TurboNet: Faithfully Emulating Networks with Programmable Switches

Jiamin Cao*†‡, Yu Zhou*†‡, Ying Liu*†‡, Mingwei Xu*†‡, Yongkai Zhou§
*Institute for Network Sciences and Cyberspace, Tsinghua University
†Beijing National Research Center for Information Science and Technology (BNRist)
‡Department of Computer Science, Tsinghua University §UnionPay

***Abstract*—Faithfully emulating networks is critical for verifying the correctness and effectiveness of new networking-related designs. Existing network experiment platforms either cannot faithfully emulate functionality and performance of production networks or cannot scale well because of cost limitations. In this paper, we propose *TurboNet*, a new network emulator that leverages one programmable switch to enable faithful emulation of both network data plane and control plane. For data plane emulation, we present a series of key designs such as port mapper, queue mapper, and delayed queue to emulate network topologies and performance metrics with high flexibility and accuracy. For control plane emulation, we support static routing configurations, distributed routing agents, and the centralized routing controller. Meanwhile, we provide API for operators to simplify network emulation tasks. We implement *TurboNet* on a Tofino switch. The evaluation results show that: (1) *TurboNet* can flexibly emulate various topologies such as the 8-ary fat-tree on the data plane and support about 200 BGP agents with 25% CPU usage on the control plane; (2) *TurboNet* can accurately emulate different network performance metrics, including 400Gbps line-rate background traffic injection, as small as 10^{-8} link loss, and microsecond-level to millisecond-level link delay.**

***Index Terms*—Network emulation, programmable switch.**

I. Introduction

Verifying new networking-related designs, such as protocols, routing algorithms, and proof-of-concept systems, work correctly on real production networks is critical for both researchers and engineers. As production networks are ossified and cannot run unverified designs for reliability and security concerns, researchers and engineers have to rely on modeling or prototyping production networks to conduct experiments and validate their designs. Then, how to faithfully mimic networks, especially in functionality, scale, and performance, becomes a significant problem for verifying the correctness of networking-related designs.

Existing network experiment approaches can be categorized into three types. First, simulators [1]–[6] leverage the computation power of CPUs to model the real networks. They support flexible customization and can easily scale to large networks, but their model cannot fully represent the real networks in both functionality and performance. Second, emulators [7]–[11] run the same code on CPUs as real platforms. They are customizable like simulators and show great functional fidelity. However, their experiment sizes rely on the available CPU cycles and memory, and cannot provide accurate performance results when emulating Gbps-level networks [12]. Third, we can also build test-beds composed of real devices. Limited by costs and resources, test-beds may only support small-scale experiments, whose results might not apply to large production networks. There are also some public test-beds [13]–[16]. They provide fidelity guarantees, but generally lack customizability for topologies, forwarding behaviors, and metrics. Furthermore, as networks evolve, test-beds shall be upgraded correspondingly, requiring repetitive capital investment. In summary, there remains a gap between real networks and existing network experiment approaches.

978-1-7281-6992-7/20/$31.00 ©2020 IEEE

To address this gap, we explore a new direction to reform network experiment approaches: emulating large networks within one real switch. BNV [10] leverages multiple OpenFlow [17] switches to emulate networks. However, due to the inflexibility of OpenFlow switches, BNV is fundamentally limited and fails in emulating large topologies and network performance metrics, such as link delay. In this paper, we propose *TurboNet*, a novel network emulator which leverages the power of programmable switches [18]–[22] to faithfully mimic functionality, scale, and performance of production networks. Programmable switches conduct re-configurable packet processing on data planes with high bandwidth. The flexibility of programmable switches enables agile customization of emulated networks, allowing users to validate their design in various network environments.

TurboNet aims at emulating both the data plane and control plane. For the data plane, *TurboNet* needs to emulate both the network topology and performance metrics such as background traffic, link delay, and link loss. First, to enable flexible topology emulation, *TurboNet* slices the programmable switch into multiple emulated switches via allocating separate ports for each emulated switch and generating the respective processing rules. Thus the networks that *TurboNet* emulates can have at most the same number of ports as the programmable switch ports. However, this design raises the first challenge: *how to emulate larger networks considering the limited ports in one programmable switch?* To solve this, we resort to the richer queue resources and take the physical queues as logical ports to emulate larger topologies. Further, *TurboNet* should be able to emulate various performance metrics, which brings the second challenge: *the gap between the limited data plane programmability of programmable ASIC and the logic of emulating performance metrics, especially for link delay*. It is non-trivial to delay a packet in a programmable switch for a fixed amount of time. To address this challenge, *TurboNet* designs the *delayed queue* with fixed queue length and queuing time via dynamically packet injection, and then

proposes the delayed queue-based link delay emulation. To accurately emulate network behaviors on the control plane, *TurboNet* provides both static and dynamic routing emulation. In particular, as for dynamic routing, *TurboNet* supports both distributed control plane, such as BGP agents, and centralized control plane, such as the SDN controller [23].

TurboNet is a practical network emulator that can be fully implemented by commodity programmable switches. In this paper, we make four major contributions.

- *TurboNet* is the first one that leverages the power of new-generation programmable switches for network emulation, which aims at helping users efficiently and faithfully verify their designs work correctly on production networks.
- We provide *TurboNet* API for operators to describe their emulation tasks (§III).
- We present the *TurboNet* design. On the data plane, *TurboNet* introduces port mapper, queue mapper, delayed queue, and so on to faithfully emulate network topologies and performance metrics (§IV). On the control plane, *TurboNet* supports static routing configurations, distributed routing agents, and centralized routing controller (§V).
- We implement *TurboNet* on a Tofino [21] switch. Evaluation results show that: (1) *TurboNet* API can effectively simplify network emulation tasks and reduce the code size by 10x; (2) *TurboNet* can flexibly emulate various topologies, including the 8-ary fat-tree and 260 real-world Internet topologies, and support almost 200 BGP agents with 25% peak CPU usage on the control plane; (3) *TurboNet* can accurately emulate different network performance metrics, such as 400Gbps background traffic injection, 10^{-8} loss, and µs-level to ms-level link delay (§VII).

II. Motivation and Related Work

A. Motivation

Operators rely on conducting network experiments to verify new network protocols, testify new routing algorithms, evaluate the performance of new proof-of-concept systems, and so on. The network experiment platforms should satisfy three requirements to verify the new networking-related designs can work correctly on production networks, as listed in Table I.

Functionality Fidelity. Functionality fidelity means that the experiment platforms should have the same implementation as the actual production networks. Unlike test-beds and emulators which run real devices or the same code as real devices, simulators model the real environment as a result of simulated events. As simulators cannot correctly model all details of real networks [24], the simulation results cannot represent real network environments and cannot guarantee fidelity.

TABLE I
NETWORK EXPERIMENT PLATFORMS.

	Simulator	Emulator	Test-Bed	NetProver
Functionality Fidelity	✗	✓	✓	✓
Performance Fidelity	✗	✗	✓	✓
Scale	✓	✗	✗	✓

Performance Fidelity. Based on functionality fidelity, performance fidelity further requires that experiment results should match the performance in real networks. Test-beds, which run on real devices, can achieve performance fidelity, while Emulators cannot since they typically run on CPUs. For example, in Mininet [7], resources are multiplexed in time by the default scheduler, which cannot guarantee that a host that is ready to send a packet will be scheduled promptly, or that all switches will forward at the same rate [25].

Scale. Besides emulating network behaviors with high precision and accuracy, the experiment platforms should also be scalable to support larger networks. Simulators achieves high scalability via modeling networks. While the experiment size of emulators usually depends on the available CPU cycles and memory [12]. Similarly, test-beds rely on hardware infrastructures to conduct experiments and are difficult to scale. Only expanding to more machines can solve this scalability issue under a fidelity premise.

