Toward Low-Latency and Accurate State Synchronization for Programmable Networks

Xiang Chen, *Member, IEEE*, Hongyan Liu, Qun Huang, *Member, IEEE*, Dong Zhang, *Member, IEEE*, Haifeng Zhou, Chunming Wu, Xuan Liu, *Senior Member, IEEE*, and Qiang Yang, *Senior Member, IEEE*

***Abstract*— Programmable switches empower stateful packet processing, in which incoming packets continuously update states in the data plane, while applications in the control plane read and write states. However, since the data plane and control plane are separated, a consistent view of states in both planes is required for stateful packet processing. Existing approaches suffer from either high latency or low accuracy. In this paper, we propose ApproSync, a framework that offers *approximate* state synchronization with low latency and high accuracy. To achieve low latency, ApproSync directly transfers states between switch ASICs and the control plane by bypassing switch operating systems. To achieve high accuracy, ApproSync utilizes the resources in the switch ASIC to realize rate control in state synchronization, such that it avoids potential state loss. It also bounds the divergence between the states in the data plane and that in the control plane under limited link capacity. We prototype ApproSync on Barefoot Tofino switches. The experimental results indicate that compared to existing approaches, ApproSync achieves order-of-magnitude latency reduction while maintaining high accuracy of state synchronization. Also, our experiments demonstrate that ApproSync provides significant latency benefits to existing network management applications and well preserves high application-level accuracy.**

***Index Terms*— State synchronization, programmable switches, approximate techniques.**

Manuscript received 24 April 2022; revised 18 August 2022 and 25 September 2022; accepted 24 October 2022; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor X. Luo. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB2901305, in part by the National Natural Science Foundation of China under Grant 61902362 and Grant 62172007, in part by the Key Research and Development Program of Zhejiang Province under Grant 2021C01036, in part by the Joint Funds of the National Natural Science Foundation of China under Grant U20A20179, in part by the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform), and in part by the Major Science and Technology Infrastructure Project of Zhejiang Laboratory (Large-Scale Experimental Device for Information Security of New Generation Industrial Control System). A preliminary version of this paper appears as a conference paper published by the IEEE ICNP 2020 conference [DOI: 10.1109/ICNP49622.2020.9259414]. *(Corresponding author: Haifeng Zhou.)*

Xiang Chen, Hongyan Liu, and Chunming Wu are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310007, China (e-mail: wasdnsxchen@gmail.com; hyliu20@zju.edu.cn; wuchunming@zju.edu.cn).

Qun Huang is with the Department of Computer Science and Technology, Peking University, Beijing 100871, China (e-mail: huangqun@pku.edu.cn).

Dong Zhang is with the College of Computer Science and Big Data, Fuzhou University, Fuzhou 350116, China (e-mail: zhangdong@fzu.edu.cn).

Haifeng Zhou is with the College of Control Science and Engineering, Zhejiang University, Hangzhou 310007, China (e-mail: zhouhaifeng@zju.edu.cn).

Xuan Liu is with the College of Information Engineering (College of Artificial Intelligence), Yangzhou University, Yangzhou 225012, China, and also with the School of Computer Science and Engineering, Southeast University, Nanjing 211189, China (e-mail: yusuf@yzu.edu.cn).

Qiang Yang is with the College of Electrical Engineering, Zhejiang University, Hangzhou 310007, China (e-mail: qyang@zju.edu.cn).

Digital Object Identifier 10.1109/TNET.2022.3218446

I. INTRODUCTION

RECENT advances in programmable networks empower network administrators to customize the packet processing logic of programmable switches. For example, with the P4 language [2], administrators are able to implement new network protocols and functions on programmable switches. Programmable switches expose a collection of stateful memory (e.g., registers) to store the *state* of packet processing. The state is a set of historical processing values (e.g., packet counts) that affect future processing decisions. By manipulating state values, administrators can build *stateful* network management applications such as traffic monitoring [3], [4], [5].

However, the separation of the data plane and control plane raises the problem of *state synchronization*. On the one hand, data plane packets continuously update the state maintained by each switch at line rate. On the other hand, the control plane applications issue state read or write operations to manipulate states. Thus, it is indispensable to keep a consistent view of states in both planes. In particular, given the huge volume and high speed of state updates, it requires synchronizing states within ultra-low latency to meet the requirements raised by latency-sensitive applications. For example, UDP flood mitigation [6] needs to collect thousands of state values from switches within a few microseconds to rapidly detect attacks. Also, such synchronization should be as accurate as possible so that applications can work on correct information.

Unfortunately, it remains a void to *efficiently* and *accurately* realize state synchronization in programmable networks. Today, state synchronization is achieved via an operating system (OS) in every switch. The switch OS manipulates state values in the underlying switch ASIC via PCIe channels and connects to the control plane via TCP-based protocols [7], [8], [9], [10]. Both PCIe channels and TCP connections are the performance bottlenecks to synchronize state updates incurred by high-speed traffic [11], [12]. Our experiments in §II-B indicate that the OS-based approach spends several

seconds to transfer a state with a normal size of 2^{16} values, which is inefficient. To achieve low latency, traffic mirroring approaches [12], [13], [14] bypass the switch OS and directly transfer state updates from the switch ASIC to the control plane. However, they lack reasonable rate control and thus suffer from serious state loss when the traffic rate exceeds link capacity.

In response, we propose ApproSync, a low-latency and accurate state synchronization framework. ApproSync bypasses the switch OS to achieve low latency. However, it is challenging to handle state loss in switch ASICs due to switch restrictions. In response, ApproSync exploits approximate strategies to achieve high accuracy. The notion behind this is that many applications tolerate a small *state divergence* between the data plane and the control plane. Thus, ApproSync allows a small state divergence, which sacrifices a portion of accuracy to alleviate the resource requirements of realizing ApproSync in switch ASICs. Also, ApproSync bounds the state divergence to limit the incurred accuracy drop.

Specifically, ApproSync offers two types of state operations for control plane applications: *state read* that collects state values from switch ASICs to the control plane, and *state write* that enforces state values from the control plane in switch ASICs. ApproSync uses two approximate strategies to realize the two types of operations, respectively.

- For state read, ApproSync monitors the state divergence in the switch ASIC and synchronizes state updates only when the divergence exceeds a threshold. It adaptively tunes the threshold based on the incoming traffic rate. (1) When the incoming traffic rate is low, it pushes *every* state update to the control plane, making synchronization error-free. (2) When incoming traffic rate is high while massive state updates need to be synchronized within a short time, it *selectively* pushes state updates to avoid link overload. This makes the state in the control plane slightly diverge from that in the data plane. However, the divergence is bounded by the threshold to keep high accuracy.
- For state write, ApproSync acknowledges each state write operation. If the ACK of an operation is timeout, it retries the operation to avoid state loss. It also ensures the atomicity of state write by preventing new state updates in the data plane from updating the state during the write. To do this, it recirculates the new state updates and eventually performs them after the write. Although such a design makes some state updates out-of-order during a write operation, their number is small since the write operation can be completed within a short time by bypassing the switch OS.

We have implemented ApproSync with Tofino switches [15]. Our experiments show that ApproSync achieves order-of-magnitude latency reduction against existing approaches while retaining high accuracy.

II. Background and Motivation

A. State Synchronization in Programmable Networks

1) Why is State Synchronization Necessary for Programmable Networks? We target programmable networks (e.g., data center networks [4], [30], [31]), where the data plane and control plane are separated in programmable networks. In the data plane, each programmable switch maintains a collection of historical packet processing information (e.g., per-flow packet counts) in its stateful elements (e.g., registers). We refer to such a collection as the *state* of the switch. Also, each switch continuously updates its state during its packet processing. Meanwhile, the control plane dynamically collects states from data plane switches and reports them to network management applications. Applications make decisions based on states and modify the states recorded in switches to perform their decisions in the data plane. For instance, the stateful firewall [32] collects state values from switches to detect malicious attacks. It also updates the detection thresholds recorded in switches with respect to traffic dynamics.

TABLE I
Examples of Network Management Applications That Require State Synchronization

Application Type	Application Examples
Network measurement [3, 4, 16, 17, 18, 19]	Flow size and entropy estimation, network-wide flow distribution analysis
Network functions (NFs) [20, 21, 22, 6, 23, 24]	Network anomaly detection, attack mitigation, dynamic routing, load balance
Network redundancy [25, 26, 27, 28, 29]	Switch state backups, switch state migration, fast failure recovery, traffic rerouting

Some may argue that the approach of entirely processing state values in the data plane is more attractive. However, it is impeded by three-fold resource limitations exhibited by programmable switches, making it impractical [13], [14], [33]. (1) The switch is equipped with a pipeline comprising a few stages, each of which has limited memory (at most 2 MB). (2) The switch only allows limited memory access (e.g., a few read-write operations for each packet). (3) The switch does not support complex operations (e.g., loop and buffering) and the operations that simultaneously manipulate multiple state values (e.g., querying). These restrictions prevent administrators from realizing their desired control logic entirely in the data plane [34], [35]. For example, in UDP flooding, administrators need to detect the UDP flow, which packet number exceeds a threshold in a given time window, as malicious. Their goal is to preserve full accuracy in attack flow detection to avoid affecting normal traffic, such that approximate data structures do not work due to their inevitable errors. Such logic cannot be realized in the switch ASIC since it requires to examine the packet counts of all flows, which consumes far more resources than the maximum capacity in the switch.

Hence, the general paradigm adopted by recent studies is to use the control plane to perform arbitrary control logic [3], [4], [13], [14], [18], [33], [35], [36], [37]. This paradigm also empowers many applications with the insight into network-wide states. Examples include the accurate analysis of network-wide traffic statistics [5] and nearly-zero-error data recovery in sketch-based measurement [38]. Also, such a distributed paradigm derives the bidirectional state synchronization that keeps the states in the data plane and the control plane consistent. (1) In the bottom-up direction, state updates incurred by data plane packets should be synchronized to the control plane. (2) In the top-down direction, the decisions of modifying states in the control plane should be reflected in the data plane.

2) Requirements: Table I shows three types of network management applications built on states. These applications

require both *low latency* and *high accuracy* in state synchronization.

- **Low latency**. We aim to reduce the latency of state synchronization in both directions (i.e., from the data plane to the control plane and vice versa). This is critical to keep pace with high-speed traffic and meet the tight latency requirements raised by applications. For example, network anomaly detection requires rapidly detecting and reacting to suspect events [18], [39], [40], [41].
- **High accuracy**. We aim to retain high accuracy for applications by bounding the state divergence between the two planes. For example, the attack detector may raise a false alarm if received states are highly noisy. Also, if a state modification is not synchronized to the data plane, switch behaviors may be wild (e.g., a firewall policy fails).

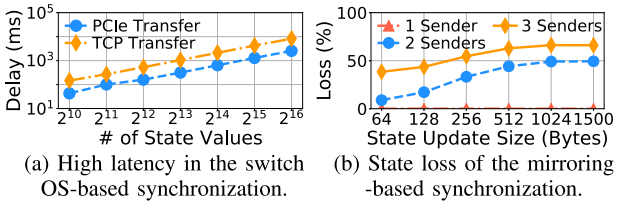
B. Limitations of Existing Approaches

Existing approaches synchronize states via either the switch OS or traffic mirroring. However, none of these approaches can achieve both low latency and high accuracy.

1) High Latency in the Switch OS-Based Approach: The OS-based approach uses the switch OS to transfer the state between the switch ASIC and the control plane. It manipulates the latest state in the switch ASIC via PCIe channels. Then it establishes TCP connections with the control plane [7], [8], [9], [10] to transfer the state. However, it suffers from high latency due to two reasons. First, due to limited bandwidth, the PCIe channels are the performance bottleneck [12]. Second, the TCP connections incur high latency in TCP stacks and reliable transmission. Figure 1(a) measures the two types of latency when synchronizing up to 2^{16} 64-bit state values in a Barefoot Tofino switch [15]. We observe that both the PCIe transfer and TCP-based transfer incur a latency of hundreds of milliseconds. For example, synchronizing 2^{16} state values takes even several seconds.

