HyperTester: High-performance Network Testing Driven by Programmable Switches

Yu Zhou[1,2,3], Zhaowei Xi[1,2,3], Dai Zhang[1,2,3], Yangyang Wang[1,3],
Jinqiu Wang[1], Mingwei Xu[1,2,3], Jianping Wu[1,2,3]
[1]Institute for Network Sciences and Cyberspace, Tsinghua University
[2]Department of Computer Science and Technology, Tsinghua University
[3]Beijing National Research Center for Information Science and Technology (BNRist)
{y-zhou16, xizw19, zhangd15}@mails.tsinghua.edu.cn, wangyy@cernet.edu.cn, jq-wang16@mails.tsinghua.edu.cn,
xumw@tsinghua.edu.cn, jianping@cernet.edu.cn

ABSTRACT

Modern network research and operations are inseparable from network testers to evaluate performance limits of proofs-of-concept, troubleshoot failures, *etc*. Existing network testers suffer from either constrained flexibility or a low performance-cost ratio. In this paper, we propose a new network tester, *HyperTester*. The core of *HyperTester* is to leverage new-generation programmable switches for generating and capturing test traffic with *high performance*, *low cost*, and *remarkable flexibility*. We design a series of efficient mechanisms, including template-based packet generation, false-positive-free counter-based queries, and stateless connections to realize various network testing tasks upon switches with limited programmability and resources. Meanwhile, to facilitate developing testing tasks upon *HyperTester*, we provide a high-level network testing API. We have implemented *HyperTester* on the Tofino switch and built dozens of network testing tasks. The evaluations on the hardware testbed show that *HyperTester* supports line-rate packet generation (400Gbps in the testbed) with highly-accurate rate control, while *HyperTester* can save $38,400 per Tbps and 7,150W per Tbps when comparing with the software network testers.

CCS CONCEPTS

• **Networks → Protocol testing and verification**; Network measurement.

KEYWORDS

Network testing, programmable switches, P4

ACM Reference Format:
Yu Zhou, Zhaowei Xi, Dai Zhang, Yangyang Wang, Jinqiu Wang, Mingwei Xu, Jianping Wu. 2019. HyperTester: High-performance Network Testing Driven by Programmable Switches. In *The 15th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '19), December 9–12, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3359989.3365406

CoNEXT '19, December 9–12, 2019, Orlando, FL, USA

ACM ISBN 978-1-4503-6998-5/19/12…$15.00
https://doi.org/10.1145/3359989.3365406

1 INTRODUCTION

Network testers proactively generate and capture test packets to evaluate network performance, which is essential for network researching and operating. On the one hand, network researchers can leverage network testers to perform Internet-wide scanning [1–3] and benchmark proofs-of-concept [4–6]. On the other hand, network operators can use network testers for measurement of latency or packet loss [7–9], failure troubleshooting [10–18], *etc*. Furthermore, the recent development of networks raises strigent demands on network testers from two perspectives. One is *high-performance packet generation* driven by ever-increasing bandwidth, the other is *flexible customization* to meet the constant appearance of new protocols and functions.

Existing network testers can be categorized into hardware and software approaches. First, *commodity network testers* [19, 20] based on proprietary hardware can achieve high throughput over 1Tbps per device. Nevertheless, commodity network testers have limited flexibility as it is hard to customize proprietary hardware to test new protocols or functions. Besides, commodity network testers are typically expensive, and a packet generation module with dual 10Gbps ports costs as much as $25,000 [21]. *Open-sourced hardware network testers* [21, 22] based on FPGA are cheap and reconfigurable but of poor performance. Second, *software network testers* [4–6, 23] support flexible customization of packet generation logic, but have to make a trade-off between performance and cost. An 8-core server can provide less than 100Gbps [5] throughput. If operators require higher throughput to test high-performance prototypes or large-scale networked systems, more servers are needed, introducing steeply-increased costs. Furthermore, software network testers suffer from low rate control accuracy, degrading their effectiveness [24]. In summary, existing network testers either have low performance-cost ratios or yield unsatisfactory flexibility.

In this paper, we present ***HyperTester***, leveraging the power of programmable switches [25–29] for high-performance network testing. Programmable switches are designed for reconfigurable packet processing while offering multi-Tbps bandwidth, which makes it ideal for executing various network testing tasks with high-performance requirements. On the one hand, *HyperTester* can efficiently process over 1Tbps test traffic with one programmable switch that is as cheap as a mid-ranged server [30]. On the other hand, *HyperTester* yields future-proof flexibility as operators can reconfigure programmable switches to test new protocols and functions. Furthermore, *HyperTester* also provides a suite of network

testing API (*NTAPI*) for operators to facilitate the specification of network testing intents. Until now, *HyperTester* supports a wide range of applications, including stress testing, Internet scanning, delay measurement, denial-of-service (DoS) attack emulation, *etc.*

However, designing *HyperTester* is non-trivial due to two challenges. First, there remains a gap between network testing logic and the programmability of switching ASIC. Switching ASIC is intended for packet forwarding and cannot generate packets without ground. Meanwhile, some network testing tasks require complex logic that is beyond the capability of switching ASIC. To address this challenge, we take advantage of switch CPU to extend switching ASIC's programmability. We propose a *template-based packet generation mechanism* leveraging switch CPU and switching ASIC simultaneously. In the mechanism, switch CPU generates a series of template packets and performs the actions (*e.g.* payload customization) that are hard for switching ASIC over template packets. Then, switching ASIC accelerates template packets by recirculation, and generates final test traffic via multicasting template packets at given rates. Template-based packet generation makes a good trade-off between performance and programmability via the co-design of switch CPU and switching ASIC.

Second, network testing might forge massive connections or conduct per-flow analysis, which requires intensive memory, but memory resources in switching ASIC are limited. To address this challenge, on the one hand, we introduce *stateless connections* to avoid storing connection states. The stateless connection generates response packets according to received packets, and we provide a general mechanism to support stateless connections entirely on data planes. On the other hand, we employ the counter-based algorithm [31] instead of sketch-based algorithms [32, 33] to perform accurate per-flow analysis. To guarantee per-flow analysis free of false positives and optimize memory usage, we introduce *exact key matching* and *cuckoo hashing* in switching ASIC. We also evict old analysis counters to switch CPU to utilize the large DRAM residing outside switching ASIC.

HyperTester is built upon the general capabilities of P4-programmable switches while being agnostic to underlying targets. The capabilities required by *HyperTester* include reconfigurable match-action tables, the *recirculate* primitive action, registers, time stamping, and multicasting. The former three capabilities are specific to P4 [34], while the others are widely supported by traditional switches [35, 36]. We make four major contributions in this paper:

- We present *HyperTester*, a high-performance, low-cost, and flexible network testing system that leverages the power of new-generation programmable switches. (§3)
- We provide a new NTAPI to help operators express their intents on network testing. (§4)
- We present the *HyperTester* design, which introduces template-based packet generation, stateless connections, and the false-positive-free counter-based algorithm to bridge the gap between challenging requirements of practical network testing and limited capabilities of switching ASIC. (§5)
- We have implemented a prototype of *HyperTester* and built dozens of network testing applications over *HyperTester*. The open-sourced code of *HyperTester* is published at [37]. We evaluate *HyperTester* on a testbed equipped with two Tofino switches. Evaluation results show that NTAPI can effectively facilitate developing network testing tasks and reduce the code size by over 74.4%. *HyperTester* can generate up to 400Gbps traffic with arbitrary packet sizes at line rate in our testbed. The rate control errors of *HyperTester* are over one order of magnitude smaller than MoonGen [5]. *HyperTester* can save $38,400 per Tbps and 7,150W per Tbps when compared with MoonGen running on commodity servers equipped with 8-core CPU. (§6 and §7)

2 BACKGROUND AND MOTIVATION

2.1 Background

P4 and programmable switches. P4 [38] is a domain-specific language for programming switching ASIC based on the reconfigurable match-action table (RMT) [25] model. RMT consists of two components, *i.e.* the parser and the pipeline. First, the parser enables decoding user-defined header formats. Second, the pipeline contains multiple sequential physical stages, by which match-action tables are implemented. With P4, programmers can specify match fields and compound actions for each table, and P4 supports control flow among tables to define the table execution sequence. Moreover, P4 also supports stateful components, *e.g.* registers, to store states consistently. A P4 program only defines a portion of the overall data plane function, while programmable switches also need to populate the tables with rules after installing the P4 program into switching ASIC. Apart from switching ASIC, programmable switches have switch CPU, which connects to switching ASIC by PCIe. Switch CPU is the control plane of switching ASIC and can execute various control programs, like the BGP agent and the statistic collector.

2.2 Related Works

In this section, we show related works on network testing.

Commodity network testers. Based on proprietary hardware, commodity network testers [19, 20] provide rich network testing functions, user-friendly GUI, and high packet generation/capture throughput, but are inevitably expensive. For example, a dual-10Gbps-port packet generation module costs as much as $25,000 [21], and the cost of a dual-100Gbps-port packet generation module could quickly rise to over $100,000 according to a network tester seller. Furthermore, commodity network testers enable crafting packets with user-defined formats but fall short of defining new communication patterns for new protocols as well as new packet capture and analysis algorithms. *HyperTester* yields much higher flexibility than commodity network testers.

