Marlin: Enabling High-Throughput Congestion Control Testing in Large-Scale Networks

Yanqing Chen[★], Li Wang[★], Jingzhi Wang[△], Songyue Liu[★], Keqiang He[†]
Jian Wang[△], Xiaoliang Wang[★], Wanchun Dou[★], Guihai Chen[★], Chen Tian[★]
[★]State Key Laboratory for Novel Software Technology, Nanjing University, China
[△]School of Electronic Science and Engineering, Nanjing University, China
[†]Shanghai Jiao Tong University, China

Abstract

Cloud providers require high-throughput traffic to test the effectiveness of congestion control (CC) configurations (*i.e.*, CC algorithm selection and their parameter settings) in networks. A network tester capable of evaluating CC configurations needs to fulfill the following requirements: (R1) Capable of generating traffic with CC behaviors. (R2) Ability to customize CC algorithms. (R3) High throughput CC traffic generation. However, existing network testers fail to meet these requirements simultaneously. The paper presents Marlin, a novel high-throughput network tester designed for CC evaluation. Marlin leverages a high-throughput, low-programmability device to amplify the traffic generated by a low-throughput, high-programmability device. The low-throughput device is responsible for complex computational tasks, such as running CC and flow scheduling algorithms, and communicates with the high-throughput device at a high frequency using small packets to instruct it to generate high-throughput traffic with CC behaviors. This hybrid approach allows for customizable, high-throughput CC testing. Our experiments demonstrate that Marlin can accurately emulate CC behaviors and replicate real-world scenarios. Marlin can generate 1.2 Tbps of CC traffic using a single programmable switch pipeline and one 100 Gbps port of an FPGA NIC, supporting up to 65,536 concurrent flows.

CCS Concepts: • **Networks** → **Protocol testing and verification**; **Network measurement**; *Programmable networks*; *Data center networks.*

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.
EuroSys '25, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1196-1/2025/03
https://doi.org/10.1145/3689031.3717486

Keywords: Network testing, Congestion control algorithms, Programmable switch, FPGA accelerator, Data center networks

ACM Reference Format:
Yanqing Chen, Li Wang, Jingzhi Wang, Songyue Liu, Keqiang He, Jian Wang, Xiaoliang Wang, Wanchun Dou, Guihai Chen, Chen Tian. 2025. Marlin: Enabling High-Throughput Congestion Control Testing in Large-Scale Networks. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3689031.3717486

1 Introduction

Congestion control (CC) configuration is critical for cloud providers. CC can regulate the quantity of packets injected into a network, thus preventing a decline in network performance. In recent years, many studies have proposed new CC algorithms to improve application performance in various networks, with some focusing on data center networks [22, 24, 27, 33, 38, 44, 45, 49, 52–55, 59, 62, 67] and others on wide area networks [28, 32, 35, 42, 43]. Additionally, many CC algorithms require switches to provide additional network information, such as explicit congestion notification (ECN), and in-band network telemetry (INT) [21, 37, 51, 64].

Cloud providers face the challenge of selecting from a multitude of CC algorithms and optimizing parameters for both CC and switches. Several studies, along with network interface card (NIC) manufacturers, have provided guidance on selecting appropriate parameters [6]. Particularly in machine learning training scenarios, several studies have evaluated the CC algorithms mentioned above and custom CC algorithms [46, 47, 56]. However, due to the varying traffic patterns of different applications, the efficacy of these CC algorithms remains inconclusive. In practical deployment, cloud providers still need to validate the effectiveness of the selected CC algorithms and parameters through high-throughput traffic that resembles their production environment [66].

Simulation-based approaches can be used to obtain initial estimates of the behavior but are insufficient for validating the many performance issues that can emerge in complex production environments. In production, issues such as code implementation errors, chip design flaws, and parameter

Table 1. Network tester implemented on various hardware platforms compared to our requirements.

Requirements	Software & FPGA	Commercial	Programmable Switch	Marlin
(R1) Capability to generate traffic with CC	✓	✓	✗	✓
(R2) Customizable CC in the network tester	✓	✗	✗	✓
(R3) High throughput CC traffic generation	✗	✗	✓	✓

misconfigurations can arise. These are problems that simulations cannot detect, making network testers more suitable for testing in such scenarios.

Ideally, cloud providers utilize network testers to generate high-throughput traffic to replace real traffic sources. Operators can configure the test traffic generated by the tester, including protocols, addresses, ports, rates, patterns, *etc.*, and send it to the tested network. The tester then measures the test traffic, obtaining metrics of interest to the operators, such as packet loss and flow throughput, to evaluate the network performance. For cloud providers and network operators, a network tester capable of testing the effectiveness of CC configurations should meet three requirements:

- **(R_1) Capability to generate traffic with CC behaviors.** Network operators are concerned with how to configure CC to improve network performance, so the traffic generated by the tester needs to comply with CC behaviors.
- **(R_2) Customizable CC in the network tester.** To select the most suitable CC algorithm for the network, the CC algorithm emulated by the tester should be customizable. Furthermore, for a given CC algorithm, network operators should be able to find the optimal configuration by adjusting CC parameters.
- **(R_3) High throughput CC traffic generation.** Cloud providers and network operators deal with networks on a large scale, necessitating that the traffic generated by the tester reaches the Tbps level. High throughput per tester also aids in conserving rack space and reducing testing costs [36].

Numerous studies have focused on network testers [12, 14, 17, 19, 25, 34, 36, 39–41, 50, 57, 58, 65], yet these have not concentrated on CC testing, thus failing to meet the previously mentioned three requirements (see Table 1). Software-based [14, 19, 34, 40, 41, 57] and FPGA-based [25, 39, 58] testers offer high programmability and flexibility, making it easy to implement CC algorithms even though these works did not initially concentrate on CC testing. But for throughput, the former is limited by the CPU capability, while the latter is limited by the number of interfaces (not satisfying $\mathbf{R_3}$). Additionally, some commercial testers support Layer 4-7 testing [12, 17]. Yet, these commercial testers do not focus on CC, nor do they support customized CC, and they do not provide Tbps-level throughput in a single device (not satisfying $\mathbf{R_2}$ and $\mathbf{R_3}$). In recent years, some works have employed programmable switches to implement network testers [36, 50, 65], which are white-box in nature and offer high throughput. Nevertheless, these works have not implemented CC testing nor provided a framework for it (not satisfying $\mathbf{R_1}$ and $\mathbf{R_2}$). Currently, there is no single hardware suitable for implementing a network tester to test the effectiveness of CC configurations (§2.1). Therefore, we decided to develop a tester that can simultaneously fulfill these three requirements.

Our approach: Marlin. This paper introduces Marlin, a high-throughput network tester designed for CC testing. The key idea of Marlin is to leverage a high-throughput, restricted-programmability device (*e.g.*, a programmable switch) to amplify the traffic generated by a low-throughput, high-programmability device (*e.g.*, an FPGA NIC). The customizable CC algorithm module and flow scheduling algorithm are implemented on the FPGA NIC. The FPGA NIC instructs the programmable switch to generate large packets from 64-byte packets, thereby amplifying the traffic with CC behavior generated by the FPGA NIC. The generated test traffic passes through the tested network, and the CC feedback returns to the programmable switch. The programmable switch compresses each CC feedback into a 64-byte packet and returns it to the FPGA NIC to execute the relevant logic of the CC algorithm (§3 & §4).

On this basis, we have designed and implemented the following three points to meet our requirements. Firstly, we realized a flow scheduling module on the FPGA NIC and exposed an interface for the CC algorithm module to generate CC traffic (satisfying $\mathbf{R_1}$). Secondly, we utilize high-level synthesis (HLS) [10] to implement customizable CC algorithm modules, allowing users to write the CC algorithm module through high-level programming languages (satisfying $\mathbf{R_2}$). Finally, through the truncation, loopback, and multicast capabilities provided by programmable switches, in conjunction with FPGA NICs, Marlin achieved Tbps-level CC traffic generation (satisfying $\mathbf{R_3}$). In addition, we implemented essential features such as fine-grained measurements required in the network tester. Since our solution involves communication and coordination between different devices, implementing Marlin requires us to address several challenges:

Challenge 1: Frequency mismatch between programmable switches and FPGA NICs. The FPGA NIC instructs the programmable switch to generate large packets using 64-byte

scheduling packets. However, for a single port on the programmable switch, the frequency of generating large packets is much lower than the processing frequency of the scheduling packets. This frequency mismatch can lead to the programmable switch failing to process the scheduling packets from the FPGA in time, resulting in erroneous operation of the CC algorithm (solved in §5.3).

