Norma: Towards Practical Network Load Testing

Yanqing Chen, *State Key Laboratory for Novel Software Technology, Nanjing University and Alibaba Group;* Bingchuan Tian, *Alibaba Group;* Chen Tian, *State Key Laboratory for Novel Software Technology, Nanjing University;* Li Dai, Yu Zhou, Mengjing Ma, and Ming Tang, *Alibaba Group;* Hao Zheng, Zhewen Yang, and Guihai Chen, *State Key Laboratory for Novel Software Technology, Nanjing University;* Dennis Cai and Ennan Zhai, *Alibaba Group*

https://www.usenix.org/conference/nsdi23/presentation/chen-yanqing

This paper is included in the
Proceedings of the 20th USENIX Symposium on
Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Open access to the Proceedings of the
20th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

Norma: Towards Practical Network Load Testing

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[‡]

[†]*State Key Laboratory for Novel Software Technology, Nanjing University* [‡]*Alibaba Group*

Abstract

Network load tester is important to daily network operation. Motivated by our experience with a major cloud provider, a practical load tester should satisfy two important requirements: **(R1)** stateful protocol customization, and **(R2)** real network traffic emulation (including high-throughput traffic generation and precise rate control). Despite the success of recent load testers, we found they fail to meet both above requirements. This paper presents Norma, a practical network load tester built upon programmable switch ASICs. To achieve the above requirements, Norma addresses three challenges: (1) modeling stateful protocols on the pipelined architecture of the ASIC, (2) generating replying packets with customized payload for stateful protocols, and (3) controlling mimicked traffic in a precise way. Specifically, first, Norma introduces a stateful protocol abstraction that allows us to program the logic of the state machine (*e.g.*, control flow and memory access) on the programmable switch ASIC. Second, Norma proposes a novel multi-queue structure to generate replying packets and customize the payload of packets. Third and finally, Norma coordinates meters and registers to construct a multi-stage rate control mechanism capable of offering precise rate and burst control. Norma has been used to test the performance of our production network devices for over two years and detected tens of performance issues. Norma can generate up to 3 Tbps TCP traffic and 1 Tbps HTTP traffic.

1 Introduction

Understanding whether the network meets expected performance is essential to today's cloud providers, especially for performance-sensitive services such as live streaming and edge cloud games [30, 35, 57]. For example, in edge cloud games, the players complain about their unsmooth feelings if the network latency reaches 50 ms, and cannot play the games when the latency exceeds 100 ms [54].

Network load tester is one of the most important testing tools that checks the performance of network devices by proactively generating various testing packets including different protocols, rates, and traffic patterns [7, 13]. A network load tester could be used by the operator to test the performance of devices, debugging the root causes of packet loss. In a typical network load testing scenario, the tester generates user-defined traffic and sends it to the Device Under Test (DUT). After receiving the testing packets, the DUT processes the traffic and forwards it back to the tester for further processing, such as dropping or replying to the incoming packets. Based on the analysis of the outgoing and incoming traffic, the tester can evaluate the performance of the DUT in multiple aspects, including throughput, latency, and packet loss.

As a major cloud provider, we also deploy load testers in production networks to test pre-online network devices and functions. Load testers have become indispensable for daily network operation tasks, such as performance monitoring, failure troubleshooting, and stress testing. Building a *practical* load tester that works for large-scale cloud networks should satisfy the following important requirements from our network operators.

- **(R1) Stateful protocol customization.** Besides switches and routers which work in a stateless way, cloud networks also have complex and stateful network functions, such as stateful packet filters and L4/L7 load balancers. The protocols used by these network functions might be stateful (*e.g.*, HTTP) or self-defined by the cloud provider. To provide the all-around testing capability, a practical load tester should be able to generate packets with not only stateless protocols (*e.g.*, UDP) but also stateful (*e.g.*, TCP) and customized (*e.g.*, private tunnel protocols) protocols.
- **(R2) Real traffic emulation.** As the scale and single-port bandwidth of cloud networks grow fast, a practical load tester should be able to mimic real, cloud-grade traffic in a cost-effective way: (1) it can generate Tbps-level traffic, and (2) it can create and send precise rate packets with customized payload and burst patterns.

A number of previous efforts have focused on network load testing [7, 9, 13, 18, 24, 31, 32, 47, 53, 55, 59]. While these state-of-the-art systems can work well in principle, in reality in our situation, they fail to simultaneously satisfy the above two requirements (see Table 1). Specifically, software load testers [9, 18, 24, 31, 32, 47] and FPGA-based load testers [60] are unable to generate Tbps-level traffic or control rate precisely (*i.e.*, fail **R2**). On the other hand, hardware load testers (*e.g.*, Keysight [7] and Spirent [13]) can only generate and emulate fixed types of protocols (*i.e.*, unable to support the customized protocols **R1**). Recently, researchers have developed load testers based on programmable switch ASICs [53, 55, 59], which are capable of sending Tbps-level traffic and are customizable. While these pioneer systems have shown the potential to partially solve the above problems, they cannot customize stateful protocols or provide precise rate control, *i.e.*, partially failing **R1** or **R2**.

We, therefore, decided to build a practical load tester to

satisfy our operators' requirements for their daily usage.

Our approach: Norma. This paper presents Norma, a high-performance network load tester, based on the RMT-based[1] programmable switch ASIC. The key idea of Norma is to execute load testing based on template packets derived from tested protocols, which enables Norma to achieve **R1** and **R2** simultaneously. First, template packets continuously loop in the pipeline and can be conditionally replicated to generate testing packets of both stateless and stateful protocols. Operators can flexibly customize headers and payload of template packets; thus, Norma can be used to test various cloud network functions. Second, through controlling rates and patterns of replicating template packets, Norma can generate load testing traffic that faithfully mimics realistic traffic. Besides programmable switch ASIC resource limitations [33, 38, 39, 41, 50–52, 56], building Norma nevertheless requires us to address the following challenges.

Challenge 1: First, RMT-based programmable switch ASICs are unfriendly to modeling stateful protocols (*e.g.*, TCP and HTTP), since the ASIC architecture is implemented as a pipeline that processes packets in a sequential way and only supports accessing states once per packet. However, most stateful protocols need to read and write a state multiple times. This, therefore, makes it difficult to model or customize stateful protocol behaviors on current programmable switch ASICs. On the other hand, testing the performance of stateful protocols (*e.g.*, stateful load balancer, DDoS defense, and ACL) is crucial, which accounts for the majority of our load testing requirements and tasks. To the best of our knowledge, none of the prior work solved this problem. Existing load testers based on programmable switch ASICs like HyperTester [59] can only support stateless protocol customization. To address this challenge, we introduce a new data structure, named *stateful protocol abstraction*, to enable programming the logic of the state machine (including control flow and memory access) on programmable switch ASICs. We construct a state machine framework via the stateful protocol abstraction. In the framework, packets are looped inside the ASIC, and each round corresponds to a step in the state machine of the emulated stateful protocol. We can use this framework to emulate arbitrarily complex protocols (including both stateful and stateless), as long as the hardware resources of the ASIC are sufficient (§4.1 and §4.2).

Challenge 2: Load testers need to reply according to state machines when receiving packets of stateful protocol from DUT. Replying packets involves packet generation with customized payload as well as header modification, which presents the second challenge. The programmable switch ASIC uses PHV[2] resources to add, delete, and modify packet headers and payload. Due to limited PHV resources, the capability of load testers to generate and modify packets with a large payload and statefully complex headers is inherently constrained. To address this challenge, we propose an efficient multi-queue structure based on registers[3] inside the programmable switch ASIC. In this structure, the stateful packet will enqueue to trigger the corresponding type of template packet to dequeue. In this way, Norma supports generating replying packets with customized payloads for most stateful protocols (§4.3 and §4.4).

Challenge 3: The final challenge is that programmable switch ASICs is hard to offer precise control of the packet rate and burst, thus resulting in unrealistic traffic emulation. The above two control capabilities are important requirements of our daily operation and testing; however, we have not seen any of state of the art systems that can achieve the above goals. The rate control relies on the specific hardware named meter; however, in practice, the speed limit of the hardware meter is coarse-grained, *i.e.*, not all target rates can be precisely achieved, which results in an error in the control of packet rate. In addition, the programmable switch ASICs do not support the generation of traffic bursts with given patterns. To address this challenge, we proposed a multi-stage rate control mechanism based on the coordination of meters and registers in the programmable switch ASIC. The meters provide coarse-grained rate control, which will be further tuned by the follow-up registers in a fine-grained manner. In this way, the special requirements of the tester for rate and burst control can be satisfied with great precision (§5).

Norma has been used to test the performance of pre-online devices that would be deployed in our production network for over two years. For example, we used Norma to test the forwarding capability and ARP learning rate of L2/L3 switches and tested stateful gateways by generating L4/L7 flows. Evaluation results show Norma can generate up to 3 Tbps TCP traffic and 1 Tbps HTTP traffic while maximizing the use of pipeline bandwidth. Experiments also show that Norma can achieve precise rate control and burst capability. The relevant rate error does not exceed 0.01% in the worst cases.

Contributions. We make the following contributions:

- It is new for us to implement the stateful responder into the programmable switch ASIC to support stateful protocols. The pipeline-folded switch ASIC and the queue implemented in P4 are the keys to make it possible.
- We propose high-precision packet rate and burst control method. This provides us the ability to reproduce traffic at accurate rates and desired burst patterns.
- We use Norma to test our pre-online devices. Norma is useful for our network developers and operators to find performance issues and system bottlenecks.

[1] RMT (Reconfigurable Match Tables) is a reconfigurable pipeline-based architecture for programmable switch ASICs. Each pipeline consists of a parser, multiple match-action stages, and a deparser [22, 40].

[2] PHV (Packet Header Vector) stores and transits parsed headers or metadata between neighboring stages. More details can be found in [10, 22].

[3] Registers are memory blocks attached to each stage, whose data can be shared by multiple packets across different ports inside a pipeline [10, 22].

Ethics. This work does not raise any ethical issues.

2 Background & Motivation

This section details our operators' requirements and discusses related work.

2.1 Requirements for Production

Based on the experience of our operators, we summarize the requirements of our network load tester in Table 1.

(1) Supporting protocol customization. In cloud networks, traffic can be carried via non-standard private protocols. These protocols are usually defined and experimentally developed by the cloud providers, *e.g.*, QUIC [42] and Multipath QUIC [28, 58], which provide great extensibility of network functions and can be quickly iterated according to the needs of upper-layer applications. These non-standard protocols and traffic are not supported by commercial hardware network testers.