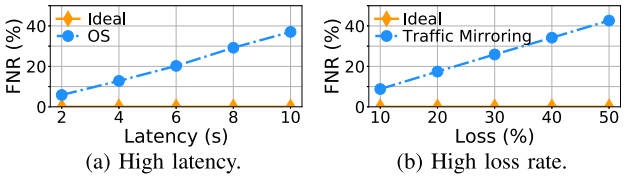
2) State Loss in the Mirroring-Based Approach: To achieve low latency, the approaches based on traffic mirroring directly mirror state updates from the switch ASIC to the control plane via a few mirroring ports [12], [13], [42], [43]. However, these approaches suffer from serious state loss when the emitted rate of state updates exceeds link capacity [12]. In Figure 1(b), we measure the loss rate in a Tofino switch. We allocate one 40-Gbps mirroring port and vary the number of traffic ports. We inject traffic to each traffic port to reach 40 Gbps. We see that with only one traffic port, there is almost no state loss. However, the loss rate rapidly rises as the number of traffic ports increases. It reaches 60% when using three traffic ports. With such a high loss rate, most state values cannot be synchronized so that the state divergence is extremely high. As a result, applications work on inaccurate states and fail to perform correct operations. Although allocating more mirroring ports can alleviate state loss, doing so unavoidably sacrifices overall switch throughput and affects normal processing.

3) Impact on Applications: We study the impact of existing approaches by testbed experiments. Our testbed uses a Tofino switch [15] that directly connects to a control plane server via a 40-Gbps link. We consider heavy hitter detection [22] as the application. A heavy hitter is a two-tuple flow whose number of packets exceeds 2^{10}. In the switch, we implement



(a) High latency in the switch OS-based synchronization. (b) State loss of the mirroring -based synchronization.

Fig. 1. Benchmarks of existing approaches.



(a) High latency. (b) High loss rate.

Fig. 2. Impact on applications.

a recently proposed hash table [13], [14] to record per-flow packet count. Due to switch resource limitations, we only equip the hash table with 2^{16} entries as suggested by the original papers. Thus, the hash table can only maintain the records of a few flows while the number of flows is excessive in practice, such that hash collisions happen frequently. To this end, it handles hash collisions with the LRU mechanism that evicts old records to the control plane and records new flows when collisions happen. Thus, it leverages the control plane to process the records of most flows. We inject a trace from CAIDA [44] into the Tofino switch. In the control plane, we collect all the state values (i.e., per-flow packet count) from the switch every second, and examine heavy hitters accordingly. We also compute the true heavy hitters using the traces as the baseline. By comparing the measured heavy hitters and true heavy hitters, we calculate the false negative rate (FNR).

We first study the impact of the high latency incurred by the OS-based approach, which can be up to several seconds (Figure 1(a)). Figure 2(a) shows that FNR significantly rises as the latency consumed by the OS-based approach increases. Next, we evaluate the impact of traffic mirroring, which suffers from high state loss rate. In Figure 2(b), the false negative rate rapidly rises as the loss rate increases.

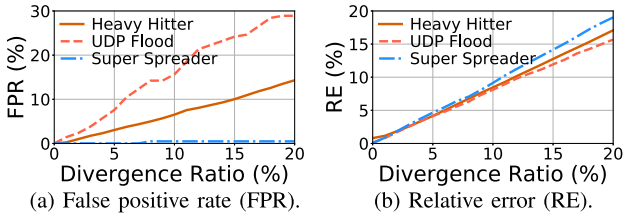
III. ApproSync Design

A. Goals

In this paper, we propose ApproSync, a framework that synchronizes states between the data plane and control plane for programmable networks. ApproSync aims to keep a consistent view of states in both the data plane and control plane. Moreover, it aims to perform its synchronization in an *accurate* manner (i.e., avoiding state loss) while retaining *low latency*, such that the two separated planes in programmable networks work as if in a single machine.

B. Challenges

There have been many solutions that handle state loss during state synchronization, e.g., timeout and retransmission mechanisms [45], [46]. Unfortunately, it is infeasible to realize these solutions in switch ASICs due to switch restrictions. Specifically, existing programmable switches typically



(a) False positive rate (FPR). (b) Relative error (RE).

Fig. 3. Impact of state divergence. We observe that a small divergence (<1%) between the states maintained by the control plane and the latest ones in the data plane only incurs a small application-level accuracy drop (<2% error).

employ specific architectures (e.g., PISA [47]) to achieve high throughput and ultra-low latency. Such architectures impose strict restrictions on their resource models due to the concern of chip footprints and heat consumption (§II-A). Given these restrictions, it is hard to handle state loss in switch ASICs.

C. Observation

Although high state divergence (e.g., >10%) seriously degrades application accuracy as shown in §II-B, we observe that it is acceptable for many applications when the divergence is small (e.g., <1%). In fact, many applications are already built on approximate algorithms such as sampling [12], [48], [49], [50], [51], [52] and sketches [3], [17], [18], [19]. Thus, a small divergence is tolerable for these applications in practice. To justify this observation, we evaluate the impact of state divergence on the accuracy of three applications: heavy hitter detection [22], UDP flood mitigation [6], and super-spreader detection [22]. A heavy hitter is a two-tuple flow whose packet count exceeds 2^{10}. UDP flood mitigation identifies all UDP flows with more than 10^5 packets within 5 seconds. A super spreader is a source IP address and targets more than 1% of the total number of distinct IP addresses.

We use the same method (i.e., the hardware-compatible hash table) and testbed as in §II-B. We explicitly drop packets between the switch and the control plane to vary the state divergence from 0% to 20%. Figure 3 shows that the application-level error as the state divergence. When the divergence is 20%, the error reaches 20% since applications rely on accurate state values to detect anomalies. However, when the divergence is small (<1%), the application-level error is small (<2%), which validates our observation.

D. Key Idea

Motivated by our observation, we design ApproSync with *approximate state synchronization*. Specifically, ApproSync directly transfers states between switch ASICs and the control plane to achieve low latency. Moreover, ApproSync allows a small state divergence during state synchronization. This enables ApproSync to satisfy the strict resource requirements in programmable switches. However, it utilizes switch resources to bound the state divergence under link capacity. Such utilization brings two-fold benefits: (1) ApproSync only requires a small portion of switch resources, which mitigates the aforementioned challenge; (2) It achieves the maximum possible accuracy for applications under the constraint of link bandwidth.

Note that approximate techniques have been widely adopted in distributed systems. Although ApproSync follows similar ideas, we address the specific challenges of adopting approximation in state synchronization for programmable networks.

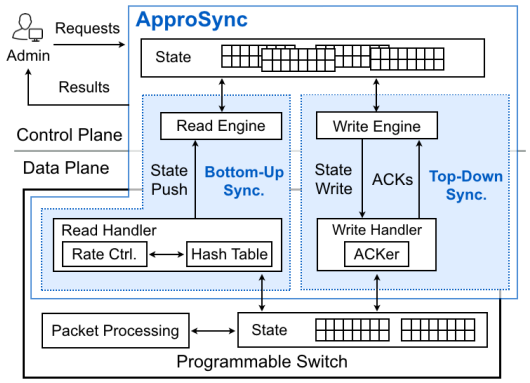


Fig. 4. Overview of ApproSync framework.

E. Architecture

As shown in Figure 4, ApproSync synchronizes states in two directions, which correspond to two types of operations, *state read* and *state write*, respectively. It offers two strategies, *bottom-up synchronization* for state read, and *top-down synchronization* for state write. Each strategy is realized by a handler in the switch ASIC and an engine in the control plane, which collectively synchronize states.

- **Bottom-up state synchronization (i.e., state read) (§IV)**. This strategy makes read operations capture the latest state updates in the data plane. The read handler aggregates state updates and pushes them to the read engine, which updates state values in the control plane. Instead of pushing all updates, it monitors the state divergence between two planes. Once the divergence exceeds a threshold, it emits state updates to make the states in both planes consistent. It adaptively tunes the threshold to keep the emitted rate below the link capacity, which avoids state loss.
- **Top-down state synchronization (i.e., state write) (§V):** This strategy writes the state modifications raised by applications to switch ASICs. The write engine exploits an acknowledgment mechanism to eliminate state loss. Also, ApproSync enables atomicity of state write. Specifically, the write handler suspends the state updates incurred by data plane packets by recirculating these updates. These updates will be eventually performed as soon as state write terminates. This makes some state updates out-of-order. However, this strategy avoids data loss and huge resource consumption of realizing complicated atomicity protocols.

IV. Bottom-Up State Synchronization

A. Synchronization Algorithm

1) Hash Table: The read handler employs a hash table H to monitor the state divergence. H uses counter indexes in the state as keys. Every entry $H[p]$ has three fields: (1) $H[p].loc$ is the state location (i.e., hash key) associated with this entry, (2) $H[p].val$ is the current state value in location $H[p].loc$, and (3) $H[p].old$ records the last state value sent to the control plane. We restrict the size of H since switch memory is scarce.

Algorithm 1 Read Handler

Input: state update (l, v)
Variables: hash table H, threshold t

```
1: function PROCESS_UPDATE(l, v)
2:     Position p = hash(l)
3:     if H[p] is empty then                ▷ Assume initial value as zero
4:         H[p].loc = l, H[p].val = v, H[p].old = 0      ▷ Insert H[p]
5:     else if H[p].loc == l then
6:         Update H[p].val = v
7:         Divergence D = |v − H[p].old|
8:         if D ≥ t then
9:             Push (H[p], t) to the control plane
10:            Update H[p].old = v
11:    else                                 ▷ H[p].loc ≠ l
12:        Push (H[p], t) to the control plane
13:        H[p].loc = l, H[p].val = v, H[p].old = 0
```

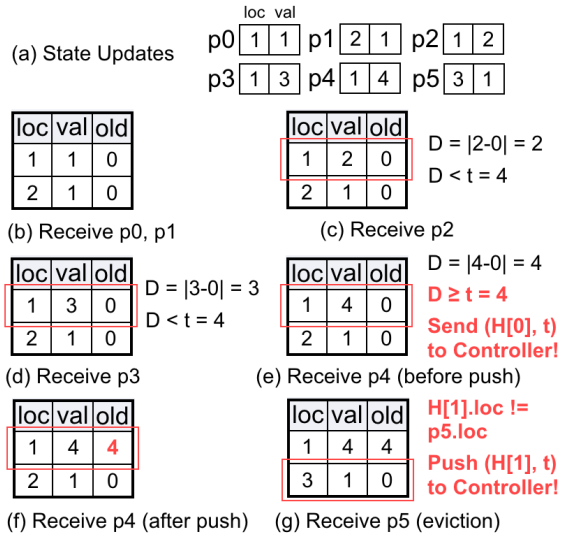


Fig. 5. Example of hash table in the read handler.

Here, a size of 2^{16} entries is enough for most applications to retain high accuracy. Moreover, when hash collisions happen, H evicts old entries to the control plane and inserts new keys.

One concern is that the single hash table in ApproSync will incur frequent hash collisions and exhaust link bandwidth. However, most traffic is contributed by a few flows due to the skewness of network traffic in modern networks [53]. Hence, most state updates are incurred by a few flows, making the probability of hash collisions small. For instance, consider the case of using 2^{16} hash table entries to synchronize a sketch that assembles ten counter arrays with 10 MB memory. In this case, we observe that the probability of hash collisions is below 5% for a CAIDA 2018 trace [44]. This leads to a peak emitted rate of 0.92 Mpps in practice, which is acceptable.

2) Algorithm: The read handler is invoked for every tuple (l, v), which indicates that the value in location l has been updated to v (Algorithm 1). It first hashes l to calculate its position p in H (line 2). If $H[p]$ is empty (line 3), it directly inserts (l, v) (line 4). Otherwise, it compares l with existing stored location $H[p].loc$ (line 5). If the two positions are the same, the read handler updates the state value (line 6). Then it computes the divergence between the current state value v and that in the control plane recorded by $H[p].old$ (line 7). If the divergence exceeds a threshold t, both the entry $H[p]$ and t are emitted to the control plane (lines 8-9). t indicates the maximum tolerable divergence between a state value recorded in the switch ASIC and that in the control plane. We detail how the read handler tunes t in §IV-B. Moreover, $H[p].old$ is changed to the last value sent to the control plane (line 10). If the stored location differs from the new location (line 12), indicating a hash collision, the read handler pushes the existing entry and t to the control plane (line 13), and then modifies the entry to store the new one (line 14).

