Understanding the Micro-Behaviors of Hardware Offloaded Network Stacks with Lumina

Zhuolong Yu*
Johns Hopkins University

Bowen Su
Peking University

Wei Bai
Microsoft

Shachar Raindel†
Microsoft

Vladimir Braverman
Rice University

Xin Jin
Peking University

ABSTRACT

Hardware offloaded network stacks are widely adopted in modern datacenters to meet the demand for high throughput, ultra-low latency and low CPU overhead. To fully leverage their exceptional performance, users need to have a deep understanding of their behaviors. Despite many efforts on testing software network stacks, hardware network stacks impose unique challenges to testing tools due to their kernel bypass nature and high performance.

In this paper, we present Lumina, a tool to test the correctness and performance of hardware network stacks. Lumina leverages network programmability to emulate various network scenarios at line rate. With user-friendly interfaces, Lumina enables developers to inject deterministic events, thus facilitating the development of precise and reproducible tests. Given the limited resource and flexibility of programmable network devices, we mirror all the packets to dedicated servers and dump them for offline analysis. We leverage Lumina to test four RDMA NICs from NVIDIA and Intel, and identify bugs that can significantly degrade performance or mislead network operations. Lumina also enables us to capture unexpected micro-behaviors which are missing or not clearly described in public documents and specifications. Vendors have confirmed the critical bugs we discovered and will include bug fixes in future releases.

CCS CONCEPTS

• **Networks → Programmable networks**; **In-network processing**; **Network performance analysis**; **Network measurement**.

KEYWORDS

Network testing, Hardware offloaded network stack, Programmable networking, RDMA, Event injection

ACM Reference Format:
Zhuolong Yu, Bowen Su, Wei Bai, Shachar Raindel, Vladimir Braverman, and Xin Jin. 2023. Understanding the Micro-Behaviors of Hardware Offloaded Network Stacks with Lumina. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3603269.3604837

*Part of this work was done when Zhuolong did an internship at Microsoft.
†Shachar is now with Google. This work was done while he was at Microsoft.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission
and/or a fee. Request permissions from permissions@acm.org.
ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 979-8-4007-0236-5/23/09. . . $15.00
https://doi.org/10.1145/3603269.3604837

1 INTRODUCTION

Modern cloud applications have increasingly demanding requirements, seeking high throughput and ultra-low processing latency, while minimizing CPU overhead. Unfortunately, the legacy TCP/IP stack in operating system (OS) kernels are not optimized to meet these specific demands. As a result, cloud providers tend to offload network stacks into network interface cards (NICs) or data processing units (DPUs) to enhance performance and free up valuable CPU cycles. Hardware offloaded networking techniques, such as Remote Direct Memory Access (RDMA), Scalable Reliable Datagram (SRD) [46] and SOLAR [38], have been widely deployed to empower various workloads, including cloud storage [14, 20, 38], high performance computing (HPC) and machine learning (ML) [11, 12, 46, 50].

To fully harness the exceptional performance benefits of hardware offloading, network operators must possess a deep understanding of hardware network stack behaviors. Presently, the common practice involves running synthetic and production workloads on testbeds and test clusters to measure end-to-end performance. While this approach can detect significant bugs that directly disrupt end-to-end workflows, it may fall short in capturing functional bugs related to congestion control, quality of service (QoS), and loss recovery. Additionally, this method is susceptible to the interference of applications and varying network conditions, leading to challenges in reliably reproducing end-to-end performance anomalies. As a result, network operators may face difficulties in effectively localizing the root causes of these issues.

There are various testing tools [16, 18, 21, 41] designed specifically for software network stack implementations. These tools offer the advantage of flexible interaction with software network stack implementations, allowing accurate capture of micro-behaviors and enabling users to conduct precise and reproducible tests. However, when it comes to testing hardware network stacks, two significant challenges arise. Firstly, hardware network stacks are directly implemented in the hardware and bypass the OS kernel, making it exceptionally difficult to directly inject events or measure behaviors on the end hosts. Consequently, conventional solutions like packetdrill [16], which rely on a shim layer for event injection, are not suitable for testing hardware network stacks. Secondly, hardware network stacks are capable of delivering much higher throughput and lower latency. As a result, any testing tool used must be able to interact with these stacks at line rate, with minimal additional delay, to accurately evaluate their performance.

In this paper, we introduce Lumina, a tool designed to thoroughly test the correctness and performance of hardware network stacks, with a primary focus on RDMA initially. Lumina employs an in-network solution, directly interacting with hardware network stack implementations. This is accomplished by connecting two hosts, each equipped with the hardware network stack under test, to a programmable switch, which plays a key role in event injection. However, due to the inherent limitations of the programmable switch, such as constrained on-chip memory and processing cycles, performing complex measurement and analysis tasks entirely on the data plane is challenging. To overcome this obstacle, we leverage the packet mirroring feature to create duplicates of all packets and forward them to dedicated servers. These servers serve as repositories for all the mirrored packets, allowing for offline analysis of the network behavior.

Lumina aims at enabling users to write precise and reproducible tests in a *user-friendly* manner. We co-design three main components: event injector (on a programmable switch), traffic generator (traffic servers equipped with the hardware network stack under test), and packet dumper (on mirror servers). Before initiating traffic, the event injector leverages metadata shared by the traffic generator, e.g., queue pair number and packet sequence number, to translate users' intents, e.g., drop the second packet of the first queue pair, into deterministic events to inject. During the mirroring, Lumina embeds essential data plane metadata, such as mirror sequence number, timestamp, and event type, into the mirrored packets. This ensures the ability to reconstruct a correct and complete network trace during offline analysis. To enable high-speed traffic dumping, Lumina adopts several optimizations, including per-packet load balancing across CPU cores and packet trimming. Additionally, a test suite is built, featuring a set of built-in analyzers for common features, along with a fuzzing-based module [49]. The fuzzing-based module automatically generates test cases to search for potential bugs, enhancing the tool's testing capabilities.

We build a prototype of Lumina using Intel Tofino switch and test four widely deployed RDMA NICs (RNICs) from NVIDIA and Intel: NVIDIA ConnectX-4 Lx, ConnectX-5, ConnectX-6 Dx Dx and Intel E810. With Lumina, we gain invaluable visibility into the micro-behaviors of packet retransmissions, a crucial aspect in lossy RDMA deployments. Through extensive testing with Lumina, we identify a few critical bugs that can lead to significant performance degradation or provide misleading information to operations. For example, we find that NVIDIA ConnectX-6 Dx's Enhanced Transmission Selection (ETS) packet scheduler fails to achieve work conservation as required by the specification [30], which can cause considerable throughput loss in production with multiple traffic classes. Furthermore, Lumina enables us to capture a set of unexpected micro-behaviors which are missing or not clearly described in vendors' documents and specifications. We have reported all of our findings to NVIDIA and Intel, and have been working closely with them in the debugging process. The vendors have confirmed the critical bugs and will include bug fixes in future releases.

With hardware network stacks becoming increasingly complex and widely adopted, we believe our community needs a comprehensive suite of testing tools and an ImageNet-like [19] benchmark to systematically assess their performance and correctness. Through the introduction of Lumina, we aim to shed light on this critical area, complementing other research efforts [17, 28, 29, 53]. The code of Lumina is publicly available at https://lumina-test.github.io/.

2 BACKGROUND AND MOTIVATION

Over the past few years, there has been a notable surge in the widespread adoption of hardware offloaded network stacks, including RDMA, SRD [46], and SOLAR [38], within cloud environments. These technologies have been used to boost performance and cost-effectiveness across various applications, such as storage [14, 20, 38], HPC, and ML [12, 46, 50]. A recent study [14] indicates that approximately 70% of the traffic in Azure is RDMA.

Given this trend, it is important for network operators to have a deep understanding of hardware network stacks' behaviors. However, the current practice of running synthetic tests (e.g., using tools like `perftest` [10]) and production workloads on testbeds and test clusters may not be sufficient to uncover all potential issues. While this approach can detect significant bugs that directly disrupt end-to-end workloads, such as failures in recovering lost packets, long-time hardware pipeline stalls, or kernel panics due to hardware bugs, it may not effectively capture functional bugs related to congestion control, Quality of Service (QoS), and loss recovery. Additionally, the presence of various applications and network conditions in the test environment can introduce noise and make it challenging to reproduce issues reliably, hindering the collection of detailed information and micro-behaviors needed to identify root causes. As a result, bugs in critical areas may go unnoticed during testing and only surface in production networks. The magnitude of these problems is likely to escalate as link speeds continue to increase and hardware network stacks become even more complex

To illustrate the problem further, we consider the performance of NVIDIA ConnectX-4 RDMA NIC over lossy networks as an example. Shpiner et al. evaluated its performance over a lossy testbed network, and found it could preserve high goodput under synthetic incast workloads (Figure 4 and 5 in [47]). Hence, the authors concluded that the ConnectX-4 could provide solid performance even in the presence of packet drops. However, our investigation with Lumina reveals a different perspective. We find that the retransmission delay of this NIC is actually around 200 microseconds (Figure 8 and 9), which translates to approximately 100 base round-trip times (RTTs). This retransmission delay can significantly impact the overall performance and latency in lossy networks. Moreover, we find that this NIC has a "noisy neighbor" problem. When multiple connections experience packet drops simultaneously, the entire NIC pipeline can stall, leading to the discarding of packets from other connections that are not involved in the packet drops (Figure 11).