In particular, the majority of protocols we need to test are stateful. The DUT keeps the state of L4/L7 sessions for stateful protocols. For example, an L4 gateway may perform a TCP relay or SYN proxy, and an L7 load-balancer balances the load of the HTTP traffic according to the HTTP header of the first packet. In these cases, the load tester should be able to emulate the establish, transmission, and release processes of a session, and reply to the incoming packets according to the specification of the protocol.

(2) Emulating realistic traffic. During the development and operation of network devices, our operators need to evaluate the device or optimize configurations by emulating realistic traffic. The volume of mixed traffic flows can be as large as O(1 Tbps), or O(1 Gpps) for small packets. It is essential to emulate the traffic similar to the realistic load. We have observed the case that a DUT works well in the experimental development with simple traffic, but suffers from continuous packet drop after it goes online. It is not acceptable for cloud providers.

To test the DUT with emulated traffic under heavy loads, the tester should support sending traffic at the line rate of DUTs. Besides, the tester needs to emulate various traffic patterns and mixed traffic flows for network operators to determine the optimal configuration like hash function, CPU allocation, and queuing policy of devices. The traffic is expected to be cheap in terms of hardware cost, power consumption, and rack size. Plus, in cloud network testing, the value of the field in the packet header is required to be editable. For example, one may expect the source IP address to be randomly chosen from the prefix 10.0.0.0/16.

In addition, since a network load tester is usually used for network checking, debugging, and troubleshooting, it is required to control the sending rate based on the determined configuration. In other words, the tester should be able to generate the random burst traffic and emulate the failure scenarios precisely. All of the testing data are collected by measuring the incoming and outgoing traffic in multiple dimensions, such as throughput and packet drops. A network load tester is required to support fine-grained bidirectional measurement of the large volume of traffic.

2.2 Related Work

Table 1 shows the comparison between Norma and the state-of-the-art load testers in terms of our production requirements.

Software network testers. Software solutions [9, 18, 24, 31, 32, 47] are highly flexible. The early software network testers [6, 11, 17, 23] are based on the standard Linux IO API which limits the performance and accuracy. There are many works [9, 18, 31, 47] that utilize the IO frameworks such as DPDK [2], Netmap [47], and PF_RING ZC [8] that are working on accelerating packet processing on various CPU architectures. However, the computing bottleneck makes it difficult to apply to 100 Gbps network test scenarios. The state of the art such as MoonGen [31] needs over 14 2.4 GHz cores to generate 64-byte packets at 100 Gbps, corresponding to only one port capability of Norma. In addition, software solutions are not stable when testing complex packet processing due to the indefinite packet processing time [59]. Therefore, they are not scalable and cost-effective in industrial scenarios that require Tbps-level load testing.

Commodity hardware testers. Vendors like Keysight [7] and Spirent [13] provide network infrastructure performance tests using their test suites with hardware-based modules. These commercial hardware testers [7, 13] are able to emulate standard traffic demands by providing rich testing functions. There are also application and security tests that cover the L4 protocols. Benefiting from the specially designed software and hardware, it achieves high throughput and accuracy on packet generation and measurements. Commercial hardware tester uses dedicated ASICs from the vendors to accelerate network traffic generation, which can provide O(1 Tbps) traffic for stateless protocols.

However, the commercial hardware tester is a black box, which makes it hard to adapt to the agile development of self-defined protocols. The vendors are aiming to provide standard tests thus the customizability of user-defined packet structures and protocols is lacking. Besides, they are expensive to deploy in a large-scale system (*e.g.*, $100,000 for a 100 Gbps dual-port packet generation module [59]) which requires a large number of testers.

Programmable hardware testers. To achieve a balance of programmability and performance, some network testers [19, 27, 48] using programmable hardware such as NetFPGA [60] are proposed. These works achieve accurate rate controlling and precise measurement results. However, the NetFPGA-based testers are still expensive to achieve Tbps-level test traffic (*e.g.*, a NetFPGA board costs $5,341 with two 100 GbE interfaces [1], and a programmable switch ASIC with 32 100 GbE interfaces only costs $2160 [5]). And it is non-trivial to develop new functions on FPGA boards.

Table 1: The required properties of a tester listed by our network operators, and the comparison between Norma and prior work.

Requirements	Meaning	Norma	CHT	ST	HT	NetFPGA
Stateful Protocol Support						
Generation	Whether the traffic of stateful protocols (*e.g.*, HTTP) can be generated?	✓	✓	✓	✗	✓
Customization	Whether the stateful protocol can be fully customized by users?	✓	✗	✓	✓	✓
Real Traffic Emulation						
Cheap High-Speed Traffic	Whether O(1 Tbps) traffic can be generated in a cheap way?	✓	✗	✗	✓	✗
Precise Rate Control	Whether the rate of generated traffic is precise?	✓	✓	✗	✗	✓
Precise Burst Control	Whether the traffic can be sent out with customized burst pattern?	✓	✓	✗	✗	✓
Precise Measurement	Whether the traffic features can be precisely measured?	✓	✓	✗	✓	✓

CHT=Commercial Hardware Testers ST=Software Testers HT=HyperTester

Programmable switch ASICs like Intel Tofino [16] provide customizable packet processing logic via programmer-friendly P4 language [21]. HyperTester [53, 55, 59] leverages the recirculate primitive in P4 language and packet replication engine to generate packets. It can generate stateless traffic at the rate of 1.6 Tbps. HyperTester confirms the feasibility to implement a stateless hardware tester via Tofino's programmable switch ASICs and proves the traffic quality via microbenchmarks. However, HyperTester cannot emulate the data plane behavior of the stateful protocol. We cannot use HyperTester to test a stateful L4/L7 gateway, because HyperTester cannot generate and maintain the session as what the TCP/HTTP specification describes. In addition, HyperTester cannot emulate realistic traffic in a high-fidelity way. We analyze the reason and conduct experiments in §9.2. Inspired by HyperTester, IMap [43] uses programmable switch ASICs for network scanning. It is not a network load tester in a general sense.

Stateful packet processing. Many works are providing stateful packet processing to offload networking functions into hardware. FlowBlaze [46], FAST [44], and OpenState [20] define state machine abstraction to describe network functions, while Domino [49], dRMT [26], SDP [34], and Ibanez *et al.* [37] propose customized RMT-based architecture using FPGA to achieve the processing ability of the stateful packet. These works are orthogonal to Norma. Norma focuses on emulating high-throughput stateful traffic to test the performance of the DUT, instead of implementing every detail of stateful protocols. This gives us the chance to implement the state machine on programmable switch ASICs like Tofino. None of the prior work focuses on this.

3 Norma Overview

Norma is a practical cloud network tester for load testing. We use the programmable switch ASIC to leverage its large capability of packet processing and programmability. In this part, we explain the reason for using the pipeline-folded [14, 15, 45] programmable switch ASIC first (§3.1). Then, we introduce the high-level architecture of Norma (§3.2). This architecture illustrates how our testing functions are arranged in the ASIC and work as a practical network tester.

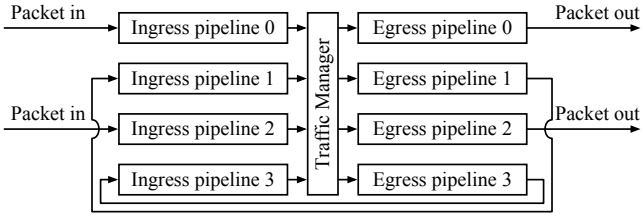


Figure 1: The packet path of pipeline-folded programmable switch ASICs. Half of the pipelines are in loopback mode.

3.1 The Pipeline-Folded Switch ASIC

As shown in Figure 1, the pipeline-folded programmable switch ASIC we use (*i.e.*, BFN-T10-032Q [5]) has 64×100G ports with maximum port bandwidth of 3.2 Tbps. It has four physical pipelines in total, but only two of them are connected to front panel ports (*e.g.*, pipeline 0/2, namely *external pipeline*). The remaining two pipelines are connected to the internal loopback ports, whose egress direction is wired to the ingress direction inside the ASIC (*e.g.*, pipeline 1/3, namely *internal pipeline*).

Norma chooses the pipeline-folded programmable switch ASIC for three reasons. (1) The internal pipelines and external pipelines can be programmed with different P4 programs. By dividing functions such as basic switching, packet editor, stateful responder, *etc.* into the above two groups, Norma can support all of them simultaneously with the limited hardware resources inside the ASIC. (2) Besides the recirculation capability provided by P4 primitives, internal pipelines provide 3.2 Tbps extra loopback bandwidths. Therefore, high-throughput traffic generation can be achieved without affecting the front-panel port throughput. (3) The folded pipelines double the stages we can use. That means we can implement more complex processing logic than the unfolded one, which is the fundamentals of stateful packet processing. The RMT-based programmable switch ASIC guarantees that these additional stages do not affect the processing rate of the traffic and only introduce negligible latency.

3.2 Norma's Architecture

Norma takes advantage of the pipeline-folded programmable switch ASIC and arranges all required functions in the architecture shown in Figure 2. The input and output represent the front-panel ports of the switch. The basic switching logic (omitted in figures) and measurement functions are imple-

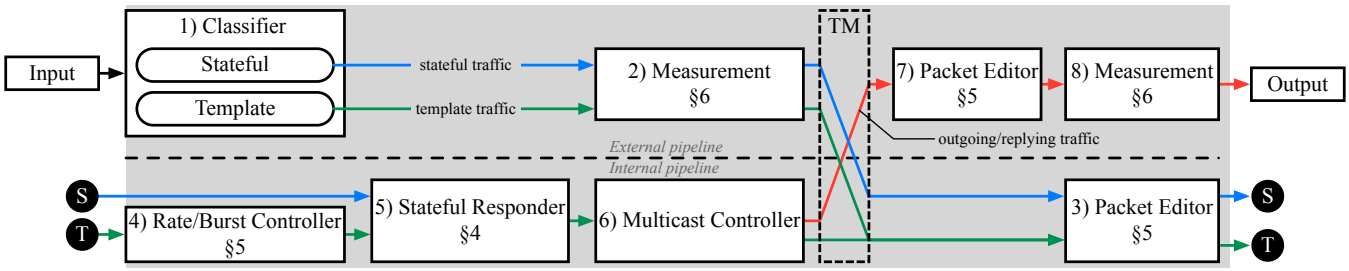


Figure 2: The function models and workflow of Norma. Different colors of arrow lines represent different traffic types. Nodes S and T are loopback ports.

mented in external pipelines, while other functions are implemented in internal pipelines. The **S** and **T** nodes represent two loopback ports in internal pipelines, allowing packets to travel from the egress back to the ingress. The traffic manager (TM) is responsible for packet replication and forwarding.