Challenge 2: Line-rate scheduling for tens of thousands of flows in FPGA NICs. The simplest method to achieve fair scheduling of each flow is to cyclically scan each flow to determine if it meets the scheduling conditions. However, our scheduling needs to reach the line rate, *i.e.*, 149 Mpps (million packets per second), while the internal clock frequency of the FPGA NIC is 322 MHz [4]. We would waste multiple clock cycles searching for a schedulable flow, especially when there are numerous flows but only a few are schedulable, preventing us from reaching line-rate (solved in §5.2).

Challenge 3: Read-Write conflicts of CC parameters in FPGA NICs. CC feedback triggers CC parameter updates in the CC algorithm, executing read-modify-write (RMW) operations in the FPGA. Because CC feedback may arrive in bursts, it can lead to a higher arrival frequency of CC feedback for the same flow than the frequency of RMW operations, resulting in read-write conflicts of CC parameters (solved in §5.3).

We implemented Marlin on programmable switches with Intel Tofino ASICs [8] and Xilinx Alevo U280 FPGA NICs [3] (§6). We conducted comprehensive experiments to assess the feasibility and correctness of CC testing on Marlin. The experiments showed that throughput can reach the line rate for a single flow. Our implementation accurately reflected the expected CC behavior and maintained high fidelity in replicating real-world scenarios. Marlin generated 1.2 Tbps of CC traffic using a single programmable switch pipeline and a 100 Gbps FPGA NIC port, supporting up to 65,536 concurrent flows (§7).

Ethics. This work does not raise any ethical issues.

2 Background

In this section, we discuss the observations that inspire Marlin's architecture. Then, we provide a list of works that are similar to or relevant to Marlin.

2.1 Observations

In scenarios testing the effectiveness of CC, it is necessary to measure various data such as the highest throughput achievable by the device or network under a given CC algorithm and parameters, the amount of packet loss, the rate of each flow, and fairness. These tests are not sensitive to the packet payload or the protocol logic. Given this, the traffic generated by Marlin can carry empty payloads and strip away protocol logic, focusing solely on the CC algorithm. Devices such as hosts, programmable switches, and FPGAs can be used to implement network testers. However, under our relaxed conditions, these devices still cannot individually meet the three requirements listed in Table 1. The reasons for this are further explained in combination with Table 2.

Table 2. The three characteristics required for implementing CC testing on various devices.

Devices	Programmability	Freq.	Throughput
Host	✓	✗	✗
Switch	✗	✓	✓
FPGA	✓	✓	✗
Marlin	✓	✓	✓

Switch=Programmable switch
Freq.=Packet processing frequency

Programmability. System programmability can be categorized into three tiers. High programmability systems such as CPUs and FPGAs provide full flexibility for implementing custom logic. Programmable switches fall under restricted programmability, offering limited capabilities for expressing logic. Dumb switches exhibit low programmability due to fixed ASIC-based data planes with minimal configurability. Programmable switches lack sufficient programmability to implement CC algorithms. The switch pipeline is divided into several match-action unit (MAU) stages, where packets match table entries within the MAU and execute certain instructions. However, programmable switches have limitations in terms of pipeline length, supported instruction operations, and register read-write capabilities, making it impossible to implement CC algorithms. For example, the Intel Tofino ASIC provides only 12 stages, restricting the number of instructions that can be executed in the pipeline. For instructions, conditional branching, jumping, and multiplication or division of any two numbers are not feasible on Tofino ASICs, as these instructions consume an indeterminate number of clock cycles. Most CC algorithms involve updating parameters (such as window size) and require read-modify-write (RWM) operations. However, for programmable switches, registers are associated with MAUs and cannot execute RWM operations, thus preventing updates to CC parameters. In contrast, CPUs have rich instructions and flexible memory access capabilities, and common network protocol stacks run on hosts. FPGAs, capable of implementing algorithms, have had multiple works[26, 29] implementing CC algorithms on FPGA NICs. Both CPUs and FPGAs meet the programmability criteria.

Packet processing frequency. The CPU's clock frequency is insufficient to meet our testing requirements. Taking a maximum transmission unit (MTU) of 1518 bytes as an example, to achieve a throughput of 1 Tbps, the ideal device should be able to process approximately 81 Mpps. Even when using DPDK [5] to bypass the operating system, and employing high-frequency CPUs with a single-core computational

frequency of 3 GHz, a highly optimized CC algorithm that completes in 50 clock cycles still cannot meet the processing requirement of 81 Mpps [7]. The 322 MHz clock provided by FPGAs like Xilinx Alevo U280 supports the 81 Mpps packet processing requirement. Programmable switches have a pipeline architecture, and the switching ASIC like Intel Tofino can forward packets at 2,400 Mpps, fully meeting the frequency requirements.

Throughput. The throughput bottleneck of a network tester built using FPGA NICs lies in the number of interfaces. Typically, there are two 100 Gbps interfaces on an FPGA NIC. A 2-rack unit server can accommodate four such NICs, providing a total throughput of 800 Gbps, which falls short of achieving Tbps-level throughput. The throughput of the host is limited by the CPU computing capacity. Only when the CC algorithm is offloaded on the NIC hardware, such as NVIDIA ConnectX NICs, is it possible to achieve high throughput. However, such a design does not meet our requirement for customizable CC. The design of programmable switches considers multi-port and high throughput from the beginning, thereby meeting the throughput criteria.

In summary, to simultaneously meet these three requirements, a device must possess high programmability, high packet processing frequency, and high throughput. However, hosts, FPGAs, and programmable switches are unable to meet all three criteria simultaneously. Therefore, we consider using a combination of these devices to create a tester supporting CC. Given that only programmable switches can uniquely meet the Tbps-level throughput criterion, our system must include a programmable switch. Additionally, we need to choose another type of hardware that can compensate for the programmability limitations of the programmable switch, and when combined, does not cause any criteria to be unmet. We choose FPGA NIC instead of the host for three reasons: (1) Only FPGA NIC can achieve line rate scheduling for a single flow. (2) The latency and jitter introduced by the host processing are much greater than FPGA, which is detrimental to delay-based congestion control and precise behaviour tracing. (3) FPGA NIC is more suitable for communicating with the programmable switch using high-frequency small packets. The FPGA NIC is responsible for programmability and computation, while programmable switches handle high-throughput traffic. We will detail Marlin's architecture and workflow in §3.

2.2 Related Work

This section lists some work related to ours in recent years.

CC on hardware. Tonic [26] implemented a programmable transport layer on FPGAs, offering a custom CC-capable data delivery engine. F4T [29] provides a full hardware-based TCP acceleration framework on FPGAs. The above efforts are aimed at offloading the transport layer to FPGA NICs, which is orthogonal to Marlin's goal. ACC-Turbo [23] implements an improved ACC algorithm on programmable switches which implements a specific CC and does not generate CC traffic.

Network testers. Software-based testers [11, 14, 16, 18, 19, 31, 40, 57] using standard Linux IO API or DPDK [5] are flexible. These testers are constrained by CPU clock frequencies and the number of network card ports, making it difficult to achieve Tbps-level throughput. More network testers are provided by manufacturers like Spirent [17] and Keysight [12]. These testers offer comprehensive testing capabilities covering L2 to L7. However, as these testers are black boxes, they cannot test custom CCs. Moreover, testers supporting L4 and above do not reach Tbps-level throughput [36]. Finally, these devices are expensive, with a dual-port 100 Gbps traffic generation module costing over $100,000 [65]. Some works [25, 39, 58] employ FPGA NICs to implement testers. These hardware-based testers are white-boxes, offering significant flexibility and the ability to provide precise measurements at the circuit level. However, FPGA NICs provide only two 100 Gbps ports and are relatively expensive (costing $5,341 [2]), which limits the throughput of these testers to Tbps levels. Programmable switches can be used to implement hardware-based white-box testers. Works like Norma [36], HyperTester [65], and IMap [50] are based on programmable switches and programmed using the user-friendly P4 language [30]. These works achieve high throughput and configurable traffic generation but do not simulate CC algorithms or generate traffic with CC behaviors.

