ApproSync: Approximate State Synchronization for Programmable Networks

Xiang Chen*[†‡], Qun Huang*, Dong Zhang[‡], Haifeng Zhou[§††], Chunming Wu[§††]
*Peking University [†]Pengcheng Lab [‡]Fuzhou University [§]Zhejiang University [††]Zhejiang Lab

***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, as 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.**

I. Introduction

Recent advances in programmable networks empower network administrators to customize the packet processing behaviors of programmable switches. For example, with the P4 language [1], administrators are able to implement new network protocols and functionalities on programmable switches. Programmable switches expose a collection of stateful memory (e.g., registers) to store the *state* of packet processing. 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 real-time traffic monitoring [2, 3, 4].

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 to synchronize states within ultra-low latency to meet the requirements raised by latency-sensitive applications. For example, UDP flood mitigation [5] needs to collect thousands of state values from switches within a few microseconds so as to rapidly detect attacks. Also, state 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) installed on every switch. The switch OS manipulates state values in its underlying switch ASIC via PCIe channels, and connects to the control plane via TCP-based protocols [6, 7, 8, 9]. Both PCIe channels and TCP connections are the bottlenecks to synchronize state updates incurred by high-speed traffic [10, 11]. 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, some traffic mirroring approaches [11, 12, 13] bypass the switch OS and directly transfer state updates from the switch ASICs to the control plane. However, without reasonable rate control, these approaches suffer from serious state loss when the traffic rate exceeds link capacity.

In this paper, 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 incorporates approximate strategies to achieve high accuracy. The notion behind this is that many applications tolerate a small divergence between the state in the data plane and that in the control plane, i.e., *state divergence*. 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 accuracy drop incurred by the state divergence.

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 incoming traffic rate. (1) When 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 and massive state updates need to be synchronized in a short time, it *selectively* pushes state updates to avoid link overload. This makes the state in the control plane diverges from that in the data plane. Nevertheless, the state divergence is bounded by the threshold to keep high accuracy. For state

Qun Huang is the corresponding author.

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

write, ApproSync acknowledges each state write operation. If the ACK of an operation is not received after a timeout, ApproSync retries the operation to avoid state loss. It also ensures the atomicity of state write by preventing all the 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 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 a prototype of ApproSync with Barefoot Tofino switches [14]. We evaluate ApproSync with 16 stateful P4 applications. Our results indicate that ApproSync achieves order-of-magnitude latency reduction against existing approaches while maintaining high accuracy.

II. Motivation

A. *Problem*

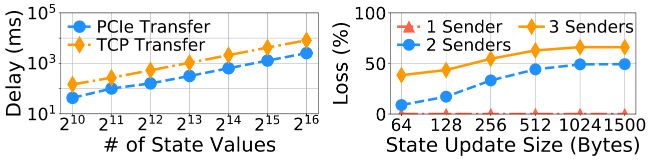
This paper targets state synchronization for programmable networks (e.g., data center networks [15, 3, 16]), where the data plane and control plane are separated. In the data plane, each programmable switch maintains a collection of values referred as *state* and continuously updates its state during packet processing. The control plane holds a copy of the state of each switch. Applications make decisions based on states and modify states to perform control actions. For instance, stateful firewall [17] collects state values from switches to detect attacks. It also updates the detection thresholds recorded in switches with respect to traffic dynamics. Such distributed processing motivates the state synchronization that keeps the states in both planes consistent. In the bottom-up direction, state updates incurred by data plane packets should be synchronized to the control plane. In the top-down direction, state modifications in the control plane should be reflected in the data plane. In particular, applications require state synchronization to achieve *low latency* and *high accuracy*.