In summary, none of the existing network experiment platforms can satisfy the above three requirements simultaneously. In this paper, we propose *TurboNet*, a network emulator which leverages programmable switches to reconcile functionality, performance, and scale for network emulation.

B. Related Work

This part presents related work on network experiment platforms.

Network Simulators. ns-2 [1] and ns-3 [2] are popular open-source network simulators and target primarily for research and educational use. NS4 [3] is a discrete-event network simulator for modeling a network containing P4-enabled [26] devices. OMNet++ [4] is another open-source C++-based simulator for modeling communication networks, multiprocessors, and other distributed or parallel systems. There are also some commercial simulators, *e.g.*, OPNET [5] and QualNet [6]. Compared with simulators which rely on CPUs to model networks, *TurboNet* provides more accurate performance emulation via emulating networks on real hardware.

Network Emulators. Mininet [7] is a popular container-based SDN emulator but faces scalability issues. Using a server with 3GHz of CPU and 3GB memory, Mininet can create at most 30 hosts connected with 100Mbps links and works poorly for 1Gbps links [12]. Compared with Mininet, *TurboNet* can emulate much larger networks with higher speeds. *TurboNet* makes full use of the high bandwidth of programmable switch via splitting one programmable switch into multiple emulated switches. Thus, the networks that *TurboNet* emulates can have at most the same total bandwidth as the programmable switch, which can be up to several Tbps. CrystalNet [8] is a cloud-scale network emulator by running real network device firmware in containers and virtual machines, but it cannot emulate real hardware forwarding behaviors for lack of data plane emulation. Emulab [9] is a time- and space- shared network emulation test-bed which provides emulation services through consistent use of virtualization and abstraction. BNV [10]

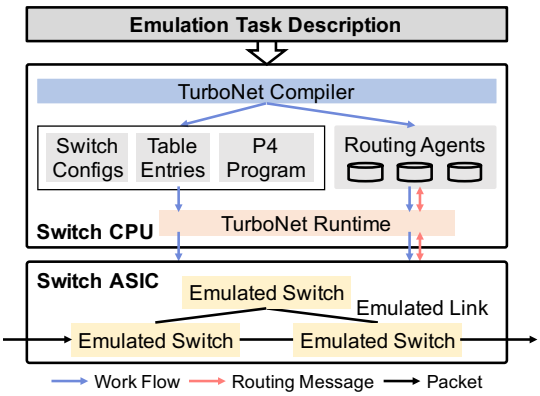


Fig. 1. **Architecture and workflow of *TurboNet*.**

aims to emulate arbitrary topologies using OpenFlow switches. OpenFlow assumes the switches have fixed behaviors and protocols. Thus BNV is not suitable for validating many new network designs, especially the ones requiring new protocols. Besides, limited by the inflexibility of OpenFlow switches, BNV cannot emulate some critical performance metrics, such as link loss and link delay.

Network Test-beds. CloudLab [13] provides an SDN environment with three moderately-sized data centers inter-connected with 100Gbps links. PlanetLab [14] is a geographically distributed network test-bed, which uses Linux vservers [27] to provide security isolation and a set of schedulers to provide resource isolation. GENI [15] is an integrated federation of testbeds throughout the United States. These hardware test-beds provide fidelity guarantees, but they are costly to scale and lack the flexibility for customizable topologies and forwarding behaviors. Compared with test-beds, *TurboNet* leverages the flexibility and high bandwidth of programmable switches to improve the customizability and scalability of network experiments.

III. Design Overview

A. Architecture and Workflow

Figure 1 shows the architecture and workflow of *TurboNet*. First, operators describe their network emulation tasks with given *TurboNet* API (§III-B). Then, for data plane emulation, *TurboNet* Compiler generates the P4 program, switch configurations, and table entries to construct topologies with specified performance metrics (§IV). Next, for control plane emulation, *TurboNet* may add or delete routing entries for static routing, create containers as routing agents for distributed routing, or establish control channels between *TurboNet* Runtime and a remote controller for centralized routing (§V). Finally, operators can conduct experiments on the emulated topology.

B. TurboNet API

TurboNet introduces a set of APIs to specify network emulation tasks, as listed in Table II.

Topology Construction. Operators can flexibly construct whatever topologies they want via adding switches, ports, and links. Meanwhile, they can specify the corresponding configurations, such as port bandwidth and queue number. We intentionally distinguish the *switch-ports* and *host-ports*, which are connected with switches and hosts, respectively, because they are differently treated when emulating topologies.

Metric Configuration. The network metrics include common link characteristics, *e.g.*, link delay and link loss, and background traffic injection. For link metrics, besides the specific link index and metric value, operators can also specify when the emulation will take effect, *i.e.*, the start time and end time. The background traffic emulation is used as a substitute for hosts. Based on a template packet, operators can inject traffic to a switch with specified injecting interval, jitter, start time, and end time. Moreover, any packet header field can be a constant value, a random value, a value from a given list, or an arithmetic regression. With these configurations, operators can customize the injected background traffic, *e.g.*, starting the SYN flood attack on a host.

Routing Configuration. *TurboNet* takes the destination IP-based routing as the default routing on each emulated switch, while operators can also specify their own routing policies with a P4 file. Notice that this P4 file can contain other network functions than routing policies. Besides, *TurboNet* provides convenient APIs to set the default table entry and add/delete entries on an emulated switch.

Example. Figure 2 gives a network emulation example. The middle portion illustrates how to describe the input topology (left) with *TurboNet* API, via adding switches (Line 2∼4), ports (Line 5∼10), and links (Line 11∼12), and then setting the corresponding metrics (Line 13∼14). The right side of the figure shows how *TurboNet* implements topology emulation, which will be introduced in §IV.

IV. Data Plane Emulation

In this section, we present how *TurboNet* emulates network data planes, *i.e.*, forwarding packets in a network with specified performance. *TurboNet* should answer the following two questions. The first one is how to emulate a complex topology within one switch (§IV-A). The second is how to emulate performance metrics to make the emulated network behave like real networks (§IV-B).

TABLE II
***TurboNet* API.**

Topology Construction	**Description**
add_switch (s)	Add a switch s.
add_hport (s,p,q,b)	Add a b (Gbps/Mbps) *host-port* p with q queues to switch s.
add_sport (s,p,q,b)	Add a b (Gbps/Mbps) *switch-port* p with q queues to switch s.
add_link (l,p_1,p_2)	Add a link l between two ports p_1 and p_2.
add_interal_host (h,sw)	Add a internal host h which connectes with switch sw.
Metric Configuration	**Description**
set_delay (l,v,t_1,t_2)	Set delay of link l as v (µs/ms) during time $[t_1,t_2]$.
set_loss (l,v,t_1,t_2)	Set loss rate v on link l during time $[t_1,t_2]$.
inject_pkt ($h,p,i,j,\{f,t,\{v\}\},t_1,t_2$)	Inject specified packets to switch s with interval i (ns/µs) and jitter j (ns/µs) during time $[t_1,t_2]$.
Routing Configuration	**Description**
set_route (s,c)	Configure routing policy c on switch s.
set_default ($s,t,spec$)	Set a default entry for routing table t on switch s.
add_entry ($s,t,spec$)	Add an entry to table t on switch s.
delete_entry ($s,t,spec$)	Delete an entry from table t on switch s.