Similar architecture. NeoBFT [60] accelerates Byzantine fault-tolerant (BFT) protocols in data centers. In NeoBFT, FPGAs serve as cryptographic coprocessors for computing SHA-256, while programmable switches handle the BFT protocol. Its architecture is similar to ours, but we focus on customizable CC, high throughput, and concurrency. μFAB [61] provides a predictable virtual fabric. It utilizes SmartNICs or NetFPGAs at the endpoints and collaborates with programmable switches to build the system. Tiara [63] achieves high-performance stateful L4 load balancing, capable of high throughput and concurrency. Its system architecture comprises three layers: programmable switches, FPGAs with HBM, and CPUs, each processing different complexity levels of logic to achieve high performance. ExoPlane [48] follows a similar concept to Tiara, providing a system for offloading functions from switches to SmartNICs, thereby achieving higher scalability. These works employ similar hardware choices but do not follow the same architecture.

3 Overview

Marlin is a high-throughput network tester that supports CC testing. Based on our observations in §2.1, we adopted an architecture that combines programmable switches and FPGA NICs, as shown in Figure 1. In this architecture, the

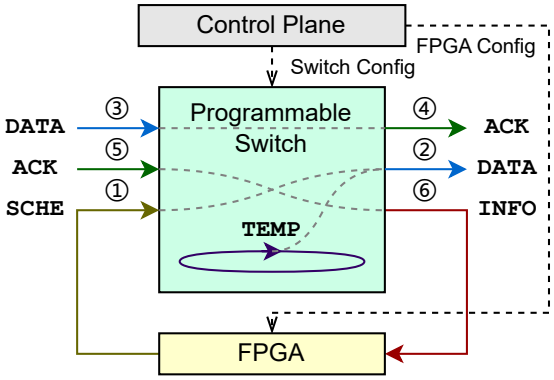


Figure 1. The system architecture and workflow of Marlin.

programmable switch is responsible for processing and generating high-volume traffic, while the FPGA NIC handles CC algorithm execution and traffic scheduling.

In this section, we will first introduce various types of packets in Marlin (§3.1). Then, we will describe the workflow of Marlin and illustrate the lifecycle of each packet (§3.2). Finally, we will discuss how the FPGA NIC and programmable switch work together to amplify throughput (§3.3).

3.1 Packet Type

The programmable switch and FPGA NIC are connected via a 100 Gbps cable, communicating by sending packets to each other. As shown in Figure 1, the types of packets in Marlin include:

- **Template packets, denoted as TEMP.** TEMP packets are sent from the control plane to the data plane of the programmable switch. These TEMP packets continuously cycle at the line rate on the data plane. When packet generation is needed, the programmable switch replicates TEMP packets to other ports via multicasting.
- **Data packets, denoted as DATA.** DATA packets are sent from the programmable switch to the tested network. These packets are transformed from replicated TEMP packets. The control plane can control the length of the generated DATA packets by adjusting the length of the TEMP packets. In addition to carrying data, DATA packets also carry the packet sequence number (PSN) or other information defined by the user's custom CC, which comes from the FPGA NIC.
- **Acknowledgment packets, denoted as ACK.** ACK packets are acknowledgment packets sent by the programmable switch upon receiving DATA packets. ACK packets do not carry data and have a length of 64 bytes. The information they carry is specified by the CC algorithm. Typically, ACK packets include the PSN of the next expected DATA packet to be received. Moreover, in some CC algorithms, ACK packets may contain other network-related information such as RTT, ECN, *etc.*
- **Information packets, denoted as INFO.** INFO packets are used by the programmable switch to convey flow state information to the FPGA NIC. INFO packets are triggered by the reception of ACK packets by the programmable switch and have a length of 64 bytes. The information they carry comes from the ACK packets but only includes the flow's information and congestion information, such as flow ID, PSN, ECN, and RTT specified by the CC algorithm.
- **Scheduling packets, denoted as SCHE.** SCHE packets are sent by the FPGA NIC to the programmable switch to pass on sending instructions, and they are 64 bytes in length. Based on the CC algorithm and scheduler, the FPGA NIC informs the programmable switch about the next packet to be generated. Thus, SCHE packets carry information about the next packet to be sent, such as flow ID, PSN, *etc.*

3.2 Workflow

Marlin provides a control plane program. Operators can use this program to configure the test, i.e., selecting the CC algorithm, setting CC parameters, choosing the test ports, and determining the number of flows per port. After that, the program generates configurations for the programmable switch and FPGA and then deploys these configurations to them. If a different CC algorithm is chosen, the corresponding firmware is written to the FPGA, and CC parameters are sent to the FPGA's BRAM via drivers. Finally, the control plane notifies the FPGA NIC to start sending traffic.

Measurement functionalities are also implemented on both the programmable switch and FPGA. The control plane can retrieve data such as port rate, flow rate, and packet loss by reading hardware registers in the programmable switch. The control plane can also track CC parameter changes in the FPGA using the fine-grained tracing and logging functionality described in §5.1.

Next, we will detail how Marlin works from a packet perspective, with the step numbers in Figure 1.

Sending. After the control plane notifies the FPGA NIC to start sending traffic, the FPGA begins scheduling flows. The scheduler within the FPGA determines whether a flow can be sent based on the CC algorithm. Information such as flow ID and PSN is written into SCHE packets and sent to the programmable switch ①. Upon receiving SCHE packets, the programmable switch writes the information from the SCHE packets into a queue implemented using registers, and then discards the SCHE packets.

TEMP packets continuously circulate at line rate within the programmable switch and are multicast to other ports. The TEMP packets after multicast are by default discarded

by the deparser. However, TEMP packets that retrieve information from this queue will not be discarded. The programmable switch will modify the fields of these TEMP packets based on this information, transforming them into DATA packets and sending them to the network under test ②.

Receiving. As the receiver, Marlin needs to process DATA packets ③. The method of handling DATA packets varies depending on the specific CC algorithm employed. Taking the traditional TCP as an example, the programmable switch updates the receive window by reading the PSN of the DATA packet. Additionally, the programmable switch can generate ACK packets by truncating DATA packets to 64 bytes and rewriting their header fields ④. This process can be fully implemented in the programmable switch.

Congestion signal processing. The ACK packets generated at the receiver are sent back to Marlin after passing through the tested network ⑤. Since both ACK and INFO packets are 64 bytes, the programmable switch can directly extract congestion information from the ACK packets and reassemble them into INFO packets, which are then sent to the FPGA NIC ⑥. Additionally, other packets containing congestion information, such as congestion notification packets (CNP) in DCQCN [67], are also uniformly encapsulated as INFO packets. Finally, the FPGA NIC updates the CC parameters based on the information conveyed in the INFO packets.

3.3 Throughput Amplifying

To achieve maximum throughput, the FPGA NIC sends SCHE packets to the programmable switch at the line rate, with a sending frequency of about 148.8 Mpps. In the case of an MTU of 1024 bytes (RoCE MTU under default Ethernet MTU [13]), the maximum sending frequency of DATA packets from a single 100 Gbps port of the switch is about 11.97 Mpps. Therefore, the FPGA NIC can schedule 100 Gbps× floor(148.8/11.97) = 1.2 Tbps of DATA packets, saturating 12×100 Gbps ports.

Furthermore, if the MTU is increased to 1518 bytes, the maximum sending frequency of DATA packets from a single port of the switch is about 8.127 Mpps. Therefore, the theoretically achievable throughput of Marlin can reach 100 Gbps × floor(148.8/8.127) = 1.8 Tbps, filling up 18×100 Gbps ports. However, since a single pipeline of ASICs like Intel Tofino supports at most 16 ports of 100 Gbps each, and Marlin requires some ports for traffic amplification, a single pipeline cannot achieve the ideal throughput of 1.8 Tbps. We will discuss the port allocation within the programmable switch and explain the maximum throughput supported by Marlin in §4.3.