Considering these limitations, it is essential to develop a tool that empowers developers to effortlessly create accurate and reproducible tests for hardware network stacks. To achieve this objective, the tool must effectively interface with hardware network stacks, allowing for versatile and predictable interactions, such as loss injection. Additionally, it should precisely capture micro-behaviors, such as packet transmission time, and crucial information like counters.

We have observed several testing tools [16, 18, 21, 41] with similar objectives, but they primarily focus on testing software network stacks. However, when it comes to hardware network stacks, testing

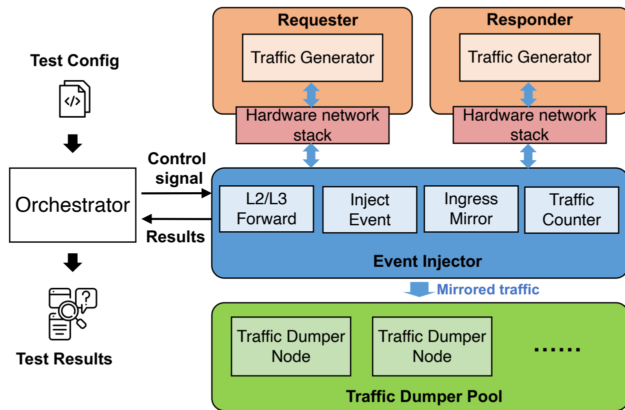


Figure 1: Lumina Overview.

tools face distinctive challenges. Firstly, hardware network stacks are predominantly implemented in hardware and bypass the kernel, making direct interaction and behavior measurement through software shim layers at the host (e.g., libpcap and TUN device used by packetdrill [16]) nearly impossible. Secondly, hardware network stacks operate at significantly higher throughputs (e.g., 400Gbps for a single flow on NVIDIA ConnectX-7 [13]) and lower latencies (a few microseconds). As a result, testing tools must meet strict performance requirements for injecting test events and conducting measurements.

3 LUMINA DESIGN

Motivated by above observations, we develop Lumina, a tool that enables testing the correctness and performance of hardware network stacks. In this paper, our primary focus lies on RDMA over Converged Ethernet (RoCE) v2[1], given its availability in commercial products from various vendors [5, 6, 9], as well as its widespread adoption among numerous cloud providers [11, 14, 20]. This section begins with an overview of the design rationale behind Lumina and then proceeds to provide a detailed introduction to its design.

3.1 Design Rationale

The inherent kernel bypass nature of hardware network stacks poses a challenge in directly injecting events and measuring behaviors at the end host. To overcome this limitation, we adopt an in-network approach. We connect two hosts with the hardware network stack under test to a programmable switch. By programming the switch, we can inject diverse events to emulate various network scenarios. However, the switch's constrained on-chip memory and processing capabilities make it impractical to entirely perform measurement and analysis tasks on the data plane. To address this, we employ packet mirroring to create duplicates of all packets, which are then forwarded to dedicated servers for offline analysis.

Figure 1 illustrates the four key components of Lumina: *Orchestrator*, *Traffic Generator*, *Event Injector*, and *Traffic Dumper*. To run a

```
requester:
  workspace: /home/foo/bar/
  control-ip: cx4-testing-traffic-requester
  nic:
    type: cx4
    if-name: enp4s0
    switch-port: 144
    ip-list: [10.0.0.2/24,10.0.0.12/24]
  roce-parameters:
    dcqcn-rp-enable: False
    dcqcn-np-enable: True
    min-time-between-cnps: 0
    adaptive-retrans: False
    slow-restart: True
```

Listing 1: Host (requester) Configuration Snippet

```
traffic:
  num-connections: 2
  rdma-verb: write
  num-msgs-per-qp: 10
  mtu: 1024
  message-size: 10240
  multi-gid: true
  barrier-sync: true
  tx-depth: 1
  min-retransmit-timeout: 14
  max-retransmit-retry: 7
  data-pkt-events:
  # Mark ECN on the 4th pkt of the 1st QP conn
  - {qpn: 1, psn: 4, type: ecn, iter: 1}
  # Drop the 5th pkt of the 2nd QP conn
  - {qpn: 2, psn: 5, type: drop, iter: 1}
  # Drop the retransmitted 5th pkt of the 2nd QP conn
  - {qpn: 2, psn: 5, type: drop, iter: 2}
```

Listing 2: Traffic and Event Configuration Snippet

test, the orchestrator takes a configuration as input, sets up the environment, and sends Remote Procedure Calls (RPCs) to coordinate different components.

Lumina employs two hosts with the same bandwidth capacity to generate traffic. Both hosts are equipped with the hardware network stack under test and are running separate instances of a traffic generator. One host serves as the requester, while the other functions as the responder. These hosts collaboratively generate network traffic based on the configurations provided by the orchestrator (§3.2).

The event injector serves the purpose of forwarding traffic while also injecting pre-configured events like ECN marks, packet losses, and corruptions (§3.3). Additionally, it mirrors all RDMA packets to the traffic dumper pool, consisting of multiple servers, allowing for offline analysis at a later stage (§3.4).

After the traffic finishes, the orchestrator gathers results from various components, such as dumped packets, NIC counters, and log files. It meticulously reconstructs the entire packet trace by assembling the dumped packets collected by the traffic dumper servers. Once this trace is complete, users have the ability to parse both the packet trace and other collected results, enabling in-depth analysis of the hardware network stack's behaviors (§3.5).

3.2 Traffic Generation

Prior to starting traffic generators, the orchestrator first configures IP addresses and apply network stack settings, e.g., congestion

[1]Throughout this paper, we will use the terms RDMA, RoCE, and RoCEv2 interchangeably.

control and loss recovery parameters, of traffic generation hosts. Listing 1 gives an example.

After the configuration process, the orchestrator initiates traffic generator instances on both hosts. Our traffic generator utilizes the Reliable Connected (RC) transport, and supports RDMA send/receive, write, and read verbs. In this paper, we use `Send`/`Recv`, `Write` and `Read` to denote them, respectively. Moreover, apart from using individual verbs, the requester has the flexibility to post verb combinations, such as `Send` and `Read`, facilitating the generation of bi-directional data traffic.

RDMA traffic generators communicate over one or multiple queue pairs (QPs). As illustrated in Listing 2, users have the flexibility to configure many parameters of traffic generators, e.g., the number of QPs, retransmission timeout, and MTU.

After two traffic generators initialize objects such as QPs and memory regions (MRs), they proceed to exchange essential metadata over a TCP connection. This metadata includes crucial information such as QP number (QPN), packet sequence number (PSN), global identifier (GID), memory address, and key. As QPNs and PSNs are generated randomly during runtime and play a critical role in allowing the event injector to identify the correct packets, the traffic generator also shares this metadata with the event injector (more details in §3.3).

Once metadata is exchanged and QP connections are established, the requester initiates the generation of RDMA traffic by posting work requests. The requester controls the total number of requests/messages and the maximum number of outstanding requests on each QP. In the case of `Send`/`Recv`, the responder continuously posts corresponding `Recv` requests to handle incoming data. Furthermore, the requester offers support for barrier synchronization among QPs. This synchronization mechanism ensures that the requester only posts the next round of requests after it receives completions of the current round of requests across all the QPs, promoting orderly and synchronized traffic generation.

Finally, once the requester receives completions of all the requests, it proceeds to calculate various metrics, including request/message completion times and goodput. Subsequently, the requester sends a completion notification to the responder through the established TCP connection, indicating the successful completion of the RDMA traffic generation process.

3.3 Event Injection

Lumina utilizes a programmable switch as the event injector to emulate real-world network scenarios, including congestion and failures. This injector can be programmed to introduce various events, such as packet corruptions, packet drops, and ECN marks, specifically targeting RDMA data packets[2]. Notably, in the context of `Read` requests, the responder generates data packets, while for other verbs, data packets are generated by the requester.

Although implementing the above events on programmable switches [7, 8] is not inherently challenging, the main difficulty lies in offering *user-friendly* interfaces that allow developers to express a sequence of *deterministic* event injections. This challenge gives rise to two specific requirements:

[2] It is important to mention that Lumina currently does not support injecting events to control packets, such as ACK and NACK.