Programmable hardware network testers. There have been many open-source network testers [21, 22] based on the programmable hardware, FPGA [39]. Leveraging the programmability of FPGA, programmable hardware network testers can be flexibly extended to test new functions and protocols, but such flexibility comes with heavy and notorious programming workload due to the non-trivial development complexity of FPGA [40, 41]. Furthermore, compared with *HyperTester*, FPGA-based network testers come with a much lower performance-cost ratio, *e.g.* a NetFPGA board equipped with four 10Gbps ports costs as much as $6,999 [42].

Software network testers. There have been many software network testers in the literature. At the early stage, software network

testers are based on raw sockets [43] or kernel functions [4]. Thus, traditional network testers come with quite limited performance and accuracy [5, 21, 24]. With the advance of new NIC I/O techniques, *e.g.* DPDK [44] and Netmap [45], the performance of software network testers [4, 5] increases by over one order of magnitude. MoonGen [5] can generate up to 80Gbps small-sized packets with eight cores, which is comparable to programmable hardware network testers [21]. Moreover, software network testers provide the highest flexibility among all types of network testers. However, many development efforts are required to optimize the network tester performance. Furthermore, as a universal computing platform, CPU has intrinsic restriction on the packet generation throughput. The networks testing tasks need more CPU cores to satisfy the requirement on higher throughput, introducing linearly-increased equipment and power cost inevitably.

2.3 Motivation of *HyperTester*

In this part, we present three significant applications of network testing as well as how *HyperTester* empowers these applications regarding performance, cost, and flexibility.

Testing network prototypes. Evaluating prototypes thoroughly is of great significance for researchers and engineers to understand design flaws, performance limits, and reliability of their prototypes. Meanwhile, the evaluation of network prototypes (*e.g.* new software router) requires high-performance packet generation. In the past, network prototypes were of poor performance (less than 1Gbps), and the traditional software network testers [1, 4] supply enough performance. Then, the emergence of fast I/O frameworks, *e.g.* DPDK [44], elevates the network prototype performance by over one order of magnitude (over 10Gbps). The new-generation network testers based on the new I/O frameworks are proposed to generate test packets with comparable performance. Nowadays, a lot of novel network prototypes [46–57] run upon switches which have much higher performance than software network testers and some hardware network testers. Thus, to guarantee evaluation soundness, researchers have to turn to commodity network testers that are too expensive to afford. We argue that *the development of existing low-cost network testers does not catch up with the performance increase of network prototypes.* Thus, we propose *HyperTester*, a low-cost and flexible network tester promising multi-Tbps throughput, which opens new opportunities for testing network prototypes.

Testing new protocols. The current advance of network programmability unleashes a trend of new protocols [38, 46–48, 58]. Inevitably, testing the networks using new protocols puts stringent requirements on the flexibility of network testers. Software network testers offer the required flexibility but fail in high-performance testing. High-performance commodity network testers enable users to customize the packets with user-defined formats. However, some protocols require generating response packets according to the received packets, just like SYN+ACK and ACK packets in the TCP initialization phase, but commodity network testers cannot customize responsive packet generation. Besides, supporting new protocols in commodity network testers need to upgrade the proprietary hardware, which brings long developing time and additional investment. Building upon protocol-independent programmable switches, *HyperTester* can be flexibly programmed to support new protocols and responsive packet generation. Furthermore, we provide NTAPI to facilitate developing network testing tasks, shortening the development lifecycle effectively.

Emulating DoS attacks. Denial of service (DoS) attacks are a disruptive presence for network services and cause a large amount of revenue penalty [59–61]. In recent years, there is a foreseeable trend that DoS attacks significantly increase in intensity, frequency, and complexity. To combat DoS attacks, a lot of defense and mitigation systems [62–64] are proposed. Testing the effectiveness of these defense systems is of great importance while requiring precious DoS attack emulation. *HyperTester* is an ideal DoS attack emulator for the following four reasons. First, *HyperTester* can generate up to 6.5Tbps attack traffic, which is much larger than the attack volume of the recorded attacks. Second, implemented upon a single programmable switch (occupying 1U for 3.2Tbps and 2U for 6.5Tbps), *HyperTester* comes with the remarkably low equipment cost, power cost, and deployment cost. Third, *HyperTester* can be flexibly programmed to emulate ever-changing DoS attacks that evolve rapidly in the attack mechanism [59]. Fourth, *HyperTester* comes with high port intensity (65 for 100Gbps ports and 260 for 25Gbps ports) and can inject traffic into multiple ingress points of the systems to emulate distributed DoS attacks.

3 *HYPERTESTER* OVERVIEW

HyperTester is a high-performance, low-cost, and flexible network tester driven by programmable switches. In this section, we present an overview of challenges, key ideas, the architecture, and the workflow of *HyperTester*.

3.1 Challenges and Key Ideas

HyperTester should overcome two challenges for high-performance packet generation and test statistic collection in P4 switches.

Flexible packet generation against limited programmability of switching ASIC. Network testing tasks can have different packet generation requirements, *e.g.* payload customization and random inter-departure time. On the one hand, switching ASIC does not support creating packets. On the other hand, to guarantee line-rate packet forwarding, switching ASIC comes with limited programmability. For example, the state-of-the-art switching ASIC (*e.g.* Tofino) fall short of payload customization and have little control over packet departure time. Thus, there remains a gap between requirements on *flexible packet generation and programmability of switching ASIC*, which challenges *HyperTester*.

Massive test states against limited resources of switching ASIC. For some network testing tasks (*e.g.* web testing and IP scanning), there could be massive connections, requiring a large number of connection states in network testers. Moreover, network testers need to analyze sent and received packets to get test statistics, such as per-port bandwidth and per-flow delay, which also needs to store analysis states. This paper refers connection states and statistic states to as *test states* uniformly. Maintaining per-flow or per-connection test states could require up to hundreds of MB or several GB memory, which is far beyond available memory (dozens of MB) in switching ASIC.

To address the challenges derived from limited programmability and resources of switching ASIC, we come up with two key ideas.

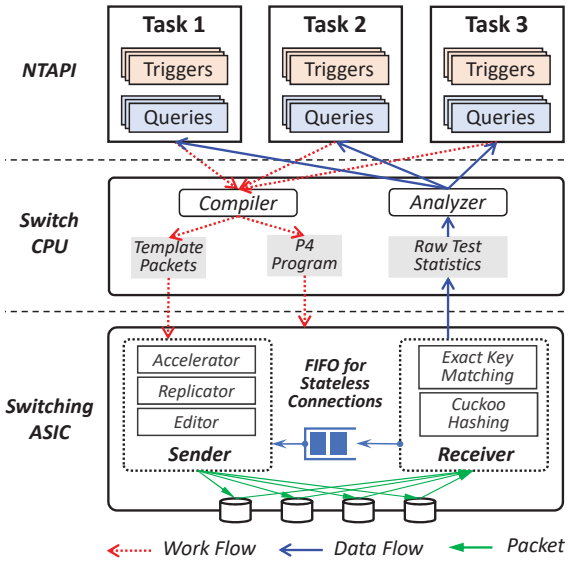


Figure 1: Architecture of HyperTester.

Co-designing switch CPU and switching ASIC. P4 switches have two programmable processors: the first one is low-performance switch CPU with high programmability, the other is high-performance switching ASIC with limited programmability. To overcome the programmability challenge, we introduce the switch CPU as the co-processor of switching ASIC, and co-design switch CPU and switching ASIC to implement packet generation logic. Thus, *HyperTester* focuses on the following question: *how to divide packet generation labor between switch CPU and switching ASIC.*

To answer this question, we propose a *template-based packet generation* mechanism. In the mechanism, switch CPU generates template packets and performs the operations, which are hard for switching ASIC, on template packets. The CPU operations include payload customization and header initialization. Then, switching ASIC amplifies template packets injected by switch CPU. Amplifying template packets involves speeding up template packets to a given rate and creating test packets based on template packets via replication. Template-based packet generation integrates benefits of switch CPU and switching ASIC, while making a good trade-off between programmability and performance.

State compression and stateless connection. We develop different mechanisms to address the challenges of storing analysis states and connection states. First, we use the *counter-based algorithm*, like HashPipe [31], to store analysis states and introduce exact key matching to remove false positives entirely. To further improve the memory efficiency, we perform *cuckoo hashing* in the counter-based algorithm. Moreover, we also evict the old analysis states and upload them to the switch CPU. Second, we adopt the *stateless connection* mechanism and do not store any connection state. In a stateless connection, *HyperTester* generates packets only according to received packets. For example, if *HyperTester* receives a TCP SYN packet, it will emit a TCP SYN+ACK packet. However, the stateless connection cannot support complex connection logic, such as TCP congestion control. Thus, *HyperTester* makes a trade-off between memory efficiency and support for complex connection logic.

3.2 Architecture and Workflow

Figure 1 presents an architectural overview of *HyperTester*. From top to down, *HyperTester* is completely implemented by a single programmable switch and is composed of three layers, *i.e.* NTAPI, switch CPU, and switching ASIC. First, network operators can define various tasks with NTAPI (§4). Second, the switch CPU compiles the tasks and generates template packets and the P4 program. Next, the sender (§5.1) generates test packets according to template packets. Moreover, the switch CPU analyzes the test statistics from switching ASIC. At last, switching ASIC will conduct acceleration and edition operations on template packets and generate final test traffic at the given speed in the sender. Meanwhile, switching ASIC captures packets from devices/networks under test and collects test statistics with the receiver (§5.2). The receiver could transfer data to the sender to trigger stateless connections (§5.3).