3) Example: Suppose that the threshold $t = 4$ and the hash table H has two buckets, and there are six state updates (Figure 5(a)). For p_0 and p_1, H directly inserts them and initials $H[p].old$ to zero (Figure 5(b)). For p_2, H maps it to the first bucket, which already stores a state update with the same location of p_2. H calculates the divergence $D = 2 < t$. Thus, H does not send p_2 to the control plane (Figure 5(c)). H processes p_3 similarly to p_2. After processing p_3, $H[p].val$ of the first bucket is three (Figure 5(d)). When p_4 arrives, H calculates the divergence D, which now reaches the threshold t (Figure 5(e)). Thus, H sends p_4 and t to the control plane and updates $H[p].old$ with $H[p].val$ (Figure 5(f)). Finally, p_5 comes in, H hashes it to the second bucket. It finds that the location of p_5 does not match the location stored in the bucket. Thus, H sends the old entry and t to the control plane and inserts p_5 (Figure 5(g)).

4) Impact of Parameters: According to Algorithm 1, there are three user-specified parameters, including w that specifies the time window of measurement, the available bandwidth capacity c, and the size s of each state update. We discuss their impacts on the efficiency of rate control as follows.

(1) w only guides the read handler of ApproSync on when to collect the counter value change Δ. It has no direct impact on the threshold t. This is because according to the definition of t, t is determined by the fraction of the counter value change Δ to w, i.e., $\frac{\Delta}{w}$, which is a metric that profiles the current incoming traffic rate. Given a fixed average incoming traffic rate, Δ changes with respect to the changes in w: Δ increases for a larger w (i.e., counting more values in a larger window) while decreasing for a smaller w (i.e., counting less values in a smaller window). As a result, $\frac{\Delta}{w}$ remains unchanged, such that t is unchanged. Hence, only changing w does not affect t and thus has no direct impact on the efficiency of rate control. To this end, we choose to quantify the impact of incoming traffic rates on rate control, which collectively considers both w and Δ, in Exp#3.

(2) Recall that c and s together determine the maximum tolerable rate M of emitting state updates without incurring state loss: $M = \frac{c}{s}$, which is negatively correlated to the threshold t. To study the impacts of these parameters, we first fix the incoming traffic rate, which fixes Δ and w. In this context, we analyze the impacts in two-fold. First, when we increase c and fix s, or decrease s and fix c, M is increased so ApproSync can use more bandwidth to emit state updates without incurring state loss. In this case, ApproSync decreases t to transfer more state updates to the control plane within the window. Second, when we decrease c and fix s, or increase s and fix c, M is reduced so that still transferring the same number of state updates as usual inevitably leads to bandwidth

saturation and state loss. To this end, ApproSync increases t to avoid state loss with the compromise of transferring less state updates within a window. In summary, in both cases, the changes in the parameters, c and s, will determine the value of the threshold t. However, ApproSync guarantees that no state loss will happen by flexibly tuning t with respect to c and s. Thus, we conclude that changing c and s does no affect the efficiency of rate control in ApproSync.

5) Support of Multi-Level Data Structures: ApproSync is able to support the synchronization of user-specified multi-level data structures such as sketches and multiple hash tables. Specifically, in ApproSync, each state update specifies (1) the location l of the updated state value in the target user-specified data structure and (2) the updated state value v. When there exists only one data structure, l stands for the index j of the updated state value in the data structure. For example, if the 5-th entry of the user-specified hash table is updated, $l = j = 5$. When the data structure is multi-level (e.g., multiple hash tables), l refers to the two-tuple (i, j), where i corresponds to the i-th level of the data structure. For example, if the 3-rd entry of the 5-th hash table is updated, $l = (3, 5)$. In this way, the read handler of ApproSync is able to process both the state updates of a standalone data structure (e.g., a single hash table) and that of multi-level data structures (e.g., multiple hash tables) with Algorithm 1.

B. Rate Control Algorithm

1) Design Decision: The threshold t controls the trade-off between accuracy and bandwidth consumption. Our intuition is that we can employ a small threshold to bound the state divergence as long as link capacity is not exhausted. With that in mind, we design a rate controller in the switch OS that adaptively tunes t instead of specifying a fixed one. In particular, we observe that the incoming traffic rate directly affects the bandwidth consumption of state updates. When the incoming traffic rate increases, the state values will be more frequently updated. In this case, more state updates are generated in a short time, which increases the emitted rate of state updates. Thus, we design the rate controller to tune t with respect to incoming traffic rate. Moreover, we employ a global threshold t instead of tuning the threshold for each entry in the hash table due to two reasons. First, per-entry thresholds occupy a large amount of memory. Second, simultaneously tuning multiple thresholds requires non-trivial computational resources, which is infeasible in restrictive switch ASICs.

2) Algorithm: As shown in Algorithm 2, the rate controller performs three steps to tune the threshold t.

Step 1: the rate controller estimates the state update rate. Specifically, the rate controller reads the total change Δ of all state values within a time window w. It employs a dedicated 64-bit counter in the switch ASIC to count the number of state updates. It reads the value change of the counter as the estimate of Δ. Such design is low-overhead, e.g., given a time window $w = 1$ms, the rate controller consumes 6.4×10^{-2} Mbps to read Δ, which is small compared to the Gbps-level switch bandwidth. Moreover, w is determined by applications. For instance, the detection of low-rate TCP denial-of-service attacks [4], [54] needs to detect microbursts that happen in a few milliseconds, so a reasonable w is 1 ms. Given Δ and w, the emitted rate of state updates is then calculated as $\frac{\Delta}{w}$. $\frac{\Delta}{w}$ is a metric that reflects incoming traffic rate: when incoming traffic rate is low, Δ is small, making $\frac{\Delta}{w}$ small; otherwise, incoming traffic rate is high.

Algorithm 2 Rate Controller

Input: Time window w, bandwidth capacity c, state update size s
Output: Threshold t for rate control
Variables: Counter value change Δ, maximum emitted rate M
1: **function** RATE_CONTROL(w, c, s)
2: Obtain Δ from the 64-bit counter in the switch ASIC
3: $M \leftarrow \frac{c}{s}$ ▷ Calculate maximum emitted rate
4: **if** $\frac{\Delta}{w} \leqslant M$ **then** ▷ The capacity is enough to transfer all updates
5: $t \leftarrow 0$ ▷ Transferring all the updates
6: **else** ▷ Transferring all the updates will saturate the bandwidth
7: $t \leftarrow \lceil \beta \frac{\Delta}{wM} \rceil$ ▷ Transferring state updates without loss
8: **return** t

Step 2: the rate controller calculates the maximum emitted rate supported by the link. Specifically, the maximum emitted rate is calculated as $M = \frac{c}{s}$. Here, c is a user-configurable parameter that indicates the maximum bandwidth allocated for state synchronization, while s is the size of a state update. c can be set to the link capacity so that the link is dedicated to transfer state values. Also, it can be set to a value smaller than link capacity to allow other traffic to use the same link.

Step 3: the rate controller tunes t by examining whether $\frac{\Delta}{w} \leqslant M$. If so, which indicates that link capacity is sufficient to support the emitted rate of state updates, the rate controller sets t to zero to send every state update to the control plane. Otherwise, the emitted rate will exceed link capacity so that the rate controller needs to set a non-zero t for avoiding link saturation. In this case, a state update is emitted when a state value is changed by t. Thus, the emitted rate of state updates can be estimated as $\frac{\Delta}{wt}$. To avoid link saturation, the rate controller tunes t to keep the emitted rate $\frac{\Delta}{wt}$ just less than M, implying $t = \lceil \frac{\Delta}{wM} \rceil$. In practice, the rate controller sets a $t = \lceil \beta \frac{\Delta}{wM} \rceil$ for some $\beta \geq 1$ to handle unexpected traffic bursts. Our experience is that a small β closed to one is sufficient. We summarize how the rate controller sets t as follows (lines 4-7).

$$t = \begin{cases} 0, & \text{if } \frac{\Delta}{w} \leqslant M \\ \lceil \beta \frac{\Delta}{wM} \rceil, & \text{otherwise.} \end{cases}$$

3) Example: We illustrate the rate control via an example. We assume that (1) ApproSync uses a 10-Gbps link to transfer 16-byte state updates so that link capacity $c = 10^{10}$bps and the size of a state value $s = 128$ bits; (2) the time interval $w = 1$ ms; (3) no microbursts happen so a $\beta = 1$ is sufficient. Thus, $M = \frac{c}{s} = 7.8125 \times 10^7$ pps. Suppose that $\Delta = 10^4$ in the first time interval. Since $\frac{\Delta}{w} = 10^7 \leqslant M$, the read controller sets the threshold t to zero, such that every state update is directly transferred to the control plane. In the second time interval, Δ is changed to 10^5, making $\frac{\Delta}{w} > M$. In this case, the rate controller sets $t = \lceil \beta \frac{\Delta}{wM} \rceil = 2$. Thus, the maximum emitted rate of state updates is $\frac{\Delta}{wt} \approx 5 \times 10^7$pps $< M$, which avoids link saturation and state loss.

4) Case Study: The read handler sends the current t with each state update to the control plane (Line 9 in Algorithm 1). With t, Applications can quantify the accuracy of the state. For example, we consider Count-Min (CM) [55] sketch, which maintains a counter array to estimate flow sizes. In the data

plane, a packet selects some counters in the array based on its flow ID, and then increments selected counters. Thus, every counter serves as an estimate for the packet count of the flow. When using ApproSync to collect counter values, the divergence between the collected counter values and the latest counter values recorded in the data plane unavoidably affects the accuracy of the CM sketch. However, ApproSync guarantees that the state divergence will not exceed the threshold t. Thus, we can fix the error bounds of the CM sketch with the threshold t, as shown in Lemma 1.

Lemma 1: Consider a CM sketch with r rows and w counters in each row. Let T_f and E_f denote the true value and estimated value of a flow f, respectively. When deployed in ApproSync, the CM sketch guarantees that: (1) $E_f \geq T_f - t$, and (2) $E_f \leq T_f + \frac{2U}{w} - t$ with a probability at least $1 - \frac{1}{2^r}$, where U is the total value of all flows.

Proof: The original CM sketch guarantees that $T_f \leq E_f \leq T_f + \frac{2U}{w}$ with a probability larger than $1 - \frac{1}{2^r}$ (Theorem 1 in [55]). With ApproSync, the counter in the control plane is smaller than that in the data plane by at most t because the value has not been synchronized yet. Thus, the lower bound and upper bound of E_f become $T_f - t$ and $T_f + \frac{2U}{w} - t$, respectively. The results follow. □

C. Discussion

1) Stale States: A problem is that some state updates may stay in the hash table for a long time, but no packets trigger the synchronization for them. To this end, we design the hash table to timely detect such stale entries based on table aging [2]. Specifically, when the time that an entry resides in the hash table exceeds a timeout value that is set by applications, the hash table raises a timeout signal to notify the switch OS. The switch OS then demands the hash table to immediately send the expired entry to the control plane. In this way, ApproSync alleviates the above concern. Note that we involve the switch OS because the logic of receiving timeout signals cannot be implemented in the switch ASIC due to switch restrictions. However, unlike the OS-based approach that brings high latency overhead, ApproSync adopts the switch OS to only forward timeout signals rather than transfer state values. Doing so neither incurs high bandwidth consumption nor affects the timeliness of state synchronization.