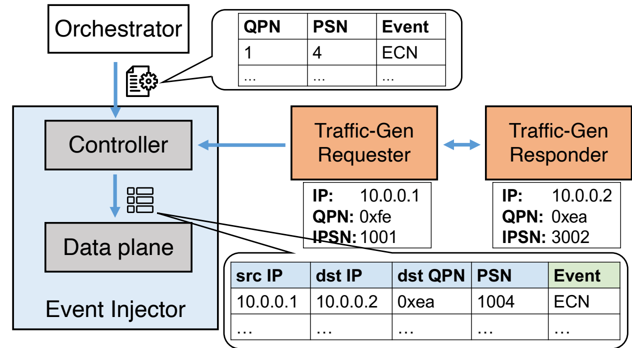


Figure 2: Lumina combines the runtime traffic metadata and intent-based traffic configuration to populate the match-action table for event injection.

Deterministic: Since Lumina aims at precise and reproducible tests to understand micro-behaviors, it exclusively accepts descriptions of deterministic injection events from users. Descriptions like "randomly drop 10% packets" are not deterministic since different rounds of testing may lead to dropping different sequences of packets. In contrast, descriptions like "drop the first packet of the first QP" generate deterministic injection behaviors.

User-friendly: Users should be able to express their high-level testing intents without delving into low-level Lumina details. For instance, users can instruct Lumina to drop the first packet of the second QP and subsequently drop its retransmission without the need to specify QPN and PSN for each QP or understand the inner workings of the event injector used for identifying retransmitted packets. The system abstracts these complexities, streamlining the testing process for users.

Listing 2 illustrates an example configuration of event injections. The setup involves three events spread across two connections. For the first connection, the fourth packet is marked with ECN. On the second connection, the fifth packet is intentionally dropped. Upon retransmission of this lost packet, we drop it again. To simplify the process for users, Lumina allows specifying *relative* QPNs and PSNs. Additionally, the *iter* field is available to identify retransmitted packets that share the same QPN and PSN as the original packets.

Next, we will describe how Lumina translates high-level test intents (e.g., relative QPN and PSN) to the low-level configuration for event injector, and uses an iteration number (ITER) to express per-connection retransmission behaviors.

Translate user intents to configurations: Since users only provide high-level intent information such as relative PSN and QPN, Lumina needs to translate this to the low-level configuration for the event injector. One straightforward solution is using the event injector to detect new QPs and parse their QPNs and PSNs in the data plane. While promising, this approach significantly complicates the data plane: In this case, the event injector needs to check if a packet belongs to a new QP and create states for the new QP.

Instead of the above stateful approach, we take a stateless approach by letting traffic generators provide runtime traffic metadata through the control plane. As mentioned above (§3.2), traffic metadata like QPN and initial PSN (IPSN) is randomly generated at

runtime. Once traffic generators finish exchanging metadata, the traffic requester sends the complete traffic metadata to the event injector through the control plane. The metadata is organized as a list of tuples. Each tuple contains the information for a certain QP connection: requester IP/QPN/IPSN and responder IP/QPN/IPSN. After that, the event injector combines the runtime traffic metadata from traffic generators and traffic configuration intents from the orchestrator to populate the match-action table for event injections. Only after the event injector populates the table, traffic generators can start RDMA traffic.

Figure 2 presents an illustrative example. In this scenario, the requester's QP has the IP address 10.0.0.1, QPN 0xfe, and IPSN 1001. On the other hand, the responder's QP has the IP address 10.0.0.2, QPN 0xea, and IPSN 3002. Data packets are transmitted from the requester to the responder, representing traffic like `Write` or `Send` operations. The user's intention is to drop the fourth packet of the first QP connection. To achieve this, the event injector combines the provided information and computes the appropriate entry to insert: ⌜srcIP=10.0.0.1, dstIP=10.0.0.2, dstQPN=0xea, PSN=1004⌟ → action="mark ECN".

Express retransmission behaviors: In certain tests, users may need to inject events specifically targeting retransmitted packets to gain insights into behaviors such as retransmission timeout backoff. However, distinguishing retransmitted packets from the original packets solely by examining packet headers becomes challenging since both packets share the same IP addresses, UDP ports, QPN, and PSN.

To address this issue, we introduce an iteration number *ITER*, which denotes the rounds of (re)transmissions for a connection. ITER starts from 1 and is maintained by the event injector. For every arriving RDMA packet, the event injector compares its PSN with *Last_PSN* (PSN of the last packet of the connection). If its PSN is *not larger than Last_PSN*, the event injector identifies this as a new round of transmissions and increases ITER by 1 for this connection. Regardless of the comparison result, the event injector always updates *Last_PSN* to the PSN of the current packet. Lumina can use (PSN, ITER) to uniquely identify every packet of a connection and inject events based on these fields.

Figure 3 provides an illustrative example of how Lumina tracks ITER. In this scenario, there is a single connection and the user intends to drop the second packet in the first round (PSN=2, ITER=1), and the third packet in the second round (PSN=3, ITER=2). The sender transmits four packets. ITER is initialized as 1 and the last PSN is set to IPSN-1, which is 0 in this case. In the first iteration, we drop packet 2. When packet 2 is retransmitted, the current PSN (2) is smaller than the last PSN (4), thus we proceed to a new round (ITER=2). Likewise, after we drop packet 3 in the second round, the retransmission of packet 3 triggers a new round (ITER=3).

3.4 Traffic Dumping

Lumina aims to dump *all* the RDMA packets between traffic generators for subsequent offline analysis. A straightforward solution is using tools like `ibdump` to dump packets at the end host. However, it is unclear if traffic dumping at the end host will impact the behaviors of network stacks. In addition, when we construct the complete packet trace from packets dumped at both traffic generation hosts, we may need nanosecond-level clock synchronization which is non-trivial [33].

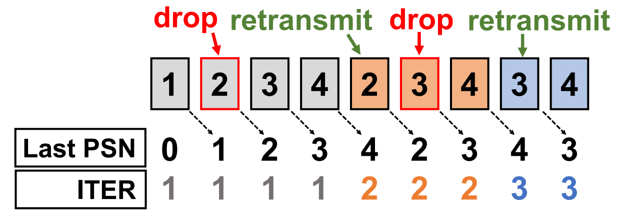


Figure 3: Lumina maintains ITER to differentiate packets in fine-grained. ITER denotes the rounds of (re)transmissions for a connection. If PSN of the current packet is no larger than that of the previous packet, ITER is increased by 1.

Realizing these challenges, Lumina utilizes the event injector to mirror all the RDMA packets to a dedicated group of servers, forming what is referred to as the "traffic dumper pool." Packet mirroring essentially clones packets of specified interfaces and forwards them to other interfaces. It has been widely used for measurement and diagnosis purposes [43, 56]. We mirror all RDMA packets at the ingress pipeline before any actual packet drops occur in the Memory Management Unit (MMU) (more details in §5).

To ensure integrity checks and facilitate traffic analysis, we leverage the event injector to embed essential metadata in the mirrored packets. To prevent losses during the packet dumping process, we develop a per-packet load balancing mechanism to evenly distribute mirrored packets across CPU cores of traffic dumpers. We will now provide detailed descriptions of these techniques.

Embedding metadata in mirrored packets: The event injector embeds three types of metadata, namely *mirror sequence number*, *event type* and *mirror timestamp*, into the mirrored packets for the following purposes:

(1) **Integrity check.** To guarantee the mirroring and dumping of all packets, the event injector maintains a global variable known as the *mirror sequence number*. This variable is incremented for each mirrored packet and is embedded into every mirrored packet. In conjunction with switch port counters, we can verify whether any packet losses occur during the mirroring and dumping process.
(2) **Indicating events.** For the ease of analyzing mirrored packets, we embed an *event type* in each packet to indicate the specific injected event, currently including ECN marking, drop, corruption, and none. It is important to note that all packets are mirrored at the ingress pipeline, before the Memory Management Unit (MMU) enforces any dropping actions.
(3) **Fine-grained measurement.** To precisely measure the behaviors, we embed a *mirror timestamp* in each mirrored packet. This timestamp carries a nanosecond-level hardware timestamp, indicating the exact moment when the original packet enters the ingress pipeline. The event injector uniformly adds timestamps to all packets, eliminating the need for clock synchronization.

Directly expanding packet headers to store these metadata might overload the bandwidth capacity of mirroring ports. To avoid this,

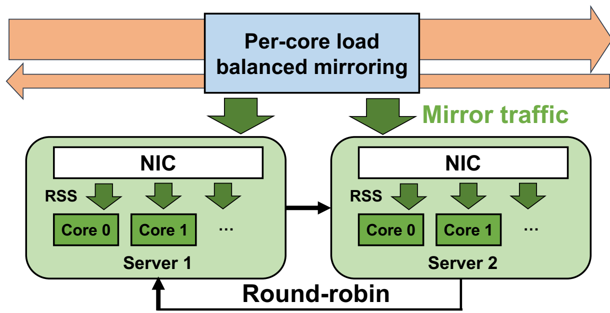


Figure 4: Per-packet load balancing to distribute mirrored packets across CPU cores of traffic dumpers.

we rewrite existing header fields that are not involved in traffic analysis to store above metadata. We use the Time to Live (TTL) field, the source MAC address field, and the destination MAC address field, to store *event type*, *mirror sequence number*, and *mirror timestamp*, respectively.