- **Low latency**. We aim to reduce the latency of state synchronization in both directions (i.e., from data plane to 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 to rapidly detect and react to suspect events [18, 19, 20, 21].
- **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*

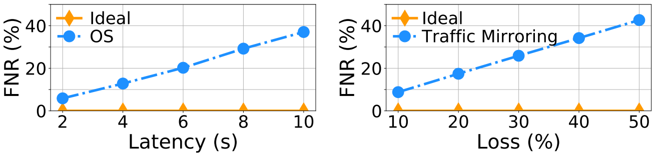
Existing approaches synchronize states via either the switch OS or traffic mirroring. However, none of these approaches can achieve both low latency and high accuracy.

High latency in OS-based approach. The OS-based approach utilizes the switch OS to transfer the state between the switch ASIC and the control plane. For the switch ASIC, the OS manipulates the state via PCIe channels. For the control plane, the OS establishes TCP connections [6, 7, 8, 9] to transfer the state. However, the OS incurs high latency in two aspects. First, due to limited bandwidth, PCIe channels could be the performance bottleneck [11]. Second, 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 [14]. We observe that both PCIe transfer and TCP-based transfer incur a latency of hundreds of milliseconds, e.g., synchronizing 2^{16} state values takes even several seconds.



(a) High latency in switch OS. (b) State loss of traffic mirroring.

Fig. 1: Benchmark of existing state transfers.

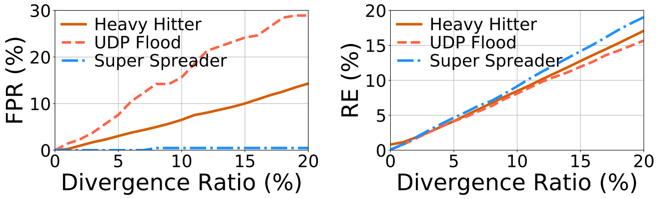


(a) High latency. (b) High loss rate.

Fig. 2: Impact on applications.

State loss in traffic mirroring. 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 [11, 22, 23, 12]. However, these approaches suffer from serious state loss when the emitted rate of state updates exceeds link capacity [11]. 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.

Impact on applications. We study the impact of existing approaches by testbed experiments. Our testbed uses a Tofino switch [14] that directly connects to a control plane server via a 40-Gbps link. We consider heavy hitter detection [24] as the application. A heavy hitter is a two-tuple flow whose number of packets exceeds 2^{10}. In the switch, we implement a recently proposed hash table [13, 12] to record per-flow packet count. We configure 2^{16} entries in the hash table. Thus, the state in this application has 2^{16} values. We inject a trace from CAIDA [25] into the Tofino switch. In the control plane, we collect all the state values (i.e., per-flow packet count) from the



(a) False positive rate.

(b) Relative error.

Fig. 3: Impact of state divergence.

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 false negative rate (FNR).

We first study the impact of the OS-based approach, which spends several seconds (Figure 1(a)) to collect state values. 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 loss rate. In Figure 2(b), false negative rate rapidly rises as the loss rate increases. In summary, due to high latency or state loss in existing approaches, the state received by the control plane significantly diverges from the actual state in the data plane (10%∼70% divergence according to Figure 1(b)), leading to low application-level accuracy.

III. ApproSync Design

Challenges. There have been many solutions in the literature that can be used to handle state loss during state synchronization, e.g., timeout and retransmission mechanisms [26, 27]. Unfortunately, it is infeasible to realize these solutions in switch ASICs due to switch restrictions. Specifically, existing programmable switches typically employ specific architectures (e.g., PISA [28]) 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 consumptions. Here, we summarize three types of restrictions [28, 29]: (1) each switch is equipped with few memory (at most 10 MB [28, 30]); (2) a switch allows limited memory access (e.g., a few read-write operations for each packet); (3) a switch does not support complex operations (e.g., loop and buffering). Given these restrictions, it is challenging to handle state loss in switch ASICs.

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 [31, 32, 33, 34, 11, 35] and sketches [2, 36, 21, 37]. Thus, a small divergence is tolerable for these applications in practice.