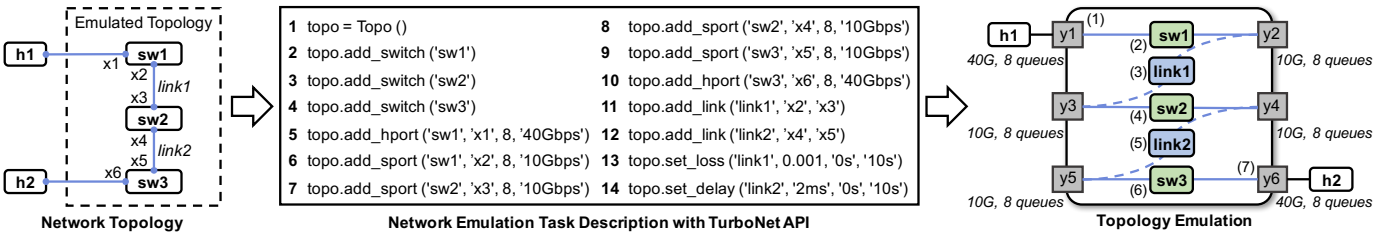


Fig. 2. **Network topology emulation example.**

A. Topology Emulation

In this part, we present how *TurboNet uses only one programmable switch to emulate various topologies flexibly*.

Topology Emulation Architecture. The key idea is to slice the programmable switch into multiple emulated switches. Specifically, we map each port in the input topology to an individual port on the programmable switch. Correspondingly, each switch in the input topology is also mapped to an individual set of ports with independent queues and bandwidth. As shown in Figure 3, the topology emulation comprises two modules. The first is Switch Emulation, which identifies the emulated switch and implements the corresponding processing logic. The second is Link Emulation, which connects the emulated switches via configuring loopback ports and uses additional match-action tables to emulate link metrics. Next, we will introduce these two modules, respectively.

Switch Emulation: The Switch Emulation module contains three parts to emulate switches. First, after receiving packets, Physical-to-Logical (P2L) Port Mapper matches with the ingress port on the programmable switch and outputs the corresponding switch and port in the input topology according to the port mapping relationship. Second, Emulated Switch contains match-action tables which set the egress port based on the routing policy of the current emulated switch. Besides routing tables, some other complex network functions, such as load balancing and firewalls, can also be implemented in each Emulated Switch. Third, Logical-to-Physical (L2P) Port Mapper reverts the egress port in the input topology to the port on the programmable switch and forwards packets.

Link Emulation: A naive method to emulate links is artificially connecting two ports on the same link with a cable, which is troublesome and adds extra cable costs. We solve this via subtly configuring the ports as loopback mode. Each loopback port acts as a unidirectional link to transfer packets to the other switch that connects to itself. Notice that only the ports connected to other switches should be configured as loopback mode, while the other ports are connected to hosts with cables. We call the former *switch-ports* and the latter *host-ports*. These two types of ports should be differentiated when operators call *TurboNet* API. To emulate link metrics, the Link Emulation module has similar processing logic to Switch Emulation: P2L Port Mapper to identify the current link and Emulated Link for metric emulation in §IV-B. The difference is that Link Emulation does not forward packets, and thus L2P Port Mapper is not needed.

Topology Emulation Workflow. The example in Figure 2

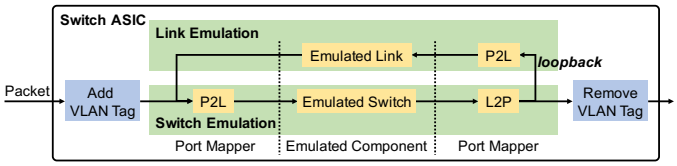


Fig. 3. **Architecture and workflow of topology emulation.**

shows how topology emulation works. We map each port in the input network topology, *i.e.*, $x1 \sim x6$, to an individual port on the programmable switch, *i.e.*, $y1 \sim y6$. The *switch-ports* ($y2 \sim y5$) are configured as loopback mode, while *host-ports* ($y1$ and $y6$) should be connected to hosts with cables. (1) When packets get into the programmable switch from a *host-port*, *e.g.*, $y1$, we add a VLAN tag, which will be introduced later. (2) Packets from $y1$ are considered to arrive at $sw1$ from $x1$. Assume that each emulated switch forwards packets from one port to the other port. Then the packets should be forwarded to $x2$, which is mapped to $y2$. Depending on the egress port type, packets either leave the emulated topology or get into the next emulated switch via a link. (3) As $y2$ is a loopback port, packets will be received from $y2$ again. Link Emulation module identifies the current link, *i.e.*, $link1$, and emulates the link metrics, *i.e.*, 10^{-3} loss rate. (4) Then packets get into the next emulated switch. As $y2$ acts as a unidirectional link which transfers packets from $sw1$ to $sw2$, packets from $y2$ are considered to be from $x3$ on $sw2$. Therefore packets are forwarded to $x4$ which is mapped to $y4$. (5) Like Step 3, packets are received by $y4$ and delayed on $link2$. (6) Like Step 4, packets are sent to $y6$ by $sw3$. (7) Finally, as $y6$ is a *host-port*, packets will leave the emulated topology. Before they are sent, we restore the original packets and remove the VLAN tag.

Emulation Optimization with Queue Mapper. A key limiting factor on topology emulation is the available ports in one programmable switch. Suppose that there are x 100G port groups in one programmable switch, and each port group can be subdivided to 1x100G, 2x40G, 4x25G, or 4x10G ports, the total port number in the emulated topology can never exceed $4x$. To emulate larger topologies, we observe that the 32 queues under each port group can be taken as logical ports. On the one hand, programmable switches support specifying the egress queue for each packet, just like the egress port. On the other hand, with maximum bandwidth shaping per queue, we can configure queue bandwidth to emulate specified input port bandwidth.

Therefore, we propose to use Queue Mapper to emulate larger topologies. Each *switch-port* is directly mapped to

TABLE III
NOTATIONS OF PM AND QM PROBLEMS.

(Input) Topology emulation requirements and usable programmable switch ports:	
$S_{1,2}$	Set of *switch-ports* (S_1) and *host-ports* (S_2) in the input topology.
S	Set of ports in the input topology. ($S = S_1 \cup S_2$)
b_k	Required bandwidth for port $k \in S$.
q_k	Required queues for port $k \in S$.
P	Set of available 100G port groups on the programmable switch.
(Output) Port mapping relationship:	
$x_i^{1,2,3,4}$	0-1 binary variable indicating port group $i \in P$ is occupied and configured as 1x100G (x_i^1), 2x40G (x_i^2), 4x25G (x_i^3), or 4x10G (x_i^4)
$x_{ik}^{1,2,3,4}$	0-1 binary variable indicating port group $i \in P$ is configured as 1x100G (x_i^1), 2x40G (x_i^2), 4x25G (x_i^3), or 4x10G (x_i^4) and has a sub-port mapped to port $k \in S$.
(Output) Queue mapping relationship:	
y_i	0-1 binary variable indicating port group $i \in P$ has queues occupied.
y_{ik}	0-1 binary variable indicating port $k \in S$ is mapped to queues of port group $i \in P$.

TABLE IV
FORMULATION OF PM AND QM PROBLEMS.