4 Programmable Switch Design

The fundamental function of Marlin is to generate traffic with CC behaviors. However, as introduced in §3, the traffic

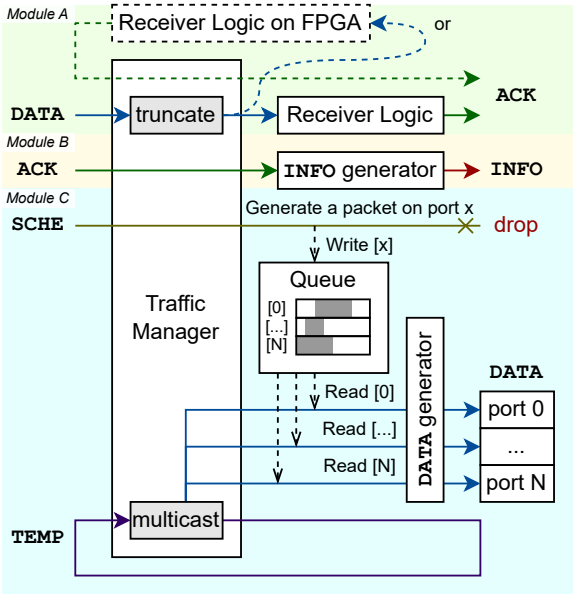


Figure 2. Modules and data paths of the programmable switch.

in Marlin is triggered by the FPGA NIC using SCHE packets. This approach is different from that used in previous works, where traffic is self-generated. Consequently, it seems that the programmable switch has less work to do. However, introducing the FPGA NIC did not simplify the programmable switch design. Instead, it introduced many challenges in their collaboration.

In this section, we first discuss the module organization on the programmable switch (§4.1). Then, we detail how the programmable switch receives SCHE packets and generates traffic (§4.2). Finally, we describe the port allocation of the programmable switch (§4.3).

4.1 Modules

In the programmable switch portion of Marlin, there are three modules, as illustrated in Figure 2.

Module A is responsible for executing the receiver logic, processing received DATA packets, and returning ACK packets. This corresponds to steps ③ and ④ in §3.2. It is important to note that the receiver logic for some CC algorithms might be too complex to be implemented in the programmable switch. However, we can still utilize the architecture of Marlin to address this issue, as illustrated by the dashed portion in Figure 2. We can implement the receiver logic on the FPGA NIC. It processes the truncated DATA packets and then returns ACK packets. In addition, this solution requires an additional port on both the programmable switch and the FPGA NIC.

Module B transforms ACK packets into INFO packets, known as the INFO generator in Figure 2. This process aligns with steps ⑤ and ⑥ in §3.2.

Module C responds to SCHE packets and generates DATA packets. Passing information from SCHE packets to DATA packets involves using queues implemented with programmable switch registers and requires coordination with the FPGA NIC. We will explain this in detail in §4.2.

4.2 Packet Generation

As shown in Figure 2, each egress port in the switch has a dedicated queue that stores metadata for the DATA packets to be generated, such as flow id and packet sequence numbers. Each flow is assigned to a specific egress port. When a SCHE packet arrives at the egress, its metadata is enqueued into the queue corresponding to the designated output port. TEMP packets continuously circulate at line rate within a loopback port and are multicast to all egress ports used for sending. After being multicast, a TEMP packet attempts to dequeue metadata from the queue associated with the port it reaches. If it succeeds, the TEMP packet uses the metadata to restore the DATA packet content and sends it out. If the queue is empty, the TEMP packet is discarded. There are several design considerations to ensure efficient and correct packet generation.

The queues are implemented using a register array, with three additional registers: header, tail, and length. Due to the hardware limitations, each packet can only perform one simple operation on a register. Consequently, once metadata is dequeued, it cannot be re-enqueued. Additionally, TEMP packets cannot be rerouted to other ports once they enter the egress stage. If there were only one queue shared by all egress ports, a TEMP packet might accidentally dequeue metadata meant for a different port, leading to incorrect packet transmission.

Placing the queues at the egress is essential. If the queues were located at the ingress, the DATA packets generated after dequeuing metadata would need to be forwarded to the correct output port corresponding to each flow. To support a throughput of 1.2 Tbps, an additional 12 ports would be needed for TEMP packet loopback to generate the DATA packets, which would significantly increase the cost.

Since each SCHE packet results in a corresponding DATA packet, queue overflow would lead to lost packets that should have been sent, which is unacceptable. At line rate, SCHE packets arrive at a rate of 148.8 Mpps, while a single port can only generate DATA packets at a rate of 8.127 Mpps. To avoid queue overflow, the FPGA NIC needs to identify the port associated with each flow and ensure that the scheduling frequency for each port does not exceed 8.127 Mpps. We will detail this design in §5.3.

4.3 Port Allocation

We use the programmable switch with Intel Tofino ASIC as an example to illustrate how to allocate ports. Since registers are not shared across different pipelines, we allocate ports on a per-pipeline basis. As analyzed in §3, with an MTU of 1024 bytes, each 100 Gbps SCHE packet can generate 1.2 Tbps of DATA traffic. Therefore, the optimal allocation is shown in Figure 3. Within one pipeline, there are a total of 16 ports, with 12 ports used for sending or receiving test traffic to and from the tested network. Additionally, one port is needed, with the ingress direction for receiving SCHE packets from the FPGA NIC, and the egress direction for sending INFO packets to the FPGA NIC. The enqueue operation for SCHE packets is performed on another port of the egress pipeline. Finally, a loopback port is needed for cycling TEMP packets.

In summary, one pipeline with one 100 Gbps port of an FPGA NIC can generate 1.2 Tbps of CC traffic. It is important to note that we have not fully utilized all 16 ports of a pipeline. The reserved one port can be used to forward truncated DATA packets to the FPGA NIC for executing the receiver logic. Moreover, when the MTU is greater than 1072 bytes, 100 Gbps SCHE packets can generate 1.3 Tbps of DATA traffic, thereby fully utilizing all ports of a pipeline.

5 FPGA Design

The functionality of FPGA NICs is more complex compared to programmable switches. It is responsible for handling the sender logic, which involves executing the CC algorithm based on received INFO packets and scheduling traffic by sending SCHE packets. We will explain this implementation in four steps.

First is the basic transport functionalities of FPGA NICs (§5.1). Based on this, we consider how to enable the FPGA NIC to schedule traffic at line rate (§5.2). Then, we discuss the solution for the FPGA NIC to resolve read-write conflicts caused by processing burst INFO packets (§5.3). Finally, we illustrate the structure of the CC algorithm module to achieve high scalability (§5.4).

5.1 Basic Process

We will use the reception of an INFO packet as an example to describe the basic workflow of the FPGA NIC and introduce several related modules, as shown in Figure 4 Step A.

When the FPGA NIC receives an INFO packet, it first parses it, extracting the flow ID, PSN, and other information related to the CC algorithm (such as NACK and ECN), and generates a reception event, which is then placed into the RX FIFO. After the CC algorithm module retrieves the reception event from the RX FIFO or a timeout event from the event generator, it obtains the CC parameters of the flow from the BRAM based on the flow ID. Next, the CC algorithm module executes the CC algorithm implemented through HLS (§5.4), updates CC parameters, and determines whether new

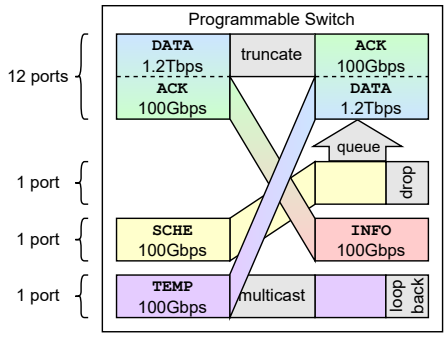


Figure 3. Port allocation on programmable switches.