Per-packet load balancing. In our initial design, we used two hosts to dump mirrored packets generated by the requester and the responder, respectively. Traffic dumping hosts had the same bandwidth capacity as traffic generation hosts. Despite our optimization efforts on the traffic dumping program, we still encountered occasional packet discards (reflected in the `rx_discards_phy` counter) on the NIC when receiving line-rate mirrored data packets. Although we can identify such *invalid* tests through integrity checks (§3.5), this situation adversely affects the efficiency of Lumina. Furthermore, this design necessitates powerful traffic dumpers that can match the performance of the hardware network stack under test. This degrades the flexibility of hardware choices of Lumina.

Realizing the above limitations, we develop a per-packet load balancing mechanism to evenly distribute mirrored packets across the CPU cores of all the traffic dumpers. Instead of relying on two powerful hosts, we organize a pool of several hosts to serve as traffic dumpers. This design provides users with greater flexibility, as they can set up hosts for this purpose as long as the total capacity of the traffic dumper pool is sufficient to handle the traffic load. As illustrated in Figure 4, the event injector implements a weighted round-robin scheduler to forward mirrored packets to different traffic dumpers based on their individual processing capacities. At each traffic dumping host, we leverage Receive Side Scaling (RSS) to distribute packets across multiple CPU cores. However, RSS maintains flow-to-CPU affinity by hashing specific packet fields to select a CPU core, leading to a dependence on the number of flows for CPU processing capacity. To fully harness CPU cores, we employ the event injector to rewrite the UDP destination port (4791, reserved for RoCEv2) to a random number. This action effectively creates an illusion of many concurrent flows for RSS, maximizing CPU usage and avoiding underutilization of specific cores. The load balancing design results in a significant improvement in the success ratio of capturing complete traffic from 30% to nearly 100%.

After the traffic generation process is completed, the orchestrator sends a TERM message to stop all the traffic dumpers. Upon receiving this message, the traffic dumper reverts the previously rewritten UDP destination port for all the mirrored packets back to its original value (4791) and then writes all the mirrored packets to a disk file.

Name	Content Description
Dumped packets	Packets collected by all the hosts of the traffic dumper pool
Network stack counters	Link/Network/Transport layer counters
Traffic generator log	Application metrics, e.g., goodput and message completion time (MCT)
Switch counters	TX/RX/mirrored packet counters for each switch port

Table 1: Results collected by the orchestrator

3.5 Result Collection and Integrity Check

Once traffic generators stop, the orchestrator terminates other components and collects various result files as listed in Table 1. The orchestrator collects dumped packets from all the traffic dumpers, network stack counters and log files from the traffic generator, and switch counters from the event injector.

Upon gathering all the result files, the orchestrator reconstructs the packet trace from packets collected by the traffic dumpers. Since the event injector maintains the sequence number for every mirrored packet (§3.4), the orchestrator simply sorts all the packets based on their mirror sequence numbers. After the packet trace reconstruction, the orchestrator initiates an *integrity check* to check the following conditions:

(1) Consecutive mirror sequence numbers are present in the trace.
(2) The total number of packets mirrored by the event injector matches the number of packets in the trace.
(3) The total number of RDMA packets received by the event injector is equal to the number of packets in the trace.

Only when all the conditions are met, we can ensure that the packet trace is *complete* and ready for analysis.

4 TEST SUITE

After passing integrity checks, users can initiate the analysis process. Lumina includes a test suite comprising various built-in analyzers for commonly used features, such as Go-back-N retransmission [22] and DCQCN congestion control [55]. Additionally, it incorporates a fuzzing-based module for generating test cases.

Retransmission logic. Retransmission performance is crucial for lossy RDMA deployments. Even in *lossless* networks, RDMA NICs (RNICs) still need effective retransmission *mechanisms* to handle non-congestion losses [22].

To this end, we develop a retransmission logic analyzer to check if the RNIC's behaviors under packet losses comply with the specifications, e.g., if the Go-back-N receiver generates a NACK packet correctly when it observes out of order arriving packets. To realize this, we represent the specification of Go-back-N, the de facto retransmission algorithm of RNICs [22], as a finite-state machine (FSM). We then utilize this FSM to process the reconstructed packet trace. If the resulting trace leads to an incorrect state, it allows us

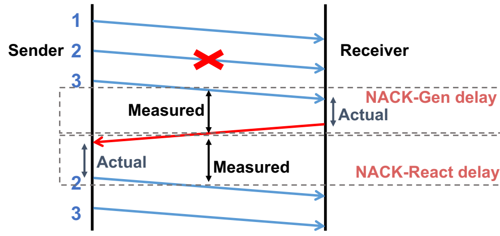


Figure 5: Retransmission latency breakdown.

to conclude that the RNIC's retransmission implementation does not fully comply with the specification.

Retransmission performance. Many efforts have been made to enable RDMA over lossy networks [3, 36, 39, 47]. Lossy RDMA technologies heavily rely on efficient retransmission *implementations*. For example, when the RNIC receives a NACK/SACK packet, it should promptly initiate the retransmission process rather than wait for an extended period. The responsiveness of retransmission significantly impacts the overall performance and reliability of RDMA in lossy network environments.

To help users understand the retransmission performance of RNICs, we develop a retransmission performance analyzer. The retransmission performance analyzer can deal with both fast retransmissions (triggered by NACK/SACK) and timeout retransmissions (due to tail losses), and provides a detailed performance breakdown to assist users in identifying potential bottlenecks.

In Figure 5, we illustrate how a NACK-triggered retransmission is divided into two distinct phases: the NACK generation phase (receiver side) and the NACK reaction phase (sender side). The NACK generation phase encompasses the time between the receiver detecting an out-of-order packet and transmitting a NACK packet. On the other hand, the NACK reaction phase refers to the time between the sender receiving the NACK packet and commencing the retransmission process. It is worth noting that there might be a half-RTT deviation due to the timestamp being added by the switch rather than the end host. To minimize this deviation, pre-measuring the round trip time (RTT) of the testbed can be employed.

Congestion notification. DCQCN [55] is the de facto congestion control protocol in RNICs. When the DCQCN notification point (NP) receives ECN-marked packets, it notifies the reaction point (RP) to reduce the rate using Congestion Notification Packets (CNPs). Recent NVIDIA RNICs extend DCQCN to lossy networks. In such scenarios, when the NP receives out-of-order packets, it generates both NACKs and CNPs to inform the RP to initiate retransmission and lower the sending rate. Additionally, NVIDIA RNICs include a CNP rate limiter at the NP side, which regulates the minimum interval between two consecutive CNPs, aiming to conserve bandwidth and NIC processing capacity [55].

To ensure that CNPs are generated as intended under diverse network conditions and CNP rate limiting configurations, we develop a CNP analyzer. This analyzer validates and verifies the correct generation of CNPs, which is crucial for efficient congestion control and performance optimization in lossy network environments.

Hardware network stack counter. We also develop a counter analyzer to verify the correct updating of hardware network stack counters. Our counter analyzer supports a wide range of counters, including those related to retransmission, timeout, congestion, and packet corruption. These counters encompass sent/received packets, sequence errors, out-of-sequences, timeouts (and retries), packets with iCRC errors, discarded packets, and CNPs sent/handled.

Algorithm 1 Genetic-based Fuzzing

```
1:  function LUMINA-FUZZ(TARGET)
2:      Γ ← initialize a pool of configs
3:      repeat
4:          γ ← randomly pick a config from Γ
5:          γ* ← mutate(γ)
6:          run Lumina with configuration γ*
7:          ϕ ← collect results (counters, timestamps, etc.)
8:          Δ ← score(γ*, ϕ)
9:          if Δ ≥ median_score(Γ) then
10:             Γ.add(γ*)
11:         else
12:             Γ.add(γ*) with a probability p.
13:     until anomaly found or timeout
```

Bugs/Hidden behaviors		Affected NICs
Non-work conserving ETS	(§6.2.1)	CX6 Dx
Noisy neighbor	(§6.2.2)	CX4 Lx
Interoperability problem	(§6.2.3)	CX5+E810
Counter inconsistency	(§6.2.4)	CX4 Lx, E810
CNP rate limiting	(§6.3)	All NICs tested
Adaptive retransmission	(§6.3)	All CX NICs

Table 2: Bugs and hidden behaviors

Test case generation. Network researchers have been utilizing fuzzing approaches in various scenarios, such as testing TCP implementations [57] and stress testing congestion control algorithms [44]. To aid users in identifying bugs, we develop an automatic test case generation module. This module employs a genetic algorithm-based fuzzing approach to discover settings and events that trigger abnormal behaviors, including bugs or performance issues. The procedure, as outlined in Algorithm 1, starts by defining a target, which can be either a general target (e.g., "finding bugs in a network setting with 0.1% loss rate") or a more specific target (e.g., "finding potential bugs where packet loss in one connection affects other co-existing connections"). The search space is smaller for more specific targets. The fuzzing approach follows four main steps:

(1) Initialization: given a target, it will generate a candidate pool of valid configurations.
(2) Mutation: During each iteration, a configuration file is randomly selected from the pool, and a new configuration file is created by applying mutations. These mutations involve modifying basic traffic settings (e.g., adjusting the number of QPs) and event settings (e.g., injecting ECN/drop to specific packets).
(3) Scoring: the scoring function assesses the "quality" of the configuration files concerning their ability to trigger anomalies. The scoring function is a multi-objective function formulated as $Score = \sum_i w_i \cdot s(i)$, where w_i is the weight for objective

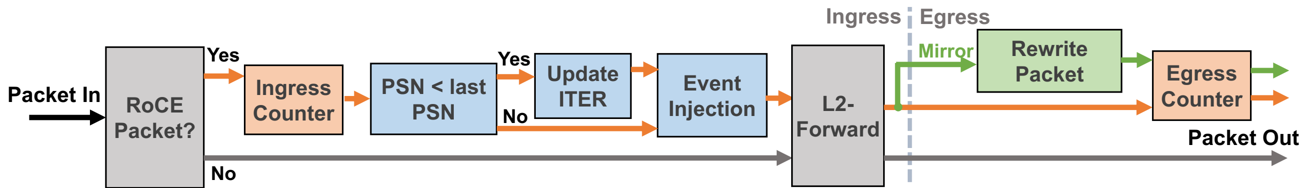


Figure 6: Pipeline layout of Lumina switch data plane.

i, and $s(i)$ implicitly models the anomaly of objective i (e.g., inconsistent counter values, large latencies, etc.).

(4) Selection: "high-quality" configuration files will be selected and put into the pool, while "low-quality" configuration files are sampled into the pool with a given probability.

Our fuzzing approach has helped us find several anomalies and the triggering conditions of the anomalies. We summarize some interesting bugs and hidden behaviors (that are not clearly described in the specification) in Table 2. In §6, we will introduce these findings in detail.

5 IMPLEMENTATION

We have built a prototype of Lumina using an Intel Tofino switch and commodity servers equipped with various RNICs. Currently Lumina can support four widely adopted RNICs from NVIDIA and Intel: NVIDIA ConnectX-4 Lx, ConnectX-5, ConnectX-6 Dx, and Intel E810. In the rest of this paper, we refer to them as CX4 Lx, CX5, CX6 Dx, and E810 respectively.

The data plane of the event injector is implemented with 708 lines of code (LoC) in P4-16 [4] and is compiled to Intel Tofino ASIC [7] using BF SDE 9.4.0. Figure 6 shows the data plane pipeline layout of the event injector. The event injection module operates at the ingress pipeline (§3.3). The egress pipeline includes a module to rewrite packet fields of mirrored packets (§3.4). Both incoming and outgoing RoCE packets, including mirrored packets, are counted on each port for integrity checks (§3.5). The switch control plane is implemented with 922 LoC in Python. It handles the translation of Remote Procedure Calls (RPCs) to configure the data plane modules and performs port counters' dumping once the experiment is completed.

The traffic generator is implemented with 5301 LoC in C. It uses `Libibverbs` to generate RDMA traffic over Reliable Connection (RC) transport. The traffic generator has the capability to control the GID (IPv4 address) associated with each QP to emulate traffic from multiple hosts. It reports total goodput and average request/message completion times for each QP.

The packet dumper is implemented with 628 LoC in C. It uses DPDK [1] with Receive Side Scaling (RSS) to process arriving packets across multiple queues and CPU cores. Instead of directly writing every arriving packet to the disk, the packet dumper leverages memory buffering and packet trimming to improve the performance. Upon receiving a packet, the packet dumper copies only the first 128 bytes of the packet into pre-allocated memory. This decision is based on the fact that Lumina does not require the IB payload content for analysis, and the first 128 bytes contain all the necessary protocol headers. Subsequently, when the packet dumper receives

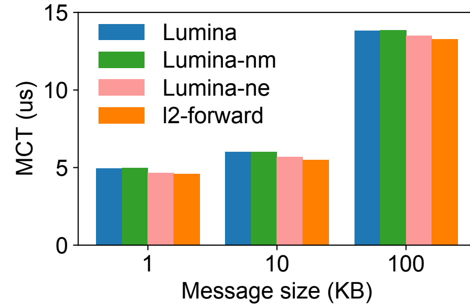


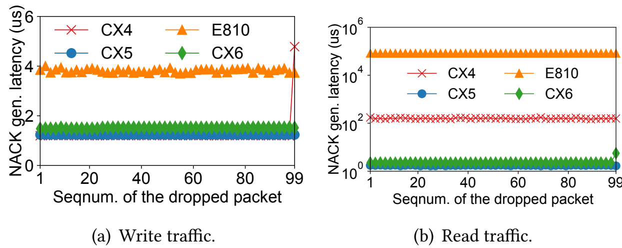
Figure 7: Lumina's impact on message completion time.

the TERM message from the orchestrator, it proceeds to write the buffered *trimmed* packets into the disk.

The orchestrator is implemented with 1933 LoC in Python. On top of the orchestrator, the built-in analyzers and the test case generation module are developed with a total of 3529 LoC in Python.

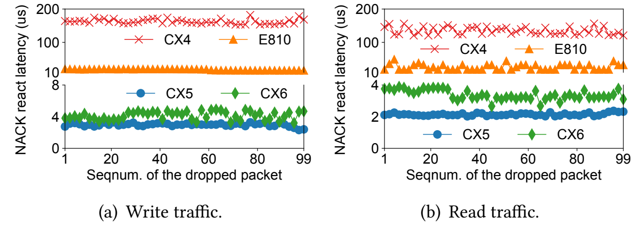
The current prototype occupies four stages of the switch's processing pipeline. It only requires approximately 1MB of on-chip memory to inject up to 100K events for 10K connections. We have pressure-tested the event injector by keeping sending traffic at full line rate. The switch is able to deliver and mirror all the packets without loss. Furthermore, measurements have demonstrated that the switch pipeline introduces less than 0.4μs of additional latency.

We have measured the overhead of event injector on the data path. We first pressure test the event injector by keeping sending traffic at full line rate. The switch is able to deliver and mirror all the packets without any losses. Subsequently, we evaluate the impact of the event injector on the under-test traffic. To achieve this, we employ a traffic generator to send 1000 messages of fixed sizes over a single connection and then measure the average message completion time (MCT). The messages are sent back-to-back with varying sizes of 1KB, 10KB, and 100KB. We use a simple L2-Forwarding program as a baseline. For Lumina, we keep all the match-action tables but disable the exact "drop" behavior to prevent retransmissions. We also implement two variants of Lumina: Lumina without event injection (Lumina-ne) and Lumina without mirroring (Lumina-nm) for comparison. Figure 7 demonstrates that event injection introduces minimal overhead. The average message completion time (MCT) of Lumina is only 4.1–7.2% higher compared to Lumina-ne and the basic L2-Forwarding. Notably, the inclusion of mirroring has negligible impact on the under-test traffic. Lumina achieves nearly the same message completion time with or without mirroring, indicating its efficiency in mirroring traffic.



(a) Write traffic. (b) Read traffic.

Figure 8: NACK generation latency v.s. sequence number of the dropped packet.



(a) Write traffic. (b) Read traffic.

Figure 9: NACK reaction latency v.s. sequence number of the dropped packet.

6 EXPERIMENTS AND FINDINGS

In this section, we present our analysis of retransmission performance, and share our experience of finding bugs and hidden behaviors using Lumina. We test four commodity RNICs: NVIDIA CX4 Lx 40GbE, CX5 100GbE, CX6 Dx 100GbE and Intel E810 100GbE. All the RNICs support Go-back-N retransmission and DCQCN.

In our testbed, the servers installed with NVIDIA NICs run Ubuntu 20.04.3 LTS and MLNX_OFED_LINUX-5.8-1.1.2.1-LTS. The servers with Intel NICs run Ubuntu 20.04.5 LTS with drivers ice 1.9.11 and irdma 1.9.30. PFC is disabled on both the switch and NICs. The RDMA MTU is set to 1024B for all the experiments. By default, each QP sends multiple messages back-to-back, thus keeping a single in-flight message.

6.1 Understanding retransmission micro-behaviors

RNICs commonly employ Go-back-N as the default fast retransmission algorithm. For `Write` and `Send`, the responder generates a NACK to trigger fast retransmission when it detects out-of-order arriving packets. However, for `Read`, the requester issues another read request to read from memory offset 'N', which serves as the equivalent of a NACK.

We use Lumina to analyze RNICs' fast retransmission behaviors by deliberately dropping packets in middle positions. First, we would like to note that all the RNICs pass our FSM-based retransmission logic check (§4) in a set of cunning and aggressive test cases. This indicates that their retransmission implementations strictly follow the specification. Then we present our findings about their fast retransmission performance, which is key for lossy RoCE [39].