4 NETWORK TESTING API

4.1 Programming Model

Network Testing API (NTAPI) has a similar programming model with the stream processing frameworks, such as Flink [65]. These frameworks abstract data as a stream of records, and execute transformations over the records one by one. Correspondingly, NTAPI takes the packet stream as the underlying programming model, and the packet stream represents a stream of header fields. Moreover, NTAPI provides *packet stream trigger* and *packet stream query* to define the logic of packet generation and statistic collection, which are the two primary functions of network testers.

Packet stream trigger. The programming model for packet generation in NTAPI is trigger-based. Specifically, each element in the packet stream defines the header fields for the generated packets, and the packet stream trigger defines how to start generating packets in the stream. In NTAPI, there are two types of packet stream triggers. The first type of triggers generates packets as soon as *HyperTester* starts. The second one is a query-based trigger, which generates packets as soon as a particular packet arrives, and we employ the query-based trigger to support stateless connections.

Packet stream query. To get test statistics, NTAPI supports queries over the packet stream. *HyperTester* supports typical query primitives, such as reduce and distinct, which have been proved to support heterogeneous network monitoring tasks [55]. In NTAPI, the packet stream query can monitor the traffic in two directions. First, the packet stream query can monitor the received traffic coming from a specific port. Second, the packet stream query can also monitor sent traffic generated by a trigger.

Category	Field	Syntax
Header	Header Field	*hdr_name.field_name*
	Payload	*payload*
Control	Packet Length	*pkt_len*
	Injection Interval	*interval*
	Injection Port	*port*
	Injection Loop	*loop*

Table 1: Fields of Network Testing API.

f	$\in$ Testing fields	Refer to Table 1
q	$\in$ Query primitives	Refer to Sonata [55]
n	$\in \mathbb{N}$	Constant value
$\mathbb{T}$	::= *trigger* ($[\mathbb{Q}]$)	Trigger primitive
$\mathbb{S}$	::= *set* ($\mathbb{L}_f \mid f, \mathbb{L}_v \mid \mathbb{V}$)	Field set operation
$\mathbb{L}_f$	::= $(f\{, f\})$	Field list
$\mathbb{L}_v$	::= $(\mathbb{V}\{, \mathbb{V}\})$	Value list
$\mathbb{V}$	::= $n \mid \mathbb{V}_r \mid \mathbb{V}_a$	Field value
$\mathbb{V}_r$	::= *random*(ALG, P, n)	Random value
$\mathbb{V}_a$	::= *range*(n, n, n)	Range array
$\mathbb{Q}$	::= *query*($[\mathbb{T}]$)	Query primitive
trigger	::= $\mathbb{T}\{.\mathbb{S}\}$	Packet stream trigger
query	::= $\mathbb{Q}\{.q\}$	Packet stream query

Table 2: Syntax of Network Testing API.

4.2 NTAPI

Based on the above programming model, we propose an easy-to-use API to define network testing tasks. Next, we will introduce the fields that can be used by NTAPI as well as the syntax. §5 introduces the compilation of NTAPI.

NTAPI fields. As shown in Table 1, there are two types of fields that can be referred by NTAPI. The first one is related to *header fields* and contains fields from parsed headers. NTAPI can use any field from all the parsed headers. Furthermore, *HyperTester* enables modifying the packet payload with a specified constant value. The second one is related to packet generation control and includes packet length, inter-departure interval, the port, and the number of loops. *HyperTester* uses the inter-departure interval to specify the packet generation rate. Furthermore, *HyperTester* re-generates a packet stream for multiple times according to the *loop* value.

NTAPI syntax. Table 2 shows the basic syntax of NTAPI. There are two types of statements in NTAPI. First, the packet stream query starts with the *query* primitive and can monitor both the sent traffic and the received traffic. NTAPI supports all the query primitives listed in Sonata [55]. Second, the packet stream trigger starts with *trigger* followed by a series of *set* primitives. The *set* primitive enables users to define the value of NTAPI fields. There are four value types, including the constant, the array, the range array, and the random array. The constant can set the field to the same value for all packets, while the other three can define the field value for a list of packets. The array contains a pre-defined field list. The range array is an arithmetic progression. The last one is to generate the value list according to some random distributions, *e.g.* normal distribution.

NTAPI by example. To show how to develop a network testing task with NTAPI, we present an example in Table 3 for testing the throughput of a network device. In the example, there are one trigger and two queries. First of all, T_1^t generates 64-byte UDP packets with specific IP addresses and ports. Then, Q_1^t and Q_2^t monitor the traffic generated by T_1^t and received packets respectively, and the queries report the throughput in bytes per second.

5 DESIGN OF *HYPERTESTER*

This section demonstrates *HyperTester*'s solutions to the following questions. The first one is *how to generate required traffic for various testing tasks based on the capabilities of P4-programmable switches* (§5.1). The second one is *how to collect accurate test statistics with limited memory resources of switching ASIC* (§5.2). The last one is *how to support stateless connections* (§5.3).

$T_1^t =$	$trigger()$
	$.set([dip, sip, proto, dport, sport], [X, Y, udp, 1, 1])$
	$.set([loop, length], [0, 64])$
$Q_1^t =$	$query(T_1^t).map(p \rightarrow (pkt\_len)).reduce(func = sum)$
$Q_2^t =$	$query().map(p \rightarrow (pkt\_len)).reduce(func = sum)$

Table 3: Throughput testing with *HyperTester*.

5.1 HyperTester Packet Sender

In this part, we present the design of *HyperTester* Packet Sender (HTPS), which generates test traffic with the capabilities of programmable switches. In brief, we use four capabilities specific to P4, *i.e.* match-action tables, registers, data plane time-stamps, and *recirculate* primitive action. Moreover, we use one general switch capability, *i.e.* multicast (mcast) engine in the queuing system, which can replicate packets to specific ports.

Figure 2 and Figure 3 show the overall design of HTPS and the component layout in RMT respectively. Logically, HTPS mimics cyclotron and keeps accelerated packets looping in cycles. The whole journey of packets in HTPS includes three sequential steps. First, HTPS accelerates template packets to 100Gbps with the *accelerator*. Second, HTPS generates replicas for template packets with the *replicator*. Last, HTPS modifies the headers of replicated packets and generates test traffic with the *editor*. In the remaining part, we will present the design of the three components.

Accelerator. Residing at the ingress pipeline, the accelerator recirculates each non-replicated template packets. Template packets travel the recirculation loop constantly, making the accelerator a stable packet source for the replicator. Our experiments testify that Tofino [28] could recirculate packets at a speed of no less than 100Gbps. Furthermore, the accelerator can recirculate multiple template packets simultaneously, and the maximum number of recirculated packets is demonstrated in §7.3. Figure 3 is not the distinctive layout for the accelerator, and the accelerator can also reside at the egress pipeline, which is further discussed in §6.

Replicator. Taking template packets in the accelerator as input, the replicator performs packet replication at given rates. As shown in Figure 3, the replicator consists of a rate control timer and a mcast engine. First, the replicator conducts rate control via adjusting the inter-departure time of test packets. Switching ASIC provides a nanosecond level timestamp for each incoming packet, and the periodic timer records the departure time of the last replicated packet. Once the difference between the current time and the recorded departure time exceeds a given threshold, the replicator will direct

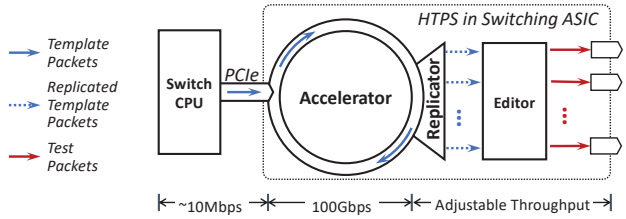


Figure 2: Design of HyperTester Packet Sender.

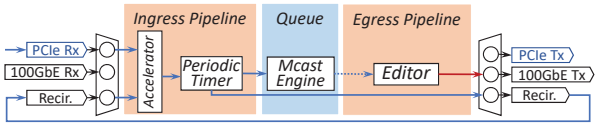


Figure 3: HTPS component layout in RMT.

the template packets to the mcast engine. Meanwhile, the timer will record the new departure time. Through changing the timer threshold, the replicator manages to control the throughput of replicated template packets with high accuracy. The rate control precision depends on the minimal arrival time of template packets and is around 6.4ns on Tofino for 64-byte packets.

Second, HTPS employs *multicast*, a general primitive widely supported by commodity switches, to create template packet replicas. Going through the mcast engine, the normal template packets will continue to be recirculated. In this manner, the replicator can constantly acquire packets from the accelerator. The mcast engine can replicate packets to multiple ports in parallel, and the replicated template packets will go to the editor for further processing.

Editor. Residing at the egress pipeline, the editor modifies header fields according to NTAPI and generates test traffic finally. The editor can modify any parsed header fields and supports four types of header field modification. The first one is to set a constant value that will be set in template packets by switch CPU. The second one is to set the header field with a given value list. The editor maintains a packet ID for each template packet with registers, and the packet ID adds 1 when the template packet is replicated. Then, the editor provides a table matching the packet ID and setting the field with the value indexed by the packet ID. For example, let us assume a task that sets the TCP source port to 80, 81, 82 sequentially. If the packet ID is 2, the editor will set the TCP source port to 81 and increment the packet ID to 3.

The third one is to set the field to an arithmetic progression. The editor records the value with registers and performs adding or subtracting over the value every time. The last one is to generate random values according to a certain distribution, such as normal distribution. P4 only supports a uniform random generator, *i.e. modify_field_rng_uniform*. We propose to implement *inverse transformation method* [66] with two tables. Through the inverse transformation method, HTPS can generate values based on arbitrary distributions as long as the cumulative distribution function is provided. Furthermore, the evaluation in §7 shows that HTPS can emulate different distributions with great accuracy.

