA and 28.6% and 56.3% improvement in IRN RDMA for the average and 99-percentile FCT slowdown to all other schemes.

B.3 Queue usage

Figure 25 shows the queue usage of ConWeave for Meta Hadoop workload. Similar to our observation from the AliCloud workload, we see that the number of queue usage is always less than 12 and the queue memory usage per switch is under 2MB for both lossless and IRN RDMA.