PM Problem	$\boldsymbol{x_i^{1,2,3,4}, x_{ik}^{1,2,3,4} = PM(S, b_k, q_k, P)}$
Objective	$min: \sum_{i \in P}(x_i^1 + x_i^2 + x_i^3 + x_i^4)$
Constraints	$\boldsymbol{C_1}: x_i^1 + x_i^2 + x_i^3 + x_i^4 \le 1, \forall i \in P$ $\boldsymbol{C_2}: \sum_{k \in S} x_{ik}^1 \le x_i^1, \sum_{k \in S} x_{ik}^2 \le 2 \cdot x_i^2, \sum_{k \in S} x_{ik}^3 \le 4 \cdot x_i^3, \sum_{k \in S} x_{ik}^4 \le 4 \cdot x_i^4, \forall i \in P$ $\boldsymbol{C_3}: \sum_{i \in P}(x_{ik}^1 + x_{ik}^2 + x_{ik}^3 + x_{ik}^4) = 1, \forall k \in S$ $\boldsymbol{C_4}: \sum_{i \in P}(100 \cdot x_{ik}^1 + 40 \cdot x_{ik}^2 + 25 \cdot x_{ik}^3 + 10 \cdot x_{ik}^4) \ge b_k, \forall k \in S$ $\boldsymbol{C_5}: \sum_{k \in S} q_{ik} \cdot (x_{ik}^1 + x_{ik}^2 + x_{ik}^3 + x_{ik}^4) \le 32 x_i, \forall i \in P$
QM Problem	$\boldsymbol{y_i, y_{ik}, x_i^{1,2,3,4}, x_{ik}^{1,2,3,4} = QM(S_1, S_2, b_k, q_k, P)}$
Objective	$min: \sum_{i \in P} x_i^1 + x_i^2 + x_i^3 + x_i^4 + y_i$
Constraints	$\boldsymbol{C_1}: x_i^1 + x_i^2 + x_i^3 + x_i^4 + y_i \le 1, \forall i \in P$ $\quad \boldsymbol{C_4}: \sum_{i \in P} y_{ik} = 1, \forall k \in S_1$ $\boldsymbol{C_2}: x_i^{1,2,3,4}, x_{ik}^{1,2,3,4} = PM(S_2, b_k, q_k, P)$ $\quad \boldsymbol{C_5}: \sum_{k \in S_1} q_k y_{ik} \le 32 y_i, \forall i \in P$ $\boldsymbol{C_3}: \sum_{k \in S_1} b_k y_{ik} \le 100 y_i, \forall i \in P$

individual queues. Notice that *host-ports* are connected to hosts by cables, so they should still be mapped to the physical ports on the programmable switch. However, after loopback ports send packets back to the programmable switch, we cannot know the previous egress queue, which is essential for P2L Queue Mapper to identify the current switch and link. Therefore, we modify the VLAN ID as the egress queue in L2P Queue Mapper. The 12-bit VLAN ID is enough to cover the queues under a port. Then P2L Queue Mapper can match with both the ingress port and VLAN ID to determine the current switch or link. Notice that VLAN is just a way to distinguish emulated switches and we can replace it with a variety of protocols, even using headers of our own design.

Performance Isolation Analysis: TurboNet emulates multiple switches on one programmable switch, and thus performance isolation is critical for emulation fidelity. When Port Mapper is employed, i.e., each emulated switch has its separate set of ports, traffic among different ports cannot affect each other since they have separate bandwidth and buffer. The same is true for Queue Mapper when queue bandwidth shaping is enabled and queue sizes are allocated for each emulated switch. Our evaluation in §VII-E will show *TurboNet*'s performance isolation effect.

Next, we will introduce how to map the ports in the input topology to the ports (Port Mapper) or queues (Queue Mapper) on the programmable switch. We formalize these two problems as 0-1 Integer Linear Programming (ILP) problems.

Port Mapper (PM) Problem. Table III shows the related notations of the PM problem. The input variables include the topology emulation requirements, *i.e.*, the bandwidth and queue number of ports in the input topology (S, b_k, q_k), and available port groups on the programmable switch (P). The output variables are binary and contain two parts. The first is how a port group should be configured if occupied ($x_i^{1,2,3,4}$). Considering that each port group can be subdivided to various sets of sub-ports, we should correctly configure them to match the actual input port bandwidth. The second is the one-to-one mapping relationship between input ports and sub-ports under each port group on the programmable switch ($x_{ik}^{1,2,3,4}$).

Object and Constraints: To reserve more bandwidth for other purposes such as link delay emulation in §IV-B, we hope to minimize the number of used ports on the programmable switch. Table IV lists the constraints. $C_1 \sim C_3$ guarantee the one-to-one mapping relationship between the input ports and sub-ports on the programmable switch. C_4 requires that the mapped port bandwidth on the programmable switch should not be smaller than the input port bandwidth. Bandwidth shaping will be employed if more bandwidth is allocated. C_5 gives the queue constraint and is based on the fact that programmable switches support arbitrary queue distribution for each sub-port under a port group.

Queue Mapper (QM) Problem. The QM problem takes the same input as PM but has different outputs. For *host-ports*, which are mapped to physical ports, QM has the same output as PM. For *switch-ports*, we define another two binary variables (y_i, y_{ik}) to represent the queue mapping relationship.

Object and Constraints: Similar to PM, QM also hopes to minimize the utilized port groups. The constraints are listed from two aspects, *i.e.*, *host-ports* and *switch-ports*. C_1 requires that a port group can be mapped to either *switch-ports* or *host-ports*. C_2 represents the mapping relationship of *host-ports*. $C_3 \sim C_5$ give the constraints for *switch-ports*. Specifically, C_4 requires that each *switch-port* should be mapped to queues from the same port group. C_3 and C_5 constrain that each port group has at most 100Gbps bandwidth and 32 queues to allocate for the input ports.

We take PuLP [28], which is an open-source linear programming modeler in Python, to solve the PM and QM problems. Then *TurboNet* Compiler will generate the corresponding table entries and switch configurations for topology emulation.

B. Emulating Network Metrics

In this part, we present how *TurboNet* emulates performance metrics to make the network behave like real networks. Table V summarizes the emulating capability of *TurboNet* in terms of background traffic, link loss, and link delay.

1) Background Traffic: An important motivation is that sometimes operators only need to monitor the traffic behaviors from a small number of hosts. In contrast, many other hosts are used to generate background traffic. Therefore, we provide background traffic emulation as internal hosts to lower emulation costs. A side benefit of background traffic emulation is that some *host-ports*, which should have been connected with hosts, are saved, and thus larger topologies can be emulated.

Programmable switches provide a way to inject packets into switch pipelines. Each pipeline has a 100Gbps packet generator engine capable of injecting packets periodically [29]. The control plane can configure the packet injection interval and jitter, and decide when the injection starts and ends. However, all injected packets are from fixed packet templates. To flexibly customize background traffic, we define match-action tables on the data plane to modify user-specified parsed packet fields. We refer to *Editor* in [30] and support four types of header field modification, *i.e.*, a constant value, a random value, a value from a given list, and an arithmetic regression. The implementation is similar to [30] and omitted here. Then operators can flexibly customize the injected traffic, *e.g.*, emulating the SYN flood attack.

To inject packets to specified switches like real hosts, we allocate a unique VLAN ID for each target switch. Before packets are injected into the pipeline, we add a VLAN tag and set VLAN ID as the corresponding target switch. The P2L Mapper can identify the current emulated switch based on this VLAN ID. Then the injected packets can be processed like normal packets in the emulated topology, *i.e.*, forwarded/received by emulated switches and finally sent to a host. However, if the destination host is another internal host, L2P Mapper in the last emulated switch will not find any matched egress port on the programmable switch, and thus packets will be dropped. That is, using background traffic to emulate hosts can only send but cannot receive packets. With 100Gbps packet generator bandwidth each pipeline, *TurboNet* provides at most 400Gbps background traffic bandwidth on a 4-pipe switch. Therefore, *TurboNet* can emulate at most $400/x$ internal hosts if each host requires xGbps bandwidth.

2) Link Loss: P4 has no direct method to perform an action according to a probability. However, it provides a primitive *modify_field_rng_uniform* to generate a random integer number v from a given range between a lower bound 0 and an upper bound $2^n - 1$. Therefore, v can be one of 2^n equally possible values, and the probability of $v < \lambda$ is $\lambda/2^n$ for a given threshold λ. To drop packets with a probability p, we just need to let $\lambda/2^n = p$. For each packet, we generate a random number v, compare v with a threshold λ, and drop the packet if $v < \lambda$. Larger n means higher link loss emulating accuracy. Our evaluation will show that $n = 32$ is enough, and the emulated loss can be as low as 10^{-8} with relative error $< 1\%$. To emulate different loss rates for various links, we need to set different thresholds and compare the generated random value with the corresponding threshold.