To justify our observation, we evaluate the impact of state divergence on the accuracy of three applications: heavy hitter detection [24], UDP flood mitigation [5], and super-spreader detection [24]. 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. In super-spreader detection, a super spreader is a source IP address, which number of distinct destination IP addresses exceeds 1% of the total number of distinct IP addresses.

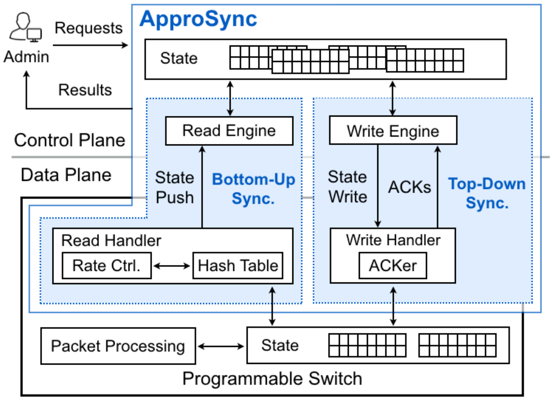


Fig. 4: Overview of ApproSync framework.

We employ 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 the 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.

Key idea. According to 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 relaxes the strict resource requirements in switches. However, it utilizes switch resources to bound the state divergence under link capacity. Doing so 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 (see §VIII for details). Although ApproSync follows similar ideas, we address the specific challenges of adopting approximation in state synchronization for programmable networks.

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 synchronization** (§**IV**). This strategy makes read operations keep pace with the state updates in the data plane. The read handler aggregates state updates and pushes them to the read engine, which extracts state values from received updates in the control plane. Instead of pushing all updates, it monitors the state divergence between two