Compiling packet stream triggers to HTPS. In *HyperTester*, a task could have multiple packet stream triggers, each of which corresponds to a template packet. To compile a packet stream trigger, we should generate template packets, mcast engine configurations, rate control code of the replicator, and packet modification code of the editor. First, as for template packet generation, we initialize the packet length, packet payload, and packet headers as specified by *payload*, *pkt_len*, and the initial values of header fields. Second, we configure the mcast group for the corresponding template packets and mcast destination port according to *port*. Third, we generate the rate control code according to *interval*, which defines the timeout threshold of the replicator timer. Fourth, we generate the header modification tables for each *set* primitive, and then arrange the generated tables to process test packets sequentially.

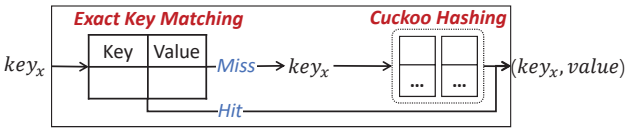


Figure 4: Accurate distinct and reduce query.

5.2 HyperTester Packet Receiver

HyperTester Packet Receiver (HTPR) has two primary functions, *i.e.* querying packet statistics and extracting trigger records for HTPS from received packets. In this section, we focus on the first function while §5.3 shows the other one. As for querying packet statistics, *HyperTester* adopts the same query primitives and the similar compilation method of Sonata [55]. However, Sonata implements *distinct* with Bloom Filter [67] and *reduce* with Count-Min Sketch [68], which compromises accuracy inevitably.

To improve querying accuracy, *HyperTester* redesigns *reduce* and *distinct* with the counter-based algorithm [31, 69] instead of the sketch-based algorithm used by Sonata [55]. At the high level, *HyperTester* stores the packet header key (*e.g.* TCP five tuples) and the corresponding counter in registers for each flow. If a packet matches the header key, some action (*e.g.* addition) will be executed over the counter. To save memory space, *HyperTester* stores the partial packet header key that is a digest generated via hashing the complete header key. The counter-based algorithm is more accurate than the sketch-based algorithm in terms of monitoring per-flow statistics at a cost of low memory efficiency. However, existing counter-based algorithms cannot guarantee complete accuracy due to false positives [49]. To guarantee complete accuracy and improve memory efficiency, *HyperTester* proposes *exact key matching* and *cuckoo hashing*, as shown in Figure 4.

False positive avoidance with exact key matching. To guarantee complete accuracy, we propose to eliminate false positives based on the following observation. The global header space of packets under querying in *HyperTester* is predictable, and the test packet headers are enumerable. As sent packets are proactively generated by *HyperTester*, we can enumerate the packet headers that will appear in *HyperTester* before the testing tasks start. This observation cannot apply to Sonata as it monitors normal traffic whose header space is non-predictable. Therefore, we can calculate all the false positives for each testing task. Then, we can use a table to execute exact matching over original header fields and resolve false positives. If a packet hits the table, the header key of the packet collides with another packet, *i.e.* leading to a false positive. *HyperTester* will directly execute *reduce* or *distinct* primitives accurately. Otherwise, packets go to cuckoo hashing for further processing. With the exact key matching, *HyperTester* can completely remove false positives. Furthermore, the false positive rate of the counter-based algorithm is always small, implying that *HyperTester* only needs a small-sized exact key matching table.

Memory efficiency improvement with cuckoo hashing. Current counter-based algorithms [31, 69] on data planes perform simple hashing and evict collided keys to the control plane for further processing. Hashing inevitably comes with limited memory utilization, which is not acceptable for resource-constrained switching ASIC. To improve memory efficiency, we propose to conduct cuckoo

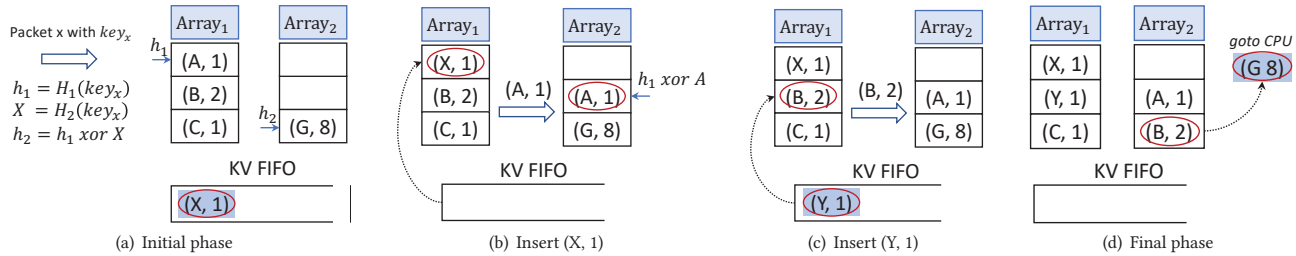


Figure 5: An example of cuckoo hashing in RMT.

hashing [70]. However, due to limited programmability, implementing cuckoo hashing is non-trivial for switching ASIC. To this end, we design a key-value (KV) FIFO and use recirculated template packets for popping key-value pairs from the FIFO. An example of cuckoo hashing is shown in Figure 5. At the initial phase shown in Figure 5(a), the packet x has key_x and calculates the indexes for two positions, *i.e.* h_1 and h_2. If any position is empty, the key of x will be inserted. Otherwise, the KV pair $(X, 1)$ is pushed into the FIFO. Then, as shown in Figure 5(b), the recirculated packet pops $(X, 1)$ and inserts its key into $Array_1$. $(A, 1)$ will be evicted and be insert into $Array_2$. Figure 5(c) and Figure 5(d) show another round of inserting new flows. When the FIFO overflows or an old KV pair (*i.e.* $(G, 8)$) is evicted from $Array_2$, HTPR directly reports the KV pairs to the switch CPU via *generate_digest*

Component layout and data collection. *HyperTester* could deploy the packet stream query at either the ingress pipeline or the egress pipeline according to which traffic to monitor. If a query monitors the sent traffic, it should be deployed at the egress pipeline. If a query monitors the received traffic, it should reside at the ingress pipeline. There are two methods for test statistic collection. Firstly, *the push mode* is to report data from the data plane to the control plane via *generate_digest*. Secondly, *the pull mode* is to collect data plane counters by switch CPU via the control plane API.

Compiling packet stream queries to HTPR. To compile packet stream queries, *HyperTester* should generate P4 code for queries and table entries for exact key matching. Query compilation is similar to Sonata but differs in three aspects. First, as *HyperTester* has different designs of *reduce* and *distinct*, *HyperTester* needs to extract the global header space for the query, calculate false positives, and to install them into the exact matching tables. The header space extraction is achieved via analyzing the *set* operations. For example, the header space of *set(tcp.dp, [80, 81, 82])* contains 80, 81, and 82. If the hash values of packets with *tcp.dp=80* or *tcp.dp=81* collide, *HyperTester* puts either *tcp.dp=80* or *tcp.dp=81* in the exact key matching table. Second, queries might be partitioned between switching ASIC and CPU. Different from Sonata using Spark computing clusters to execute CPU querying logic, *HyperTester* runs all the CPU logic within switch CPU because *HyperTester* has a much smaller workload than Sonata. In this manner, we can implement *HyperTester* within one programmable switch.

5.3 Stateless Connection

To avoid storing connection states, *HyperTester* adopts the stateless connection, which is also used by Zmap [1]. To be concise, *HyperTester* generates packets according to received packets. A naive approach to support stateless connection is to directly re-write headers of the received packets (*i.e.* swap source IP and destination IP) and forward them. However, the triggered packets might have different lengths with the received ones. Due to the limited packet header vector size, *HyperTester* falls short of changing the packet length, which disables the naive approach. To address this issue, HTPS is designed to interact with HTPR and generates packets according to triggers extracted by HTPR. We design a *trigger FIFO* to transfer trigger records from HTPR to HTPS, as shown in Figure 6. As the template packet length is easy to control, the trigger FIFO between HTPS and HTPR efficiently supports stateless connections.

Figure 6: Trigger FIFO between HTPS and HTPR.

5.4 *HyperTester* by Example

In this part, we utilize an example to let readers go through the overall workflow of *HyperTester*. The task called *web testing* is to emulate a certain number of clients who request a web page from HTTP servers, and Table 4 presents the NTAPI code for web testing. Next, we show how *HyperTester* executes web testing step by step.