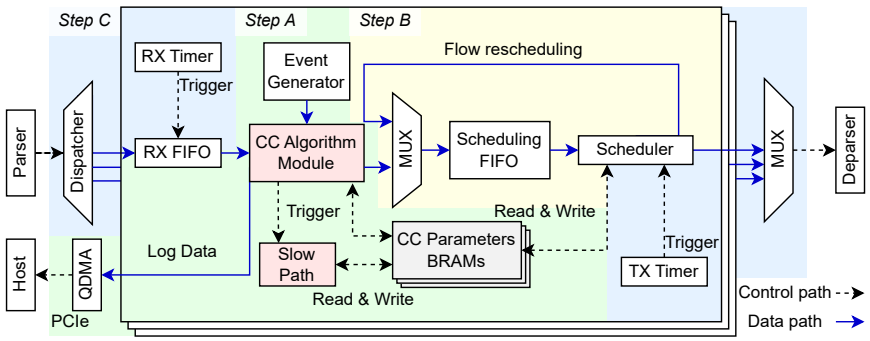


Figure 4. Modules and data paths of the FPGA NIC.

scheduling events need to be generated. These scheduling events could be for the normal transmissions or for retransmissions. These events pass all the scheduling information to the deparser for generating SCHE packets.

Fine-grained logging module. We allow the logged information to be uploaded to the host via Queue-based Direct Memory Access (QDMA). The logging logic is implemented within the CC algorithm module, with each computation capable of logging 16B of data and a timestamp derived from a 322 MHz hardware clock. For ease of use, we chose to aggregate the logged content and upload it to the host in the form of 1024B packets, with logging performance matching the host's DPDK performance.

CC parameters. The CC parameters in Figure 4 are composed of multiple BRAMs due to differing read and write requirements for these parameters by three components, the CC algorithm module, slow path (§5.4), and scheduler (§5.2). These BRAMs all use the flow ID for addressing and will be read and written by one of the three components, while the other two will perform read-only operations. Therefore, Simple Dual-Port RAM is used for implementation.

5.2 Line-Rate Scheduling

In the basic process, the CC algorithm module may not always generate a scheduling event, for example, when it encounters window or rate limitations. Moreover, even if it can generate one, a single flow can only trigger one packet in transit on the link. In order to transmit as many eligible packets as possible, we need to circulate scheduling events, as shown in Figure 4 Step B.

Rescheduling events. After the CC algorithm module generates a scheduling event, we first enqueue it into the scheduling FIFO. The scheduler will fetch the scheduling event from the scheduling FIFO and reevaluate whether to schedule the transmission based on the congestion window or rate. If so, besides triggering the transmission of SCHE packets, this scheduling event will be reinserted into the scheduling FIFO without passing through the CC algorithm module again. Although there is no difference between the recycled scheduling events and the original ones, we will refer to such events as rescheduling events to distinguish them from the latter.

It is noteworthy that rescheduling events actually represent currently active flows. If a flow lacks rescheduling events, it indicates that the flow is inactive. Otherwise, the INFO packet of that flow will trigger scheduling by the CC algorithm module, adding its scheduling event back to the scheduling FIFO. This implies that there is no need for duplicate scheduling events for the same flow in the scheduling FIFO. This approach also helps prevent scheduling FIFO overflow, ensuring that it can record all active flows and guarantee fair scheduling. Additionally, this entire loop only takes six clock cycles, much less than the time required for scheduling a single flow at line rate, ensuring that the rescheduling events have entered the scheduling FIFO before the next scheduling. For high-priority events such as retransmission and timeouts, another FIFO is utilized to prioritize scheduling.

Finally, we discuss the principles of this mechanism under two scenarios. Firstly, if the link bandwidth is not fully utilized, the scheduling FIFO may not necessarily receive scheduling events from the CC algorithm module within each scheduling period. In this case, the scheduling FIFO acquires rescheduling events from the scheduler, gradually occupying link bandwidth. Secondly, if the link bandwidth is fully utilized, the scheduling FIFO will invariably receive scheduling events from the CC algorithm module within each scheduling period. At this point, the uniqueness of events in the scheduling FIFO can ensure fair scheduling of flows.

Scheduler. In addition to facilitating the circulation of scheduling events, another function of the scheduler is to ensure the correctness of rescheduling. Without the scheduler, the rescheduling events would continuously trigger the transmission of SCHE packets. Furthermore, the PSN of these SCHE packets triggered by rescheduling events also needs to be obtained through the scheduler.

Another approach would be to return rescheduling events to the CC algorithm module. We do not adopt this method because the CC algorithm module needs to process INFO packets at line rate, and parallel processing of rescheduling events would increase the complexity of CC algorithm module. The separated design allows the scheduler to only update the sending window or target rate. The CC algorithm module can then focus on processing INFO packets.

5.3 Packet Frequency Control

Packet frequency control is used to prevent packet loss within Marlin and to facilitate the cooperation between the programmable switch and the FPGA NIC. This involves controlling the ingress direction and egress direction separately, as shown in Figure 4 Step C.

Ingress Direction. There are some read-modify-write (RMW) operations in the CC algorithm module. These RMW operations typically take several clock cycles to complete. For the FPGA NIC with an internal clock of 322 MHz, the INFO packets to be processed may arrive at a rate of 148 Mpps. If some RMW operations take more than two clock cycles, their execution frequency will be less than 107 MHz, which is lower than the packet arrival rate, leading to two choices. First, if we ensure throughput, RMW operations triggered by packets will cause read-write conflicts in CC parameters, leading to incorrect execution of the CC algorithm. Second, if we ensure the atomicity of RMW operations, packets will have to wait for the RMW operation to complete, causing a drop in throughput.

We note that for a single flow, the average arrival rate of INFO packets is much lower than the execution frequency of RMW operations. Taking an MTU of 1518 bytes as an example, the average arrival rate of INFO packets for a single flow is 8.127 Mpps. In this case, RMW operations are allowed to take a maximum of 40 clock cycles. Therefore, we can let INFO packets entering the FPGA join different RX FIFOs according to the port they arrive at the programmable switch. At this point, the average rate of INFO packets in each RX FIFO is 8.127 Mpps. However, when INFO packets from the same flow arrive at line rate, such as when DPDK sends ACKs in bursts, the peak packet rate can spike to 148 Mpps. To handle this, we also need an RX timer to control the rate at which these INFO packets are submitted to the CC algorithm module. When the RX timer is set to 8.127 Mpps, the CC algorithm module can allow RMW operations to consume 40 clock cycles without causing read-write conflicts.

Egress Direction. As mentioned in §4.2, the SCHE packets sent to the programmable switch cannot exceed 8.127 Mpps per port. Otherwise, it would lead to queue overflow in the programmable switch, resulting in false packet losses. The solution is similar to the ingress direction frequency control. We allocate one scheduling FIFO and one scheduler for each port. Each scheduling FIFO stores scheduling events for flows on a single port, and each scheduler retrieves scheduling events from the FIFO based on the TX timer we have configured. These events are then passed to the multiplexer (MUX). The MUX collects scheduling events from the schedulers and sends them to the deparser to generate SCHE packets.

It is important to note that the duration set by the RX timer must be less than or equal to the duration set by the TX timer. Otherwise, it will cause the RX FIFO to overflow and lead to incorrect execution of the CC algorithm.

5.4 Scalable CC Algorithm Module

The CC algorithm module, implemented through Vivado HLS [10], functions to run the CC algorithm. HLS enables users to implement the CC algorithm in C++, which is more convenient than traditional approaches like Verilog [20].

Programming Interface. When writing an HLS program, we can write linear processing logic, and the compiler will automatically pipeline it. The parameters of the HLS entry function are shown in Table 3, with read-only input parameters and write-only output parameters, both being struct pointer variables. The intrinsic variable is immutable and contains information such as the triggered event type, window-related or rate-related data. The customized variable, with a total length of 64B, is customized by the user and stores the parameters of CC.

Slow Path. As mentioned in §5.3, RMW operations are allowed to take a maximum of 40 clock cycles. RMW operations are limited to 40 clock cycles, enabling 16-bit division and several 32-bit multiplications but still lack programmability. We observed that in CC algorithms like DCTCP [24] and Timely [54], time-consuming logic only runs once per RTT or every few packets, so we introduced a Slow Path. The CC algorithm module triggers events for the Slow Path to update specific parameters (Table 3). With microsecond-level RTTs in data centers, the Slow Path has hundreds of clock cycles, improving performance. For example, using the Slow Path to update `alpha` in DCTCP allows increasing division and `alpha` precision from 16-bit to 32-bit.