In particular, the timeout value is determined by specific applications because various applications exhibit different latency requirements, e.g., security applications require more timely state synchronization than other applications to detect attacks as soon as possible [35], [37]. Thus, ApproSync allows its users to specify their desired timeout values of the hash table. By default, we set the timeout value to be equal to the length of two consecutive measurement windows. This is because if an entry has not been synchronized for two windows, it is likely to record finished flows. Thus, this entry may contain stale values and is set to be timeout.

2) Robustness: Another concern is that the state updates sent to the control plane may be lost, which degrades the consistency. ApproSync alleviates this concern from two aspects. First, it adaptively controls the emitted rate, making the probability of the above condition small (close to zero as empirically demonstrated in Exp#3 in §VII). Second, even if some updates were lost, the subsequent state updates will be sent to the control plane soon given the high updating rate of states, which mitigates the loss of previous updates.

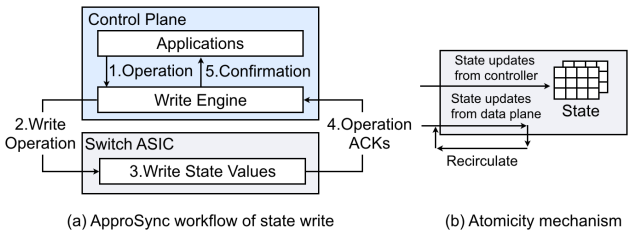


Fig. 6. State write mechanism.

V. Top-Down State Synchronization

A. Synchronization Algorithm

1) Write Acknowledgment: As shown in Figure 6(a), control plane applications issue a write operation comprising a set of state updates to the write engine. The write engine encapsulates these updates in several packets. For each packet, it allocates a dedicated timer and waits to receive an ACK after sending the packet to the destination switch ASIC. The write handler in the switch ASIC acknowledges every packet sent by the write engine. Specifically, it performs the state updates to modify state values, and then sends an ACK back. If a timer raises a timeout, which indicates the loss of a packet, the write engine immediately retransmits the packet. After all the ACKs of sent packets are received, it notifies applications of write success and informs the write handler of termination.

2) Atomicity Mechanism: At times, applications need to simultaneously write multiple state values, which requires *atomicity* to avoid unpredictable results. However, during state write, data plane packets also continuously update the state in the switch ASIC, which harms atomicity. One strawman solution is to avoid conflicts between state write and new state updates incurred by data plane packets via concurrency control [56], [57], [58]. However, existing concurrency control methods cannot be implemented on programmable switches because these methods require either excessive memory (e.g., 2PL [59], timestamps ordering [59], optimistic concurrency control [60], 2PC [61], 3PC [62]) or complex queue scheduling (e.g., deterministic system [63]).

To this end, ApproSync offers a hardware-compatible atomicity mechanism, i.e., STATE_WRITE in Algorithm 3. It guarantees that the state write driven by the control plane will not be interrupted by data plane packets. As depicted in Figure 6(b), the main idea is to lock the *entire* data plane state and recirculate new state updates during state write. Specifically, the data plane packets that enter switches during state write, i.e., inflight packets, are normally forwarded, i.e., no packets will be recirculated. In contrast, since the state is locked until state write finishes, the new state updates incurred by inflight packets will be recirculated for second-pass processing. The recirculation continues until the state write is completed and the lock is free. When state write completes, the write handler eventually performs the recirculated state updates on its state values (lines 10-11, Algorithm 3). Thus, it avoids the conflicts between state write and new updates, ensuring atomicity.

Our rationale behind is two-fold. First, in most cases, the priority of state write is higher than data plane updates.

Algorithm 3 Write Handler

Input: state update (l, v), termination flag γ
Variables: lock L

```
1: function STATE_WRITE((l, v), γ)
2:     if (l, v) is from the control plane then
3:         Lock L
4:         Update the state value in location l to value v
5:         Send ACK to the control plane
6:         if γ is True then ▷ Write engine stops state write
7:             Unlock L
8:     else ▷ Update by a data plane packet
9:         if L is unlocked then
10:            Update the state value in location l to value v
11:        else
12:            Recirculate (l, v)
13: function STATE_RESET((l, v), γ)
14:    if (l, v) is from the control plane and v = 0 then
15:        Lock L
16:        Reset the state value in location l to zero
17:        if l points to the last state value then ▷ Reset done
18:            Unlock L
19:            Send ACK to the control plane
20:        else
21:            (l, v) ← (l + 1, 0)
22:            Recirculate (l, v)
23:    else ▷ Update by a data plane packet
24:        if L is unlocked then
25:            Update the state value in location l to value v
26:        else
27:            Recirculate (l, v)
```

This is because state write raised by applications usually changes processing strategies (e.g., resetting sketches). Thus, prioritizing state write ensures that merely using one lock is sufficient for resolving concurrency conflicts. It also naturally avoids deadlocks since only state write can obtain the lock. Second, the operations of ApproSync should remain simple so that they can be implemented in switch ASICs.

B. State Reset Algorithm

1) Motivation: A common case of state write is to reset the entire state (i.e., resetting all state values to zero) maintained by the target switch. For instance, before starting a new time interval, network measurement applications reset the entire counter array in the switch ASIC to prevent legacy statistics from disturbing on-going measurement [34], [64]. In particular, we observe that the state updates for state reset share the same intent, i.e., resetting a specific value to zero. This brings us an opportunity of merging these state updates into a single update that orders the target switch to reset all state values. Such merging avoids the need of generating and acknowledging massive state updates. Thus, it brings more latency benefits in state reset than the state write algorithm in §V-A.

2) Workflow: We design a state reset algorithm in ApproSync, i.e., STATE_RESET in Algorithm 3. More precisely, when applications issue a request of state reset, the write engine generates a control plane packet that encapsulates a special state update $(l = 0, v = 0)$ to indicate state reset. When the write handler in the target switch receives this update, it performs the same atomicity mechanism mentioned in §V-A through a lock L of the entire state. Meanwhile, it identifies the state value based on the location l in the state update and resets the value to zero (line 16). Next, it increments the location l by one so that the state update points to the next value (lines 20-22). It repeats its processing by recirculating the state update. If the state update points to the last state value, indicating that all state values have been reset, the write handler returns an ACK of state write to the control plane and releases the lock L (lines 17-19). In particular, since the state update for state reset may be lost during transmission, we set ApproSync to simultaneously send several identical state updates to the switch ASIC. In this way, even when one state update is lost, the subsequent state updates will mitigate the loss and still perform state reset. By default, we set the number of state updates to 20, which ensures high success probability of state reset. For example, even when the loss rate becomes 80%, the success probability of state reset is bounded to 99%.

Although our algorithm needs to recirculate the state update of state reset multiple times in the switch ASIC, it reduces the latency of normal state write by one order of magnitude (see Exp#4 in §VII). This is because the time of recirculation is nanosecond-level. Thus, even though ApproSync needs to recirculate the state update for thousands of times to reset the entire state, its time is still less than 1 ms.

C. Discussion

1) Robustness: If the control plane or links failed in the middle of state write, the lock will never be freed and the switch bandwidth will be saturated soon by recirculated updates. To this end, we design the write handler in the switch ASIC to wait for a time period after receiving a control plane packet. If no new control plane packets arrive during the period, the write handler will release the lock and perform the recirculated updates. After the link or control plane is recovered, the control plane can perform the same state write operation to eventually write the demanded state.

2) Impact of Out-of-Order State Updates: Due to its recirculation, ApproSync may incur out-of-order state updates during its state write. However, it completes within a small time period (e.g., less than 20 ms for writing 2^{16} state values in Exp#2). Thus, the number of recirculated state updates is limited. According to Exp#4, ApproSync recirculates at most 4.5K updates when writing a state with the normal size of 2^{16} values. This indicates that the maximum number of out-of-order state updates is a few thousands during state write. Such a number is small and has a limited impact on application-level accuracy because many applications (e.g., measurement [3], [4] and load balancing [65]) are insensitive to such reordering. According to Exp#5, the application-level accuracy drop is below 0.1% in practice. Thus, we conclude that the out-of-order state updates incurred by ApproSync during state write only have a negligible impact on application-level accuracy.

VI. IMPLEMENTATION

3) Protocol: Figure 7 shows the format of state transfer, which follows the Ethernet header with a reserved protocol

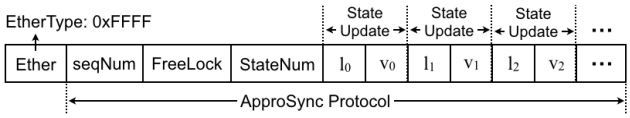


Fig. 7. Protocol format of state transfer.

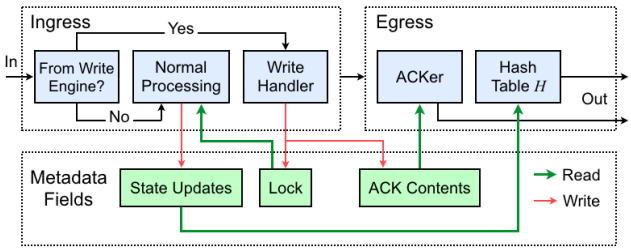


Fig. 8. Workflow of the switch ASIC.

type "0xFFFF". The first field *seqNum* is the sequence number used by the write engine to ensure operation reliability. The second field *FreeLock* indicates whether to release the write lock or not. Moreover, the third field *StateNum* records the total number of state updates appended in the packet header. The remaining fields record state updates, each of which comprises a 16-bit location and a 64-bit state value. Here, 16 bits and 64 bits correspond to the maximum size of location and the maximum size of state value, respectively. Thus, such a design can support arbitrary validate sizes of state updates.

4) Data Plane Handlers: Figure 8 details the implementation of ApproSync handlers. For the read handler, ApproSync records every state update in metadata fields in the ingress pipeline. The updates are sent to the hash table resided in the egress pipeline. We implement the hash table with match-action tables (MATs) and registers. The egress pipeline reserves a port that pushes state updates to the control plane based on the processing results of hash table. Each state update comprises a 16-bit location and a 64-bit state value. As mentioned above, we place the state update between the Ethernet and IP headers. For the write handler, ApproSync employs an MAT at the beginning of the ingress pipeline to identify the type of each received packet. Normal packets are processed by the user program. Otherwise, when the packet indicates a write operation, the write handler is invoked to handle it. We implement both state lock signal and ACK components with metadata fields and registers.

Note that our implementation uses both an ingress pipeline and an egress pipeline rather than a single switch pipeline. The reason is that the ingress pipeline determines which egress port to forward packets based on state values, while the egress pipeline is binding to the port connected to the control plane. Thus, we place the functions that access state values on the ingress pipeline and put the functions that interact with the control plane on the egress pipeline.

5) Control Plane Engines: We implement ApproSync engines in C. Our prototype offers both southbound APIs and northbound APIs. We implement southbound APIs on DPDK [66] to avoid the overhead incurred by kernel stacks. We employ Redis [67] as the state storage and use HiRedis [68] to manage the storage. For northbound APIs, we design four intuitive APIs for applications to manage states via ApproSync.

- READ_ALL_VALUES(SID). This API is used to obtain all the values of the state maintained by the target switch (identified by SID). It returns a complete state.
- WRITE_ALL_VALUES(SID,v). This API rewrites all the state values maintained by the target switch with the specific value v specified by applications. It returns a boolean flag indicating the success of writing.
- READ_VALUE(SID,i). This API returns the state value that locates in the target switch and is indexed by i.
- WRITE_VALUE(SID,i,v). This API rewrites the state value that locates in the target switch and is indexed by i with the specific value v specified by applications. It returns a boolean flag indicating the success of writing.

6) Compiler: We have implemented a compiler to automatically integrate ApproSync into the user program written in $P4_{14}$ or $P4_{16}$. The compiler first inserts the P4 codes that implement ApproSync handlers to the user program. It then augments the user program to connect it with ApproSync handlers: (1) For the read handler, the compiler identifies each state update in the program and records the update in metadata fields, which are delivered to the read handler for further processing; (2) For the write handler, the compiler adds an additional logic that handles state updates via the write handler. Our compiler enables administrators to select registers to be synchronized. By default, it chooses to synchronize all registers.