- Operators develop the triggers and queries for web testing with NTAPI. There are six triggers that emit or acknowledge packets. The queries can be categorized into two types. One (*e.g.* Q_1^w) is for capturing specific packets for stateless connections, while the other one (*e.g.* Q_5^w) is for collecting test statistics.
- Taking triggers and queries as input, *HyperTester* generates the P4 program, table entries, and template packets. Then, *HyperTester* configures the switch with the generated materials.
- Suppose that the task creates 100K new clients per second. Then, *HyperTester* configures T_1^w to generate SYN packets whose inter-departure interval is 10us, and the sequence number is 1.
- Q_1^w monitors SYN+ACK packets and transfers the record to T_2^w, and T_2^w generates ACK packets with IP addresses, TCP ports, sequence numbers and so forth from trigger records. For example, if the destination port of an SYN+ACK packet is 4096, T_2^w generates an ACK packet whose source port is 4096.
- T_3^w is also triggered by Q_2^w and generates PSH+ACK packets whose payload carries an HTTP request. The second query monitors ACK packets with data from servers and triggers T_4^w to acknowledge the data packets.
- For connection release, on the one hand, Q_3^w monitors ACK packets whose acknowledge number is beyond a threshold and triggers T_5^w to generate FIN packets. We assume that the web page

T_1^w = $trigger().set([dip, dport, proto, flag, seq\_no], [X, 80, tcp, SYN, 1]).set(sip, range(1.1.0.1, 1.1.1.0, 1))$
$.set(sport, range(1024, 65535, 1)).set(interval, 10us)$

Q_1^w = $query().filter(tcp\_flag == SYN + ACK)$

T_2^w = $trigger(Q_1^w).set([dip, sip, dport, sport, tcp\_flag, seq\_no, ack\_no],$
$[Q_1^w.sip, Q_1^w.dip, Q_1^w.sport, Q_1^w.dport, ACK, Q_1^w.ack\_no, Q_1^w.seq\_no + 1])$

T_3^w = $trigger(Q_1^w).set([dip, sip, dport, sport, tcp\_flag, seq\_no, ack\_no],$
$[Q_1^w.sip, Q_1^w.dip, Q_1^w.sport, Q_1^w.dport, PSH + ACK, Q_1^w.ack\_no, Q_1^w.seq\_no + 1]).set(payload, "GET\ index.html")$

Q_2^w = $query().filter(tcp\_flag == ACK).reduce(func = sum).filter(count < 5)$

T_4^w = $trigger(Q_2^w).set([dip, sip, dp, sp, tcp\_flag, seq\_no, ack\_no],$
$[Q_2^w.sip, Q_2^w.dip, Q_2^w.sp, Q_2^w.dp, ACK, Q_2^w.ack\_no, Q_2^w.seq\_no + 1])$

Q_3^w = $query().filter(tcp\_flag == ACK).reduce(func = sum).filter(count >= 5)$

T_5^w = $trigger(Q_3^w).set([dip, sip, dport, sport, tcp\_flag, seq\_no, ack\_no],$
$[Q_3^w.sip, Q_3^w.dip, Q_3^w.sport, Q_3^w.dport, FIN, Q_3^w.ack\_no, Q_3^w.seq\_no + 1])$

Q_4^w = $query().filter(tcp\_flag == FIN)$

T_6^w = $trigger(Q_4^w).set([dip, sip, dport, sport, tcp\_flag, seq\_no, ack\_no],$
$[Q_4^w.sip, Q_4^w.dip, Q_4^w.sport, Q_4^w.dport, FIN + ACK, Q_4^w.ack\_no, Q_4^w.seq\_no + 1])$

Q_5^w = $query().filter(tcp\_flag == SYN + ACK).reduce(func = sum)$

Table 4: The web testing application running upon *HyperTester*. *sip*, *dip*, *sport*, and *dport* respectively stand for the source IP address, the destination IP address, the source TCP port, and the destination TCP port.

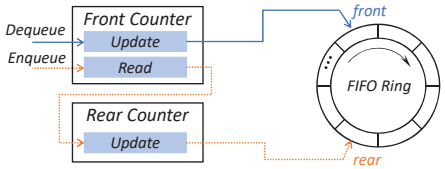


Figure 7: FIFO implementation. The blue solid line and the dashed orange line respectively denote the two FIFO operations, *i.e.* enqueue and dequeue.

can be loaded in 5 packets (count data packets only). On the other hand, Q_4^w monitors FIN packets from servers and triggers T_6^w to generate acknowledgment packets.

- *HyperTester* could also install some queries (*e.g.* Q_5^w) to monitor the HTTP server performance metrics, such as answered connections and delay of HTTP responses. These performance monitoring queries are agnostic to the above queries for stateless connections and upload data to the switch CPU.

6 IMPLEMENTATION AND LIMITATION

6.1 Implementation of *HyperTester*

First of all, we show the implementation of FIFO in *HyperTester*.

FIFO Implementation. In *HyperTester*, we use FIFO for cuckoo hashing as well as for the interaction between HTPR and HTPS. FIFO is important for *HyperTester*, but implementing a FIFO with acceptable resource usage is non-trivial. To address this issue, we present a FIFO implementation shown in Figure 7. Each FIFO is composed of two parts. The first part is the 32-bit counters for queue front and queue rear. For the two counters, there are two operations, *i.e. read* and *update*. *read* returns the counter value without changing the counter, while *update* increments the counter by 1 and returns the updated value. Note that *update* of the rear counter depends on the value of the front counter to prevent queue underflows. Currently, our FIFO implementation in *HyperTester* has a limitation that it cannot guarantee freedom of queue overflows. In our future work, we will optimize the dequeue speeds (*i.e.* the recirculation speed of template packets) to prevent queue overflows.

Second, we discuss some feasibility considerations of implementing *HyperTester* on realistic P4 targets.

Limited implementation of standard P4 primitive actions. Throughout the design of *HyperTester*, we have used several standard primitive actions provided by P4. Although these primitive actions are mandatory, P4 targets may implement them with compromises. We mainly encounter two types of compromises. The first one is the *parameter limitation*. The parameters of some primitive actions are more limited than what the P4 specification defines. For example, *HyperTester* assumes that *lower_bound* and *upper_bound* in *modify_field_rng_uniform* are arbitrary integers, but in some targets, *lower_bound* must be 0, and *upper_bound* must be power of two to simplify hardware implementation. To address this issue, *HyperTester* limits the scope of generated values to the power of two and further increments the generated value with a specific offset. The second one is the *control limitation*, *i.e.* some primitive actions can only be invoked in the ingress pipeline or the egress pipeline. For example, *recirculate* is only supported by the egress pipeline in the P4 specification but can only be invoked by the ingress pipeline in some P4 targets. Fortunately, the accelerator, the only one component of *HyperTester* relying on *recirculate*, can be placed either at the ingress pipeline or the egress pipeline.

Limited recirculation capacity. There could be multiple template packets for one testing task. Thus, *HyperTester* requires recirculating multiple packets simultaneously. We refer the maximum number of recirculated packets as the recirculation capacity. Inevitably, the recirculation capacity of switching ASIC is finite, which limits the number of template packets in one testing task. To address this issue, we could implement the recirculation function via configuring a port to the loopback mode. Then, we recirculate packets by just forwarding the packets to the loopback port. If there

are more template packets, we can configure more ports to the loopback mode and amortize template packets among the ports. In this manner, *HyperTester* trades the available bandwidth and ports for the feasibility of complex testing tasks.

Errors in network testing tasks defined by NTAPI. Users might make some errors or conflicts when using NTAPI to specify their testing intents. For example, users might specify the TCP port with a value that is larger than 65536. During the NTAPI compilation, *HyperTester* will reject the mistaken testing tasks. Moreover, the test task might be too complex to be implemented in the switching ASIC, *e.g.* requiring too many physical stages. *HyperTester* will reject the testing tasks that cannot be accommodated by switching ASIC. Our future work will focus on optimizing testing task compilation to accommodate as many testing tasks with switching ASIC as possible.

6.2 Limitation of *HyperTester*

Template-based packet generation and stateless connections provide great generality for *HyperTester* to support various network testing tasks, and they enable *HyperTester* to reconcile programmability, resources, and high performance for network testing. However, they come with some limitations, which make it hard for *HyperTester* to support some complex network testing tasks.

Limitation of template-based packet generation. Template-based packet generation requires that (1) the network testing task can provide a small number of template packets, and (2) the task only needs to conduct the actions supported by switching ASIC over template packets. Thus, template-based packet generation cannot support the tasks that cannot extract template packets or need complex actions. For example, packet trace replaying generates packets according to the captured traffic, and it is hardly possible to perform template packet extraction over real-world traffic. Furthermore, some testing tasks, *e.g.* IPSec testing, might require decryption or encryption over the payload, which is too complex for switching ASIC.

Limitation of stateless connections. If connection state transitions in a network testing task can be explicitly triggered by packets, such as SYN and SYN+ACK for TCP handshaking, *HyperTester* can use stateless connections to avoid storing connection states. However, *HyperTester* cannot support the tasks whose connection state transitions are not triggered by packets. An example is to emulate duplicated ACK behaviors when some packets are lost in TCP connections. *HyperTester* cannot infer the information of lost packets from arrived packets. For such tasks, *HyperTester* has to store states for each connection on resource-constrained data planes, which cannot scale.

7 EVALUATION

Testbed setup. We deploy *HyperTester* in the testbed shown in Figure 8. The testbed is composed of two Tofino switches with 32 100Gbps ports and Intel Pentium 4-core 1.60GHz CPU as well as two commodity servers equipped with Intel Xeon 12-core 2.4GHz CPU and 64GB DRAM. NICs on both servers are DPDK-compatible. The testbed has three types of cables, *i.e.* 100Gbps, 40Gbps, and 10Gbps. Moreover, we use the widely-used DPDK-based packet generator, MoonGen [5] as the evaluation baseline of *HyperTester*. We acknowledge that *HyperTester* is a hardware-based solution, while MoonGen is a software-based one. Thus this comparison is unfair. Ideally, we would also compare HyperTester with hardware-based packet generators. However, these are commercial solutions for network testing, and we did not have access to them.