6 Implementation

The hardware used by Marlin includes a 32×100 Gbps ports P4 programmable ethernet switch with 2 pipelines [9], as well as a Xilinx Alveo U280 data center accelerator card [3]. The programmable switch and FPGA NIC are connected via a single 100 Gbps link.

Programmable switch. The implementation of the programmable switch includes the control plane and data plane. The control plane is written in Python and interacts with the data plane tables through gRPC. Additionally, the control plane sends TEMP packets to the data plane via PCIe. The control plane consists of 1,103 lines of code. The data plane implementation is written in P4 [30], consisting of 1,581 lines of code. It utilizes 58/960 of SRAM resources, 3/288 of TCAM resources, spanning across 4 stages.

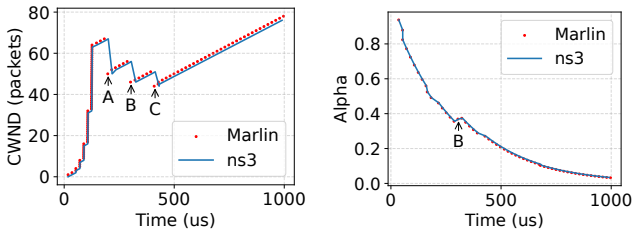
Table 3. CC Algorithm Module Programming Interface.

Variable		Description
INPUT	intr-var	struct for intrinsic variable
	- psn	packet sequence number
	- cwnd/rate	congestion window size or rate
	- una	psn of the next unacknowledged packet
	- nxt	psn of the next packet to be sent
	- flags	flag bits, including ack, ecn, etc
	- evt-typ	including packet reception and timeout
	- prb-rtt	probed round-trip time
	- tstamp	timestamp of receiving this event
	cust-var	user-defined variables for CC modules
	slwpth-var	variables modified in Slow Path
OUTPUT	int-var	struct for intrinsic variable
	- rtx-psn	the psn of the retransmitted packet
	- cwnd/rate	congestion window size or rate
	- rst-timer	reset specific timer
	cust-var	consistent with the input parameters
	slwpth-evt	events that trigger the Slow Path
	log-content	upload logged content to the host

Table 4. Summary of lines of code and clock cycles required for implementing different CC algorithms, along with the resource usage (as a percentage) for both the CC module and the entire project.

	LoC	clk	CC Module		Total		
			LUT	FF	LUT	FF	BRAM
Reno	156	2	1.1	0.7	10	11	59
DCTCP	175	24	3.5	2.1	13	12	63
DCQCN	98	6	1.4	0.9	12	10	46

FPGA NIC. The FPGA NIC is implemented based on the Xilinx OpenNIC shell [4] and synthesized using Vivado 2021.2, with all modules operating at 322 MHz. We have implemented window-based Reno and DCTCP, as well as rate-based DCQCN. Table 4 presents the number of lines of code written for each algorithm's CC module, excluding fixed formats, the number of clock cycles required for algorithm execution, the LUT and FF resources consumed by the CC module and the entire project, as well as the total BRAM resources consumed. All resource consumption is given as a percentage of the total resources on the FPGA card. Through an analysis similar to that in §5.3, when the template packet size is 1024B, the CC module has 27 clock cycles for processing. The implemented DCTCP here performs one 16-bit division, two 32-bit multiplications, and several additions, subtractions, and shifts on the critical path, all while meeting the clock constraints.



(a) Congestion Window **(b)** Alpha

Figure 5. Comparison of CWND and Alpha changes between Marlin and ns3 in CC Module Correctness Test

7 Evaluation

We evaluate Marlin from three perspectives: First, we verify the correctness of the implementation. In §7.1, we validate basic TCP state transitions, window and parameter updates, and demonstrate the fine-grained tracing capabilities. In §7.2, we assess the correctness of the scheduling mechanism designed in §5.2. In §7.3, we evaluate flow fairness under congestion, further confirming the correctness of the implemented CC algorithms. Next, we verify that Marlin's implementation of the DCQCN algorithm shows high fidelity when compared to commercial NICs (§7.4). Finally, we activate the maximum number of concurrent flows supported by Marlin and subject them to competition over a bottleneck link, demonstrating that Marlin can reliably handle and correctly test up to 65,536 flows (§7.5).

7.1 CC Module Correctness Test

In the correctness testing of the CC module, we used Marlin to generate a single TCP DCTCP flow and tested the consistency of its parameter changes with ns3 [1] simulation results. The sender and receiver are connected with a programmable switch via twelve 100Gbps links each, and this topology was used in all experiments except §7.4. For the sake of determinism and interpretability, we deliberately introduced packet loss events and modified ECN markings at specific points in the switch's transmission to the receiver. Using Marlin's data tracing capabilities, we monitored every change in the flow's congestion window (`cwnd`) and `alpha` parameter. In this experiment, the initial `ssthreshold` was set to 64, and the initial `cwnd` was set to 1, with other parameters matching ns3 defaults.

The changes in window size and `alpha` for both Marlin and the ns3 simulation are shown in Figure 5. The flow initially undergoes a slow start phase, where the window grows exponentially. After reaching the initial `ssthreshold` of 64, it enters congestion avoidance, with the window growing linearly. At points A and C in the figure, we introduced packet loss, while at point B, we marked some packets with ECN. After detecting these events, the algorithm enters Fast Recovery and Congestion Window Reduced phases, then updates

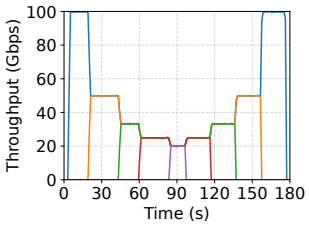


Figure 6. Single port multi-flow scheduling test.

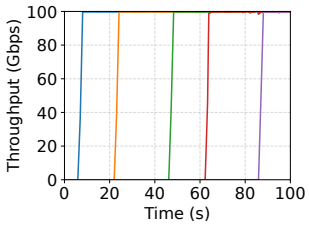


Figure 7. Multi-port scheduling test.

the parameters and returns to congestion avoidance. The `alpha` parameter, which reflects the degree of network congestion, decreases gradually throughout the process, except at point B where ECN was marked.

This experiment validated the correctness of the CC module implementation and demonstrated its fine-grained tracing capabilities.

7.2 Flow Scheduling Test

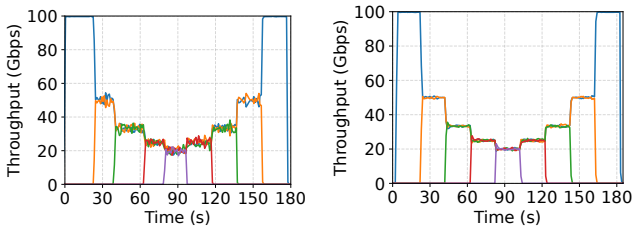
Since we are only testing the flow scheduling mechanism, the congestion control algorithm used in these experiments does not affect the results. For consistency, we employed the DCTCP algorithm in this part.

Single port multi-flow scheduling. First, we had the tester initiate multiple flows on a single port and send them concurrently to the receiver. The intermediate switch directly forwards this traffic, equivalent to a pass-through link. In this setup, the tester acts as a single host, sending multiple flows to another host via a single port. Since these flows do not encounter congestion, according to Marlin's scheduling mechanism, we expected these flows to evenly share the port bandwidth. As shown in Figure 6, the rates of each flow remain consistent, and the total throughput reaches near 100 Gbps. This indicates that the scheduling mechanism of the scheduling FIFO and scheduler is fair.

Multi-port scheduling. Next, we had the tester initiate one flow on each port, and these flows are forwarded one-to-one to different ports on the receiver via the intermediate switch. In this setup, the tester acts as multiple hosts, each port sending traffic to other different hosts. This experiment can be used to verify that the scheduling on each port does not interfere with each other, and we expected these flows to individually occupy the bandwidth of each port. As shown in Figure 7, the rate of each flow can reach 100 Gbps, aligning with our expectations.