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 state 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 $D \geq t$ then
 9:             Push ($H[p]$, $t$) to the control plane
10:             Update $H[p].old = v$
11:         end if
12:     else                                        ▷ $H[p].loc \neq l$
13:         Push ($H[p]$, $t$) to the control plane
14:         $H[p].loc = l, H[p].val = v, H[p].old = 0$
15:     end if
16: end function
```

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 synchronization (§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 are eventually performed after state write. 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 Synchronization for State Read

In this section, we present the synchronization algorithm of state read in §IV-A, and introduce the rate control in state read in §IV-B. Then we discuss some practical issues in §IV-C.

A. Synchronization Algorithm

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. 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 frequent hash collisions may exhaust link bandwidth. However, in practice, most traffic is contributed by a few flows due to the skewness of network traffic [38]. Thus, most state updates are incurred by a few flows, making the probability of hash collisions small. For instance, when using 2^{16} entries, the probability of hash collisions is below 5% for a one-hour CAIDA trace. This leads to a peak emitted rate of 0.92 Mpps, which is acceptable.

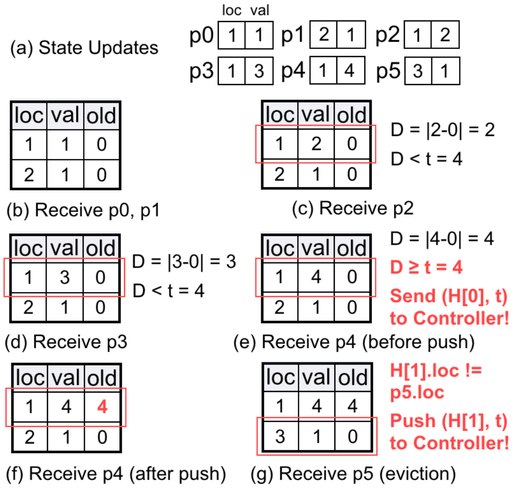

(a) State Updates
(b) Receive p0, p1
(c) Receive p2
(d) Receive p3
(e) Receive p4 (before push)
(f) Receive p4 (after push)
(g) Receive p5 (eviction)

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

Algorithm. The read handler is invoked for every tuple (l, v), which indicates that the value in location l has been updated to v, as shown in 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 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).

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)).

B. Rate Control

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 incoming traffic rate directly affects the bandwidth consumption of state updates. When 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.

Algorithm. The rate controller performs three steps to tune the threshold t as follows.

Step 1: the rate controller first 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 [39, 3] 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.

Step 2: the rate controller then calculates the maximum emitted rate supported by the link. Specifically, the maximum emitted rate is $M = \frac{c}{s}$. 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.

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

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 β=1 is sufficient. Thus, $M = \frac{c}{s} = 7.8125 \times 10^7$ pps. Suppose that Δ=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 $\frac{\Delta}{wt} \approx 5\times10^7$pps $< M$, which avoids link saturation and state loss.

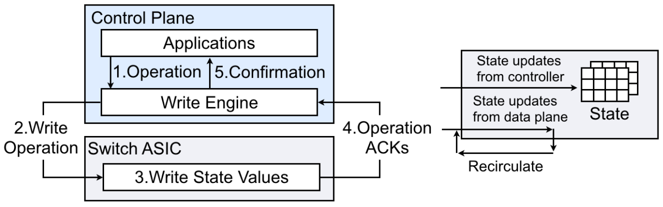
Case study. The read handler sends the current t with each state update to the control plane (Line 9 in Algorithm 1). With t, Applications can quantify the accuracy of the state. For example, we consider Count-Min (CM) [40] sketch, which maintains a counter array to estimate flow sizes. In the data plane, 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 CM sketch. However, ApproSync guarantees that the state divergence will not exceed the threshold t. Thus, we can fix the lower bound and upper bound of error of 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 [40]). 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