VII. EVALUATION

We conduct experiments to evaluate ApproSync. We present the average after 100 runs. We highlight our results as follows.

- ApproSync uses less than 15% switch resources, making it compatible with existing stateful applications (Exp#1).
- Compared to the switch OS-based approach, ApproSync reduces the latency by several orders of magnitude when reading the state with a normal size of 2^{16} values (Exp#2).
- Compared to a representative traffic mirroring-based approach, *Flow [13], ApproSync avoids link saturation and state loss via its rate control in state read (Exp#3).
- Even in a link with 80% loss rate, ApproSync writes 2^{16} updates within 10 ms, whereas the OS-based approach spends two orders of magnitude higher latency (Exp#4).
- The state write of ApproSync preserves high accuracy for network management applications (Exp#5).
- ApproSync does not affect throughput. It increases the latency of packet forwarding by less than 5% (Exp#6).
- ApproSync achieves ultra-low (at most 23 ms) state read and write latency for 16 real-world applications (Exp#7).
- We present a case study of ApproSync, in which we build a low-latency sketch collector on the state synchronization offered by ApproSync. The experimental results indicate that the sketch collector outperforms existing approaches with orders-of-magnitude lower collection latency, thus preserving high application-level accuracy (Exp#8-10).

A. Experimental Setup

1) Platforms: We build a testbed comprising two 32×100 Gbps Barefoot Tofino switches [15] and six servers. Each server has 36-core Intel(R) Xeon(R) Gold 6240C CPU

TABLE II

STATEFUL P4 APPLICATIONS USED IN §VII ("SIZE" INDICATES THE SIZE (IN BYTES) OF A STATE UPDATE)

Name	P4 LoC	# of counters	Size
Packet counter (PC) [70]	265	2^{16}	6
Flowlet switching (FL) [70]	251	2^{14}	10
Malicious DNS domain detection (MD) [20]	358	3×2^{16}	4
Snort flowbits (FB) [20]	296	3×2^{16}	4
Affine LB (AL) [20]	345	2^{17}	4
DNS TTL change tracking (TC) [21]	357	3×2^{16}	6
DNS tunnel detection (TD) [21]	530	3×2^{16}	4
Stateful firewall (FW) [22]	349	2^{17}	4
FTP monitoring (FM) [22]	303	2^{16}	4
Heavy hitter detection (HH) [22]	310	2^{17}	6
Super-spreader detection (SS) [22]	313	2^{17}	4
Sampling based on flow size (FS) [22]	560	5×2^{16}	4
SYN flood detection (SF) [6]	313	2^{17}	4
DNS amplification mitigation (AM) [6]	360	2^{16}	4
UDP flood mitigation (UF) [6]	309	2^{17}	4
Elephant flows detection (EF) [6]	553	5×2^{16}	4

TABLE III

(EXP#1) SWITCH RESOURCE USAGE OF APPROSYNC

Type	SRAM	TCAM	mALU	sALU	VLIW	Stage
Only Read	6.77%	0%	14.58%	0%	4.69%	91.6%
Only Write	2.40%	0%	0%	3.65%	3.82%	100%
Overall (Read+Write)	9.17%	0%	14.58%	3.65%	5.21%	100%

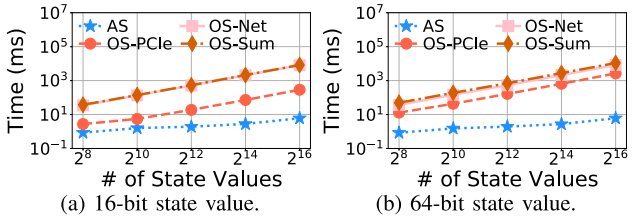
(2.60 GHz), 128GB RAM and a two-port 40-Gbps NIC. We run the control plane on a server based on DPDK [66]. In the data plane, we connect the two switches to compose a linear topologic, while using the remaining five servers as traffic testers. The control plane and traffic testers are directly connected to the two switches via 40-Gbps ports.

2) Workloads: We select a CAIDA 2018 trace [44] with 38M packets, and use PktGen [69] to replay the trace. We select 16 stateful P4 applications that vary in size and complexity to evaluate the performance of state read and write operations (Exp#7). Each application employs a counter array for packet processing, which configurations are shown in Table II.

3) Parameters: By default, the hash table H has 2^{16} entries. We fix the time window w to 1 ms and use a $\beta = 1$. For each application, we obtain the size s of a state update and calculate the maximum emitted rate M as $\frac{c}{s}$, where c equals the capacity of a 40 Gbps link. Unless specified otherwise, ApproSync will adaptively tune t via its rate control.

B. Microbenchmarks

1) (Exp#1) Switch Resource Usage: We validate that ApproSync is hardware-compatible and only consumes a small amount of switch resources. To do this, we measure the total usage of switch resources, including memory resources, computational resources, and match-action stages. More precisely, we focus on (1) the usage of SRAM that supports the data structures of ApproSync handlers, (2) the usage of meter ALUs (mALUs), stateful ALUs (sALUs), and very long instruction words (VLIWs) that realize the operations of ApproSync handlers in the switch ASIC, and (3) the number of switch stages. Note that ApproSync (AS) provides one primitive for state read, and one primitive for state write. We measure their resource consumption individually and the overall consumption of both primitives. Table III presents the switch resource consumption when only ApproSync is deployed. It indicates that ApproSync uses an average of less than 15% resources even when using both the primitives in one programmable switch. Note that the TCAM consumption equals zero because ApproSync handlers do not involve any ternary matches. Moreover, ApproSync uses all the stages since interdependent MATs must be placed in different stages due to switch restrictions. Nevertheless, it uses limited resources and remains sufficient resources in each stage for other logics.



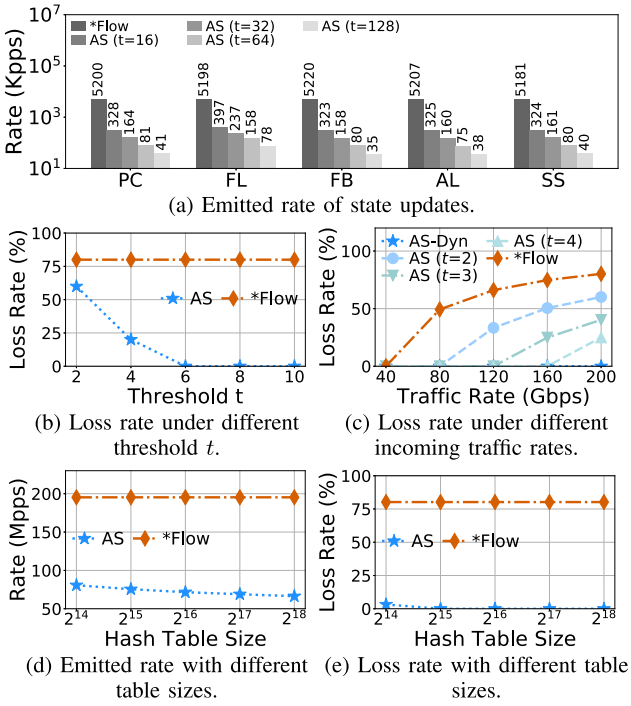
(a) 16-bit state value. (b) 64-bit state value.

Fig. 9. (Exp#2) Performance of state read.

2) (Exp#2) Performance of State Read: We measure the latency of state read. We vary the number of state values from 2^8 to 2^{16}. We employ two types of state: 16-bit state and 64-bit state, where 16-bit is widely used by applications and 64-bit is the largest size supported by our switches. The OS-based approach is built on the interfaces exposed by our switches [15] and ZeroMQ [10]. We measure three types of latency in the OS-based approach: the latency of reading state values from the switch ASIC to the switch OS via PCIe channels (OS-PCIe), the latency of transferring state values via TCP connections (OS-Net), and the overall latency of state write (OS-Sum). Figure 9 shows that the latency of the OS-based approach (OS-Sum) exceeds 1s when a state has 2^{16} values. When using 64-bit state, even reading state via PCIe channels takes tens of milliseconds. In contrast, ApproSync spends less than 10 ms to collect 2^{16} values, which outperforms the OS-based approach with order-of-magnitude latency reduction.

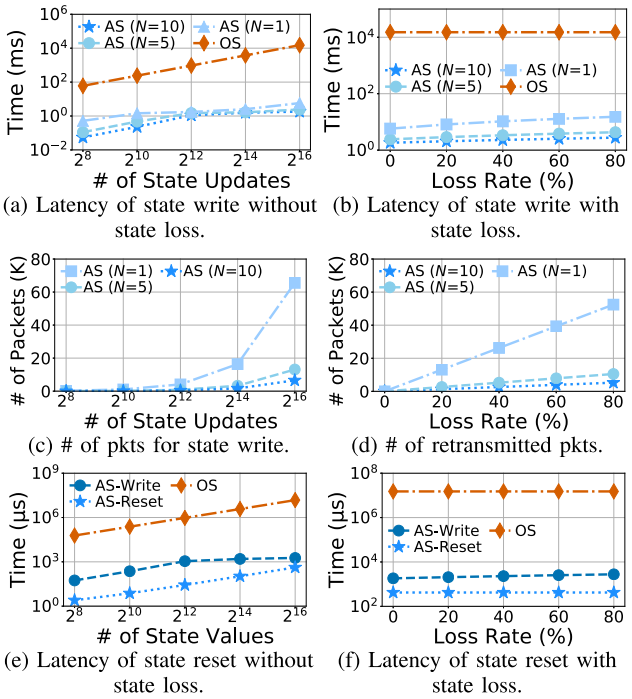
3) (Exp#3) Accuracy of State Read: We evaluate the accuracy of state read. First, we validate that ApproSync avoids link saturation. We replay our trace at 40 Gbps and vary t from 16 to 128. Since the emitted rate of state updates depends on how an application updates state, we select five applications, in which every packet triggers an update, from Table II. We compare ApproSync with *Flow [13], which uses an LRU cache for rate control. We choose *Flow due to two main reasons. (1) *Flow synchronizes its flow records from switches to the control plane, which is the same as the state read of ApproSync. (2) *Flow realizes an in-switch LRU cache. It caches flow records inside the switch to reduce the rate of emitting flow records and decrease link bandwidth consumption, which is similar to the state read of ApproSync. We set the LRU cache in the same way as the hash table of ApproSync. Figure 10(a) shows that *Flow brings limited benefits because its cache is frequently evicted given that the number of flows far exceeds the cache size. Instead, ApproSync achieves low bandwidth consumption (e.g., 35 Kpps for *Snort flowbits* [20]).

Second, we validate that ApproSync can offer accurate state read even with congested networks. More precisely, we deploy the packet counter (PC) that generates a state update for every packet. We inject traffic at 200 Gbps so that the



(a) Emitted rate of state updates.
(b) Loss rate under different threshold t.
(c) Loss rate under different incoming traffic rates.
(d) Emitted rate with different table sizes.
(e) Loss rate with different table sizes.

Fig. 10. (Exp#3) Accuracy of state read.



(a) Latency of state write without state loss.
(b) Latency of state write with state loss.
(c) # of pkts for state write.
(d) # of retransmitted pkts.
(e) Latency of state reset without state loss.
(f) Latency of state reset with state loss.

Fig. 11. (Exp#4) Performance of state write.

emitted rate of state updates far exceeds the available link bandwidth capacity. In this context, directly emitting state updates, which rate becomes 200 Gbps, inevitably congests the 40-Gbps link and incurs significant state loss. In response, ApproSync will increase its threshold t to perform tight rate control. As shown in Figure 10(b), if ApproSync employs fixed and small values of t (e.g., 2), it inevitably suffers from link congestion and significant state loss (e.g., 60% loss rate if $t = 2$). However, ApproSync automatically tunes its t to perform rate control with respect to incoming traffic rate (e.g., increasing t from 2 to 6), which avoids congestion and state loss. We further validate this by gradually increasing the incoming traffic rate from 40 Gbps (a moderate rate that avoids link congestion) to 200 Gbps (an extreme rate that easily causes heavy link congestion). Figure 10(c) indicates that ApproSync (AS-Dyn) avoids congestion and offers loss-free state read since it dynamically tunes t to adapt to incoming traffic rate, which validates our claim. In contrast, ApproSync with fixed t ($t < 5$) and *Flow cannot guarantee no state loss, which emphasizes the importance of rate control.