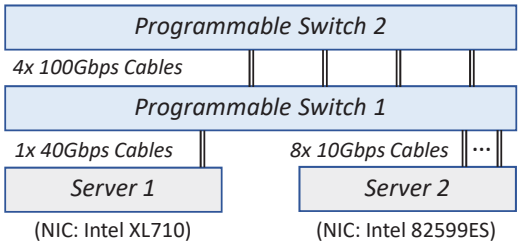


Figure 8: Evaluation testbed with two Tofino switches and two commodity servers.

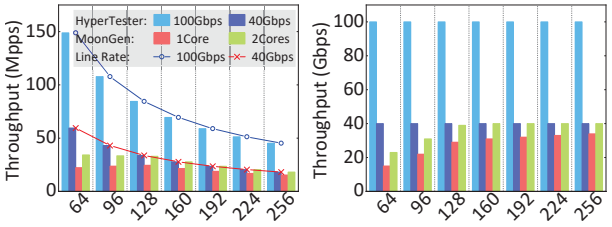
Evaluation objectives. We evaluate *HyperTester* with five objectives. First, we demonstrate the expressibility of NTAPI by comparing NTAPI with Lua used by MoonGen (§7.1). Second, we evaluate the overall performance of *HyperTester* in terms of throughput and rate control accuracy (§7.2). Third, we perform micro-benchmarking on *HyperTester* components, including the accelerator, the replicator, and test statistic collection (§7.3). Fourth, we conduct the quantitative analysis on the resource usage and the cost of *HyperTester* (§7.4). Finally, we study two cases, including delay measurement and SYN flood attack emulation (§7.5).

Result overview. We summarize a portion of results below.

- **Expressibility**: NTAPI can effectively facilitate developing network testing tasks and reduce the code size by over 74.4% when comparing with Lua used by MoonGen.
- **Accelerator**: The round trip time of template packets is less than 590ns in the accelerator, and a testing task can accelerate 89 64-byte template packets with the accelerator simultaneously.
- **Replicator**: The mcast engine brings around 400ns delay for replicated packets and introduces about 4ns jitters on inter-departure time of replicated packets.
- **Throughput**: *HyperTester* can generate 400Gbps traffic with arbitrary packet sizes at line rate in our testbed.
- **Rate control accuracy**: *HyperTester* provides much better rate control accuracy than MoonGen, and the errors of *HyperTester* are over one order of magnitude smaller than those of MoonGen.
- **Equipment and power cost**: *HyperTester* can save as much as $38,400 per Tbps and 7,150W per Tbps when comparing with MoonGen running on servers equipped with 8-core CPU.
- **Resource usage**: The packet stream trigger consumes a small amount of data plane resources, while the resource usage of querying is moderate when compared with switch.p4 [71].
- **SYN flood attack emulation**: *HyperTester* implemented on a 6.5Tbps programmable switch can emulate around 5.2×10^6 SYN flood attack agents.

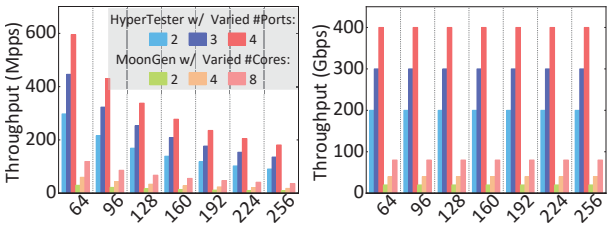
7.1 Expressibility of NTAPI

In this part, we present how NTAPI simplifies developing network testing tasks. We testify the expressibility of NTAPI via comparing the lines of NTAPI code (LoC) with the generated P4 code and the



(a) Mpps vs. Packet sizes (b) Gbps vs. Packet sizes

Figure 9: Single-port throughput comparison.



(a) Mpps vs. Packet sizes (b) Gbps vs. Packet sizes

Figure 10: Multi-port throughput comparison.

MoonGen Lua code when implementing the same applications. To guarantee fairness, the counted lines of generated P4 code only include control flow, tables, and actions. We use four applications whose code is at [37]. As shown in Table 5, NTAPI only needs around ten LoC to specify various testing tasks, while MoonGen requires 3x to 6x more LoC, indicating that NTAPI comes with strong expressibility. When comparing with P4, the LoC of NTAPI is over one order of magnitude lower, revealing that NTAPI can simplify developing network testing tasks significantly.

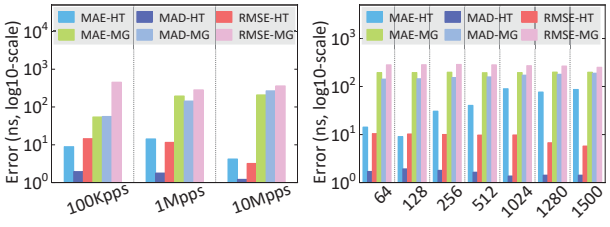
7.2 Marco-benchmark of HyperTester

In this part, we evaluate the overall performance of *HyperTester* in terms of throughput, the accuracy of rate control, and random number generation.

Throughput. We compare the throughput of *HyperTester* (HT) and MoonGen (MG) when changing generated packet sizes and port numbers. First, we use *HyperTester* to generate small-sized packets for a single port whose speed can be 100Gbps and 40Gbps in our testbed and use MoonGen for a single 40Gbps port. As shown in Figure 9(a) and Figure 9(b), *HyperTester* can generate packets at line rate, while MoonGen cannot generate small-sized packets with one core at line rate. Second, we add available ports for *HyperTester* and available CPU cores for MoonGen (using eight 10Gbps ports) to show the multi-port throughput. As shown in Figure 10(a) and Figure 10(b), *HyperTester* keeps line-rate packet generation in our testbed, while MoonGen needs one core for 10Gbps and can generate up to 80Gbps with 8 CPU cores.

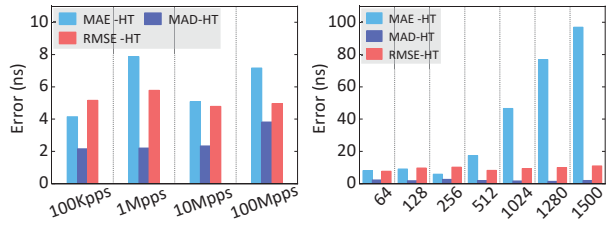
Applications	NTAPI	P4	MoonGen Lua
Throughput Testing	9	172	43
Delay Testing	10	134	71
IP Scanning	7	133	48
SYN Flood Attack	5	94	63

Table 5: Lines of code for different applications.



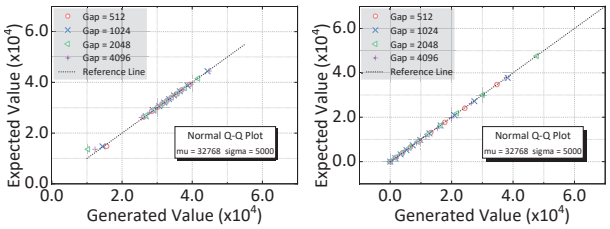
(a) Expected rates vs. Error (b) Packet sizes vs. Error

Figure 11: Rate control accuracy for 40Gbps.



(a) Expected rates vs. Error (b) Packet sizes vs. Error

Figure 12: Rate control accuracy of *HyperTester* for 100Gbps.

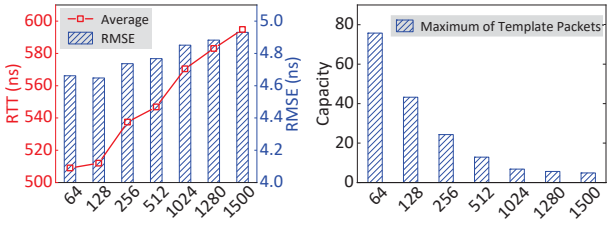


(a) Normal distribution (b) Exponential distribution

Figure 13: Accuracy of random number generation.

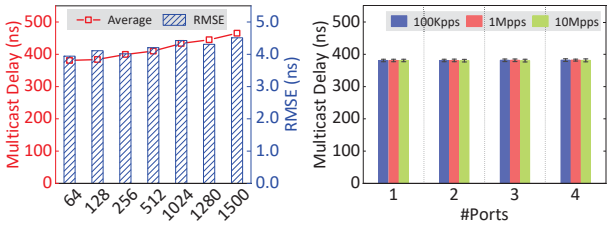
Rate control. We compare the rate control accuracy of *HyperTester* and MoonGen when changing packet generation speeds (from 100Kpps to 100Mpps) and packet sizes. In the experiments, MoonGen is configured to use the hardware rate control function of NIC. We conduct the experiments on 40Gbps and 100Gbps ports. To quantify rate control accuracy, we use three error metrics, including mean absolute error (MAE), mean absolute difference (MAD), and root mean squared error (RMSE), and the metrics are calculated based on inter-departure time. Figure 11 compares *HyperTester* with MoonGen for 40Gbps. As shown in Figure 11(a) and Figure 11(b), all the errors of *HyperTester* are over one order of magnitude lower than MoonGen, indicating that *HyperTester* has better rate control accuracy. Figure 12 shows rate control errors of *HyperTester* for 100Gbps. As shown in Figure 12(a) and Figure 12(b), the packet generation speed does not bring an obvious influence on rate control accuracy, but the errors grow with the size of generated packets.

Random number generation. *HyperTester* employs the inverse transformation method to realize random number generation entirely on data planes. We show the accuracy of random number generation through the Q-Q plot, and we use *HyperTester* to generate two well-known distributions, *i.e.* normal distribution and exponential distribution. Based on Figure 13(a) and Figure 13(b), we can reasonably claim that *HyperTester* can generate random numbers according to specified distributions with very strong similarity and generality.