In this experiment, we transfer a series of 100KB messages using a single connection. For each message, we intentionally drop a packet with a specific *relative* packet sequence number (PSN) and then measure the retransmission latency. To simplify expression, we refer to the `Read` request that triggers fast retransmission as NACK as well. As shown in Figure 5, we divide the Go-back-N retransmission latency into two parts: the NACK generation latency, and the NACK reaction latency. We show NACK generation latency results in Figure 8 and NACK reaction latency results in Figure 9. According to our experiment results, `Send` traffic delivers similar results as `Write` traffic, so we only present the results for `Write` and `Read` traffic here. We have the following observations:

- **CX5 and CX6 Dx achieve the best retransmisison performance.** Figure 8 and Figure 9 reveal that CX5 and CX6 Dx exhibit significantly lower retransmission latencies compared to CX4 Lx and E810. The NACK generation latency of CX5 and CX6 Dx is around 2μs, and the NACK reaction latency ranges from 2μs to 6μs. As a result, the retransmission delay of CX5 and CX6 Dx is approximately 4–8μs. Conversely, CX4 Lx experiences retransmission latencies in the hundreds of μs range, primarily due to slow NACK reactions. Regarding E810, the NACK generation latency for `Write` is around 10μs, while for `Read`, it remarkably increases to around 83ms.
- **Different behaviors of `Read` and `Write`/`Send`.** Figure 8 and Figure 9 demonstrate distinct results for `Read` and `Write` cases. The NACK generation latency for `Write` traffic is consistently low across all four NICs. However, for `Read` traffic, the NACK generation latency is significantly higher, especially on CX4 Lx (approximately 150μs) and E810 (around 83ms). This discrepancy suggests that RNICs may utilize a different and potentially slower processing pipeline to handle out-of-order `Read` packets, a factor that requires careful testing and investigation.

6.2 Experience with Lumina to find bugs

Here we share our experience using Lumina to find bugs and hidden behaviors. We have reported all the bugs to NVIDIA and Intel, and most of the bugs have been acknowledged. We are working closely with vendors on the rest of them. Lumina also enables us to find some *hidden* behaviors that are missing or not clearly described in specifications and vendors' documents. While we cannot reveal all the details due to our nondisclosure agreement (NDA) with vendors, we believe our experience can help readers have basic ideas of typical RNIC bugs and how to find them using Lumina.

6.2.1 *Non-work conserving ETS on CX6 Dx*

Enhanced Transmission Selection (ETS) [30] is a packet scheduling algorithm specified by IEEE 802.1Q and widely supported by commodity RNICs. It is a hierarchical scheduler that combines strict priority scheduling and a weighted fair queueing-based algorithm [27, 48]. ETS is work conserving. If higher-priority queues do not have packets ready for transmission, then packets from weighted fair queues can be transmitted. If a weighted fair queue cannot use its guaranteed bandwidth, then other queues will use its leftover bandwidth. ETS is widely used in production to allocate bandwidth between different types of traffic, e.g., TCP, storage frontend RDMA, and storage backend RDMA [14].

We design the following three experiments to test if ETS implementation can achieve work conserving in basic scenarios. In all the experiments, we create two QPs, named QP0 and QP1, and post

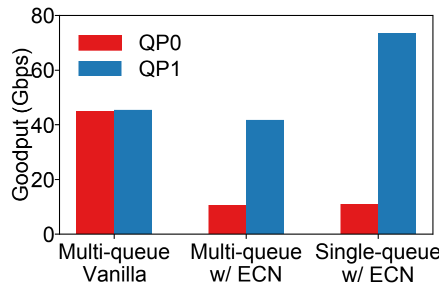


Figure 10: Goodput of two QPs under three settings on 100Gbps CX6 Dx. For the first and second experiments, there are two ETS queues, each with a guaranteed bandwidth of 50Gbps (weight=50%). We mark ECN on packets of QP0 for the second and third experiments.

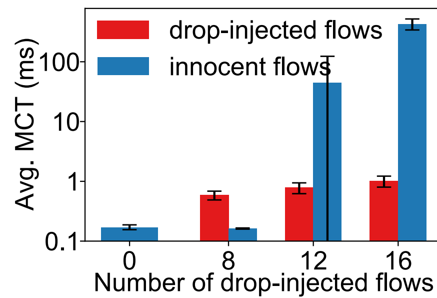


Figure 11: Average message completion times (MCTs) of innocent flows and flows with injected drops. There are 36 flows in total. When there are 12 flows experiencing injected drops simultaneously, innocent flows suffer from timeouts, thus leading to large MCTs.

20 `Write` requests with 1MB data per message on each QP. DCQCN is enabled.

(1) **Multi-queue vanilla:** We map two QPs to two ETS queues with identical weights. Hence, each ETS queue has a guaranteed bandwidth of 50Gbps.
(2) **Multi-queue with ECN marking:** We map two QPs to two ETS queues with identical weights. For QP0, we mark ECN for one out of every 50 packets.
(3) **Single queue with ECN marking:** We map two QPs to a single ETS queue. For QP0, we mark ECN on one out of every 50 packets.

In the first experiment, we anticipate both QPs achieving similar throughput results, approximately 50% of the line rate, due to the equal allocation of bandwidth in the two ETS queues. In the second and third experiments, we expect QP0 to have lower throughput due to DCQCN rate limiting, and QP1 to have larger goodput by utilizing the spare bandwidth left by QP0. However, as shown in Figure 10, we get unexpected results for the second experiment on NVIDIA CX6 Dx. While QP0's goodput is significantly reduced, QP1's goodput is still close to that of the first experiment. It seems that QP1 can not use the spare bandwidth left by QP0 when it is mapped to a different ETS queue. We further conduct a series of targeted tests and conclude that, in many scenarios, *ETS queues of CX6 Dx are strictly limited by their minimum guaranteed bandwidth regardless of bandwidth usage of other queues.* **This can significantly reduce CX6 Dx's throughput in production.**

We have reported this bug to NVIDIA. NVIDIA has confirmed our observation and closely collaborated with us in the debugging process. NVIDIA will include a bug fix for the issue in the future general available (GA) firmware version.

6.2.2 "Noisy neighbor" on CX4 Lx

Our experiment results in Figure 8 and 9 indicate that packet drops may trigger some slow paths of RNICs, thus causing large performance degradation, e.g., hundreds of μs NACK reaction latency on CX4 Lx. Given many types of shared micro-architecture resources in RNICs [25, 28, 29], we are wondering if slow paths of some connections can stall the whole RNIC pipeline, thus affecting other seemingly unrelated connections. This phenomenon is referred to as the "noisy neighbor" problem.

We use the fuzzing method (Algorithm 1) to search for "noisy neighbor" on all the RNICs. We split the connections into two sets: one set with injected drops, and the other without any events (innocent connections). The fuzzing module initializes a set of valid inputs (configurations) by randomly picking the number of connections for each set, choosing the traffic type (`Read`, `Send` or `Write`), setting the message size, and generating a list of packet drop events, etc. The mutation step modifies some fields of existing configurations to create new inputs. The scoring function accesses counter values and the performance of each connection (especially the message completion time of innocent connections).

We observe "noisy neighbor" problem on NVIDIA CX4 Lx. When more than a certain number of `Read` connections experience packet drops simultaneously, the other innocent connections also suffer from packet drops and even timeouts. Note that such concurrent packet drops are common in incast congestion. For example, we establish 36 connections and use `Read` to transfer ten 20KB messages per connection. On the first i (i=0,8,12,16) connections, Lumina drops the fifth data packet for each connection. The rest of 36-i connections are *innocent*. As Figure 11 shows, when there are only 8 connections with injected drops, the innocent connections perform normally and their message completion times are around 160μs. However, when there are more flows (i=12,16) suffering from injected drops, innocent flows start to suffer from performance degradation. The average message completion time of innocent connections reaches 430ms. To understand the causes of performance penalty, we look into NIC counters dumped by Lumina and find that the requestor discards many arriving packets (*rx_discards_phy* counter is increased by 107 when there are 12 connections with injected drops). When some "tail" packets are dropped by the NIC, timeouts will happen, thus leading to extremely high message completion times.

We have reported this bug to NVIDIA. NVIDIA has confirmed that this was a limitation of CX4 Lx.