Third, we study the impact of hash table size on accuracy. We vary the size from 2^{14} to 2^{18} entries. We conduct the same experiment as above. Figure 10(d)-(e) show that ApproSync loses a few state updates (3.22%) when using 2^{14} entries. This is because hash collisions happen frequently given the high traffic rate and relatively small table size. However, ApproSync alleviates this problem via its rate control: when using 2^{15} entries, ApproSync ensures loss-free state read, while *Flow suffers from high state loss.

4) (Exp#4) Performance of State Write: We first measure the performance of state write when writing different values. ApproSync splits a write operation into several sub-operations, each of which is completed by a single packet. Thus, its performance depends on the number N of state updates encapsulated in a packet. We set N to 1, 5, and 10, and vary the number of state updates from 2^8 to 2^{16}. Figure 11(a) shows that ApproSync reduces the latency by orders of magnitude even when $N = 1$ by eliminating the overhead of switch OS.

We next study the robustness of state write in a congested link. In this case, ApproSync needs to retransmit dropped state updates, which increases the latency. Here, we use ApproSync to write 2^{16} state updates and measure its latency when the state loss rate ranges from 0% to 80%. Figure 11(b) presents that even with an 80% loss rate, ApproSync completes state write in a few milliseconds. Also, we measure the bandwidth consumed by state write. Figure 11(c)-(d) present the number of packets for a write operation and that for retransmission under different loss rates. Even in the worst case, the number of generated packets is at most 65K, which is far below link capacity (e.g., 14.88 Mpps of a 10 Gbps link).

We further evaluate the performance of state reset. We use the same settings as mentioned above and compare the state reset algorithm (AS-Reset) with the state write algorithm of ApproSync (AS-Write, $N = 10$) and the OS-based approach. Figure 11(e)-(f) indicate that the state reset algorithm offered by ApproSync achieves significantly lower latency than the comparison methods. The reason is that ApproSync only uses a small number of packets to complete state reset (see §V-B), which brings remarkable latency benefits.

5) (Exp#5) Accuracy of State Write: We measure the accuracy of ApproSync in state write. We first count the number of out-of-order updates during state write. We deploy HashPipe [16], a heavy hitter detection algorithm, with 5K counters on a switch. The accuracy of HashPipe is associated with every state update so that it can accurately reflect the impact of state write. As shown in Figure 12(a), ApproSync affects at most 4.5K updates. Even in the worst case, it only consumes 300 Kpps bandwidth, which is far below Mpps-level switch bandwidth. This is because its state write is low-latency, so the number of affected updates is small. Next, we study

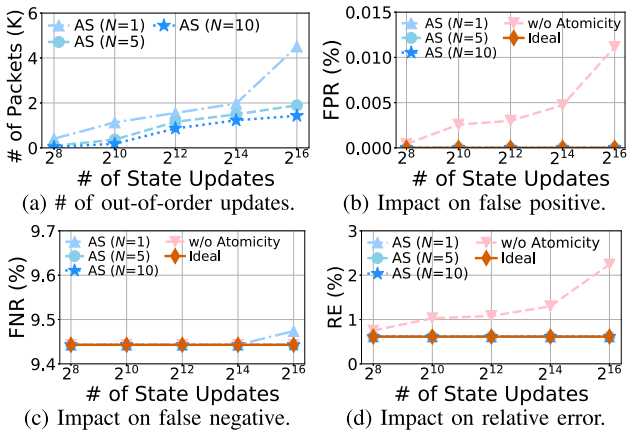


Fig. 12. (Exp#5) Accuracy of state write.

TABLE IV
(EXP#6) IMPACT ON PACKET FORWARDING

Name	Thpt.	Latency	Thpt. cost	Latency cost
NoApproSync	40 Gbps	1073 ns	-	-
Only Read	40 Gbps	1123 ns	-0.0%	+4.6%
Only Write	40 Gbps	1101 ns	-0.0%	+2.6%
Overall	40 Gbps	1141 ns	-0.0%	+6.3%

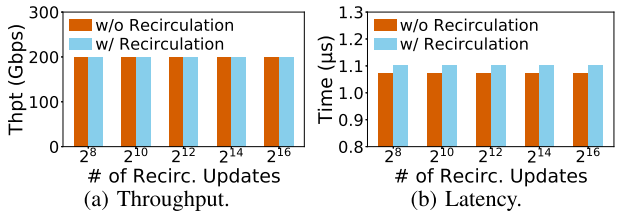


Fig. 13. (Exp#6) Impact of recirculation.

the impact on HashPipe accuracy. Figure 12(b)-(d) show that: (1) the out-of-order updates only increase false negative rate (FNR) and relative error (RE) by at most 0.2% and 0.03%; (2) compared to the OS-based approach, ApproSync offers accurate state write with atomicity guarantees.

6) (Exp#6) Impact on Packet Forwarding: Table IV examines how ApproSync affects throughput and per-packet latency. We consider four cases. (1) "NoApproSync" disables ApproSync and presents the original performance. (2) "Only read" presents the results when only the read handler is activated. (3) "Only write" shows the results of state write. (4) "Overall" enables full functionalities. We observe that ApproSync incurs zero throughput drop while adding the per-packet processing latency by at most 6.3%.

We also quantify the overheads of recirculating state updates on normal packet processing performance. Specifically, we repeat Exp#5 and inject 200-Gbps traffic to the switch. We set the write handler to recirculate a fixed number of state updates varying from 2^8 to 2^{16}. We quantify the overheads incurred by recirculation by measuring the switch throughput and per-packet processing latency. In Figure 13, the recirculation does not affect throughput and incurs an average latency overhead of 30 ns, which is negligible.

7) (Exp#7) Latency for Stateful P4 Applications: We measure the application-perceived latency of state read and write of ApproSync. We implement the 16 applications in Table II, and measure the latency of reading states from the counters used by

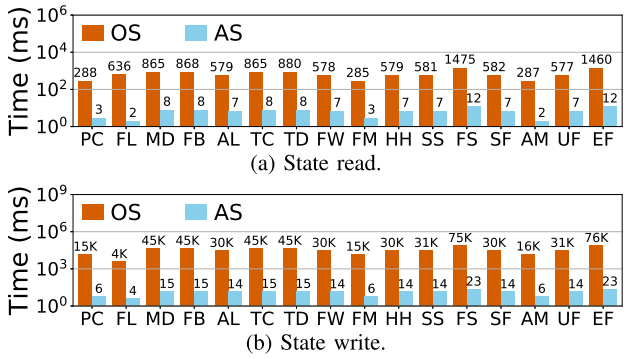


Fig. 14. (Exp#7) Application-perceived latency.

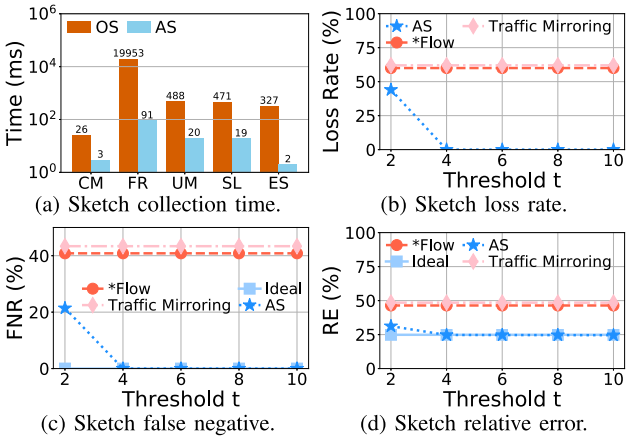


Fig. 15. (Exp#8) Microbenchmarks of sketch collector.

applications and resetting counters. We compare ApproSync with the OS-based approach in Figure 14. We observe that ApproSync completes state read within 12 ms, while the OS-based approach takes at least 285 ms. Also, ApproSync requires at most 23 ms for state write, while the OS-based approach requires several seconds.

C. Case Study: Fast and Accurate Sketch Collector

1) (Exp#8) Microbenchmarks: Sketches are widely-used algorithms in network measurement [3], [4], [17], [18], [19], [37]. A sketch algorithm maintains a compact counter array in the switch ASIC to measure flow statistics. At the end of a time interval, applications collect the counter array to obtain statistics. However, existing approaches collect counter values via the switch OS, which incurs high latency and hurts the timeliness of network management. To this end, we build a sketch collector on the northbound APIs offered by ApproSync. In detail, ApproSync executes the following two steps to support the sketch collector. (1) When a packet updates counter values, the read handler inserts those updates to its hash table; (2) When the state divergence exceeds the threshold, it pushes the latest sketch values to the control plane, where applications make corresponding decisions.

We first evaluate the performance of the sketch collector. We implement five sketch algorithms, Count-Min sketch (CM) [55], FlowRadar (FR) [4], UnivMon (UM) [3], SketchLearn (SL) [18] and ElasticSketch (ES) [19]. We set these sketch algorithms with the following realistic settings [18], [19]. (1) Count-Min sketch uses three hash functions and 2^{16} 4-byte counters. (2) FlowRadar uses three

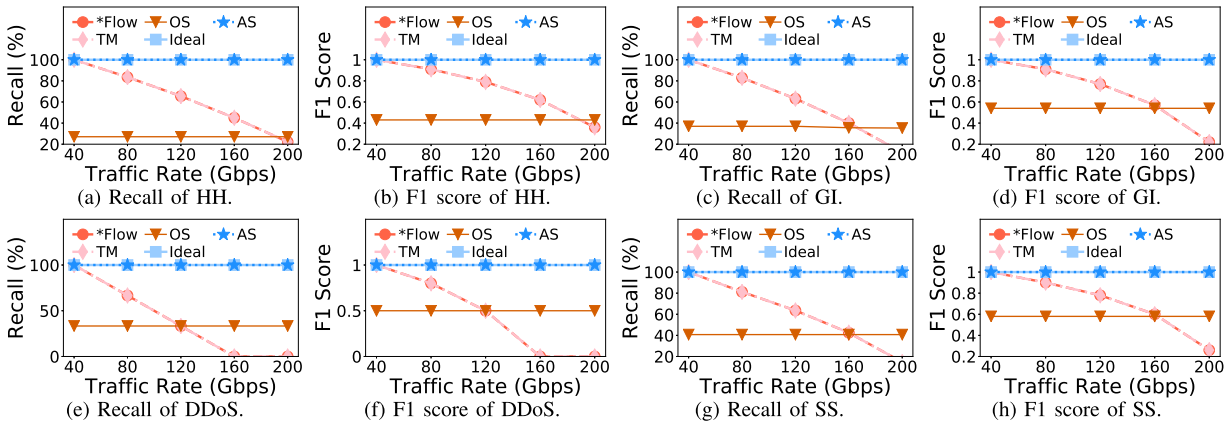


Fig. 16. (Exp#9) Impact on application-level accuracy (higher values are better).

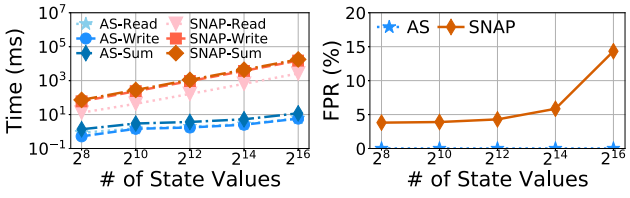
hash functions in both the Bloom filter [71] and the invertible Bloom lookup table (IBLT) [72]. It uses 1 MB memory in total, and allocates $\frac{1}{10}$ of the memory for the Bloom filter and the rest for the IBLT part. (3) Both UnivMon and SketchLearn use a 32-level sketch, where each level uses one hash function and 2^{15} 4-byte counters. (5) ElasticSketch has a heavy part of 2^{12} entries and a light part of 2^{19} entries, and the total memory usage is 0.69 MB. We deploy these algorithms on a switch, and use the sketch collector to collect counter values from the switch. Figure 15(a) indicates that compared to the OS-based approach, ApproSync reduces the collection time by orders of magnitude. Next, we inject traffic at 120 Gbps to examine the loss rate of state updates. Figure 15(b) presents the loss rate with respect to the threshold t that determines the emitted rate of state updates. We compare ApproSync with traffic mirroring and *Flow. We observe that traffic mirroring and *Flow loss around 60% state updates due to the lack of a reasonable rate control. ApproSync also suffers from a high loss rate when $t = 2$. However, the state loss is completely eliminated as the threshold increases.