Traffic. Norma needs to classify incoming traffic to decide which function to enable according to the user's test task. We detail three kinds of traffic shown in Figure 2 as follows.

- **Stateful traffic** includes incoming packets of stateful protocols such as TCP and HTTP, represented as blue lines. Once received, these packets will be preprocessed by the packet editor (*e.g.*, updating the TCP sequence number) and then handled by the stateful responder, which triggers replying traffic.
- **Template traffic** includes template packets sent from the control plane, represented as green lines. Control plane programs construct template packets according to the user's test task and send them to the ASIC via PCIe. Then Norma keeps these packets looping all the time in loopback ports of internal pipelines. If a template packet is marked by the multicast controller, it will be replicated by the packet replication engine (PRE) in the ASIC's TM module. Then the replicated packet will be forwarded to the DUT as outgoing traffic via external egress pipelines. In this way, Norma can generate line rate traffic on the data plane, though there is no memory for the ASIC to store packets. The looping template packets determine what kind of traffic can be generated.
- **Outgoing/Replying traffic** is represented as red lines. The two traffic types can be regarded as the same because they go through the same paths. The replying traffic is triggered by the stateful responder, while the outgoing traffic is not.

Modules and workflow. We now show the function modules of Norma in turn by the following workflow.

1) **Classifier**. The traffic is classified by the classifier first. Norma mainly focuses on stateful traffic and template traffic. Other traffic is also classified, but omitted in Figure 2.
2) **Measurement**. All traffic enters the measurement module and is measured according to the user's measurement rules. Measurement functions are described in §6.
3) **Packet Editor**. Stateful traffic and template traffic are forwarded to internal egress pipelines. The packet editor preprocesses the stateful traffic or modifies template packet fields. The packet modification function is introduced in §5.
4) **Rate/Burst Controller**. After looping back to internal ingress pipelines, the rate/burst controller marks template packets to control the rate and burst pattern of the generated traffic. This part is detailed in §5.
5) **Stateful Responder**. Norma uses the extended finite-state machine (EFSM) [25] to abstract the stateful protocol. The stateful responder triggers the EFSM according to the input stateful traffic. The replying traffic is generated with the help of the template traffic. This part is detailed in §4.
6) **Multicast Controller**. The multicast controller marks the template traffic and forwards it to the internal egress pipeline it comes from to complete the high-speed looping of the template traffic. In addition, the marked template packets are replicated to the target output port through TM.
7) **Packet Editor**. Outgoing traffic needs to go through the packet editor on the external egress pipeline one more time. The supported actions of these two packet editors are different for sophisticated traffic generation capabilities.
8) **Measurement**. Finally, all outgoing traffic enters the measurement module in the egress direction.

4 Emulating Stateful Protocol

The essence of emulating a protocol is to run the processing program on the programmable switch ASIC. Although, it is not easy to port programs that originally run on CPUs to the ASIC. The protocol implemented on Norma for testing the DUT can be reduced to a human-descriptive sequence of packet interactions, as long as the DUT does not perceive the differences. Therefore, our high-level idea is to convert such a program into a state machine and write it into the match-action table.

In this section, we first introduce the EFSM [25] abstraction of stateful responder, which helps us establish a general stateful protocol programming pattern for Norma (§4.1). Then we take the HTTP protocol as an example to show the implementation details of the stateful responder in three steps: executing

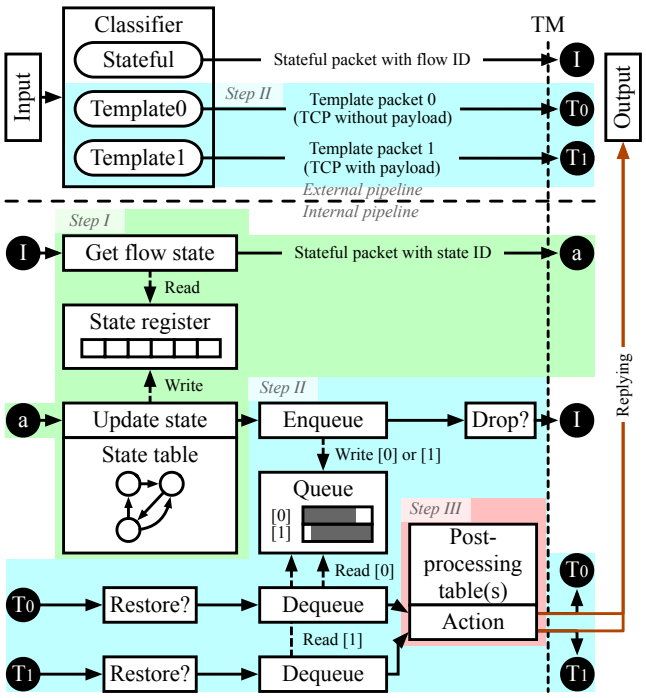


Figure 3: A detailed view of the components of the stateful responder. The classifier is not included in the stateful responder but used to describe where the inputs come from.

the state machine (§4.2), generating replying traffic (§4.3), and postprocessing of the replying traffic (§4.4). The brief packet path and table placement are shown in Figure 3.

4.1 The Stateful Protocol Abstraction

To handle stateful protocols, we need to program the processing logic of the protocol in the programmable switch ASIC. The main process is to generate replying packets according to the flow state and stateful packets, and the abstraction of this process can be represented by the EFSM.

EFSM abstraction of Norma. An EFSM in Norma is defined as a 7-tuple $M = (I, O, S, D, F, U, T)$. S is a set of flow states. I is the current flow state. O is the next flow state. D is a set of variables, such as packet fields, metadata, and registers. F is a set of enabling functions that trigger transitions based on the variables ($f_i : D \to \{0,1\}$). U is a set of update functions that update variables ($u_i : D \to D$). T is a transition relation ($T : S \times F \times I \to S \times U \times O$). Table 2 shows the state table of the EFSM that handles HTTP GET requests. The conditions are the enabling functions in set F, and the actions are update functions in set U.

Hardware bases and limitations. The EFSM abstraction provides an interface for users to customize stateful protocols. Specifically, users need to construct the following three parts in Norma to realize the EFSM. The first part is variables D and flow states S. The parser and deparser provide the ability to locate packet fields and metadata. Because programmable switch ASICs natively support registers, the flow states are saved in registers and updated by register actions [14].

Table 2: A part of the simplified HTTP state table.

Curr S	Condition	Next S	Action
0	RST	0	`ig_md.skip_mc = 1;`
0	SYN	2	`hdr.tcp.flags = R;`
0	Unknown	1	`hdr.tcp.flags = R;`
1	SYN	2	`hdr.tcp.flags = S|A;`
1	Unknown	1	`ig_md.skip_mc = 1;`
2	RST	0	`ig_md.skip_mc = 1;`
2	HTTP GET	3	`hdr.payload = 0x48···`
2	Unknown	1	`hdr.tcp.flags = R;`
		...	

The second part is conditions F. The conditions can be implemented as keys in the match-action table. State transitions and actions are triggered by matching the variables like the current flow state and the input stateful packet. However, it should be noted that the matching ability of the key is limited. The relationship between keys in a table can only be a logical AND relationship, so complex matching rules need to be expressed using multiple entries.

The final part is actions U. The actions modify and send the template packets to reply to the stateful packet. It should be noted that the implementation of actions is limited by the resource constraints of the programmable switch ASIC. Excessive register action and table execution may exceed the number of stages in the switch. And it is hard to implement actions that require iteration, such as sorting headers in a packet. A possible solution is splitting the action that requires iteration into multiple steps. The stateful packet loops in the internal pipeline through node I in the internal pipeline in Figure 3 and executes these steps. But this solution reduces the throughput of the flow processing.

4.2 Executing State Machine

Implementing the EFSM on programmable switch ASICs is challenging due to limitations on registers, and table actions. Next, we will introduce the details of the state machine implementation, as shown in Figure 3 Step I.

State register. Norma stores the state of each flow in a register array. Initially, a flow ID is assigned by the classifier to each flow and used as an index to access registers. Then, the stateful packet is forwarded to the internal pipeline with its flow ID for indexing its state register. The flow state registers store current state ID in Table 2 and other flow information such as TCP sequence number, which can be read by the template packet directly for generating the replying packet.

A side benefit of the flow ID is that it releases the fields of 5-tuples and MAC addresses in the stateful packet. When we generate the replying packet, these fields can be retrieved via a post-processing table with flow IDs. Therefore, we can write the flow ID and other bridged metadata[4] into these fields to avoid additional bandwidth consumption when the packet

[4]Bridged metadata is a temporary header carrying the data calculated in current and previous pipelines to the next pipeline [14].

carries the metadata looping from egress pipelines to ingress pipelines.

State updating. As described in §4.1, the state table stores the conditions for state transition. For the stateful packet, it first reads its state ID from the state registers. Then, it gets the next state ID and action ID from the state table. And finally, the stateful packet writes a new state ID back into the register. However, this process cannot be done simply because of the register access restriction that a register cannot be accessed more than once in one pass (a packet goes through the ingress pipeline and egress pipeline). Reading and updating are two accesses that cannot be merged into one due to complex dependencies. To break this limitation, our idea is to let the stateful packet do another pass. In the first pass, the stateful packet reads the state register to get the state ID. In the second pass, the stateful packet gets the new state ID through the state table and then writes back to the state register. Note that this design can make state register updating non-atomic, further discussed in Appendix A.

State machine bypass. Some protocols can be implemented with state machine bypass. Taking the SYN flood test as an example, the tester receives a SYN packet and replies with an RST packet. The SYN packet can be preprocessed by the packet editor to obtain the action ID. Then, the packet skips Step I in Figure 3 and directly passes the relevant packet information to generate replying packets. For the stateless traffic generation like UDP and ARP reply, the entire stateful responder can be skipped.