TABLE V
SUMMARY OF METRIC EMULATING CAPABILITY OF *TurboNet*.

Network Metric	Emulating Capability
Background Traffic	Up to 400Gbps line-rate customizable traffic injection, *i.e.*, emulating y hosts, each with $\frac{400}{y}$Gbps throughput.
Link Loss	As small as 10^{-8} link loss emulation with relative error $< 1\%$.
Link Delay	μs-level to ms-level link delay emulation with per-ms deviation $< 9\mu$s.

3) Link Delay: Link delay varies a lot in real networks, which can be from nanosecond-level in data center networks to millisecond-level in wide-area networks [31]. Therefore, how to emulate link delay for different networks is a challenge. In *TurboNet*, loopback ports act as links to transfer packets between switches with nanosecond-level delay t_l. To emulate larger link delay t_p, we should bridge the delay gap $\tau = t_p - t_l$. A key observation is that the queuing time is in proportion to the queue depth in programmable switches (§VII-C). Therefore, we can maintain the queue depth to achieve target queuing time. The high-level idea is to *(1) implement delayed queues with desired queuing time, and (2) send packets to the delayed queues to emulate target link delay*. Notice that delayed queues should be from *reserved ports*, which are not used in topology emulation, to isolate delayed traffic with other traffic.

Delayed Queue with Target Queuing Time. The key idea for delayed queue implementation is to dynamically inject *backup packets* to the delayed queue for target queuing time. In particular, we inject backup packets to the switch and decide whether to send these packets to the delayed queue based on the up-to-date queuing time. However, the queue information can only be obtained in the egress pipeline, while the egress port and egress queue must be specified in the ingress pipeline. To address this challenge, we first specify another reserved loopback port as the egress port for these injected packets. These packets will read the queuing time of the delayed queue in the egress pipeline and decide whether to go to the delayed queue when they return to the switch again. Figure 4 shows the implementation of a delayed queue. We place a register, *i.e.*, *QTime*, in the egress pipeline to record the up-to-date queuing time of the delayed queue. When packets are dequeued from the delayed queue, they update *QTime* as their queuing time. Meanwhile, we use the internal packet generator to inject *backup packets* to another *reserved* loopback port R_0. In the egress pipeline, the *backup packets* read *QTime*, compare the register value q with the desired queuing time τ, and record the result in a 1-bit header field. When they return to the switch via R_0, they may be sent to the delayed queue based on this 1-bit value. If $q < \tau$, they should go to the delayed queue and also update *QTime*. Otherwise, they go to R_0 to continually check the queuing time.

Link Delay Emulation. As the queuing time has upper limits, packets may go to multiple delayed queues for higher delay. Figure 5 shows how *TurboNet* emulates link delay with

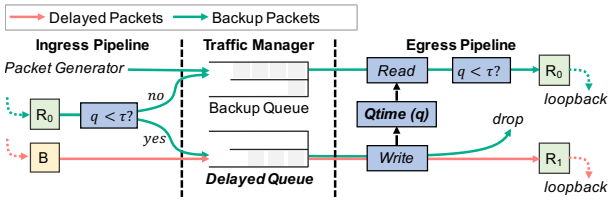


Fig. 4. **Delayed queue with target queuing time τ.**

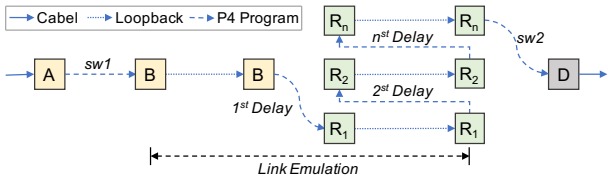


Fig. 5. **Link delay emulation.**

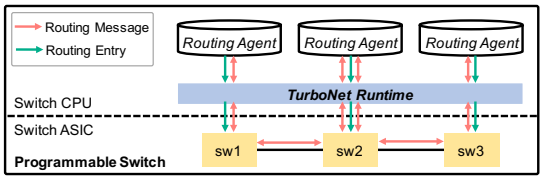


Fig. 6. **Distributed routing emulation.**

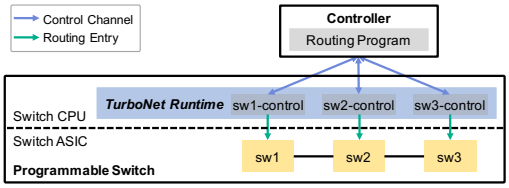


Fig. 7. **Centralized routing emulation.**

delayed queues. Normally, the loopback port B acts as a link to transfer packets from the emulated switch *sw1* to *sw2*. To emulate the target link delay, packets received from port B are sent to n delayed queues in turn before they get into *sw2*. $R_1 \sim R_n$ are n *reserved* loopback ports and each has a delayed queue with queuing time τ_i. Then the total emulated delay includes queuing time $\sum_{i=1}^{n} \tau_i$, pipeline time except queuing time $t_p * n$, and port loopback time $t_l * n$. As t_l and t_p of a P4 program are fixed (§VII-C), to bridge the delay gap τ, the queuing time of these n delayed queues should satisfy $\sum_{i=1}^{n} \tau_i = \tau - n * (t_l + t_p)$. When packets finish link delay emulation and go to the next emulated switch *sw2*, the ingress port becomes R_n instead of B. However, the emulated switch relies on the ingress port for port mapping in P2L Mapper. To make the delayed packets identified by *sw2*, we add a table entry, which matches with R_n and outputs the corresponding switch and port index.

Accuracy and Resource Analysis: Stable queuing time guarantees that all *delayed packets* obtain the same link delay, and thus throughput is not affected. There are three premises for stable and accurate queuing time. First, the delayed queue bandwidth should not be smaller than the emulated link bandwidth. Otherwise, the queue depth might keep increasing, even if no *backup packets* are enqueued. Second, the bandwidth of *backup packets* should be larger than the delayed queue bandwidth. This guarantees that enough *backup packets* can be injected to maintain queue depth, even if there are few *delayed packets* on the link. Third, the time from reading queuing time to getting into the delayed queue should be small enough to make timely decisions. According to our measurements in §VII-C, the time from dequeuing to enqueuing again via a loopback port is no more than 620ns. Therefore, queue depth and queuing time variations are minimal, when the target queuing time is much larger than 620ns. Based on the above, each delayed queue implementation requires double emulated link bandwidth for *reserved ports*. Therefore, we should minimize the used ports in topology emulation to satisfy delay emulating requirements for more links. According to our evaluation in §VII-C, one 10Gbps delayed queue can provide 1ms delay emulation with deviation $<$ 9µs. *TurboNet* can further utilize multiple delayed queues to emulate µs-level to ms-level link delay with per-ms deviation $<$ 9µs.

V. Control Plane Emulation

Control plane emulation is also important and mimics how routing protocols or programs control the production networks. To accurately emulate the network behaviors on the control plane, *TurboNet* provides both static and dynamic routing.

The routing tables reside at Emulated Switch in Figure 3. Each emulated switch takes the destination IP-based routing by default. Users can also define their own routing policies, *e.g.*, ECMP and MPLS, for each emulated switch via specifying routing tables. For each received packet, *TurboNet* performs the corresponding routing policies based on the current switch index acquired from P2L Mapper.

Static Routing. For static routing, routing entries are added or deleted manually. Operators need not know how the ports in the input topology are mapped to ports on the programmable switch, but just need to call the routing configuration APIs in Table II to manage table entries in emulated switches.