(a) Accelerator RTT (b) Accelerator capacity

Figure 14: Evaluation of the accelerator.



(a) Impact of packet size (b) Impact of port number and speed

Figure 15: Evaluation of the replicator.

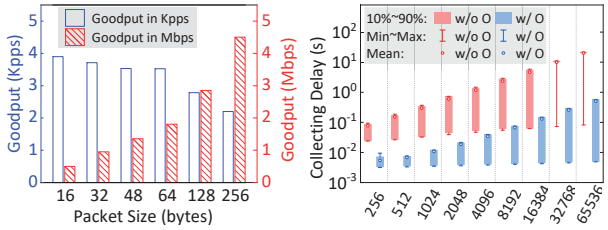
7.3 Micro-benchmark of HyperTester

In this part, we conduct micro-benchmarking experiments on *HyperTester* components, including the accelerator, the replicator, data collection, and exact key matching.

Accelerator. We present how fast the accelerator speeds up template packets via round trip time (RTT) and how many template packets a testing task can have (accelerator capacity). In the experiments, we change the size of template packets from 64 bytes to 1500 bytes. Figure 14 shows the experiment results of RTT and capacity. As for RTT, we present the average RTT as well as RMSE when letting a template packet recirculate for 10^6 times. As shown in Figure 14(a), a 64-byte packet can complete a loop within 570ns while the RMSE is lower than 5ns. Next, we get accelerator capacity in the switching ASIC via dividing the average RTT by minimal arrival interval between template packets. The recirculation bandwidth is no less than 100Gbps, and the minimal arrival interval for the 64-byte packet is 6.4ns. As shown in Figure 14(b), the accelerator can hold 89 64-byte template packets. We can use loopback ports to linearly extend the accelerator capacity at a price of overall bandwidth.

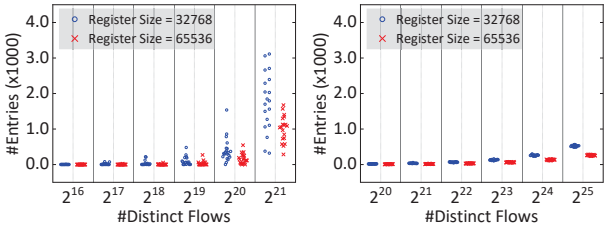
Replicator. As for the evaluation of the replicator, we measure the mcast engine delay, which might influence the accuracy of rate control. Figure 15 presents the impact of packet sizes, mcast ports, and mcast speed on the multicast delay. As shown in Figure 15(a), 64-byte packets have about 389ns multicast delay, and the delay increases by about 65ns when the packet size rises to 1280 bytes. The RMSE is lower than 4.5ns, indicating small inter-arrival time jitters and accurate rate control. Figure 15(b) shows the mcast delay of 64-byte packets, revealing that multicast ports and speed have a close-to-zero impact on the mcast delay.

Test statistic collection. Test statistics can be continuously pushed by switch ASIC or pulled by switch CPU at runtime. Thus, we evaluate *pull speed* and *push bandwidth* of test statistics, as shown in Figure 16. First, we measure received messages emitted by *generate_digest* in switch CPU when changing the message size from 16 bytes to 256 bytes. As shown in Figure 16(a), the goodput grows with the message size and can achieve 4.5Mbps. Second, we measure the delay when employing switching CPU to query different numbers of counters. In the experiment, we compare two querying methods. The first one is to query the counters one by one (*w/o O*), while the other is batch-based querying (*w/ O*). As shown in Figure 16(b), switch CPU could get 65536 counters within 0.2s under the batch-based querying, which largely outperforms the one-by-one querying method.



(a) Push bandwidth (b) Pull speed

Figure 16: Evaluation of test statistic collection.



(a) 16-bit flow digest (b) 32-bit flow digest

Figure 17: Evaluation of exact key matching.

Exact key matching. We measure the number of required exact key matching entries when the overall flow number (*i.e.* global space) and the hashing array size change. We execute each experiment for 20 times, and Figure 17 shows the data points. As shown in Figure 17(a), when the digest size is 16bits, exact key matching only needs no more than 3000 entries for over 2M flows, consuming 39KB memory. Figure 17(b) shows that the 32-bit digest could effectively reduce required entries at the cost of doubling memory usage when compared with the 16-bit digest.

7.4 Cost and Resource Analysis

In this part, we evaluate power cost, equipment cost, and data plane resource consumption of *HyperTester*.

Cost. We present a comparative analysis of equipment and power cost. According to [30], a programmable switch costs about $3600 and 150 Watts (W) per Tbps, while an 8-core CPU server costs about $3500 and 750W under full load. Based on Figure 10(b), an 8-core CPU server could generate 80Gbps traffic. Based on the above data, we analyze the cost and present the results (normalized by throughput) in Table 6. *HyperTester* running on a 6.5Tbps switch

Metrics (per Tbps)	Equipment Cost	Power Cost
MoonGen	$42000	7200W
HyperTester	$3600	150W
HyperTester Saving	**$38400, 7150W per Tbps**	

Table 6: Power and equipment cost comparison.

Category	Metric	Match Crossbar	SRAM	TCAM	VLIW	Hash Bits	SALU	Gateway
Trigger	accelerator	0.25%	0.36%	0%	0.71%	0.62%	0%	0%
	replicator(0)	0.13%	0.71%	0%	1.41%	0.62%	0%	0%
	replicator(100)	0.98%	2.12%	0%	2.82%	1.90%	5.56%	1.43%
	set(tcp.dp,range(80,100,2))	0.74%	1.06%	2.16%	2.82%	0.62%	0%	0%
	set(tcp.dp,rand('E',128,16))	0.74%	1.06%	3.23%	2.12%	1.23%	0%	0%
Query	filter(tcp.flag==SYN)	0.12%	0%	0%	0%	0.37%	0%	1.43%
	distinct(keys={5-tuple})	9.3%	12.0%	0%	12.0%	13.5%	33.4%	10%
	reduce(keys={ipv4.dip},func=sum)	13.3%	20.5%	4.3%	15.5%	13.5%	44.5%	8.6%

Table 7: Hardware resources consumed by HyperTester components. The values are normalized by switch.p4.

can replace 81 8-core CPU servers while reducing both equipment and power cost by over one order of magnitude.

Resource usage. We measure the data plane resource usage when deploying different NTAPI statements on Tofino switches, and results are normalized by the resource usage of switch.p4 [71], as shown Table 7. The accelerator consumes a small number of resources, occupying less than 1% of switch.p4. The replicator consumes different resources under different inter-departure time configurations. As for packet stream queries, we evaluate three types of statements, including *filter*, *distinct*, and *reduce*. To guarantee accuracy and improve memory efficiency, *distinct* and *reduce* support exact key matching and KV FIFO, which consume additional SRAM and SALU. Note that switch.p4 is designed for stateless packet forwarding, so it consumes a small count of SALU, which makes the normalized SALU usage of *distinct* and *reduce* seem large. In fact, *distinct* and *reduce* only consume a small portion of overall SALU resources.

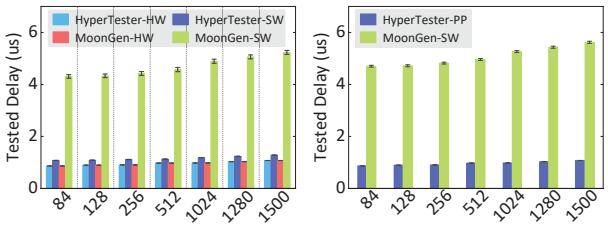
7.5 Case Study

In this part, we study two cases including delay testing and DoS attack emulation to show the benefits of *HyperTester*.

Delay testing. We implement the delay testing application [37] on both *HyperTester* and MoonGen to test Tofino switch forwarding delay. We can reasonably infer that the smaller measured delay is, the better test accuracy is. We conduct two types of delay testing experiments shown in Figure 18. The first one is to get the delay via the timestamp piggybacked in packets, while the other is to store the delay states. For timestamp-based delay testing, we can use both hardware (HW, NIC for MoonGen, and MAC for *HyperTester*) and software (SW, CPU for MoonGen, and P4 pipeline for *HyperTester*) to piggyback timestamps. Figure 18(a) shows the timestamp-based delay testing results. We can see that HW timestamps have the best accuracy, and the accuracy of *HyperTester*-SW is a little lower than HW, while MoonGen-SW performs worst and deviates from the HW results by over 3x. As for the state-based experiments shown in Figure 18(b), *HyperTester* keeps a similar accuracy as timestamp-based testing and outperforms MoonGen.

Metrics	Testbed	Estimation (80%)
Throughput	400Gbps	5.2Tbps
SYN Packets	595Mpps	7737Mpps
# emulated attack agents	4×10^5	5.2×10^6

Table 8: Analysis on SYN flood attack emulation.



(a) Timestamp-based (b) State-based

Figure 18: Delay testing results.

Attack emulation. We implement the SYN flood attack emulation [37] on *HyperTester*. The attack emulation is of great significance for evaluating the reliability of distributed systems. We measure the throughput of generated SYN packets on our testbed with four 100Gbps ports. Moreover, we estimate that a 6.5Tbps Tofino switch can achieve up to 80% of the total 6.5Tbps bandwidth when generating 64-byte SYN packets. Furthermore, we assume that a distributed attack agent can generate up to 1Mbps SYN flood traffic [72]. Table 8 indicates that *HyperTester* implemented on a 6.5Tbps switch can emulate 5.2×10^6 attack agents.