6.2.3 Interoperability problem between CX5 and E810

As the cloud infrastructure keeps evolving incrementally, different servers may have different types of RNICs. Hence, communication between different types of RNICs, especially those from different vendors, becomes a new challenge. To this end, we run

basic tests by sending traffic from Intel E810 to NVIDIA CX5 without injecting any events. We vary the number of QPs and send five 100KB `Send` messages on each QP. We find that CX5 discards ~500 RX packets (by checking `rx_discards_phy` counter) when the number of QPs reaches 16. By analyzing the dumped packet trace, we find that most packet drops happen on the first message of each QP. Packet drops trigger timeouts and significantly affect message completion times. The average completion time of messages with and without packet drops are 20460μs and 156μs, respectively. This problem gets worse when we increase the number of QPs. However, if we send traffic from CX5 to CX5 under the same settings, this problem disappears. **This bug can degrade the performance of communication between CX5 and E810, especially when two nodes establish many new QPs to transfer data simultaneously.**

We dump complete packet traces of the above experiments using Lumina. We find a key difference between packets from E810 and CX5: packets sent by E810 set `MigReq` field of InfiniBand (IB) header to 0 while CX5's packets set `MigReq` to 1. According to IB specification [42], `MigReq` field is used for automatic path migration (APM) in IB networks and should be set to 1 at the initial state. However, since RoCE relies on IP and Ethernet routing, it is unclear how APM should work in IP/Ethernet networks. It seems that E810 and CX5 have different APM behaviors on RoCE mode and their communication might trigger some slow paths in APM processing logic of CX5. To confirm our hypothesis, we extend Lumina by adding a new action to modify `MigReq` to 1. We find that once we set `MigReq` of all the packets from E810 to 1, CX5 does not discard packets.

We have reported this bug to NVIDIA and Intel. They have confirmed our observation and closely collaborated with us in the debugging process.

6.2.4 *Incorrect RNIC counters.*

We also find several bugs about NIC counters. While these bugs do not directly cause performance impairments, they can significantly mislead operators, thus affecting debugging and diagnosis. We are working closely with NVIDIA and Intel to debug these counter bugs now.

- **Intel E810's `cnpSent`:** In DCQCN, the receiver generates congestion notification packets (CNPs) to notify the sender of congestion. All the RNICs have a counter to track the number of CNPs sent, e.g., `cnpSent` for Intel E810 and `np_cnp_sent` for NVIDIA RNICs. When we use Lumina to inject ECNs, we find that E810's `cnpSent` counter remains unchanged while the receiver does generate CNPs as shown in the dumped packet trace. **This bug can mislead operators' estimation of network congestion.**
- **NVIDIA CX4 Lx's `implied_nak_seq_err`:** NVIDIA NICs use this counter to indicate the number of times the requester detects out-of-order arriving packets for RDMA `Read` responses. When we use Lumina to inject packet drops on `Read` response traffic, we find that CX4 Lx's `implied_nak_seq_err` remains unchanged while drops and retransmissions do happen as shown in the packet trace. In contrast, `implied_nak_seq_err` keep increasing on CX5 and CX6 Dx as expected under the same settings. **This bug can mislead operators' estimation of network congestion and failures.**

6.3 Hidden behaviors

Lumina also enables us to capture unexpected micro-behaviors which are missing or not clearly described in specifications and vendors' documents. While these *hidden* behaviors are not necessarily bugs, understanding them can help operators better operate networks.

CNP generation interval. Instead of necessarily generating a CNP for every ECN-marked packet, NVIDIA NICs use a parameter *min_time_between_cnps* (4μs by default) to control the interval between two consecutive CNPs generated. CNP coalescing can effectively reduce the network bandwidth and NIC processing capacity consumed by CNPs. In contrast, Intel E810 does not have such a parameter. However, when we use Lumina to inject ECN marks and measure CNP intervals for all the NICs, we find that E810 does not generate a CNP for every arriving ECN-marked packet as expected. To further understand E810's CNP generation behaviors, we use Lumina to mark every packet and discover that there is a ~50μs interval between CNPs generated by E810. We reported our observation to Intel. Intel confirmed that E810 does have a hidden 50μs CNP minimum generation interval.

Different CNP rate limiting modes. Public documents of vendors do not clearly describe how RNICs enforce CNP rate limiting, e.g., per NIC port, per IP, or per QP. RNICs do not expose corresponding parameters either. By deliberately injecting ECN marks in different scenarios, we find that our RNICs use three CNP rate limiting modes: CX4 Lx limits CNP generations on a per destination IP basis; E810 on a per QP basis; and CX5 and CX6 Dx on a per NIC port basis. While per NIC port CNP rate limiting can effectively reduce resources consumed by excessive CNPs, it will increase the lag in DCQCN's control loop when many flows experience congestion simultaneously, such as in an incast scenario. We hope RNIC vendors expose configuration parameters to allow users to choose the desired CNP rate limiting mode.

Unexpected retransmission timeouts and times to retry in adaptive retransmission mode of NVIDIA NICs. According to IB specification [2], users can configure `timeout` and `retry_cnt` parameters to specify the minimum retransmission timeout ($4.096\mu s * 2^{timeout}$) and the maximum number of times that the QP tries to resend packets. We note that NVIDIA NICs have a new feature called adaptive retransmission to improve RDMA's resiliency over lossy networks. However, we do not find any public documents and references describing its mechanisms in detail. Through a series of targeted experiments, we find that actual retransmission timeouts and maximum times to retry in adaptive retransmission mode do not follow the IB specification. For example, when we set `retry_cnt` to 7, we observe that CX4 Lx, CX5 and CX6 Dx retry 8–13 times. When we set `timeout` to 14 (which means the *minimum* retransmission timeout is $4.096\mu s * 2^{timeout} = 0.0671s$), we find that the actual retransmission timeouts are smaller than 0.0671s for the first message. Taking CX6 Dx as an example, if we keep dropping the last packet of the first message for 7 times, their actual retransmission timeouts are 0.0056s, 0.0041s, 0.0084s, 0.0167s, 0.0251s, 0.0671s and 0.1342s. In contrast, if we disable adaptive retransmission, all the retransmission behaviors follow the IB

specification. We hope NVIDIA can reveal more details about this adaptive retransmission.

7 DISCUSSION

Lossy RoCE. The original implementation of RoCE depended on PFC to achieve a lossless fabric. However, ongoing debates [39, 47] regarding the feasibility of lossy RoCE networks have highlighted the need for a thorough understanding of the retransmission behavior and performance of RDMA NICs. Through a comprehensive analysis of the retransmission process, Lumina offers valuable insights into the underlying micro-behaviors, making a significant contribution to advancing our overall understanding of this subject.

Extensibility towards other traffic patterns. As of now, Lumina primarily focuses on studying RDMA transport behaviors. Consequently, it generates simple yet representative RDMA traffic, while avoiding complex operations that could strain RNIC microarchitecture resources. By combining straightforward traffic generation with the injection of various network events, Lumina aims to provide a clear and focused analysis of RDMA transport behaviors. This distinctive approach sets Lumina apart from other RDMA testing efforts [28, 29], that use more complex operations to explore challenging RDMA workloads can trigger performance anomalies of RNICs. Yet, one of the key advantages of Lumina is its extensibility, allowing for the integration of RDMA workloads from other research endeavors [28, 29]. This capability enables researchers to augment Lumina's capabilities and explore a broader range of RDMA scenarios and behaviors, contributing to a more comprehensive understanding of RDMA performance.

Extensibility towards other protocols. While Lumina initially targets RDMA, it can be extended with reasonable efforts to support other transport protocols and network stacks. By expanding the set of supported events, it will facilitate analysis of various settings like delay-based congestion control and multi-path load balancing. Although currently lacking support for events such as quantitatively adding delay and packet reordering, we plan to include these features as part of our future work.

Depolyment flexibility. We choose programmable switches as our event injector solution due to their user-friendly functionalities and accessibility. However, our design is not restricted solely to programmable switches; any programmable high-performance hardware can be employed as an event injector for Lumina. In the future, we aim to deploy Lumina on an FPGA board, creating a more lightweight solution that allows users to plug-and-test directly, simplifying the testing process further.

8 RELATED WORK

Network protocol testing. Several tools and research works are dedicated to testing network protocol implementations [16, 18, 21, 34, 41, 44]. Among them, packetdrill [16] is most closely related to Lumina. Packetdrill is a scripting tool that facilitates tests for the entire TCP/IP network stack. It interacts with the local and remote network stack using libpcap and TUN device as a "shim layer" to inject or consume packets. Additionally, packetdrill has been applied for testing QUIC [21] and utilized in educational contexts [15]. While all of these works concentrate on testing software network stacks, Lumina diverges by focusing on hardware offloaded network stacks, addressing a distinct aspect of network testing.

Understanding the performance of RDMA. Recent years have witnessed numerous efforts focused on understanding and optimizing the performance of RDMA in various scenarios [24–26, 28, 29]. Among these endeavors, Collie [29] and Husky [28] stand out as the most related works. Collie employs simulated annealing to explore a comprehensive search space, identifying workloads that trigger performance anomalies. Husky aims to systematically understand the impact of RNIC microarchitecture resources on performance isolation. Both works aim to stress commodity RNICs or performance isolation solutions by identifying challenging workloads. In contrast, Lumina currently focuses primarily on RDMA transport behaviors. It employs simple RDMA workloads while injecting various network events to uncover bugs and hidden behaviors.