For dynamic routing, operators can run any routing algorithms in a centralized or distributed manner as they desire.

Distributed Routing. To enable distributed routing, *TurboNet* creates a separate container as a routing agent for each emulated switch, as shown in Figure 6. Operators can specify the routing protocol, *e.g.*, OSPF, and run their routing algorithms, *e.g.*, Dijkstra, in routing agents. The routing agents inject routing messages to the emulated switch and manage routing entries on the emulated switch. *TurboNet* Runtime on the switch CPU acts as a proxy to transfer routing messages and add/delete routing entries on emulated switches.

Centralized Routing. *TurboNet* supports centralized routing with a real remote controller, as shown in Figure 7. Operators can run their routing program on the controller. Each emulated switch has a local control plane on *TurboNet* Runtime to establish a control channel with the controller. The controller collects network status from local control planes, calculates routing entries, and distributes the entries to different local control planes. Then the control planes add or delete routing entries on the corresponding emulated switches.

VI. Discussions

In this part, we discuss four issues related to further extension of *TurboNet*. The first issue is about the scalability of *TurboNet*, *i.e.*, how to emulate larger networks which requires massive switch resources. Since *TurboNet* emulates a whole network on one programmable switch, the limited switch resources becomes a bottleneck for emulation. There are four types of resources required by different emulated components of *TurboNet* and may become the emulation bottleneck. First, the available *switch bandwidth or ports* is the key resource for topology and link delay emulation. Emulating larger topologies means fewer switch ports for delay emulation, and the insufficient switch ports may cause the emulation to fail. Second, the SRAM and TCAM may be the key resources for routing when routing tables contain massive entries. Third, the packet generator bandwidth is important for background traffic generation and link delay emulation. Fourth, the CPU and memory on the control plane may become scarce resources required by the dynamic routing, either to create distributed routing agents or to establish control channels with a remote controller. In general, not all emulation tasks can be performed by *TurboNet* for resource limitations, but there are some approaches to overcome these limitations. For example, we can use external memory to extend no-chip memory [32], combine multiple programmable switches for more switch ports, and replace the original switch CPU with a more powerful CPU.

The second issue is about the fidelity of *TurboNet*, *i.e.*, whether the emulation are accurate when switch resources are insufficient. Given an emulation task, *TurboNet* Compiler will check whether there are adequate hardware resources for network emulation and whether the generated P4 program can be deployed on the programmable switch. *TurboNet* can guarantee emulation fidelity with enough resources, while the emulation will not be performed if resources are insufficient.

The third issue is about the generality of *TurboNet*, *i.e.*, whether *TurboNet* can accommodate to other programmable devices. Besides re-configurable match-action tables in P4, *TurboNet* also requires three other hardware capabilities including port/queue bandwidth shaping, port loopback modes, and packet generator. The former two capabilities are widely supported by programmable switches [33], [34], while the third one, *i.e.*, packet generator, is not. We admit that packet generator is a requisite to emulate background traffic and link delay, but it will not affect topology and link loss emulation. For programmable switches without internal packet generator, we can resort to external packet generator if necessary.

The fourth issue is about *TurboNet*'s two inherent shortcomings. First, *TurboNet* mainly focuses on emulating the intermediate network, *i.e.*, the switches, while host emulation is beyond programmable switches. Although we provide background traffic injection as an alternative, the background traffic only supports sending packets with simple header field modification but cannot receive packets. Therefore, *TurboNet* cannot emulate client/server applications. In other words, we have to rely on external packet generator for complex traffic patterns. Second, since *TurboNet* emulates network-related designs via P4 programs, *TurboNet* cannot reflect device-dependent details. Thus, *TurboNet* may miss some hardware failures, *e.g.*, faulty memory modules, processors, and ports, which can be found by test-beds composed of true devices.

VII. Evaluation

A. Overview

We evaluate *TurboNet* with one Barefoot Tofino switch and two servers. Equipped with an Intel Pentium 4-core 1.60GHz CPU and 8GiB memory, the switch has two packet processing pipelines for 3.2Tbps port bandwidth and 200Gbps packet generator bandwidth. The servers both have a 12-core Intel Xeon E5-2620 2.40GHz CPU and connect to the switch via an Intel XL710 (40Gbps) and two Intel 82599ES (10Gbps) NICs, respectively. To evaluate *TurboNet*, we first emulate various data center and Internet topologies to prove *TurboNet*'s topology emulating capability. Then we take the 4-ary fat-tree as an example and demonstrate *TurboNet*'s emulation capability for various network components. In total, we have the following five evaluation goals.

- **Topology emulating capability**: How large are the topologies that *TurboNet* can emulate (§VII-B)?
- **Metric emulating accuracy**: How accurately can *TurboNet* emulate various network metrics, including link delay, link loss, link bandwidth, and background traffic (§VII-C)?
- **Control plane emulating capability**: How complex are the control plane *TurboNet* can emulate with limited CPU and memory resources (§VII-D)?
- **Performance isolation**: How much isolation can *TurboNet* guarantee (§VII-E)?

B. Topology Emulating Capability

In this part, we evaluate the topology emulating capability of *TurboNet* and compare *TurboNet* with BNV [10]. We take various real-world topologies as input. The topologies include data center topologies, *e.g.*, fat-tree [35] and VL2 [36], and 261 Internet topologies from [37]. For *TurboNet*, we evaluate both PM-based and QM-based topology emulation (hereafter referred to as PM and QM). For BNV, we present the emulating capability with static loopback links and software-configurable loopback. The results are shown in Table VI.

PM and QM Comparison for *TurboNet*. The topology emulating capability can be divided into two aspects: topology size and link bandwidth. We compare the emulation capability of PM and QM by changing the input topology size and link bandwidth, respectively. First, we change k in the fat-tree, *i.e.*, the topology size, and show the maximum link bandwidth that *TurboNet* can emulate. A k-ary fat-tree has $k^3/4$ *host-ports* and k^3 *switch-ports* [35]. We can see that on a 4-pipe switch, PM can only emulate the 4-ary fat-tree, while QM can emulate the 8-ary fat-tree. As the total available port bandwidth is fixed, the maximum link bandwidth for QM decreases with k increasing. Second, we change the link bandwidth in VL2 and show the maximum $D_A D_1$, *i.e.*, the topology size, that

TABLE VI
TOPOLOGY EMULATING CAPABILITY COMPARISON.

Topology		TurboNet				BNV	
		2-Pipe (32*100G) Programmable Switch		4-Pipe (64*100G) Programmable Switch		64*100G SDN switch	
		PM	QM	PM	QM	Static Loopback	L2-Switch
Fat-Tree	$k=4$	$Link \le 25G$	$Link \le 25G$	$Link \le 40G$	$Link \le 40G$	$Link \le 40G$	$Link \le 40G$
	$k=6$	×	$Link \le 8.3G$	×	$Link \le 20G$	×	×
	$k=8$	×	×	×	$Link \le 6.2G$	×	×
VL2	$Link = 1G, 10G$	$D_A D_1 \le 16$	$D_A D_1 \le 20$	$D_A D_1 \le 36$	$D_A D_1 \le 44$	$D_A D_1 \le 24$	$D_A D_1 \le 36$
	$Link = 10G, 40G$	$D_A D_1 \le 20$	$D_A D_1 \le 20$	$D_A D_1 \le 40$	$D_A D_1 \le 40$	$D_A D_1 \le 32$	$D_A D_1 \le 40$
261 Internet Topologies		199 (76.2%)	259 (99.2%)	248 (95.0%)	260 (99.6%)	248 (95.0%)	248 (95.0%)

TurboNet can emulate. According to [36], a VL2 topology has $5D_AD_1$ *host-ports* and $2D_AD_1$ *switch-ports* when the server links and switch links are 1Gbps and 10Gbps, or $2D_AD_1$ *host-ports* and $2D_AD_1$ *switch-ports* when the links are 10Gbps and 40Gbps. We can see that when link bandwidth is small, QM can emulate larger topologies than PM. However, when link bandwidth increases to 10Gbps and 40Gbps, the bandwidth instead of port number becomes the limiting factor, and thus QM presents no improvement compared with PM.