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 [41]. 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.



(a) ApproSync workflow of state write (b) Atomicity mechanism

Fig. 6: State write mechanism.

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 incur high bandwidth consumption nor affect the timeliness of state synchronization.

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.

V. Top-Down Synchronization for State Write

Write acknowledgment. As shown in Figure 6(a), applications issue a write operation comprising a set of state updates to the write engine. The write engine encapsulates these updates in several control plane 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 conforms 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 with write success and informs the write handler of termination.

Atomicity mechanism. At times, applications need to simultaneously write multiple state values, which requires *atomicity* to avoid unpredictable results. 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 [42, 43]. 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 [44, 45, 46]. However, existing concurrency control methods cannot be implemented on programmable switches because these methods require either excessive memory (e.g., 2PL [47], timestamps ordering [47], optimistic concurrency control [48], 2PC [49], 3PC [50]) or complex queue scheduling (e.g., deterministic system [51]).

To this end, ApproSync offers a hardware-compatible atomicity mechanism. 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, packets arriving during state write are normally forwarded. However, the new state updates incurred by these packets are recirculated for a second-pass processing, which eventually performs state updates. The recirculation continues until state write is completed and the lock is free.

Our rationale behind is two-fold. First, in most cases, the priority of state write is higher than data plane updates. This is because state write raised by applications usually changes processing strategies (e.g., reset a data structure). Thus, by prioritizing state write, ApproSync 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, such that it can be implemented in switches.

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.

Efficiency. The recirculation-based mechanism brings limited degradations on throughput and accuracy. Recall that ApproSync bypasses the switch OS, such that a write operation can be completed in short time, e.g., less than 20 ms for writing 2^{16} state values (Exp#2 in §VII). Within such a short time, the number of recirculated updates is small (Exp#4 in §VII). Thus, the throughput drop due to recirculation is also small ($<1\%$). Also, a small number of out-of-order updates cause limited accuracy drop. In particular, many applications (e.g., measurement [2, 3] and load balancing [30]) are insensitive to such reordering. Our experience is that the application-level accuracy drop is below 0.1% in practice (Exp#5 in §VII).

VI. Implementation

We have implemented a prototype of ApproSync, which targets P4-compatible switches. We maintain state values in registers. Note that P4 also offers another two components, i.e., counters and meters, to support stateful processing. ApproSync only targets registers because registers can not only realize the functions of counters and meters, but also support customizable operations towards state values.