Network testing with programmable networks. Programmable networks are gaining increasing relevance [23, 32, 35, 37, 40, 45, 51, 54]. Researchers have been exploring the use of programmable switches to enhance networking testing [17, 31, 52]. These efforts mainly focus on achieving high-throughput traffic generation and precise rate control using programmable switches. In contrast, Lumina takes a different approach by leveraging programmable switches to inject various network events, enabling a deeper understanding of the transport behaviors of hardware network stacks.

9 CONCLUSION

We present Lumina, a tool designed for testing the correctness and performance of hardware network stack implementations. By offering developers a user-friendly environment to write reproducible tests and inject deterministic events, Lumina simplifies the testing process. Additionally, we include a test suite with analyzers for common features and a fuzzing-based test case generation module. With Lumina, we have discovered multiple critical bugs in commodity RDMA NICs and confirmed the bugs with the vendors. We firmly believe that Lumina holds the potential to significantly assist network developers in comprehending the micro-behaviors of intricate hardware network stacks.

Ethics. This work does not raise any ethical issues.

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China under the grant number 2022YFB4500700, the National Natural Science Foundation of China under the grant number 62172008, the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas), and National Science Foundation Grant under award 2244870 and 2333887.

We sincerely thank the anonymous reviewers for their valuable feedback. Additionally, we would like to express our appreciation to Mahmoud Elhaddad, Jitendra Padhye, and Abdul Kabbani for their essential insights and feedback on this research. Our special thanks go to NVIDIA and Intel for providing us with their technical support throughout this project. Xin Jin is the corresponding author. Xin Jin is also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

REFERENCES

[1] 2018. Intel Data Plane Development Kit (DPDK). http://dpdk.org/.

[2] 2021. InfiniBand Architecture Specification Volume 1 Release 1.5. https://www.infinibandta.org/ibta-specification/.

[3] 2021. NVIDIA Zero Touch RoCE (ZTR). https://tinyurl.com/yc6tnv7h/.

[4] 2021. P4-16 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.2.2.html.

[5] 2022. Broadcom M1100G16 100GbE OCP 2.0 Adapter. https://www.broadcom.com/products/ethernet-connectivity/network-adapters/100gb-nic-ocp/m1100g.

[6] 2022. Intel Ethernet Network Adapter E810. https://www.intel.com/content/www/us/en/products/details/ethernet/800-network-adapters/e810-network-adapters.html.

[7] 2022. Intel Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[8] 2022. Intel Tofino 2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[9] 2022. NVIDIA ConnectX-6 Dx. https://www.nvidia.com/en-us/networking/ethernet/connectx-6-dx/.

[10] 2022. OFED perftest. https://github.com/linux-rdma/perftest.

[11] 2022. Oracle Cloud Infrastructure Blog: First principles: Building a high-performance network in the public cloud. https://blogs.oracle.com/cloud-infrastructure/post/building-high-performance-network-in-the-cloud.

[12] 2023. Azure high-performance computing. https://azure.microsoft.com/en-us/solutions/high-performance-computing/.

[13] 2023. NVIDIA ConnectX-7 400G Adapters. https://nvdam.widen.net/s/csf8rmnqwl/infiniband-ethernet-datasheet-connectx-7-ds-nv-us-2544471.

[14] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. 2023. Empowering Azure Storage with RDMA. In *USENIX NSDI*.

[15] Olivier Bonaventure, Quentin De Coninck, Fabien Duchêne, Anthony Gego, Mathieu Jadin, François Michel, Maxime Piraux, Chantal Poncin, and Olivier Tilmans. 2020. Open educational resources for computer networking. *SIGCOMM CCR* (August 2020).

[16] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao-keng Jerry Chu, Andreas Terzis, and Tom Herbert. 2013. packetdrill: Scriptable network stack testing, from sockets to packets. In *USENIX ATC*.

[17] Yanqing Chen, Bingchuan Tian, Chen Tian, Li Dai, Yu Zhou, Mengjing Ma, Ming Tang, Hao Zheng, Zhewen Yang, Guihai Chen, Dennis Cai, and Ennan Zhai. 2023. Norma: Towards Practical Network Load Testing. In *USENIX NSDI*.

[18] Scott Dawson, Farnam Jahanian, and Todd Mitton. 1997. Experiments on six commercial TCP implementations using a software fault injection tool. *Software: Practice and Experience* (1997).

[19] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[20] 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 *USENIX NSDI*.

[21] Vidhi Goel, Rui Paulo, and Christoph Paasch. 2020. Testing QUIC with packetdrill. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*.

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

[23] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. 2019. Qpipe: Quantiles sketch fully in the data plane. In *ACM CoNEXT*.

[24] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *ACM SIGCOMM*.

[25] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *USENIX ATC*.

[26] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *USENIX OSDI*.

[27] Manolis Katevenis, Stefanos Sidiropoulos, and Costas Courcoubetis. 1991. Weighted round-robin cell multiplexing in a general-purpose ATM switch chip. *IEEE Journal on selected Areas in Communications* 9, 8 (1991), 1265–1279.

[28] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, and Alvin R Lebeck Danyang Zhuo. 2023. Understanding RDMA Microarchitecture Resources for Performance Isolation. In *USENIX NSDI*.

[29] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. 2022. Collie: Finding Performance Anomalies in RDMA Subsystems. In *USENIX NSDI*.

[30] Victor Lama. 2011. Enhanced Transmission Selection–IEEE 802.1 Qaz. (2011). https://community.cisco.com/legacyfs/online/legacy/3/7/8/74873-Enhanced%20Transmission%20Selection%20v1.0.pdf.

[31] Guanyu Li, Menghao Zhang, Cheng Guo, Han Bao, Mingwei Xu, Hongxin Hu, and Fenghua Li. 2022. IMap: Fast and Scalable In-Network Scanning with Programmable Switches. In *USENIX NSDI*.

[32] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R.K. Ports. 2016. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *USENIX OSDI*.

[33] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. 2020. Sundial: Fault-tolerant clock synchronization for datacenters. In *USENIX OSDI*.

[34] Yuliang Li, Rui Miao, Mohammad Alizadeh, and Minlan Yu. 2019. DETER: Deterministic TCP Replay for Performance Diagnosis. In *USENIX NSDI*.

[35] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019.. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *USENIX FAST*.

[36] Yuanwei Lu, Guo Chen, Zhenyuan Ruan, Wencong Xiao, Bojie Li, Jiansong Zhang, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2017. Memory efficient loss recovery for hardware-based transport in datacenter. In *Asia-Pacific Workshop on Networking*.

[37] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *ACM SIGCOMM*.

[38] 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 *ACM SIGCOMM*.

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

[40] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *ACM SIGCOMM*.

[41] Steve Parker and Chris Schmechel. 1996. The packet shell protocol testing tool. *Software distribution at http://playground.sun.com/psh* (1996).

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

[43] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. 2014. Planck: Millisecond-scale monitoring and control for commodity networks. *ACM SIGCOMM*.

[44] Devdeep Ray and Srinivasan Seshan. 2022. CC-fuzz: genetic algorithm-based fuzzing for stress testing congestion control algorithms. In *ACM SIGCOMM HotNets Workshop*.

[45] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. 2019. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701* (February 2019).

[46] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. 2020. A Cloud-Optimized Transport Protocol for Elastic and Scalable HPC. *IEEE/ACM MICRO* (2020).

[47] Alexander Shpiner, Eitan Zahavi, Omar Dahley, Aviv Barnea, Rotem Damsker, Gennady Yekelis, Michael Zus, Eitan Kuta, and Dean Baram. 2017. RoCE rocks without PFC: Detailed evaluation. In *Proceedings of the Workshop on Kernel-Bypass Networks*.

[48] Madhavapeddi Shreedhar and George Varghese. 1996. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking* 4, 3 (1996), 375–385.

[49] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: brute force vulnerability discovery*. Pearson Education.

[50] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut

Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:arXiv:2307.09288

[51] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. 2020. Netlock: Fast, centralized lock management using programmable switches. In *ACM SIGCOMM*.

[52] Dai Zhang, Yu Zhou, Zhaowei Xi, Yangyang Wang, Mingwei Xu, and Jianping Wu. 2021. Hypertester: high-performance network testing driven by programmable switches. *IEEE/ACM Transactions on Networking* (2021).

[53] Naiqian Zheng, Mengqi Liu, Ennan Zhai, Hongqiang Harry Liu, Yifan Li, Kaicheng Yang, Xuanzhe Liu, and Xin Jin. 2022. Meissa: scalable network testing for programmable data planes. In *ACM SIGCOMM*.

[54] Hang Zhu, Zhihao Bai, Jialin Li, Ellis Michael, Dan Ports, Ion Stoica, and Xin Jin. 2019. Harmonia: Near-Linear Scalability for Replicated Storage with In-Network Conflict Detection. In *Proceedings of the VLDB Endowment*.

[55] 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. In *ACM SIGCOMM*.

[56] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. 2015. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM*.

[57] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. 2021. TCP-Fuzz: Detecting memory and semantic bugs in {TCP} stacks with fuzzing. In *USENIX ATC*.