We also compare the emulating capability of PM and QM over 261 Internet topologies. The link bandwidth is 10Gbps by default if not specified in the input topology. As the table shows, QM can emulate 60 more topologies than PM on a 2-pipe switch. This is because that these 60 topologies have many links with small bandwidth, which can be further mapped to queues using QM. The only topology that cannot be emulated using QM on a 4-pipe switch is the Kdl topology, which contains 899 links, and each link bandwidth is 10Gbps by default. The total port bandwidth then comes to 18Tbps and is beyond the programmable switch. In summary, the above results prove that compared with PM, QM improves the topology emulating capability in terms of topology size, especially for smaller link bandwidth.

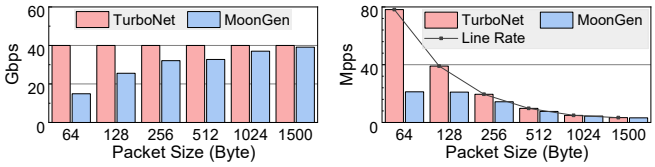
***TurboNet* and BNV Comparison**. To achieve fairness, we assume that BNV has the same port bandwidth as *TurboNet*, *i.e.*, one SDN switch with 64 100Gbps port groups, instead of five 1Gbps SDN switches in the original BNV test-bed. BNV supports two types of loopback links. The first is static loopback links. We take the original configuration in the BNV test-bed. Half of the ports are connected to servers, and the remaining half is used to form loopback links. As the loopback links are fixed, the topology that BNV can emulate is smaller than *TurboNet* PM. The second is software-configurable loopback, which requires another L2 intermediate switch to dynamically configure loopback links. With dynamical loopback links, BNV achieves the same emulating capability as *TurboNet* PM. However, whether utilizing static or dynamic loopback links, BNV's topology emulating capability is always not at par with *TurboNet* QM.

C. Metric Emulating Accuracy

In this part, we evaluate the network metric emulating accuracy of *TurboNet* in terms of background traffic, link delay, link loss, and link bandwidth.

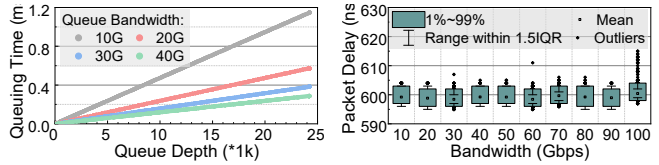
TABLE VII
TOPOLOGY EMULATING CAPABILITY WITH HOST EMULATION.

Topology		Without Host Emulation		With Host Emulation	
		PM	QM	PM	QM
VL2	$Link = 1G, 10G$	$D_A D_1 \le 36$	$D_A D_1 \le 44$	$D_A D_1 \le 92$	$D_A D_1 \le 112$
	$Link = 10G, 40G$	$D_A D_1 \le 40$	$D_A D_1 \le 40$	$D_A D_1 \le 48$	$D_A D_1 \le 48$
Fat-Tree	$Link = 1G$	$k \le 4$	$k \le 8$	$k \le 6$	$k \le 12$
	$Link = 10G$	$k \le 4$	$k \le 6$	$k \le 6$	$k \le 6$
	$Link = 40G$	$k \le 4$	$k \le 4$	$k \le 4$	$k \le 4$



(a) Gbps (b) Mpps

Fig. 8. **Background traffic throughput comparison.**



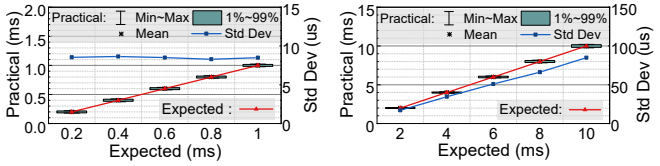
(a) Queuing time vs. enqueue depth (b) Delay from dequeuing to enqueuing via a loopback port

Fig. 9. **Measurements of queuing time and packet delay.**

Background Traffic. *TurboNet* introduces background traffic emulation as a substitute for hosts to lower emulation costs. Figure 8 compares the throughput between background traffic emulation in *TurboNet* and MoonGen [38] for different packet sizes. The throughput test of MoonGen is performed on a 40Gbps port. To guarantee fairness, we test the throughput of *TurboNet* using the packet generator on a single pipeline with bandwidth limited within 40Gbps. We can see that *TurboNet* can always achieve line-rate for all packet sizes, while MoonGen cannot generate small-sized packets at line-rate. For 64-byte packets, the throughput of MoonGen is 21.2Mpps, 3.7x smaller than *TurboNet*.

Besides lowering host costs, another benefit of background traffic emulation is that larger topologies can be emulated, as the ports that should have been connected with hosts are saved now. Table VII shows the topology emulating capability improvement when using 400Gbps background traffic to emulate hosts. We can see that host emulation can significantly increase the emulated topology size especially for smaller link bandwidth, since more hosts can be emulated. For the VL2 topology with 1Gbps server links and 10Gbps switch links, using background traffic to emulate hosts can increase the emulated size by $>$ 2x.

Link Delay. Figure 9 demonstrates some measurement results, which are the premises of link delay emulation. Specifically, Figure 9(a) shows the proportionality relationship between queue depth at the packet enqueue time and queuing time. We can see that queue bandwidth also affects queuing time. At the same enqueue depth, queuing time is inversely proportional to queue bandwidth. Besides, the maximum queuing time of a 10Gbps queue is smaller than 1.2ms, and thus multiple delayed queues are necessary to emulate larger delay. Figure 9(b)



(a) One-pass link delay accuracy (b) Multi-pass link delay accuracy

Fig. 10. **Link delay accuracy.**

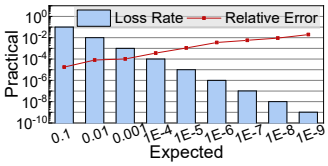


Fig. 11. **Link loss accuracy.**

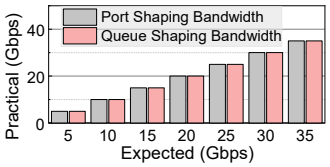


Fig. 12. **Link bandwidth accuracy.**

shows the packet delay from dequeuing to enqueuing via a loopback port. As the queue bandwidth ranges from 10Gbps to 100Gbps, the packet delay always fluctuates around 600ns, and the variations are smaller than 20ns. This small delay is the premise for accurate link delay emulation.

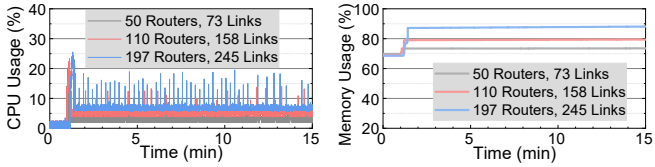
We test the practical link delay when emulating different target link delay. Figure 10(a) shows the one-pass delay, *i.e.*, packets pass a 10Gbps delayed queue only once. As the expected delay ranges from 0.2ms to 1ms, the practical delay presents minimal difference from the expected delay with standard deviation $<$ 9µs. Figure 10(b) shows the multi-pass delay, *i.e.*, packets pass the delayed queues more than once. To achieve from 2ms to 10ms target delay, packets pass the delayed queues with 1ms queuing time from 2 to 10 times. As we can see, with packets passing the delayed queues more, delay variations also increase linearly, but the per-ms deviation is always $<$ 9µs, *i.e.*, the relative deviation is $<$ 0.9%.