4.3 Generating Replying Packets

Norma generates replying packets by replicating the template packet looping in the internal pipeline, as shown in Figure 3 Step II. Take the HTTP GET test task as an example. Norma needs to generate a PUSH packet when receiving an ACK packet. The PUSH packet has a 1460-byte payload, and the ACK packet does not. If the programmable switch ASIC can add or delete the payload, we can generate these two kinds of packets by modifying the input stateful packet. However, PHV resources limit the total length of packet headers that can be parsed. The 1460-byte payload cannot be completely stored in the PHV. So we have to prepare ACK template traffic and PUSH template traffic with 1460-byte payload separately to generate corresponding traffic. Since the stateful packets and template packets do not share packet fields and metadata, Norma needs to transfer information from the stateful packets to the template packets, such as the sequence number and action ID. Then in post-processing tables, Norma can modify the template packets according to the transferred information to generate required replying packets.

Next, we will detail the approach to transferring information across different packets.

A strawman solution. A straightforward way to transfer information across packets is using a queue. The incoming stateful packet pushes necessary information into the queue, and then the template packet pops the information to its corresponding fields or metadata. The register array in programmable switch ASICs can be used to implement the queue. Besides the register array for the elements, the queue needs three more registers to maintain the data structure, one for the head pointer, one for the tail pointer, and one for the queue length. Compared with the common queue data structure, the one in programmable switch ASICs has two critical limitations.

First, the template packet cannot decide whether to dequeue according to the head element. To dequeue according to the head element, the tester must first check whether the queue length is zero, then read the queue element via the head pointer, and finally decide whether to decrease the queue length. In this way, the queue length register is accessed twice in one packet path, which is not allowed by programmable switch ASICs. Since template packets will inevitably take out information from the queue, the queue cannot be shared by multiple types of template packets.

Second, there is no way to prevent the queue from overflowing. Since the packet paths of the stateful packet and the template packet are parallel, to ensure parallel safety, the instructions for dequeuing and enqueuing must be executed in the following order. For dequeuing, reading elements must occur after decreasing the queue length; and for enqueuing, writing elements must occur before increasing the queue length. To ensure that the queue does not overflow, the queue length is compared with its capacity to get the enqueuing permission first. Then the stateful packet writes its information to the queue. And finally, the queue length increases. In this way, the tester accesses the queue length register twice. So there is no method to check the queue length before enqueuing.

In Norma, the consumer of the queue is the template packets, and the producer of the queue is the stateful packets. Template packets poll from the queue to generate replying packets. For example, the PUSH template packets poll the queue for ACK packets. Therefore we must ensure that the polling rate is higher than the arrival rate of the stateful packets to avoid queue overflowing.

Multi-queue for multiple template traffic types. Because of the limitations of the queue, only the stateful packet can choose what kind of template packets to reply to. It is straightforward to allocate a queue for each type of template packets. As shown in Figure 3 Step II, the stateful packet enqueues using queue ID which is obtained from the state table and then triggers the corresponding type of template packet to dequeue. Compared to the strawman solution, there are two changes to the multi-queue data structure. The first change is that three register arrays are used as head pointers, tail pointers, and queue lengths. Each queue ID identifies an element in these register arrays. The second change is that the index of the queue elements needs to be re-planned. A typical method is using some lower bits of the pointer to represent the queue ID, and the remaining bits to represent the element offset in the queue. For example, one register array with a capacity of

256 is used as 16 queues. The lower four bits are the queue ID and the higher four bits are the offset.

Modifying payload. For some test cases where the complete or partial payload of the template packet is capable to be parsed in the PHV, we can treat this payload as a normal packet header. First, the parser needs to parse the payload as a header according to the *total length* in the IP header. Before entering the post-processing table, the previously added payload header must be set to invalid to avoid adding payloads repeatedly. Second, the payload header needs to participate in the calculation of the TCP checksum, which should be implemented in both the parser and the deparser. In this way, Norma can add, delete, and modify the payload without queues.

4.4 Post-Processing

As shown in Figure 3 Step III, post-processing tables are used to modify the template packet and finally generate the replying packet. First, the stateful packet enqueues the flow ID and action ID. Then, the template packet obtains the action ID from the queue and executes state actions according to this action ID in post-processing tables. State actions must implement the following functions: (1) restoring the MAC addresses, IP addresses, and TCP ports according to the flow ID; and (2) setting the correct egress port to the replying packet. Other instructions such as updating the TCP sequence number depend on the user's testing requirements.

5 Emulating Realistic Traffic

Now we introduce how Norma emulates realistic traffic. We have two requirements for real traffic generation. First, the types of outgoing traffic generated by Norma cover our test scenarios (§5.1). The packet editor can modify the fields of the outgoing packet fields according to the user's test tasks such as port scanning and host probing. Second, the rate of outgoing traffic controlled by Norma covers our test scenarios (§5.2). Norma leverages the packet header compression technique to overcome the bandwidth bottleneck caused by bridged metadata conveyance, thus achieves the line-rate traffic generation (Appendix B). On this basis, the rate/burst controller is able to control the outgoing traffic rate accurately and the burst pattern can be customized by the controller to emulate the network traffic in corner cases.

5.1 Two-Stage Packet Editor

If the resources of the external pipeline are sufficient, the packet editor only needs to be implemented on the external egress pipeline. However, most resources of the external pipeline has been occupied by switching functions. There are no more stages available to support register operations like generating random numbers and execution of the packet editor table actions like packet field assignment.

We propose a two-stage editing mechanism to overcome this limitation. Instead of editing the outgoing packet on the external egress pipeline only, Norma splits the packet editor into two parts. Because the internal egress pipeline is not used by the switching function and measurement functions. We implement a major packet editor that edits the template packets on the internal pipeline first to complete most of the work. However, modifying the template packet alone is not enough. The outgoing traffic on multiple front-panel ports is identical if these modified packets are replicated to these ports through the PRE. So there is a minor one that edits the outgoing traffic on the external pipeline to differentiate them on each port. We leave implementation details in Appendix C.

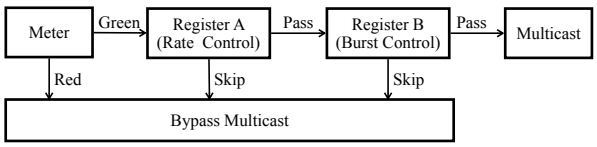


Figure 4: The multi-stage rate control mechanism. Meters and registers determine whether to multicast the template packets or not to control the traffic rate and burst pattern.

5.2 Precise Rate & Burst Control

The traffic rate is an important feature in realistic traffic emulation. Norma is required to send the traffic exactly at the configured rate with diverse burst patterns. The meter [36] (usually implemented by a token bucket) is a common rate control component provided by the programmable switch ASIC. In Norma, the meter first colors the template packets that loop in the internal pipeline at line rate to red or green. The multicast controller then marks the drop flag to the red packets and writes the multicast metadata to the green packets to generate outgoing traffic at the target rate. But the following two limitations make the meter not quite practical. 1) The meter colors the packets of the stable traffic with an equal time interval. It is impossible to generate burst traffic where packets are expected to be sent in batches. 2) The target rate has a finite precision, *i.e.*, not all target rates can be precisely configured. The actual rate of the meter can be different from what we set, and the error grows even larger if we choose a shallower bucket depth to avoid unexpected bursts.

Norma proposes a multi-stage rate control mechanism in the rate/burst controller to obtain the ability of accurate rate control and bursts. As shown in Figure 4, the mechanism is based on a meter, appended with multiple register units (*e.g.*, 2 units in the figure). Each unit is implemented in the same way, which skips the following m packets after passing n successive packets. These units are connected in a cascaded way, and the parameters m and n can be configured individually. Next, we show how the design solves both two problems.

First, for the case of accurate rate control, the user wants to generate 10 Gbps traffic. However, the two closest rates supported by the meter are 9.9 Gbps and 10.1 Gbps. In this case, we can configure the meter to the larger rate. The parameters m and n of register A in Figure 4 are set to 1 and 100, respectively, which means skipping 1 packet after passing 100 packets. Therefore, the final rate becomes $\frac{10.1\ \text{Gbps} \times 100}{1+100} = 10$ Gbps.

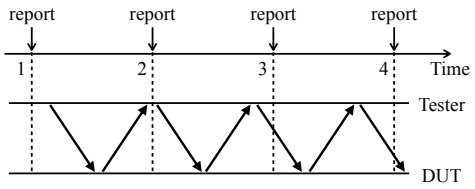


Figure 5: Cross-interval packets make counters out of sync. In time 2-3, the tester sends two packets and reports one is lost, but it receives the lost one in time 3-4.

Second, for the case of burst pattern control, the user wants to send burst traffic whose peak rate and average rate are 10 Gbps and 4 Gbps, respectively, which means there are 1,000 packets in a burst batch. To meet this goal, we only need to configure the m and n of register B to 1,000 and 1,500, respectively. These registers can be used flexibly. We can use two or more registers together to get a more precise rate or to construct complex burst patterns.

6 More Practical Considerations

To use Norma in practice, we also need to address some engineering challenges. For a load tester, measurement is one of the most important engineering challenges, since it is used to estimate the performance of tested networks. We have two requirements for the measurement. First, the measurement must be accurate, which means that the measurement must be done on the data plane as much as possible. Second, the measurement should not affect the functionality of the DUT and the pattern of the outgoing traffic. This means that the outgoing packet should not carry additional headers to store information such as timestamps.

This section first presents a measurement technique based on a flapped version bit, which can obtain high-precision traffic metrics in real time (§6.1). Then, we detail the blind measurement technique used in the delay measurement that avoids adding extra information to the outgoing packets (§6.2).

6.1 High-Precision Real-Time Measurement

A straightforward way is to let the programmable switch ASIC periodically report the counter value to control plane programs. Then, the metrics (*e.g.*, throughput) can be calculated as the quotient of the counter value difference and the reporting period. However, this method is inaccurate for complex metrics relying on bidirectional measurements. Consider the measurement of packet drop rate in Figure 5, which counts in both ingress and egress directions and reports the difference. Note that there is a *cross-interval packet* that is sent before the third report (at time 3) and received after it. Thus the ASIC knows that there are two packets sent in total and only one packet received during the reporting interval 2-3, and reports "one packet is lost" to the control plane at time 3. But in fact, there is no packet loss.