8 CONCLUSION

In this paper, we present a novel network tester, *HyperTester*, which empowers many significant network testing applications. Driven by new-generation programmable switches, the core idea of *HyperTester* is to co-design switching ASIC and switch CPU for high-performance, flexible, and low-cost network testing. *HyperTester* provides several key designs to overcome the challenges of implementing network testing with programmable switches, including template-based packet generation, stateless connections, and the counter-based algorithm. Our experiments testify that the designs of *HyperTester* work efficiently on programmable switches with acceptable resource overheads.

ACKNOWLEDGEMENT

We thank Prof. Jelena Mirkovic for shepherding the paper and Prof. Ramon Fontes for reviewing the paper reproducibility. We thank all anonymous reviewers for their constructive comments. We thank Chen Sun, Zhilong Zheng, Heng Yu, Yunsenxiao Lin, Yiran Zhang, Zili Meng, Yimin Jiang, Weibin Meng, and Ya Su for their important suggestions on this work. Prof. Mingwei Xu is the corresponding author. This research is supported by National Key R&D Program of China (2017YFB0801701) and the National Science Foundation of China (No. 61872426, No. 61625203, and No. 61832013).

REFERENCES

[1] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *Proceedings of NDSS*, 2013.

[2] Jan Rüth, Christian Bormann, and Oliver Hohlfeld. Large-scale scanning of tcp's initial window. In *Proceedings of IMC*, 2017.

[3] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G. Paterson, Narseo Vallina-Rodriguez, and Juan Caballero. Coming of age: A longitudinal study of tls deployment. In *Proceedings of IMC*, 2018.

[4] pktgen. Website, 2019. https://www.kernel.org/doc/Documentation/networking/pktgen.txt.

[5] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of IMC*, 2015.

[6] pktgen-dpdk. Website, 2019. http://git.dpdk.org/apps/pktgen-dpdk.

[7] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of SIGCOMM*, 2015.

[8] Yilong Geng, Shiyu Liu, Zi Yin, Balaji Prabhakar, Mendel Rosenblum, Ashish Naik, and Amin Vahdat. Simon: A simple and scalable method for sensing, inference and measurement in data center networks. In *Proceedings of NSDI*, 2019.

[9] Tian Bu, Nick Duffield, Francesco Lo Presti, and Don Towsley. Network tomography on general topologies. In *Proceedings of SIGMETRICS*, 2002.

[10] Yanghua Peng, Ji Yang, Chuan Wu, Chuanxiong Guo, Chengchen Hu, and Zongpeng Li. detector: a topology-aware monitoring system for data center networks. In *Proceedings of ATC*, 2017.

[11] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. *IEEE/ACM Trans. Netw.*, 22(2), 2014.

[12] Yu Zhao, Huazhe Wang, Xin Lin, Tingting Yu, and Chen Qian. Pronto: Efficient test packet generation for dynamic network data planes. In *Proceedings of ICDCS*, 2017.

[13] Peter Perešíni, Maciej Kuźniar, and Dejan Kostić. Monocle: Dynamic, fine-grained data plane monitoring. In *Proceedings of CoNEXT*, 2015.

[14] Peng Zhang, Cheng Zhang, and Chengchen Hu. Fast testing network data plane with rulechecker. In *Proceedings of ICNP*, 2017.

[15] Xitao Wen, Kai Bu, Bo Yang, Yan Chen, Li Erran Li, Xiaolin Chen, Jianfeng Yang, and Xue Leng. Is every flow on the right track?: Inspect sdn forwarding with rulescope. In *Proceedings of INFOCOM*, 2016.

[16] Yibo Zhu, Ben Y. Zhao, Haitao Zheng, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, and Ming Zhang. Packet-level telemetry in large datacenter networks. In *Proceedings of SIGCOMM*, 2015.

[17] Francois Aubry, David Lebrun, Stefano Vissicchio, Minh Thanh Khong, Yves Deville, and Olivier Bonaventure. Scmon: Leveraging segment routing to improve network monitoring. In *Proceedings of INFOCOM*, 2016.

[18] Yu Zhou, Jun Bi, Yunsenxiao Lin, Yangyang Wang, Dai Zhang, Zhaowei Xi, Jiamin Cao, and Chen Sun. P4tester: Efficient runtime rule fault detection for programmable data planes. In *Proceedings of IWQoS*, 2019.

[19] IXIA. Test hardware. Website, 2019. https://ixia.keysight.com/products/test-hardware.

[20] Spirent. Spirent testcenter. Website, 2019. https://www.spirent.com/products/testcenter.

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

[22] G. Adam Covington, Glen Gibb, John W. Lockwood, and Nick McKeown. A packet generator on the netfpga platform. In *Proceedings of FCCM*, 2009.

[23] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W Moore. Oflops: An open framework for openflow switch evaluation. In *Proceedings of PAM*, 2012.

[24] Paul Emmerich, Sebastian Gallenmuller, Gianni Antichi, Andrew W. Moore, and Georg Carle. Mind the gap - a comparison of software packet generators. In *Proceedings of ANCS*, 2017.

[25] 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.

[26] 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.

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

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

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

[30] John Sonchack, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of EuroSys*, 2018.

[31] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, SOSR, pages 164–176. ACM, 2017.

[32] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and Fast Network-wide Measurements. In *Proceedings of SIGCOMM*, 2018.

[33] Qun Huang, Patrick P. C. Lee, and Yungang Bao. Sketchlearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference. In *Proceedings of SIGCOMM*, 2018.

[34] The P4 Language Consortium. The p4 language specificatio. Website. https://goo.gl/Sq3TRK.

[35] Muhammad Shahbaz, Lalith Suresh, Jennifer Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo: Source routed multicast for public clouds. In *Proceedings of SIGCOMM*, 2019.

[36] Marvell. Zen and the art of network timestamping. Website. https://www.marvell.com/documents/7a47sa82moydu7z8uvam/.

[37] Repositories for hypertester with strict anonymity. Website, 2019. https://github.com/hypertester/.

[38] Pat Bosshart, Glen Gibb, Hun-seok Kim, George Varghese, Nick Mckeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. P4: Programming protocol-independent packet processors. *SIGCOMM CCR*, 44(3), 2014.

[39] Noa Zilberman, Yury Audzevich, Georgina Kalogeridou, Neelakandan Manihatty-Bojan, Jingyun Zhang, and Andrew Moore. Netfpga: Rapid prototyping of networking devices in open source. *SIGCOMM CCR*, 45(4):363–364, August 2015.

[40] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of SIGCOMM*, 2016.

[41] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4fpga: A rapid prototyping framework for p4. In *Proceedings of SOSR*, pages 122–135, 2017.

[42] Dijilent. Netfpga-sume virtex-7 fpga development board. Website. https://store.digilentinc.com/netfpga-sume-virtex-7-fpga-development-board/.

[43] Scapy. Scapy project. Website. https://scapy.net.

[44] Data plane development kit. Website, 2019. https://www.dpdk.org.

[45] Luigi Rizzo. netmap: A novel framework for fast packet i/o. In *Proceedings of ATC*, 2012.

[46] 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 SOSP*, 2017.

[47] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *Proceedings of NSDI*, 2018.

[48] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proceedings of SOSR*, 2015.

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

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

[51] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of tcp. In *Proceedings of SOSR*, 2017.

[52] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In *Proceedings of CoNEXT*, 2016.

[53] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *Proceedings of NSDI*, 2016.

[54] Li Chen, Ge Chen, Justinas Lingys, and Kai Chen. Programmable switch as a parallel computing device. *CoRR*, abs/1803.01491, 2018.

[55] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven network telemetry. In *Proceedings of SIGCOMM*, 2018.

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

[57] Sheng Liu, Theophilus A. Benson, and Michael K. Reiter. Efficient and safe network updates with suffix causal consistency. In *Proceedings of EuroSys*, 2019.

[58] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *Proceedings of NSDI*, 2018.

[59] Eric Osterweil, Angelos Stavrou, and Lixia Zhang. 20 years of ddos: a call to action. *CoRR*, abs/1904.02739, 2019.

[60] David Moore, Geoffrey M. Voelker, and Stefan Savage. Inferring internet denial-of-service activity. In *Proceedings of SSYM*, 2001.
[61] Akamai. Akamai ddos protection. Website, 2019. https://www.akamai.com/us/en/resources/ddos-protection.jsp.
[62] A10 Networks. Thunder tps. Website, 2019. https://www.a10networks.com/products/thunder-tps/.
[63] Akamai. Akamai ddos protection. Website, 2019. https://www.akamai.com/us/en/resources/ddos-protection.jsp.
[64] CloudFlare. Cloudflare advanced ddos attack protection. Website, 2019. https://www.cloudflare.com/ddos/.
[65] The Apache Software Foundation. Flink: Stateful computations over data streams. Website. https://flink.apache.org.
[66] Wikipedia. Inverse transform sampling. Website. https://en.wikipedia.org/wiki/Inverse_transform_sampling.
[67] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), July 1970.
[68] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
[69] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *Proceedings of ICNP*, 2018.
[70] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of CoNEXT*, 2014.
[71] The P4 Language Consortium. Consolidated switch repository. Website, 2019. https://github.com/p4lang/switch.
[72] A10. Testing ddos defense effectiveness at 300 gbps scale and beyond. Website. https://www.a10networks.com/marketing-comms/white-papers/testing-ddos-defense-effectiveness-300-gbps-scale/.