Link Loss. Figure 11 shows the link loss emulating accuracy using the 32-bit random number generation. We set different loss rates and check the deviation of practical link losses from expected ones. As the figure shows, when the expected loss changes from 0.1 to 10^{-8}, *TurboNet* can always accurately emulate link loss with relative error $<$ 1%.

Link Bandwidth. In §IV-A we employ port/queue bandwidth shaping to achieve specified input port bandwidth, *i.e.*, link bandwidth. Therefore, the accuracy of port/queue bandwidth shaping decides link bandwidth emulating accuracy. Figure 12 shows the practical port and queue bandwidth on a 40G port when different shaping thresholds are configured. As we can see, both the port and queue bandwidth shaping are accurate for different shaped bandwidth.

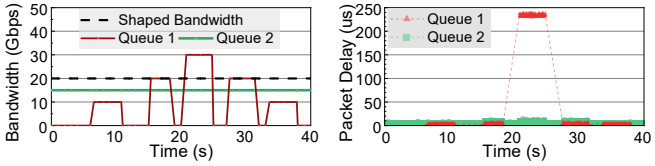
D. Control Plane Emulating Capability

Compared with static routing and centralized routing, the distributed routing requires more control plane resources because an individual routing agent is created for each router. Therefore, we mainly test the distributed routing emulating capability for control plane evaluation. We take three real Internet topologies from [37] which represent small, median, and large networks with router and link numbers are 50 and 73, 110 and 158, and 197 and 245, respectively. BGP is used as the only routing policy and we assume that all connected routers



(a) CPU (b) Memory

Fig. 13. **CPU and memory usage of distributed routing on control plane.**



(a) Bandwidth (b) Packet Delay

Fig. 14. **Queue performance isolation.**

are BGP peers. Figure 13 shows the CPU and memory usage for different topology sizes. The CPU usage peaks in the first few minutes and then decreases but still periodically reaches smaller peaks. The memory usage changes differently, which increases at first to create routing agents and then remains stable. We can see that *TurboNet* can easily support almost 200 BGP routing agents with 25% peak CPU usage.

E. Performance Isolation

TurboNet emulates topologies via allocating separate port and queue resources for each switch in the input topology. The ports are physically isolated in programmable switches, while the queues share bandwidth resources on the same port. Figure 14 shows the performance isolation effect with queue bandwidth shaping. Queue 1 and Queue 2 are two queues under the same port. We configure the maximum bandwidth shaping to limit the bandwidth of both queues should not exceed 20Gbps, and set the overall port bandwidth as 40Gbps. In theory, the port should cease to schedule traffic from the queue if the queue reaches the configured bandwidth. To test the performance isolation effect, we always send 15Gbps traffic to Queue 1 while changing the traffic sent to Queue 2, and observe the practical receiving bandwidth and packet delay of both queues. We can see that the traffic variations of Queue 1 have little impact on Queue 2. That is, queue bandwidth shaping strictly guarantees performance isolation.

VIII. Conclusion

This paper presents *TurboNet*, a network emulator that leverages one programmable switch to faithfully mimic functionality, scale, and performance of production networks. In our future work, we will combine more programmable switches to emulate larger networks with more complex network functions. Furthermore, we will consider how to accommodate to dynamically changing network emulation requirements without interrupting the current emulation process.

Acknowledgments. We thank our shepherd, Marinho Barcellos, and the anonymous reviewers for their valuable comments. This research is supported by National Key R&D Program of China (2018YFB1800405) and the National Natural Science Foundation of China (61772307). Ying Liu is the corresponding author.

REFERENCES

[1] Teerawat Issariyakul and Ekram Hossain. Introduction to network simulator 2 (ns2). In *Introduction to network simulator NS2*, pages 1–18. Springer, 2009.

[2] ns-3. https://www.nsnam.org/.

[3] Jiasong Bai, Jun Bi, Peng Kuang, Chengze Fan, Yu Zhou, and Cheng Zhang. Ns4: Enabling programmable data plane simulation. In *Proceedings of the Symposium on SDN Research*, SOSR '18, New York, NY, USA, 2018. Association for Computing Machinery.

[4] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and ..., 2008.

[5] Xinjie Chang. Network simulations with opnet. In *WSC'99. 1999 Winter Simulation Conference Proceedings.'Simulation-A Bridge to the Future'(Cat. No. 99CH37038)*, volume 1, pages 307–314. IEEE, 1999.

[6] SCALABLE Network Technologies. Qualnet official site. Website. https://www.scalable-networks.com/products/qualnet-network-simulation-software-tool/.

[7] Mininet Team. Mininet: An instant virtual network on your laptop (or other pc). Website. http://mininet.org.

[8] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP. ACM, 2017.

[9] Emulab. http://emulab.net/.

[10] Pravein Govindan Kannan, Ahmad Soltani, Mun Choon Chan, and Ee-Chien Chang. {BNV}: Enabling scalable network experimentation through bare-metal network virtualization. In *11th {USENIX} Workshop on Cyber Security Experimentation and Test ({CSET} 18)*, 2018.

[11] Jelena Mirkovic and Terry Benzel. Teaching cybersecurity with deterlab. *IEEE Security & Privacy*, 10(1):73–76, 2012.

[12] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 253–264, 2012.

[13] Robert Ricci, Eric Eide, and CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[14] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[15] Mark Berman, Jeffrey S Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. Geni: A federated testbed for innovative network experiments. *Computer Networks*, 61:5–23, 2014.

[16] Marc Suñé, Leonardo Bergesio, Hagen Woesner, Tom Rothe, Andreas Köpsel, Didier Colle, Bart Puype, Dimitra Simeonidou, Reza Nejabati, Mayur Channegowda, et al. Design and implementation of the ofelia fp7 facility: The european openflow testbed. *Computer Networks*, 61:132–150, 2014.

[17] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[18] 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. In *Proceedings of SIGCOMM*, 2013.

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

[20] Cavium. Xpliant ethernet switch product family. Website. https://www.cavium.com/xpliant-ethernet-switch-product-family.html.

[21] Barefoot Networks. Tofino. Website, 2019. https://www.barefootnetworks.com/products/brief-tofino/.

[22] Barefoot Networks. Tofino2. Website, 2019. https://www.barefootnetworks.com/products/brief-tofino-2/.

[23] Floodlight sdn openflow controller. Website. https://github.com/floodlight/floodlight.

[24] Stephen Naicken, Anirban Basu, Barnaby Livingston, and Sethalat Rodhetbhai. A survey of peer-to-peer network simulators. In *Proceedings of The Seventh Annual Postgraduate Symposium, Liverpool, UK*, volume 2, 2006.

[25] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6, 2010.

[26] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[27] Benoit des Ligneris. Virtualization of linux based computers: the linux-vserver project. In *19th International Symposium on High Performance Computing Systems and Applications (HPCS'05)*, pages 340–346. IEEE, 2005.

[28] Stuart Mitchell, Michael OSullivan, and Iain Dunning. Pulp: a linear programming toolkit for python. *The University of Auckland, Auckland, New Zealand*, 2011.

[29] Raj Joshi, Ben Leong, and Mun Choon Chan. Timertasks: Towards time-driven execution in programmable dataplanes. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, pages 69–71, 2019.

[30] 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*, pages 30–43, 2019.

[31] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–14, 2012.

[32] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. Generic external memory for switch data planes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2018.

[33] Broadcom trident 3 - programmable, varied and volume. http://packetpushers.net/broadcom-trident3-programmable-varied-volume/.

[34] Intel flexpipe. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[35] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4):63–74, 2008.

[36] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, 2009.

[37] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[38] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*, pages 275–287, 2015.