Norma synchronizes the counters in two directions by embedding a version bit in packet headers (*e.g.*, one bit in the IPv4 *identification* field). The version bit flaps every time a report happens. For example, in Figure 5, the version bit values in three reporting intervals can be (1,0,1) or (0,1,0), respectively. In the meanwhile, each original counter will be replaced by a counter group composed of two counters, corresponding to two versions. When receiving a packet, the ASIC reads the version it belongs to from the packet header, and updates the counter indexed by the version. Choosing the reporting period to a value larger than the maximum forwarding time of the DUT, the cross-interval packets will disappear.

While Norma achieves high-precision real-time measurement, it also adds a delay in reporting period to the data report. In addition, the SRAM used by counters is doubled.

6.2 Blind Measurement of Forwarding Delay

Typically, the forwarding delay can be measured by embedding a timestamp at the end of the packet. When Norma receives it, the timestamp will be parsed out and then compared with the current timestamp. However, this method cannot be applied to the programmable switch ASIC, whose pipelines are unaware of the packet payload. An alternative way is to embed to timestamp between the headers where the ASIC can parse, but it does not work for stateful protocols. For example, the HTTP header cannot be parsed by the ASIC due to the variable header length, and such embedding inserts the timestamp between the TCP header and the HTTP header. When the DUT (*e.g.*, an L7 gateway) receives the packet, it incorrectly treats the timestamp as an HTTP header. Another way is to embed the timestamp in packet headers, but there is not enough room for a 32-bit nanosecond timestamp, which is necessary for measurement precision.

Norma proposes the *blind measurement* technique, which does not embed or add a timestamp into the packet but sends blindly. Our approach is based on the synchronization framework (§6.1). Within each reporting interval, the ASIC records the timestamp of the first outgoing packet in a register and regards the value as base timestamp B_o. For following outgoing packets in the interval, the ASIC calculates relevant time as the difference between its timestamp and the base timestamp and adds it to a time register. Here we use the relevant time to avoid arithmetic overflow and denote T_o and P_o as the sum of relevant time and the outgoing packet number, respectively. The ASIC processes the incoming packets in the same way, and we denote corresponding values as B_i, T_i, and P_i, respectively. When the interval passes, the ASIC reports the all above values to the control plane, and then the control plane calculates the average delay d as: $d = (B_i + T_i/P_i) - (B_o + T_o/P_o)$. It is noticeable that packet loss can affect the precision of blind measurement. We leave the analysis in Appendix D.

We use the example in Figure 6 to illustrate how it works. Assume that the base timestamps of outgoing and incoming packets are 1000 and 1005, respectively. Three outgoing packets are sent at the time 1010, 1020, and 1030, so the time

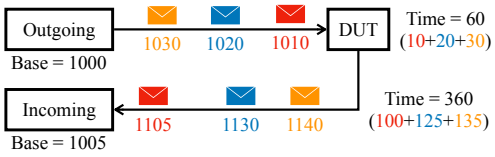


Figure 6: An example of blind measurement. The time offsets are accumulated by Norma and not carried with packets.

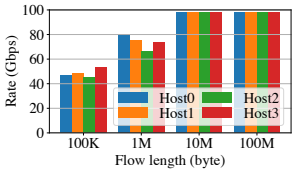


Figure 7: CDN load balance throughput.

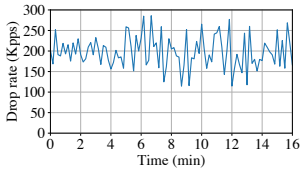


Figure 8: Packet drop on the traffic manager.

register of outgoing packets is added by 10, 20, and 30, respectively, which sum up to 60. Similarly, the time counter of incoming packets is 360. Therefore, the control plane calculates the average delay as $(1005+360/3)-(1000+60/3)=105$.

7 Implementation

Different from prior work, Norma depends on the pipeline-folded programmable switch ASIC, and therefore, is built from scratch. We write about 1,200 and 1,000 lines of P4 code to implement HTTP and TCP traffic responding functions, respectively. Except that, we write about 8,400 lines of P4 code to implement the rest of the data plane, as well as basic switch functions such as routing and ACL.

The control plane of Norma is implemented with about 2,500 lines of Python code and runs in a SONiC-like operating system [12]. It uses internal gRPC to communicate with the ASIC and provides HTTP APIs to users for job management. With these APIs, users can create a traffic-sending job with a desired packet header stack, traffic rate, burst pattern, measurement, stateful responder, *etc.*, and then submit it to Norma. After that, the job manager automatically allocates loopback ports (Appendix E) to these jobs and starts sending packets, until the job is terminated by users.

8 Case Study

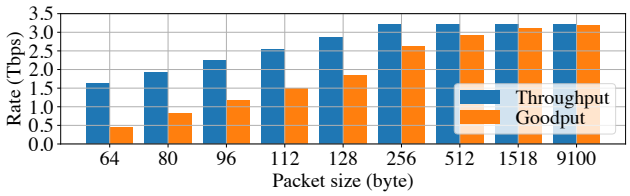
Norma has been used to test our pre-online devices for over two years. We present three real usage experiences.

CDN load balancer stress test. In a CDN system, the load balancer (LB) is responsible for distributing user requests to backend servers and then returning the servers' responses to the user. The LB, therefore, needs to afford a large amount of traffic. Its performance is critical to the CDN system. In one of our production pre-deployment, the load balancer employs four 100 Gbps links to connect to the ISP network and uses the other four 100 Gbps links to connect to the backend servers. To test the LB, we used Norma to emulate the traffic from both the ISP network and the backend servers. The traffic of the emulated HTTP clients was sent to the ISP ports of the LB, and the traffic of the emulated HTTP server was sent to the backend server ports of the LB. Norma initiated and maintained 4,000 HTTP connections. If one connection ends normally, Norma re-initiates the connection; if one connection ends abnormally, Norma shuts down the connection. Therefore we changed the connection establishing frequency by tuning the flow length to test whether there existed any performance issue. As shown in Figure 7, we observed that when the flow length was greater than 10 MB, the throughput of each backend server was close to 100 Gbps; however, when the flow length was less than 1 MB, the throughput was lower than 80 Gbps due to the limitation of the HTTP connection establishment capability of the LB. We therefore successfully measured the performance specifications of the LB under different types of loads.

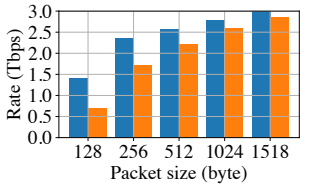
Traffic manager burst traffic test. In another LB setup, our switch should connect to the ISP network with two 100 Gbps links and 32 backend servers with 25 Gbps links. Normally, the throughput of the user requests from the ISP to the backend servers should be 50 Gbps. These requests trigger about 90 Gbps replying traffic, and 200 Mbps synchronization traffic from each backend server to other servers. But long-term operation in practice showed that there was packet loss on the LB. In troubleshooting, we find that requests sometimes generate bursts of 7 Gbps lasting 9-10 ms on one host, and drive the burst of synchronization traffic. Even with the bursts, this level of traffic should not cause a significant packet loss on the DUT. However, the QAC (Queue Admission Control) drop counters of backend servers increased irregularly.

The root cause is the improperly configured traffic manager. The burst traffic can rapidly fill the queue up and then cause packet loss. To tune the traffic manager configuration, we set the burst mode to sending 3,000 packets at 2.5 times the average throughput intermittently on the request traffic and the synchronization traffic, which can reproduce the bursts and packet loss. Figure 8 shows our result. There were about 197,000 packets dropped by the traffic manager every second. And therefore, Norma assisted our operators to optimize the traffic manager configuration.

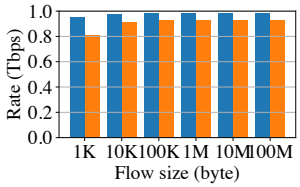
ARP learning rate test. We have deployed many programmable switches in our network. We need to ensure the correctness and speed of L2/L3 forwarding functions. To this end, we used Norma to connect these DUTs (*i.e.*, tested programmable switches) with two links. On one link, Norma generated ARP-reply traffic at line rate. It announced the MAC address of a segment of free IPs as the tester itself's. On another link, Norma generated UDP traffic at line rate, where the destination IPs were the free IP addresses announced by the ARP traffic. The DUTs must be able to learn ARP entries correctly first. Second, the DUTs need to forward UDP traffic according to the learned ARP entries. Finally, Norma judged whether the DUTs had learned all ARP entries by measuring the throughput of the forwarded UDP traffic, and then



(a) UDP traffic.



(b) TCP traffic.



(c) HTTP traffic.

Figure 9: Maximum throughput of Norma.

calculated the ARP learning rate.

In our testing, Norma showed that when the number of tested IP addresses was 2^{13}, the learning rate was about 462 ARP entries per second. However, when the number of IP addresses reached 2^{14}, the DUTs failed to learn all ARP entries. After our troubleshooting, we found that the control plane used a hash value of the entry to locate the ARP entry. If a hash conflict of ARP entries occurred, the control plane returned an exception. In addition, the control plane only supported up to 2^{14} entries, which made our test trigger hash collisions. These bugs were hard to find in unit tests because the number of ARP entries in unit tests is limited. No assertion can be triggered without hash collision.

9 Evaluation

All of our experiments were conducted in two programmable switches with Tofino programmable switch ASICs, where one was the tester, and the other was the DUT. Two switches were connected via 32 100 Gbps optical fibers, which provided 3.2 Tbps bidirectional bandwidth in total.

The traffic quality of Tofino ASICs and software testers has been well-learned in HyperTester [55] (see Appendix F). So we omit these experiments in the evaluation. Norma generates traffic only via dedicated hardware, rather than software. Therefore, the traffic generated is quite stable and very easy to reproduce. We got exactly the same results from multiple runs in our experiments.

9.1 Traffic Throughput

In this part, we evaluated the maximum throughput of three typical traffic, including UDP, TCP, and HTTP. Packets or flows with different sizes were required to generate to test the performance limit of Norma. And loopback ports were allocated by the algorithms in Appendix E. Unless otherwise specified, the Ethernet header and frame check sequence are taken into account when we describe packet or flow sizes, while the inter-packet gap and preambles are not.

UDP traffic. We first evaluated Norma's throughput of UDP traffic with different packet sizes, including small packets ranging from 64 to 512 bytes, MTU-sized packets (1518 bytes), and jumbo packets (9100 bytes). The allocation of loopback ports was straightforward for UDP, whose outgoing packets were directly multicast from template packets, and did not need other packets to trigger. Therefore, only one loopback port was occupied by template packets.