Data plane handlers. Figure 7 details the implementation of ApproSync handlers. For the read handler, ApproSync records every state update in metadata fields in the ingress pipeline.

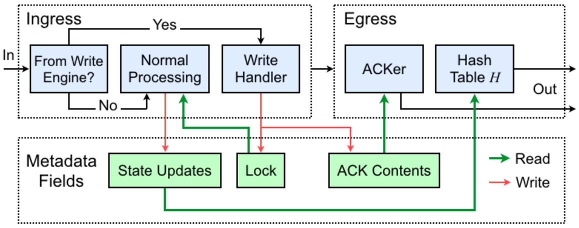


Fig. 7: Workflow of the switch ASIC.

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. 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.

Control plane engines. We implement ApproSync engines in C. Our prototype offers both southbound APIs and northbound APIs. We implement southbound APIs on DPDK [52] to avoid the overhead incurred by kernel stacks. We employ Redis [53] as the state storage and use HiRedis [54] to manage the storage. For northbound APIs, we design a suite of intuitive interfaces for applications to manage states stored in Redis.

Compiler. We implement a compiler to integrate ApproSync handlers 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

In this section, we conduct experiments to evaluate our ApproSync prototype. In each experiment, we present the average after 100 runs. We highlight our results as follows.

- ApproSync uses less than 15% switch resources (Exp#1).
- Compared to the OS-based approach, ApproSync achieves order-of-magnitude latency reduction in state read (Exp#2).
- Compared to *Flow [12], 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 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).
- ApproSync enables low-latency and accurate sketch collection (Exp#8).

TABLE I: 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) [56]	265	2^{16}	6
Flowlet switching (FL) [56]	251	2^{14}	10
Malicious DNS domain detection (MD) [57]	358	3×2^{16}	4
Snort flowbits (FB) [57]	296	3×2^{16}	4
Affine LB (AL) [57]	345	2^{17}	4
DNS TTL change tracking (TC) [58]	357	3×2^{16}	6
DNS tunnel detection (TD) [58]	530	3×2^{16}	4
Stateful firewall (FW) [24]	349	2^{17}	4
FTP monitoring (FM) [24]	303	2^{16}	4
Heavy hitter detection (HH) [24]	310	2^{17}	6
Super-spreader detection (SS) [24]	313	2^{17}	4
Sampling based on flow size (FS) [24]	560	5×2^{16}	4
SYN flood detection (SF) [5]	313	2^{17}	4
DNS amplification mitigation (AM) [5]	360	2^{16}	4
UDP flood mitigation (UF) [5]	309	2^{17}	4
Elephant flows detection (EF) [5]	553	5×2^{16}	4

TABLE II: (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	9.17%	0%	14.58%	3.65%	5.21%	100%

A. Setup

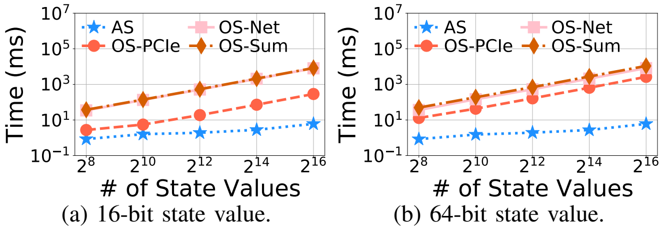
Platforms. We build a testbed comprising two 32×100 Gbps Barefoot Tofino switches [14] and six servers. Each server has 36-core Intel(R) Xeon(R) Gold 6240C CPU (2.60 GHz), 128GB RAM and a two-port 40-Gbps NIC. We run the control plane on a server based on DPDK [52]. In the data plane, we connect the 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.

Workloads. We select a one-hour CAIDA trace [25] with 38M packets, and use PktGen [55] 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 I.

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. Experimental Results

(Exp#1) Switch resource usage. This experiment measures the total usage of switch resources, including memory resources, computational resources, and match-action stages. Here, memory includes both SRAM and TCAM, while computational resources include meter ALUs (mALUs), stateful



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

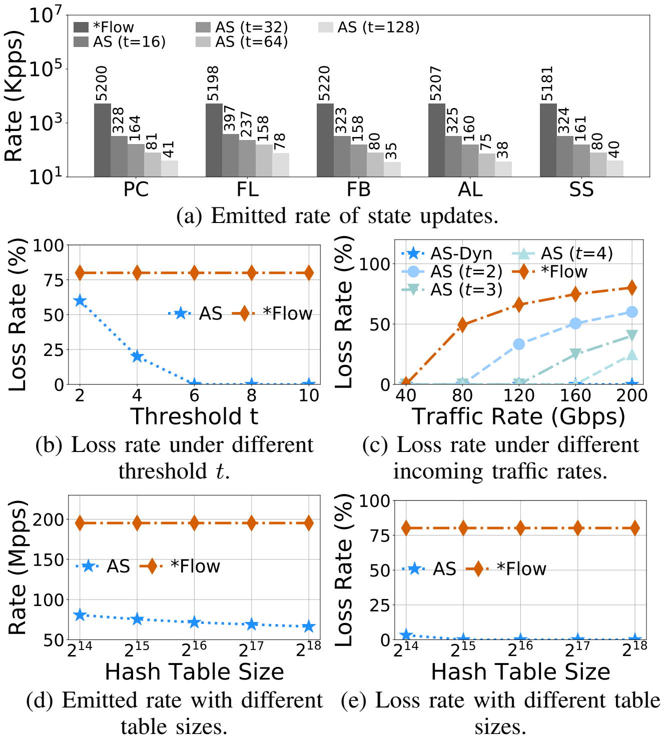
Fig. 8: (Exp#2) Performance of state read.

ALUs (sALUs), and very long instruction words (VLIWs). 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 II shows that ApproSync uses less than 15% resources even when using both the primitives in one switch. 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.