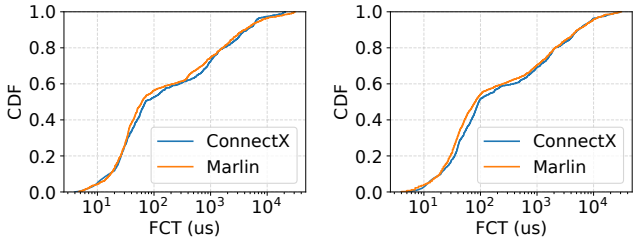
7.3 Congestion Test

In this experiment, we had the tester sequentially initiate one flow on each port and then sequentially terminate one flow on each port. These flows are forwarded through the intermediate switch to the same destination port, thereby creating congestion. We tested the performance of DCTCP and DCQCN in this scenario. For DCTCP, the parameter `ssthreshold` was set to a value that allows the single-flow throughput to quickly reach the port bandwidth. The parameters for DCQCN were configured according to the documentation provided by NVIDIA [6].



(a) DCTCP **(b)** DCQCN

Figure 8. Multi-flow performance under congestion conditions.



(a) 2 cast 1 **(b)** 3 cast 1

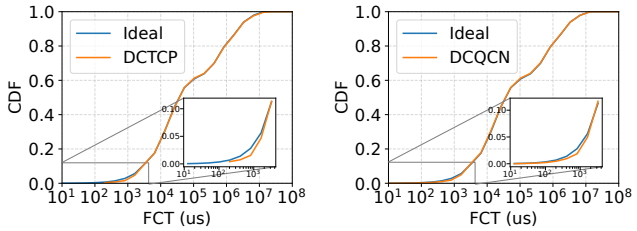
Figure 9. CDF of FCT in n-cast-1 scenario, Marlin vs ConnectX-5. Each of n ports runs 5 flows.

According to the design of DCTCP and DCQCN, these flows will relinquish bandwidth after experiencing congestion and eventually evenly share the bandwidth at the bottleneck. Moreover, after terminating some flows, the remaining active flows will gradually occupy the available bandwidth. The results, as shown in Figure 8, indicate that both DCTCP and DCQCN are capable of evenly sharing the bandwidth of 100 Gbps across these flows and evenly sharing the available bandwidth once some of them release their bandwidth. DCTCP exhibits more obvious throughput oscillations compared to DCQCN. This experiment demonstrates the correctness of both the CC algorithm module and the scheduling mechanism.

7.4 Flow Fidelity Test

In the flow fidelity experiments, we set up an n-cast-1 scenario to compare the flow completion time (FCT) performance of our implemented DCQCN algorithm with the algorithm implemented in commercial RDMA NICs.

Three hosts equipped with Mellanox ConnectX-5 are configured with recommended parameters as mentioned above. Each NIC has two ports, with port 0 connected to Switch



(a) DCTCP **(b)** DCQCN

Figure 10. WebSearch FCT for a maximum of 65,536 concurrent flows.

A and port 1 connected to Switch B. Switch A and B are connected with one link. All links are 100Gbps.

We developed a FCT testing tool using the verbs API [15], with data transmitted through RDMA Write operations. Five Queue Pairs are created on each host, and use WebSearch [24] traffic model to generate flows. A new flow is initiated immediately after the completion of the previous one. After testing the 2-cast-1 and 3-cast-1 scenarios, we replaced the hosts with our tester, with each port corresponding to the original port, and ran the same tests with identical parameters. After each flow completes, the FPGA NIC will calculate the completion time of that flow and send it to the control plane program for statistics.

As depicted in Figure 9, our implementation demonstrated consistent performance with commercial NICs in terms of flow completion time. Due to the proprietary nature of the DCQCN implementation in commercial NICs, it was not possible to achieve complete equivalence. In summary, this experiment validated the fidelity of our tester's multi-flow scheduling within a single port and multi-flow competition compared to commercial NICs.

7.5 Comprehensive Test

Finally, we evaluated the performance of the tester in real-world scenarios. The tester run both DCTCP and DCQCN algorithms with maximum number of concurrent flows and then measured the flow completion time (FCT). The parameters for these two CC algorithms are consistent with the previous experiment.

The tester uses WebSearch traffic model to generate flows. To maintain the number of concurrent flows and maximize the throughput of the tester, a new flow will be created based on the chosen traffic model after each flow completes. Therefore the arrival time of the flow is determined by the completion time of the previous flow, rather than following a Poisson distribution. As a reference, we calculated the ideal FCT under this scheduling, where each flow evenly shares the bandwidth at all times, and reflected it in Figure 10 to validate the correctness of our flow generation.

The tester is designed to support a maximum of 65,536 concurrent flows and achieve a throughput of 1.2 Tbps. The cumulative distribution of FCT for each CC algorithm and traffic model measured by the tester is shown in Figure 10. Both the DCTCP and DCQCN algorithms perform worse than the ideal algorithm, and DCQCN shows a significant improvement in performance compared to DCTCP when sending short flows, as expected. We measured that each port can send traffic at close to line rate, with a total throughput close to 1.2 Tbps.

8 Discussion

The complexity of implementing CC algorithms on FPGA NIC. The implementation of CC algorithms on FPGA NIC is primarily constrained by the target single-flow throughput. For CC algorithms like DCQCN used in data center networks, the tester needs to provide line-rate throughput for a single flow on a single port. Taking a single flow with an MTU of 1518 bytes as an example, achieving a throughput of 100 Gbps requires the FPGA NIC to process at a rate of 8.127 Mpps. At an internal FPGA clock frequency of 332 MHz, RMW operations must be completed in fewer than 40 clock cycles. Simple operations consisting of multiple additions and subtractions can be completed within a single clock cycle, and multiplication typically requires several clock cycles. The clock cycle is primarily consumed by division operations. For example, after optimizing the cubic root calculation using lookup tables, Cubic still requires around 100 clock cycles to process a single packet. However, these types of algorithms can still operate properly by reducing the packets-per-second (PPS) per flow and leveraging multiple flows to achieve line rate. Fortunately, most CCAs in data center networks do not require such complex operations.

Scalability. The number of flows supported by Marlin depends on the size of the on-chip memory resources and how much data the CCA needs to store for each flow. In this paper, we utilized 72 Mb of BRAM to support 65,536 flows, and there remains an additional 276 Mb of URAM available for scaling to even more flows.

Alternatives to FPGA NIC. The primary reason for using FPGA NIC in Marlin is its ability to provide extremely high packet processing rates. If some performance goals of the tester can be relaxed, other alternatives with lower cost and lower entry barriers can be considered. For scenarios where traffic characteristics involve high concurrency rather than high throughput for individual flows, devices based on many-core architectures can be used to achieve high throughput for the entire tester through concurrent processing of multiple flows. Additionally, for scenarios where higher packet processing latency is acceptable, it may be possible to use GPUs to execute CC algorithms by merging multiple congestion information to increase concurrency in processing.

Feasibility of implementing receiver-driven CC algorithms. The design goal of Marlin is to validate that the combination of programmable switches and FPGA NICs can generate CC traffic at the Tbps level. The FPGA NIC performs the sender-side algorithm, controlling the single-flow rate through a congestion window or sending rate (such as Reno and DCQCN). Receiver-driven CC algorithms can also be implemented using programmable switches and FPGA NICs, as indicated by the dashed lines in Figure 2. Common receiver-driven algorithms control the sending rate at the receiver end, which can still be achieved with the FPGA design shown in Figure 4.

9 Conclusion

We present Marlin, the first network tester capable of supporting CC deployment effectiveness testing. Marlin leverages a hardware architecture that combines programmable switches and FPGA NICs to generate Tbps-level CC traffic. Using a single programmable switch pipeline and one 100 Gbps port of an FPGA NIC, our tester achieved concurrency of 65,536 flows and a throughput of 1.2 Tbps, while maintaining high fidelity in replicating real-world scenarios and enabling fine-grained tracing.

Acknowledgments

We would like to thank our shepherd, Rishabh Iyer, and the anonymous EuroSys '25 reviewers for all their constructive feedback and comments. This research is supported by the National Key R&D Program of China (2022YFB2901502), the Key Program of Natural Science Foundation of Jiangsu under grant No. BK20243053, the National Natural Science Foundation of China under Grant Numbers 62325205 and 62172204, the Nanjing University-China Mobile Communications Group Co.,Ltd. Joint Institute. Chen Tian is the corresponding author.