Figure 9a shows our results. Norma can generate traffic at the rate of at least 1.6 Tbps and reaches 3.2 Tbps for packets longer than 256 bytes. Two bottlenecks limit the performance of Norma. For small packets, the throughput was bounded by the operating frequency of the ASIC because there were more headers for the pipelines to process. And for large packets, however, the throughput was bounded by the 100 Gbps port rate. We used goodput to represent the transmission rate of the payload. As the packet size increased, the proportion of the packet header decreased, so the goodput increased. These results indicated that Norma's performance reached the limit of ASIC's capability. In the meanwhile, HyperTester can generate UDP traffic at the rate of 1.6 Tbps [55] and can be simply extended to 3.2 Tbps for large packets, similar to Norma.

TCP traffic. Next, we evaluated Norma's throughput of TCP traffic with state machine bypass. The TCP packet received was forwarded to the loopback pipeline where packet information was enqueued directly. Then the template packets looped in the pipeline read the information from the queue and generated a replying packet based on TCP flags. For example, when receiving a pure ACK packet, Norma would send back a PUSH packet with the payload of a specified size. Since the size of ACK packets was much smaller than that of PUSH packets, one loopback port was enough to process the ACK traffic received from multiple front-panel ports. For example, one loopback port processing ACK packets can support three front-panel ports that sent out 256-byte PUSH packets. That means, every four loopback ports can support up to 300 Gbps TCP traffic.

Figure 9b shows our results. According to the loopback port allocation algorithm in Appendix E, the expected throughput of PUSH packets sized by 128, 256, 512, 1024, and 1518 bytes was 1.6, 2.4, 2.6, 2.8, and 3.0 Tbps, respectively. However, the throughput of 128- and 256-byte packets was slightly lower than expected due to pipeline throughput limitations. For the rest of the PUSH packet sizes, each front-panel port generated near-line-rate TCP traffic.

HTTP traffic. Finally, we evaluated Norma's throughput of HTTP traffic by emulating HTTP sessions with different flow sizes, ranging from 1 KB to 100 MB. There were two types of packets in one HTTP session. One was the packets with HTTP content, and the other was the TCP control packets. Norma generated the packets with HTTP content by duplicating the template packets with the 1024-byte payload. For other packets, Norma used the template packet with no pay-

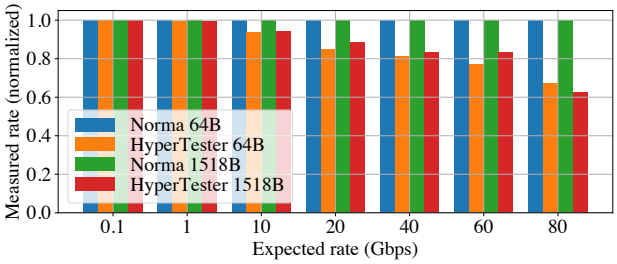
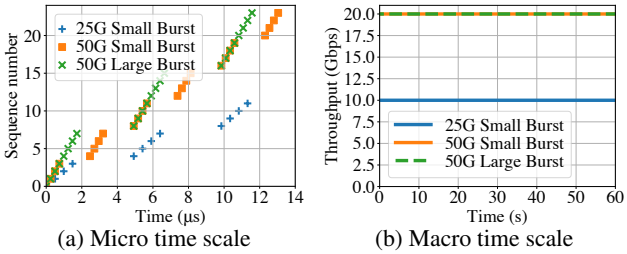


Figure 10: Comparison of rate control accuracy.



(a) Micro time scale (b) Macro time scale

Figure 11: The traffic generated by Norma's burst control.

load. To achieve the maximum throughput, in each 16-port internal pipeline, five ports were used for reading state registers, five ports were used for writing state registers, five ports were used for generating PUSH packets, and one port was used for generating TCP control packets. Therefore, we expected that Norma should generate 1 Tbps HTTP traffic.

Figure 9c shows our results. Norma generated HTTP traffic with over 950 Gbps throughput, which was slightly lower than expected. This was because of the gap between the looping frequency of large template packets (*e.g.*, 11 Mpps) and the arrival frequency of small packets (*e.g.*, 150 Mpps), which made the enqueue time and dequeue time unaligned. For example, a small HTTP control packet may wait in the queue for triggering a large data packet. This phenomenon was obvious when small packets dominate in short HTTP sessions and led to lower throughput.

Stability. We evaluated the stability of Norma by sending UDP, TCP, and HTTP traffic continuously over 24 hours, at the rate of 3.2, 3.0 and, 1.0 Tbps, respectively. We recorded the throughput of Norma periodically and found that it kept stable during the long-term run.

9.2 Traffic Control

In this part, we evaluated the traffic control capabilities in Norma in terms of rate control accuracy and traffic bursts.

Rate control. We evaluated rate control on two types of UDP traffic generated by Norma. One was composed of 64-byte packets and the other was MTU-sized packets. We measured the actual throughput of generated traffic and compared it with the expected rate. For clarity, the actual throughput was normalized by the expected rate. As shown in Figure 10, Norma achieved nearly 100% accuracy in all cases. However, the actual throughput of HyperTester became inaccurate as the expected rate grew larger. For example, the rate error reached 38% when the expected rate was 80 Gbps.

HyperTester controlled the rate of generated traffic by comparing the timestamp gap with a dropping threshold, which was calculated based on the expected rate. For any two successive packets, if the packet gap was less than the threshold, the second packet would be dropped. However, considering the scenario when the expected rate was more than half of the full rate, the dropping threshold would always be larger than the transmission time of a single packet. It meant there was always one packet being dropped for any two successive packets, and the actual rate cannot exceed 50% of the full rate. When the expected rate reached 80 Gbps, the actual rate of HyperTester was 80 Gbps $\times$ 62% $\approx$ 50 Gbps, which was the same as what we measured above.

We further evaluated the accuracy of our multi-stage rate control. We used the meter-based rate limiter provided by the programmable switch ASIC as a baseline. The actual rate is measured by a counter that counts how many packets pass through the egress pipeline in a range of time. The error is the ratio of the difference between the actual rate and the target rate to the target rate. For Norma, the rate control accuracy can be further guaranteed by our multi-stage rate control design. As shown in Table 3, the error of the meter-based rate limiter ranged between 0.1% and 1%, because the rate to limit supported by the hardware meter was not continuous. After applying the multi-stage rate control, the accuracy was promoted by at least 10$\times$, and the rate error was less than 0.01% in the worst cases.

Table 3: The rate error of our multi-stage rate control and pure meter when generating packets of different sizes.

Rate (Gbps)	64 Bytes		1518 Bytes	
	Multi-Stage	Pure Meter	Multi-Stage	Pure Meter
0.1	6×10^{-6}	3×10^{-3}	2×10^{-5}	1×10^{-3}
1	9×10^{-7}	2×10^{-3}	4×10^{-6}	5×10^{-4}
10	8×10^{-8}	4×10^{-3}	1×10^{-6}	3×10^{-3}
20	5×10^{-8}	4×10^{-3}	4×10^{-8}	3×10^{-3}
40	3×10^{-8}	4×10^{-3}	2×10^{-7}	3×10^{-3}
60	5×10^{-5}	3×10^{-3}	2×10^{-6}	8×10^{-4}
80	2×10^{-4}	4×10^{-3}	1×10^{-7}	3×10^{-3}

Burst control. To test the burst control, we made the Norma to generate three kinds of traffic with different burst patterns. For traffic A, B, and C, we set the expected average rate to 10 Gbps, 20 Gbps, and 20 Gbps, and the burst scale (*i.e.*, the number of packets in a burst batch) to 4, 4, and 8, respectively. In addition, the burst rate (*i.e.*, peak rate) of all the traffic was required to be 2.5$\times$ the average rate. The performance of burst control was evaluated on two scales. The micro time scale showed the packet-level burst pattern, while the macro time scale showed the traffic-level throughput. For the micro time scale, we gave each packet an increasing sequence number and recorded their transmission time. The result was shown in Figure 11a. In the two burst patterns with an average rate of

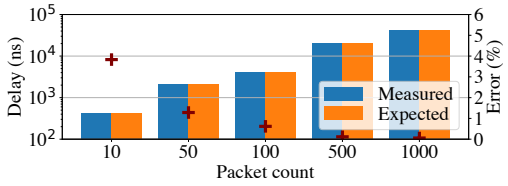


Figure 12: The packet delay when DUT meets bursts.

50 Gbps, there were 8 packets sent continuously in each batch under the large burst setting, and there were 4 packets under the small burst setting at the same rate with half the batch size. For the burst pattern with an average rate of 25 Gbps, there were 4 packets sent continuously at half rate in the same batch size as the 50 Gbps large burst pattern. At the macro time scale shown in Figure 11b, they all stayed at the target average rate.

9.3 Measurement

In this part, we combined traffic generation and burst control to evaluate the measurement function of Norma. First, Norma was connected to the switch under test and generated 100 Gbps line-rate UDP background traffic. The switch then forwarded background traffic back to one specified port of the tester. Second, Norma generated burst traffic at 100 Gbps with different burst scales. And burst traffic was also forwarded by the switch back to the same port of the tester. The packet size of burst traffic and background traffic was both 1024 bytes. Finally, Norma measured the average delay of packets in the burst traffic and compared it with the expected queuing time to judge the error of Norma's measurement.

Note that in this case, both burst traffic and background traffic were queued at the same egress port of the switch. We denoted the burst scale as n packets, and then the theoretical average packet queuing time was $\frac{(1024+20)\text{ Bytes}}{100\text{ Gbps}} \times \frac{n}{2}$[5]. In addition, the delay measured by Norma included the link propagation time, which is about 1454 ns.

Results are shown in Figure 12, where the link delay has been removed. For all of the burst scales, the queuing delay measured by Norma was very close to the theoretical value. When the burst scale was greater than 10 packets, the error of the average delay was less than 4%. The larger the burst size, the more accurate the measurement of average delay was.

10 Limitation & Discussion

Can Norma emulate full functions of stateful protocols? It depends on the complexity of the protocol. Besides the hardware limitations we detail in §4.1, the hardware resources also constrain the implementation of the stateful protocol. A stateful protocol in Norma consists of its state transitions, the replying traffic types, and the state actions. The capacity of the state table determines how many state transitions Norma can support. The loopback ports determine how many template packet types Norma can support and the maximum throughput of replying traffic Norma can generate. And most importantly, the state action may be too complex to implement into the programmable switch ASIC, because the switch resource allocation algorithms and related optimizations in compilers are unknown to developers. Without trying to implement the protocol and compile it, it is hard to know whether a stateful protocol can be fully emulated. Therefore, for complex protocols, we need to simplify them under the premise of being able to complete the test task.