Also, we qualify the accuracy of heavy hitter detection, whose settings are the same as that in §II-B. We present the results of Count-Min sketch. Since the results of other sketches are similar to that of Count-Min, we elide them here. We also build an original version of Count-Min without state loss ("Ideal") as the baseline. Figure 15(c)-(d) presents false negative rate and relative error, respectively. We observe that state loss in traffic mirroring and *Flow seriously improves false negative rate and relative error. Instead, ApproSync achieves near-optimal accuracy.

2) (Exp#9) Impact on Application-Level Accuracy: We further demonstrate the benefits of ApproSync in practice by considering the impact of the low-latency sketch collector on the accuracy of four common applications in network measurement, including heavy hitter detection (HH) [17], global iceberg detection (GI) [3], DDoS flow detection (DDoS) [73], and superspreader detection (SS) [73]. These applications run on the control plane and periodically acquire values from the Count-Min sketch running on a programmable switch. We configure the Count-Min sketch with 10 MB, which is close to the total memory capacity of a single switch. We setup applications with practical settings [3], [17]. More precisely, for HH and GI, we set their thresholds to 10 K packets. For DDoS and SS, We set the thresholds of the two applications to 0.5% of the total number of five-tuples and that of IP addresses, respectively. We set the collection window to 10 ms [74], i.e., each application collects all the sketch values every 10 ms. Moreover, we inject the workload to the sketch at different rates varying from 40 Gbps to 200 Gbps. Meanwhile, we use the sketch collector built on ApproSync, the OS-based approach, traffic mirroring (TM), and *Flow, which details are given in the above experiments, to collect sketch values, respectively. Next, we input the sketch values into applications and obtain their measurement results. By comparing the measurement results with the true results exhibited by the workload, we compute the application-level accuracy in terms of recall and F1 score. We use the same method as Exp#8 to calculate the ideal results as the ground truth.

As shown in Figure 16, since ApproSync preserves both low latency and high accuracy in state synchronization, it enables the sketch collector to retain high application-level accuracy. In contrast, other approaches either suffer from high OS-level latency overheads or fail to avoid state loss during synchronization, leading to a non-trivial accuracy drop. Specifically, for the OS-based approach, since its collection takes a long time to complete (see Figure 15(a)), it fails to collect all the sketch values within a collection window. Thus, it cannot keep up with high-speed sketch updates in the data plane and inevitably misses the latest updates of sketch values. As a result, the state divergence is significantly high in all cases, resulting in poor application-level accuracy. Moreover, for traffic mirroring and *Flow, when the incoming traffic rate increases, their state updates inevitably exhaust the limited capacity of link bandwidth. Thus, massive sketch values are dropped, reducing application-level accuracy.

3) (Exp#10) Timely UDP Flood Prevention: We further demonstrate the usage of sketch collector in the scenario of network security. More precisely, we consider the UDP flood attack, which is a common type of DDoS attacks. In such attacks, attackers randomly generate numerous UDP packets that target random UDP ports of innocent hosts. Thus, attackers can easily exhaust the available resources in victims. To defend against UDP flood attacks, existing approaches exploit per-flow counters resided in switches to detect such attacks [6], [20]. They rely on the OS-based approach to transfer counter values to the control plane for attack detection and periodically reset counters to mitigate detection errors. However, the OS-based approach incurs non-trivial latency that delays attack detection. Also, due to the lack of atomicity,



(a) Detection Time. (b) Detection accuracy.

Fig. 17. (Exp#10) Timely UDP flood prevention.

the reset operations will be affected by data plane packets, leading to false alarms.

To this end, we propose to leverage the sketch collector built on ApproSync in UDP flood prevention. We build a network management application, namely UDP flood prevention (UFP), on the sketch collector. UFP employs FlowRadar [4] on the switch ASIC to record flow statistics. At runtime, UFP invokes the sketch collector to collect the values of FlowRadar counters and detect attacks by analyzing collected values. It also periodically resets FlowRadar counters with atomicity guarantees offered by the state write of ApproSync.

We evaluate the performance of UFP. We vary the number of state values from 2^8 to 2^{16}, and measure the latency of collecting and resetting these values ("AS-Read" and "AS-Write") via UFP. We also quantify the accuracy of UFP by measuring its false positive rate. We set the detection threshold to 10^5, and compare UFP with the SNAP method [20] based on the OS-based approach. Figure 17 shows that UFP achieves orders-of-magnitude latency reduction compared to SNAP. Moreover, with atomicity guarantees, it achieves zero false positive rate, while SNAP suffers from a few errors.

VIII. Related Work

A. Bottom-up State Read

Prior solutions [12], [48], [49], [50], [51], [52] exploit packet sampling to reduce bandwidth consumption. However, sampling techniques inevitably degrade accuracy. TurboFlow [36] processes state values in the switch OS. However, a switch OS fails to process multi-Tbps state updates and incurs high latency. Instead, ApproSync bypasses the switch OS to mitigate performance overhead. Moreover, Marple [14] caches flow records in the switch ASIC before mirroring states to remote servers. However, states are frequently evicted, which saturates link capacity (see Exp#3 in §VII). ApproSync improves in-switch caching by adaptively controlling the trade-off between accuracy and bandwidth consumption. KeySight [64] aggregates packets based on packet processing behaviors to reduce overhead. MAFIA [34] proposes intent-based primitives to ease the deployment of measurement tasks on programmable switches. Sonata [33] pre-processes packets in switches to reduce workloads in the control plane. OmniMon [5] coordinates different types of network elements to jointly perform measurement tasks. ApproSync is complementary to these solutions.

In addition, MTP [74] aims to avoid control plane overload through its measurement task placement. Compared to MTP, ApproSync exhibits two benefits. First, MTP only targets network measurement, while ApproSync can be activated in arbitrary scenarios. Second, the placement made by MTP may be affected by dynamic topology changes, while ApproSync is agnostic to such changes. NetSeer [75] shares a similar idea to ApproSync, i.e., deduplicating and batching data within data plane switches to reduce the load of data transfers. However, its focus is to detect network performance anomalies such as packet drops, making it different from the state synchronization offered by ApproSync.

B. Top-Down State Write

Current network controllers modify states via TCP-based protocols such as OpenFlow [7], Thrift [8] and gRPC [9], which require a switch OS for complicated processing. Applications built on these protocols (e.g., P4NFV [29]) suffer from high latency. Swing State [28] directly migrates state values in the data plane to achieve rapid state migration. P4Sync [76] offers strong authenticity guarantees when migrating state values among switches. P4State [77] takes a step further to only migrate essential state values to reduce migration overhead. However, the write operations issued by these solutions could be lost. Instead, ApproSync provides low-latency and loss-free write. Moreover, RedPlane [78] offers inter-switch mechanisms to migrate state values among different switches to achieve fault tolerance in the data plane. ApproSync aims to achieve the inter-plane consistency via synchronizing state values between the control plane and data plane, making it complementary to RedPlane.

C. Approximate Systems

Approximation is a well-studied topic in distributed systems. AF-Stream [79] provides approximate fault tolerance in the content of stream processing. Sketchlearn [18] integrates the statistical properties of resource conflicts in sketches to improve the overall accuracy. HashPipe [16] proposes algorithms that approximate the common space-saving method to track heavy hitters in the data plane. In contrast, ApproSync exploits approximate techniques to achieve low-latency and accurate state synchronization.

IX. Conclusion

We propose ApproSync, a framework that synchronizes states between the data plane and control plane. ApproSync utilizes approximate techniques to reduce resource consumption and bound errors of state synchronization. We implement ApproSync on Tofino switches. Our experiments indicate that ApproSync achieves order-of-magnitude latency reduction and preserves high application-level accuracy.

In the near future, we plan to evaluate ApproSync with the distributed applications that collaborate multiple switches to demonstrate its effectiveness in network-wide scenarios.

References

[1] X. Chen, Q. Huang, D. Zhang, H. Zhou, and C. Wu, "ApproSync: Approximate state synchronization for programmable networks," in *Proc. IEEE 28th Int. Conf. Netw. Protocols (ICNP)*, Oct. 2020, pp. 1–12.

[2] P. Bosshart et al., "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[3] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 101–114.

[4] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A better netflow for data centers," in *Proc. USENIX NSDI*, 2016, pp. 311–324.

[5] Q. Huang, H. Sun, P. P. C. Lee, W. Bai, F. Zhu, and Y. Bao, "OmniMon: Re-architecting network telemetry with resource efficiency and full accuracy," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protocols Comput. Commun.*, Jul. 2020, pp. 404–421.

[6] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and elastic DDoS defense," in *Proc. USENIX Secur.*, 2015, pp. 817–832.

[7] N. McKeown et al., "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[8] *Apache Software Foundation, Thrift*. Accessed: Oct. 24, 2022. [Online]. Available: http://thrift.apache.org/

[9] *GRPC*. Accessed: Oct. 24, 2022. [Online]. Available: https://www.grpc.io/

[10] *Zeromq*. Accessed: Oct. 24, 2022. [Online]. Available: http://zeromq.org/

[11] J. C. Mogul and P. Congdon, "Hey, you darned counters!: Get off my ASIC!" in *Proc. 1st Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2012, pp. 25–30.

[12] J. Rasley et al., "Planck: Millisecond-scale monitoring and control for commodity networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 407–418, 2014.

[13] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, "Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow," in *Proc. USENIX ATC*, 2018, pp. 823–835.

[14] S. Narayana et al., "Language-directed hardware design for network performance monitoring," in *Proc. ACM SIGCOMM*, 2017, pp. 85–98.

[15] *Tofino*. Accessed: Oct. 24, 2022. [Online]. Available: https://www.barefootnetworks.com/technology/#tofino

[16] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proc. Symp. SDN Res.*, Apr. 2017, pp. 164–176.

[17] Q. Huang et al., "SketchVisor: Robust network measurement for software packet processing," in *Proc. ACM SIGCOMM*, 2017, pp. 113–126.

[18] Q. Huang, P. P. C. Lee, and Y. Bao, "SketchLearn: Relieving user burdens in approximate measurement with automated statistical inference," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 576–590.

[19] T. Yang et al., "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. ACM SIGCOMM*, 2018, pp. 561–575.

[20] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "SNAP: Stateful network-wide abstractions for packet processing," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 29–43.

[21] K. Borders et al., "Chimera: A declarative language for streaming network traffic analysis," in *Proc. USENIX Secur.*, 2012, pp. 365–379.

[22] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for SDN," in *Proc. ACM Conf. SIGCOMM*, Aug. 2014, pp. 61–66.

[23] X. Chen, D. Zhang, X. Wang, K. Zhu, and H. Zhou, "P4SC: Towards high-performance service function chain implementation on the P4-capable device," in *Proc. IFIP/IEEE IM*, Dec. 2019, pp. 1–9.

[24] S. Pontarelli et al., "FlowBlaze: Stateful packet processing in hardware," in *Proc. USENIX NSDI*, 2019, pp. 531–548.

[25] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proc. ACM SIGCOMM Conf.*, 2011, pp. 350–361.

[26] J. Sherry et al., "Rollback-recovery for middleboxes," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 227–240, 2015.

[27] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or die: High-availability design principles drawn from Googles network infrastructure," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 58–72.

[28] S. Luo, H. Yu, and L. Vanbever, "Swing state: Consistent updates for stateful and programmable data planes," in *Proc. Symp. SDN Res.*, Apr. 2017, pp. 115–121.