References

[1] 2017. NS3 Network Simulator. https://www.nsnam.org/.
[2] 2024. Alveo U200 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u200.html.
[3] 2024. Alveo U280 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.
[4] 2024. AMD OpenNIC Shell includes the HDL source files. https://github.com/Xilinx/open-nic-shell.
[5] 2024. Data Plane Development Kit. https://www.dpdk.org/.
[6] 2024. DCQCN Parameters. https://enterprise-support.nvidia.com/s/article/dcqcn-parameters.
[7] 2024. High Performance Networking Leveraging Community. https://www.dpdk.org/wp-content/uploads/sites/35/2014/09/DPDK-SFSummit2014-HighPerformanceNetworkingLeveragingCommunity.pdf.
[8] 2024. Intel Tofino. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html.
[9] 2024. Intel Tofino 3.2 Tbps, 2 pipelines. https://www.intel.com/content/www/us/en/products/sku/218641/intel-tofino-3-2-tbps-2-pipelines/specifications.html.
[10] 2024. Introduction to FPGA Design with Vivado High-Level Synthesis. https://docs.amd.com/v/u/en-US/ug998-vivado-intro-fpga-design-hls.
[11] 2024. iPerf. https://iperf.fr/.
[12] 2024. Keysight. https://www.keysight.com/us/en/products/network-test.html.
[13] 2024. MTU considerations for RoCE based applications. https://enterprise-support.nvidia.com/s/article/mtu-considerations-for-roce-based-applications.
[14] 2024. Pktgen. https://github.com/pktgen/Pktgen-DPDK.
[15] 2024. rdma-core. https://github.com/linux-rdma/rdma-core.
[16] 2024. Scapy. https://scapy.net/.
[17] 2024. Spirent. https://www.spirent.com/products/testcenter-ethernet-ip-cloud-test.
[18] 2024. Trafgen. http://netsniff-ng.org/.
[19] 2024. TRex. https://trex-tgn.cisco.com/.
[20] 2024. Verilog. https://www.verilog.com/.
[21] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. 2022. ABM: active buffer management in datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.
[22] Vamsi Addanki, Oliver Michel, and Stefan Schmid. 2022. PowerTCP: Pushing the Performance Limits of Datacenter Networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[23] Albert Gran Alcoz, Martin Strohmeier, Vincent Lenders, and Laurent Vanbever. 2022. Aggregate-Based Congestion Control for Pulse-Wave DDoS Defense. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.
[24] 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 the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.
[25] Gianni Antichi, Charalampos Rotsos, and Andrew W. Moore. 2015. Enabling Performance Evaluation Beyond 10 Gbps. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.
[26] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[27] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkipati. 2023. Bolt: Sub-RTT Congestion Control for Ultra-Low Latency. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[28] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical Delay-Based Congestion Control for the Internet. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[29] Junehyuk Boo, Yujin Chung, Eunjin Baek, Seongmin Na, Changsu Kim, and Jangwoo Kim. 2023. F4T: A Fast and Flexible FPGA-Based Full-Stack TCP Acceleration Framework. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.
[30] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (jul 2014), 87–95. doi:10.1145/2656877.2656890
[31] Alessio Botta, Alberto Dainotti, and Antonio Pescapé. 2012. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks* 56, 15 (2012), 3531–3547. doi:10.1016/j.comnet.2012.02.019

[32] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. 1994. TCP Vegas: new techniques for congestion detection and avoidance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[33] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. 2022. dcPIM: near-optimal proactive datacenter transport. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[34] 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 Annual Technical Conference (ATC)*.

[35] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, September-October (2016), 20 – 53. http://queue.acm.org/detail.cfm?id=3022184

[36] 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 Symposium on Networked Systems Design and Implementation (NSDI)*.

[37] Wenxue Cheng, Kun Qian, Wanchun Jiang, Tong Zhang, and Fengyuan Ren. 2020. Re-architecting Congestion Management in Lossless Ethernet. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[38] Inho Cho, Keon Jang, and Dongsu Han. 2017. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[39] G. Adam Covington, Glenn Gibb, John W. Lockwood, and Nick Mckeown. 2009. A Packet Generator on the NetFPGA Platform. In *IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

[40] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *Proceedings of the Internet Measurement Conference (IMC)*.

[41] Paul Emmerich, Sebastian Gallenmüller, Gianni Antichi, Andrew W. Moore, and Georg Carle. 2017. Mind the Gap - A Comparison of Software Packet Generators. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*.

[42] S. Floyd and T. Henderson. 1999. RFC2582: The NewReno Modification to TCP's Fast Recovery Algorithm.

[43] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.* 42, 5 (jul 2008), 64–74. doi:10.1145/1400097.1400105

[44] 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 the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[45] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. 2020. Aeolus: A Building Block for Proactive Transport in Datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[46] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs.

[47] T. Khan, S. Rashidi, S. Sridharan, P. Shurpali, A. Akella, and T. Krishna. 2022. Impact of RoCE Congestion Control Policies on Distributed Training of DNNs. In *IEEE Symposium on High-Performance Interconnects (HOTI)*.

[48] Daehyeok Kim, Vyas Sekar, and Srinivasan Seshan. 2023. ExoPlane: An Operating System for On-Rack Switch Resource Augmentation. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[49] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[50] 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 Symposium on Networked Systems Design and Implementation (NSDI)*.

[51] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: high precision congestion control. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[52] Hwijoon Lim, Jaehong Kim, Inho Cho, Keon Jang, Wei Bai, and Dongsu Han. 2023. FlexPass: A Case for Flexible Credit-based Transport for Datacenter Networks. In *Proceedings of the European Conference on Computer Systems (EuroSys)*.

[53] Shiyu Liu, Ahmad Ghalayini, Mohammad Alizadeh, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. 2021. Breaking the Transience-Equilibrium Nexus: A New Approach to Datacenter Packet Transport. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[54] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[55] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: a receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[56] Sudarsanan Rajasekaran, Manya Ghobadi, Gautam Kumar, and Aditya Akella. 2022. Congestion control in machine learning clusters. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*.

[57] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference (ATC)*.

[58] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W. Moore. 2012. OFLOPS: An Open Framework for OpenFlow Switch Evaluation. In *Passive and Active Measurement (PAM)*.

[59] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa Ammar, Ellen Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, and Amin Vahdat. 2020. Annulus: A Dual Congestion Control Loop for Datacenter and WAN Traffic Aggregates. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[60] Guangda Sun, Mingliang Jiang, Xin Zhe Khooi, Yunfan Li, and Jialin Li. 2023. NeoBFT: Accelerating Byzantine Fault Tolerance Using Authenticated In-Network Ordering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[61] Shuai Wang, Kaihui Gao, Kun Qian, Dan Li, Rui Miao, Bo Li, Yu Zhou, Ennan Zhai, Chen Sun, Jiaqi Gao, Dai Zhang, Binzhang Fu, Frank Kelly, Dennis Cai, Hongqiang Harry Liu, and Ming Zhang. 2022. Predictable VFabric on Informative Data Plane. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[62] Weitao Wang, Masoud Moshref, Yuliang Li, Gautam Kumar, T. S. Eugene Ng, Neal Cardwell, and Nandita Dukkipati. 2023. Poseidon:

Efficient, Robust, and Practical Datacenter CC via Deployable INT. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[63] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei Geng, Tao Feng, Feng Ning, Kai Chen, and Chuanxiong Guo. 2022. Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[64] Yiran Zhang, Yifan Liu, Qingkai Meng, and Fengyuan Ren. 2021. Congestion detection in lossless networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[65] Yu Zhou, Zhaowei Xi, Dai Zhang, Yangyang Wang, Jinqiu Wang, Mingwei Xu, and Jianping Wu. 2019. HyperTester: high-performance network testing driven by programmable switches. In *Proceedings of the International Conference on Emerging Networking Experiments And Technologies (CoNEXT)*.

[66] Lingjun Zhu, Yifan Shen, Erci Xu, Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, Zhendan Yang, Zhongqing Chen, Wei Lin, Yijun Hou, Rong Liu, Chao Shi, Jiaji Zhu, and Jiesheng Wu. 2023. Deploying User-space TCP at Cloud Scale with LUNA. In *USENIX Annual Technical Conference (ATC)*.

[67] 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 *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*.