Can Norma support testing customized protocols? Users can customize the packet structure and the processing logic of the protocol in most cases. For example, if we want to measure the forwarding performance of the GPRS tunneling protocol [3], we can modify the parser and deparser to support it. If we want to measure the RDMA write-only throughput of a host, things become complex. The process of exchanging information, congestion control algorithm and packet loss recovery are difficult to express with the EFSM. Even if possible, it is difficult to implement within limited instructions. Our approach is to retain only the process of transferring content in RDMA and remove other logic such as congestion control. However, Norma cannot support protocols with encryption due to the limitation of programmable switch ASICs, unless the encrypted data can be regarded as a fixed payload, or special acceleration cards such as IPU [4] are available.

Can Norma localize the root cause of performance issues? Norma cannot localize the root cause of performance issues because the DUT is typically a black or gray box that cannot be simply modeled. For example, a performance problem can come from misconfigurations, ASIC capabilities, bottleneck of switch CPUs, and even signal strength when wireless links involve. It might be possible to extend Norma to root cause localization if sufficient information is provided.

11 Conclusion

We present Norma, the first practical network load tester used in production. Norma employs the programmable switch ASIC to support stateful protocol generation and customization and realistic traffic emulation such as high precise rate control. Norma has been used in our operation for over two years and successfully detected many performance issues.

Acknowledgments

We thank our shepherd, Muhammad Shahbaz, and NSDI reviewers for their insightful comments. We also thank Xiaoliang Wang for his valuable feedback on earlier drafts of this paper. This work is supported by Alibaba Group through Alibaba Research Intern Program. Yanqing Chen, Chen Tian, and Guihai Chen are also supported in part by the National Key R&D Program of China (2022YFB2702803), the National Natural Science Foundation of China under Grant Numbers 62072228, and the Fundamental Research Funds for the Central Universities. Chen Tian and Ennan Zhai are co-corresponding authors.

[5] The inter-packet gap and preambles (20 bytes) should be considered.

References

[1] Alveo U200 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u200.html#buy-from-xilinx.

[2] DPDK. https://www.dpdk.org.

[3] GPRS tunnelling protocol. https://www.3gpp.org/DynaReport/29274.htm.

[4] Intel IPU. https://www.intel.com/content/www/us/en/products/network-io/smartnic.html.

[5] Intel Tofino 3.2 Tbps, 4 pipelines. https://www.intel.com/content/www/us/en/products/sku/218642/intel-tofino-3-2-tbps-4-pipelines/specifications.html.

[6] iPerf. https://iperf.fr.

[7] Keysight. https://www.keysight.com/us/en/products/network-test/network-test-hardware.html.

[8] PF_RING ZC. https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/.

[9] Pktgen. https://github.com/pktgen/Pktgen-DPDK.

[10] Programmable data plane at terabit speeds. https://conferences.sigcomm.org/sigcomm/2018/files/slides/p4/P4Barefoot.pdf.

[11] Scapy. https://scapy.net/.

[12] SONiC. https://sonic-net.github.io/SONiC.

[13] Spirent. https://www.spirent.com/products/testcenter-ethernet-ip-cloud-test.

[14] Tofino native architecture - public version. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf.

[15] Tofino product family brochure. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/tofino-product-family-brochure.pdf.

[16] Tofino programmable Ethernet switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[17] Trafgen. http://netsniff-ng.org.

[18] TRex. https://trex-tgn.cisco.com.

[19] Gianni Antichi, Charalampos Rotsos, and Andrew W. Moore. Enabling performance evaluation beyond 10 gbps. *SIGCOMM Comput. Commun. Rev.*, 45(4):369–370, aug 2015.

[20] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. *SIGCOMM Comput. Commun. Rev.*, 44(2):44–51, apr 2014.

[21] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, jul 2014.

[22] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[23] Alessio Botta, Alberto Dainotti, and Antonio Pescapè. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks*, 56(15):3531–3547, 2012.

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

[25] Kwang Ting Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th International Design Automation Conference*, 1993.

[26] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. Drmt: Disaggregated programmable switching. In *ACM SIGCOMM (SIGCOMM)*, 2017.

[27] G. Adam Covington, Glenn Gibb, John W. Lockwood, and Nick Mckeown. A packet generator on the netfpga platform. In *17th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, 2009.

[28] Quentin De Coninck and Olivier Bonaventure. Multipath quic: Design and evaluation. In *Proceedings of the 13th international conference on emerging networking experiments and technologies (CoNEXT)*, 2017.

[29] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[30] Andrea Di Domenico, Gianluca Perna, Martino Trevisan, Luca Vassio, and Danilo Giordano. A network analysis on cloud gaming: Stadia, geforce now and psnow. *Network*, 1(3):247–260, 2021.

[31] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the Internet Measurement Conference (IMC)*, 2015.

[32] Paul Emmerich, Sebastian Gallenmüller, Gianni Antichi, Andrew W. Moore, and Georg Carle. Mind the gap - a comparison of software packet generators. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2017.

[33] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *ACM SIGCOMM (SIGCOMM)*, 2020.

[34] Nadeen Gebara, Alberto Lerner, Mingran Yang, Minlan Yu, Paolo Costa, and Manya Ghobadi. Challenging the stateless quo of programmable switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets)*, 2020.

[35] Philippe Graff, Xavier Marchal, Thibault Cholez, Stéphane Tuffin, Bertrand Mathieu, and Olivier Festor. An analysis of cloud gaming platforms behavior under different network constraints. In *17th International Conference on Network and Service Management (CNSM)*. IEEE, 2021.

[36] J. Heinanen and R. Guerin. Rfc2698: A two rate three color marker. Technical report, RFC Editor, USA, 1999. https://www.rfc-editor.org/rfc/rfc2698.html.

[37] Stephen Ibanez, Gianni Antichi, Gordon Brebner, and Nick McKeown. Event-driven packet processing. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets)*, 2019.

[38] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[39] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[40] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[41] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *ACM SIGCOMM (SIGCOMM)*, 2020.

[42] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *ACM SIGCOMM (SIGCOMM)*, 2017.

[43] Guanyu Li, Menghao Zhang, Cheng Guo, Han Bao, Mingwei Xu, Hongxin Hu, and Fenghua Li. IMap: Fast and scalable in-network scanning with programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[44] Masoud Moshref, Apoorv Bhargava, Adhip Gupta, Minlan Yu, and Ramesh Govindan. Flow-level state transition as a new switch primitive for sdn. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014.

[45] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *ACM SIGCOMM (SIGCOMM)*, 2021.

[46] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. FlowBlaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[47] Luigi Rizzo. netmap: A novel framework for fast packet I/O. In *USENIX Annual Technical Conference (ATC)*, 2012.

[48] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W. Moore. Oflops: An open framework for openflow switch evaluation. In *Passive and Active Measurement (PAM)*, 2012.

[49] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM (SIGCOMM)*, 2016.

[50] Junji Takemasa, Ryoma Yamada, Yuki Koizumi, and Toru Hasegawa. Ccngen: A high-speed generator of bidirectional ccn traffic using a programmable switch. In *Proceedings of the 8th ACM Conference on Information-Centric Networking (ICN)*, 2021.

[51] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. Cheetah: Accelerating database queries with switch pruning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2020.

[52] Michael D. Wong, Aatish Kishan Varma, and Anirudh Sivaraman. *Testing Compilers for Programmable Switches through Switch Hardware Simulation*. 2020.

[53] Zhaowei Xi, Yu Zhou, Dai Zhang, Jinqiu Wang, Sun Chen, Yangyang Wang, Xinrui Li, HaoMing Wang, and Jianping Wu. Hypergen: High-performance flexible packet generator using programmable switching asic. In *ACM SIGCOMM Posters and Demos*, 2019.

[54] Yiling Xu, Qiu Shen, Xin Li, and Zhan Ma. A cost-efficient cloud gaming system at scale. *IEEE Network*, 32(1):42–47, 2018.

[55] Dai Zhang, Yu Zhou, Zhaowei Xi, Yangyang Wang, Mingwei Xu, and Jianping Wu. Hypertester: High-performance network testing driven by programmable switches. *IEEE/ACM Transactions on Networking*, 29(5):2005–2018, 2021.

[56] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. Gallium: Automated software middlebox offloading to programmable switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2020.

[57] Xu Zhang, Hao Chen, Yangchao Zhao, Zhan Ma, Yiling Xu, Haojun Huang, Hao Yin, and Dapeng Oliver Wu. Improving cloud gaming experience through mobile edge computing. *IEEE Wireless Communications*, 26(4):178–183, 2019.

[58] Zhilong Zheng, Yunfei Ma, Yanmei Liu, Furong Yang, Zhenyu Li, Yuanbo Zhang, Jiuhai Zhang, Wei Shi, Wentao Chen, Ding Li, et al. Xlink: QoE-driven multi-path QUIC transport in large-scale video services. In *ACM SIGCOMM (SIGCOMM)*, 2021.

[59] Yu Zhou, Zhaowei Xi, Dai Zhang, Yangyang Wang, Jinqiu Wang, Mingwei Xu, and Jianping Wu. HyperTester: high-performance network testing driven by programmable switches. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT)*, 2019.

[60] Noa Zilberman, Yury Audzevich, Georgina Kalogeridou, Neelakandan Manihatty-Bojan, Jingyun Zhang, and Andrew Moore. NetFPGA: Rapid prototyping of networking devices in open source. In *ACM SIGCOMM (SIGCOMM)*, 2015.

APPENDIX

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

A Non-Atomic State Updating

The state updating we detailed in §4.2 is non-atomic, which brings two drawbacks. First, this inevitably consumes an extra loopback port and increases the packet processing delay. Second, because the updating of the state registers is done in two passes, if the new state ID is too late to be written to the state register, the next stateful packet may read the old one. In this situation, the state register can be mistakenly written with the wrong value. Because the programmable switch ASIC does not provide the ability to schedule packets, this error can only be avoided by locking the state register and delaying the processing of the following stateful packets. This will bring more problems like keeping following stateful packets looping in the internal pipeline buffer and then handling them in order, which makes it impractical. Therefore, a more acceptable solution is to take the continuously incoming packets into account when designing the EFSM.