[29] M. He et al., "P4NFV: An NFV architecture with flexible data plane reconfiguration," in *Proc. CNSM*, 2018, pp. 90–98.

[30] A. Singh et al., "Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 183–197, 2015.

[31] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian, "DRILL: Micro load balancing for low-latency data center networks," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 225–238.

[32] M. G. Gouda and A. X. Liu, "A model of stateful firewalls and its properties," in *Proc. Int. Conf. Dependable Syst. Netw. (DSN)*, 2005, pp. 128–137.

[33] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 357–371.

[34] P. Laffranchini, L. Rodrigues, M. Canini, and B. Krishnamurthy, "Measurements as first-class artifacts," in *Proc. IEEE INFOCOM*, Apr. 2019, pp. 415–423.

[35] S. Wang et al., "Martini: Bridging the gap between network measurement and control using switching ASICs," in *Proc. IEEE ICNP*, Oct. 2020, pp. 1–12.

[36] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "TurboFlow: Information rich flow record generation on commodity switches," in *Proc. 13th EuroSys Conf.*, Apr. 2018, p. 11.

[37] L. Tang, Q. Huang, and P. P. C. Lee, "MV-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams," in *Proc. IEEE INFOCOM*, Apr. 2019, pp. 2026–2034.

[38] Q. Huang et al., "Toward nearly-zero-error sketching via compressive sensing," in *Proc. USENIX NSDI*, 2021, pp. 1027–1044.

[39] R. Joshi, T. Qu, M. C. Chan, B. Leong, and B. T. Loo, "BurstRadar: Practical real-time microburst monitoring for datacenter networks," in *Proc. 9th Asia–Pacific Workshop Syst.*, Aug. 2018, pp. 1–8.

[40] D. Shan, F. Ren, P. Cheng, R. Shu, and C. Guo, "Micro-burst in data centers: Observations, analysis, and mitigations," in *Proc. IEEE 26th Int. Conf. Netw. Protocols (ICNP)*, Sep. 2018, pp. 88–98.

[41] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, and O. Rottenstreich, "Catching the microburst culprits with snappy," in *Proc. Afternoon Workshop Self-Driving Netw.*, Aug. 2018, pp. 22–28.

[42] Y. Zhu et al., "Packet-level telemetry in large datacenter networks," in *Proc. ACM SIGCOMM Comput. Commun. Rev.*, 2015, vol. 45, no. 4, pp. 479–491.

[43] O. Tilmans, T. Bühler, S. Vissicchio, and L. Vanbever, "Mille-Feuille: Putting ISP traffic under the scalpel," in *Proc. 15th ACM Workshop Hot Topics Netw.*, Nov. 2016, pp. 113–119.

[44] *Caida Traces*. Accessed: Oct. 24, 2022. [Online]. Available: http://www.caida.org/data/overview/

[45] M. Allman and V. Paxson, "On estimating end-to-end network path properties," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 263–274, 1999.

[46] P. Sarolahti, M. Kojo, and K. Raatikainen, "F-RTO: An enhanced recovery algorithm for TCP retransmission timeouts," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 2, pp. 51–63, 2003.

[47] P. Bosshart et al., "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 99–110, 2013.

[48] *Sflow*. Accessed: Oct. 24, 2022. [Online]. Available: http://sflow.org/about/index.php

[49] V. Sekar, M. K. Reiter, and H. Zhang, "Revisiting the case for a minimalist approach for network flow monitoring," in *Proc. 10th Annu. Conf. Internet Meas.*, 2010, pp. 328–341.

[50] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," *Comput. Commun. Rev.*, vol. 41, no. 4, pp. 254–265, Aug. 2011.

[51] N. Handigol et al., "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proc. USENIX NSDI*, 2014, pp. 71–85.

[52] C. Zhang et al., "P4DB: On-the-fly debugging of the programmable data plane," in *Proc. IEEE 25th Int. Conf. Netw. Protocols (ICNP)*, Oct. 2017, pp. 1–10.

[53] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proc. 9th ACM SIGCOMM Conf. Internet Meas. Conf.*, 2009, pp. 202–208.

[54] A. Kuzmanovic and E. W. Knightly, "Low-rate TCP-targeted denial of service attacks: The shrew vs. the mice and elephants," in *Proc. ACM SIGCOMM*, 2003, pp. 75–86.

[55] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[56] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the abyss: An evaluation of concurrency control with one thousand cores," *Proc. VLDB Endowment*, vol. 8, no. 3, pp. 209–220, Nov. 2014.

[57] R. Harding et al., "An evaluation of distributed concurrency control," *Proc. VLDB Endowment*, vol. 10, no. 5, pp. 553–564, 2017.

[58] C. Barthels, I. Müller, K. Taranov, G. Alonso, and T. Hoefler, "Strong consistency is not hard to get: Two-phase locking and two-phase commit on thousands of cores," *Proc. VLDB Endowment*, vol. 12, no. 13, pp. 2325–2338, Sep. 2019.

[59] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surveys*, vol. 13, no. 2, pp. 185–221, 1981.

[60] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, Jun. 1981.

[61] D. Skeen, "Nonblocking commit protocols," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1981, pp. 133–142.

[62] D. Skeen, "A quorum-based commit protocol," Cornell Univ., Ithaca, NY, USA, Tech. Rep. 1982-02, 1982.

[63] K. Ren, A. Thomson, and D. J. Abadi, "An evaluation of the advantages and disadvantages of deterministic database systems," *Proc. VLDB Endowment*, vol. 7, no. 10, pp. 821–832, Jun. 2014.

[64] Y. Zhou et al., "KeySight: Troubleshooting programmable switches via scalable high-coverage behavior tracking," in *Proc. IEEE 26th Int. Conf. Netw. Protocols (ICNP)*, Sep. 2018, pp. 291–301.
[65] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making stateful Layer-4 load balancing fast and cheap using switching ASICs," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 15–28.
[66] *Intel. Data Plane Development Kit*. Accessed: Oct. 24, 2022. [Online]. Available: http://dpdk.org
[67] *Redis*. Accessed: Oct. 24, 2022. [Online]. Available: http://redis.io/
[68] *Hiredis*. Accessed: Oct. 24, 2022. [Online]. Available: http://redis.io/
[69] *Pktgen*. Accessed: Oct. 24, 2022. [Online]. Available: https://pktgen-dpdk.readthedocs.io/
[70] *P4 Tutorials*. Accessed: Oct. 24, 2022. [Online]. Available: https://github.com/p4lang/tutorials
[71] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.
[72] M. T. Goodrich and M. Mitzenmacher, "Invertible Bloom lookup tables," in *Proc. 49th Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, Sep. 2011, pp. 792–799.
[73] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with OpenSketch," in *Proc. USENIX NSDI*, 2013, pp. 29–42.
[74] X. Chen et al., "MTP: Avoiding control plane overload with measurement task placement," in *Proc. IEEE INFOCOM*, May 2021, pp. 1–10.
[75] Y. Zhou et al., "Flow event telemetry on programmable data plane," in *Proc. ACM SIGCOMM*, Jul. 2020, pp. 76–89.
[76] J. Xing, A. Chen, and T. S. E. Ng, "Secure state migration in the data plane," in *Proc. Workshop Secure Program. Netw. Infrastructure*, Aug. 2020, pp. 28–34.
[77] M. He, A. Blenk, W. Kellerer, and S. Schmid, "Toward consistent state management of adaptive programmable networks based on P4," in *Proc. ACM SIGCOMM Workshop Netw. Emerg. Appl. Technol.*, 2019, pp. 29–35.
[78] D. Kim, J. Nelson, D. R. K. Ports, V. Sekar, and S. Seshan, "RedPlane: Enabling fault-tolerant stateful in-switch applications," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 223–244.
[79] Q. Huang and P. P. C. Lee, "Toward high-performance distributed stream processing via approximate fault tolerance," *Proc. VLDB Endowment*, vol. 10, no. 3, pp. 73–84, Nov. 2016.

Xiang Chen (Member, IEEE) received the B.Eng. and M.Eng. degrees from Fuzhou University in 2019 and 2022, respectively. He is currently pursuing the Ph.D. degree with the College of Computer Science and Technology, Zhejiang University, China. He has published papers in IEEE INFOCOM, IEEE ICNP, IEEE ICDCS, and IEEE/ACM TRANSACTIONS ON NETWORKING. His research interests include programmable networks and network security. He received the Best Paper Award from IEEE/ACM IWQoS 2021, and the Best Paper Candidate from IEEE INFOCOM 2021.

Hongyan Liu received the B.Eng. degree from the College of Mathematics and Computer Science, Fuzhou University. He is currently pursuing the M.Eng. degree with the College of Computer Science, Zhejiang University. His current research interests include network measurement and programmable networks.

Qun Huang (Member, IEEE) received the bachelor's degree in computer science from Peking University in 2011 and the Ph.D. degree from The Chinese University of Hong Kong in August 2015. He is currently an Assistant Professor at the Department of Computer Science and Technology, Peking University (PKU). Before joining PKU, he worked at the Institute of Computing Technology, Chinese Academy of Sciences (ICT-CAS), from September 2017 to May 2020, and at Huawei from September 2015 to September 2017.

Dong Zhang (Member, IEEE) received the B.S. and Ph.D. degrees from Zhejiang University, China, in 2005 and 2010, respectively. He visited Alabama University, USA, as a Visiting Scholar, from 2018 to 2019. He is currently a Professor with the College of Computer Science and Big Data, Fuzhou University, China. His research interests include software-defined networking, network virtualization, and internet QoS.

Haifeng Zhou received the Ph.D. degree in computer science and technology from Zhejiang University in 2018. He is currently an Associate Research Fellow with the College of Control Science and Engineering, Zhejiang University. His current research interests include P4 and security, software-defined network and security, cloud security, industrial internet and security, intelligent networks and security systems, and innovative network and security technologies.

Chunming Wu received the Ph.D. degree in computer science from Zhejiang University in 1995. He is currently a Professor with the College of Computer Science and Technology, Zhejiang University. He is also the Associate Director of the Research Institute of Computer System Architecture and Network Security, Zhejiang University, and the Director of the NGNT Laboratory. His research interests include software-defined networking, reconfigurable networks, proactive network defense, network security, network virtualization, the architecture of next-generation internet, and intelligent networks.

Xuan Liu (Senior Member, IEEE) received the Ph.D. degree in computer science and engineering from Southeast University, China. He is currently a Lecturer and a Master's Supervisor with the College of Information Engineering (College of Artificial Intelligence), Yangzhou University, China, and on-the-job Post-Doctoral Researcher with the School of Computer Science and Engineering, Southeast University. His main research interests include future content networking and security (future internet architecture, DAN, ICN, CDN, SDN, VLC, and V2R). Furthermore, he served(s) as a TPC Member for ACM MobiCom, IEEE INFOCOM, IEEE ICC, IEEE GLOBECOM, IEEE WCNC, IFIP/IEEE IM, IEEE NOMS, IEEE PIMRC, IEEE MSN, IEEE VTC, IEEE ICIN, IEEE GIIS, IEEE DASC, APNOMS, and CollaborateCom. He is serving as an Editorial Board Member for *Computer Communications*, *TELS*, *IET Networks*, and *IET Smart Cities*.

Qiang Yang (Senior Member, IEEE) received the Ph.D. degree in electronic engineering and computer science from the Queen Mary University of London, London, U.K., in 2007. He worked with the Department of Electrical and Electronic Engineering, Imperial College London, U.K., from 2007 to 2010. He visited The University of British Columbia and the University of Victoria, Canada, as a Visiting Scholar, in 2015 and 2016. He is currently a Full Professor at the College of Electrical Engineering, Zhejiang University, China. He has published more than 200 technical articles, applied 60 national patents, coauthored two books, edited two books, and several book chapters. His research interests include smart energy systems, large-scale complex network modeling, control and optimization, and learning-based optimization and control. He is a Fellow of the British Computer Society (BCS), a Senior Member of the IET, and a Senior Member of the China Computer Federation (CCF).