B Full-Speed Traffic Generation

How to elimate bandwidth consumption of bridged metadata? Norma generates traffic by multicasting the template packets that loop inside the internal pipeline. However, there is no extra bandwidth reserved for the bridged metadata when the loopback port conveys the packet from the egress back to the ingress, so all metadata must be packed into the packet, which increases the length of the packet and forms a bandwidth bottleneck. For example, consider the scenario when we want to send outgoing traffic composed of 64-byte small packets at the rate of 100 Gbps from one front panel port. Assume the egress of the loopback port adds 12-byte metadata to the template packet, so the packet loops at the rate of $\frac{100\text{ Gbps}}{(64+20+12)\times 8} = 130$ Mpps. Although the ingress will parse and remove the metadata header, the packet rate could not increase anymore. As a result, the throughput of outgoing traffic is only $130\text{ Mpps} \times (64+20) \times 8 = 87$ Gbps. This is a severe problem for the network tester, which makes it impossible to test the DUT under 100% workload.

The key idea of Norma is to compress the packet header. We noticed that the bandwidth bottleneck only exists in the internal pipeline, instead of the external pipeline, because only the internal pipeline has egress-to-ingress forwarding. This enables Norma to borrow some header bits to temporally store the metadata in the internal pipeline, and pay them back at the egress of the external pipeline. The key point is to find the "traffic/port-invariant" fields, whose value is determined once the flow ID and outgoing port are given. Practically, Norma compresses the Ethernet header and borrows 12 bytes in total.

- The 2-byte *Ethernet Type* field can be compressed to one byte. This field indicates the type of the next header in the packet. For example, 0x0800 represents the IPv4 header. In cloud scenarios, there are no more than 10 possible values for this field, which can be encoded in one byte.
- The *Src MAC* and *Dst MAC* fields can be compressed together to one byte, which occupy 12 bytes in the original Ethernet header. MAC addresses are usually fixed given the traffic ID and the outgoing port, so storing the 1-byte traffic ID is enough. When the traffic arrives at the egress of the external pipeline, these fields can be recovered.

How many template packets are needed in one loopback port? In the RMT-based programmable switch ASIC, one template packet is not enough to make full use of the hardware pipeline of a loopback port. Multiple template packets are required to guarantee the line-rate looping. But unfortunately, the exact number of template packets we need depends on many factors, including the packet size, the packet header depth, and how the compiler arranges P4 tables, which makes it hard to predict. Based on our experience, the number of template packets can be represented as a function like $y = Ax^{-B} + C$, where A, B, and C are unknown constants, and x is the packet size. When the packet type and the P4 program are fixed, the function can be determined via curve fitting. In general, 10 and 120 packets are enough when packet sizes are 1500 and 64 bytes, respectively.

C Implementation Details of Packet Editors

The major editor can apply step-based or random-based field editing. There are five modes Norma supports:

1) The **direct step mode** simply adds a constant to the initial value. If the value exceeds the bound provided by the user, it will be subtracted by the bound.
2) The **indirect step mode** is similar to the direct step mode. Differently, the value is not directly outputted but used as an index to access a register array, which saves the real value to the output provided by the user.
3) The **cascaded step mode** can be regarded as a combination of two editors working in the direct step mode. The first one works as normal, but the second one is triggered only when the bound excess happens in the first one.
4) The **direct random mode** simply fills some bits of a field with random bits. For example, filling the rightmost 8 bits of the source IP field means randomly choosing an IP address under a /24 prefix.
5) The **ranged random mode** also relies on random bits. Differently, Norma is required to choose a number from a given range. For example, choose a number for the source port field uniformly from the range 2000-3000.

Among all these modes, the ranged random is the most complex one. It is the random bit instead of the random number that is provided by the programmable switch ASIC, which

means the length of the range must be a power of 2. For example, with 8 random bits, we can only get a uniform random number from 0 to 255, instead of any other range. A straightforward way is to keep trying until a number in the required range is acquired. However, it is not suitable for the ASIC, because there is no cheap way to emulate the while-loop. So we need a concise method that does not rely on loops. Norma solves this problem by using the equation $n := (n' + r) \bmod l$, where n' is the random number generated in the last execution, stored in the corresponding field of the template packet. Here, r is a k-bit random number generated by the ASIC satisfying $2^k \leq l$, where l is the length of the range. Note that the mod operator is not supported by the ASIC, but the restriction to k makes it representable with no more than one subtraction and thus can be implemented in the ASIC. We evaluated the quality of generated random numbers as follows.

We applied the packet editor to UDP packets, whose source ports were randomly chosen from the range 0-999, so the length of the range was 1000, and k (*i.e.*, the number of random bits) should be set to a number no more than 9 to satisfy the constraint $2^k \leq 1000$. From Figure 13a and Figure 13b, we can observe that the quality of the generated random number improved as k becomes larger, benefiting from more random bits provided. Figure 13c shows the frequency of each number generated from 100,000 packets. Each number occurred at a frequency of around 0.1% as expected, which indicated the uniformity of the generated numbers.

D Analysis of Blind Measurement

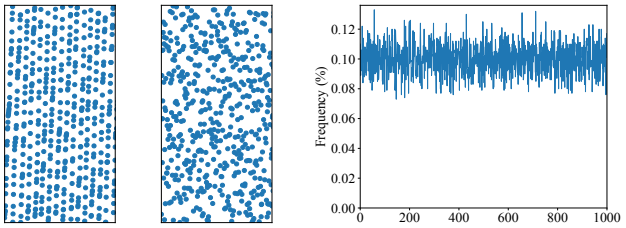
We use the sets $\mathbb{S}$ and $\mathbb{R}$ to represent the packets sent by the egress pipelines, and received from the ingress pipelines, so we have $\mathbb{R} \subseteq \mathbb{S}$, and $drops = |\mathbb{S}| - |\mathbb{R}|$ is the number of packets dropped by the DUT. The real average delay can be represented as $delay = \frac{1}{|\mathbb{R}|}\sum_{i\in\mathbb{R}}(r_i - s_i)$, where the dropped packets are excluded due to incomplete information. However, the blind average delay becomes $delay' = \frac{1}{|\mathbb{R}|}\sum_{i\in\mathbb{R}} r_i - \frac{1}{|\mathbb{S}|}\sum_{i\in\mathbb{S}} s_i$. Now, we define the absolute measurement error e as the difference between $delay$ and $delay'$, and we have

$$
\begin{aligned}
e =& |delay' - delay| \\
=& \left| \frac{1}{|\mathbb{R}|}\sum_{i\in\mathbb{R}} s_i - \frac{1}{|\mathbb{S}|}\sum_{i\in\mathbb{S}} s_i \right| \\
=& \left| \frac{1}{|\mathbb{S}|}\left(\sum_{i\in\mathbb{R}} s_i + \sum_{i\in\mathbb{S}\setminus\mathbb{R}}\left(\frac{1}{|\mathbb{R}|}\sum_{j\in\mathbb{R}} s_j\right)\right) - \frac{1}{|\mathbb{S}|}\left(\sum_{i\in\mathbb{R}} s_i + \sum_{i\in\mathbb{S}\setminus\mathbb{R}} s_i\right) \right| \\
=& \left| \frac{1}{|\mathbb{S}|}\sum_{i\in\mathbb{S}\setminus\mathbb{R}}\left(\frac{1}{|\mathbb{R}|}\sum_{j\in\mathbb{R}} s_j - s_i\right) \right| \leq \frac{1}{|\mathbb{S}|}\sum_{i\in\mathbb{S}\setminus\mathbb{R}}\left| \frac{1}{|\mathbb{R}|}\sum_{j\in\mathbb{R}} s_j - s_i \right| \\
\leq& \frac{|\mathbb{S}| - |\mathbb{R}|}{|\mathbb{S}|}\left(\max_{i\in\mathbb{S}} s_i - \min_{i\in\mathbb{S}} s_i\right) = \frac{drops}{pps_{tx}},
\end{aligned}
$$

which means, the more the packets are dropped, or the slower the packets are sent, the larger the error could be.

E Loopback Port Allocation

As shown in Figure 1, Norma uses two internal pipelines and each of which contains 16 loopback ports. Each loopback port has a maximum BPS rate (*e.g.*, 100 Gbps) while each pipeline has a maximum PPS rate shared by all ports belonging to it.



(a) $k = 6$ (b) $k = 9$ (c) Frequency

Figure 13: Pseudo-random ports ranging in 0-999.

Norma models the loopback port allocation as a satisfiability problem, which can be solved efficiently by SMT solvers such as Z3 [29]. Consider there are n flows using loopback ports. We use zero-one variables $x_{i,j,k}$ to indicate whether flow i should be placed to port k in pipeline j. The BPS rate and PPS rate of flow i are represented as b_i and p_i, respectively. The maximum BPS rate of a port and the maximum PPS rate of a pipeline are represented as B and P, respectively. Then loopback port allocation can be modeled as the following integer linear satisfiability problem:

$$\sum_i \sum_k x_{i,j,k} p_i \leq P \qquad \text{for each } j, \tag{1}$$

$$\sum_i x_{i,j,k} b_i \leq B \qquad \text{for each pair of } (j,k), \tag{2}$$

$$\sum_k x_{i,j,k} = \sum_k x_{i',j,k} \qquad \text{if } i \text{ shares data with } i', \tag{3}$$

$$\sum_j \sum_k x_{i,j,k} = 1 \qquad \text{for each } i. \tag{4}$$

Equations (1) and (2) describe the constraints of the PPS rate and BPS rate, respectively. Equation (3) enforces flow i and flow i' in the same pipeline if they share data via registers, such as the enqueuing flow and dequeuing flow in §4.3. Equation (4) guarantees that each flow will be placed to a port and a port is allowed to be shared by more than one flow.

For all of the cases we met, the satisfiability problems were solved by Z3 in less than 1 second. Then Norma can allocate loopback ports according to the solved variables $x_{i,j,k}$.

F Performance of Software Testers

Zhang *et al.* [55] evaluated software testers such as MoonGen [31] and TRex [18]. Results are summarized as follows. First, software testers cannot generate traffic at more than 300 Gbps due to PCIe bandwidth limitations. For small packets, even with 12 CPU cores, software testers can only generate traffic at about 40 Gbps. Second, the rate control of MoonGen relies on NIC meters, which means not all target rates can be precisely configured. TRex uses software timestamps for rate control, which leads to unstable inter-packet gaps. These results show that software testers cannot generate precisely-controlled high-speed traffic.