(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 [14] and ZeroMQ [9]. 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 8 shows that the latency of 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.

(Exp#3) Accuracy of state read. We evaluate the accuracy of state read. First, we validate that ApproSync can avoid 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 I. We compare ApproSync with *Flow [12], a traffic mirroring system that uses an LRU cache for rate control. We configure the LRU cache in the same way as the hash table of ApproSync. Figure 9(a) shows that *Flow brings limited benefits because its LRU 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* [57] with t=128) via its rate control.

Second, we validate that ApproSync can offer accurate state read. We deploy *packet counter* (PC) that generates a state update for every packet. We inject traffic at 200 Gbps so that the emitted rate of state updates far exceeds link capacity. We vary t from 2 to 10. Figure 9(b) shows that ApproSync gradually reduces emitted rate as t increases, and avoids state loss when $t \geq 6$. Then we repeat the experiment without any manual settings. Figure 9(c) shows that ApproSync (AS-Dyn) offers loss-free state read since it dynamically tunes t to adapt to incoming traffic rate. In contrast, ApproSync with fixed t ($t < 5$) and *Flow cannot guarantee no state loss, which emphasizes the importance of rate control.



(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.

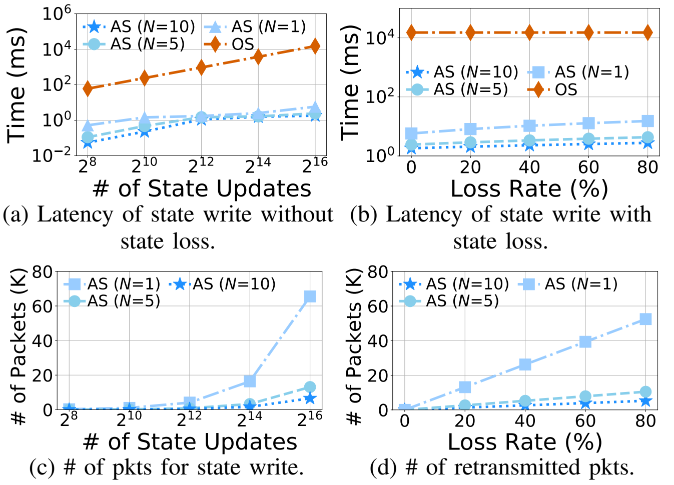
(e) Loss rate with different table sizes.

Fig. 9: (Exp#3) Accuracy of state read.

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 9(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.

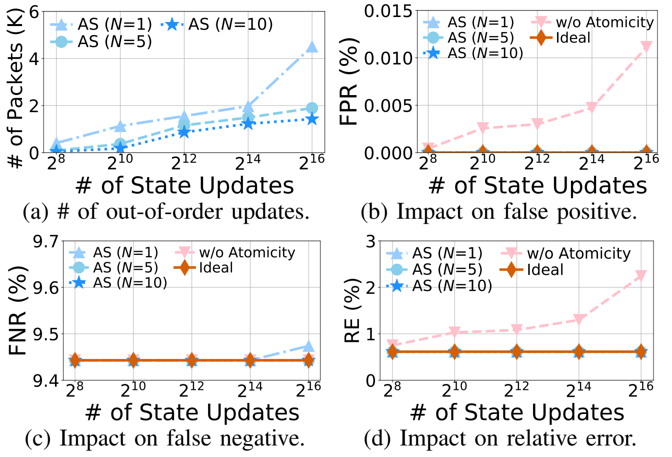
(Exp#4) Performance of state write. We measure the performance of state write. ApproSync splits a write operation into several partial operations, each of which is completed by a single packet. Thus, the performance of state write depends on the number N of state updates encapsulated in a packet. We set N to 1, 5, and 10, and vary the number of state updates from 2^8 to 2^{16}. Figure 10(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 lose rate ranges from 0% to 80%. Figure 10(b) presents that even with 80% loss rate, ApproSync completes state write in a few milliseconds. In addition, we measure the bandwidth



(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.

Fig. 10: (Exp#4) Performance of state write.



(a) # of out-of-order updates. (b) Impact on false positive. (c) Impact on false negative. (d) Impact on relative error.

Fig. 11: (Exp#5) Accuracy of state write.

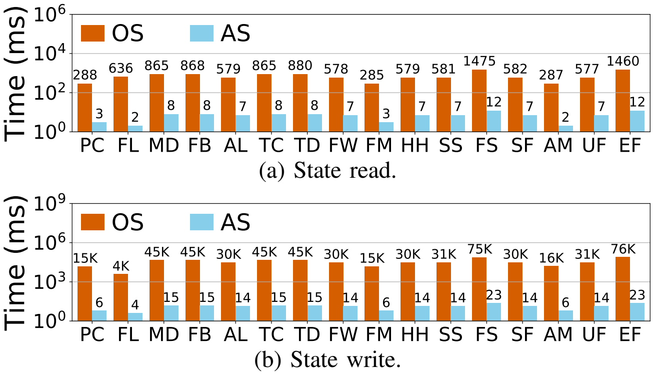
consumed by state write. Figure 10(c)-(d) present the number of packets for a write operation and that for retransmission under different loss rates. Even in the worst case, the number of generated packets is at most 65K, which is far below link capacity (e.g., 14.88 Mpps of a 10 Gbps link).

(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 [29], 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 11(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 and bandwidth consumption is small. Next, we examine the impact on HashPipe accuracy. Figure 11(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%, which is small; (2) compared to the OS-based approach, ApproSync offers highly accurate state write with atomicity guarantees.

(Exp#6) Impact on packet forwarding. Table III 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%.

TABLE III: (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%



(a) State read.

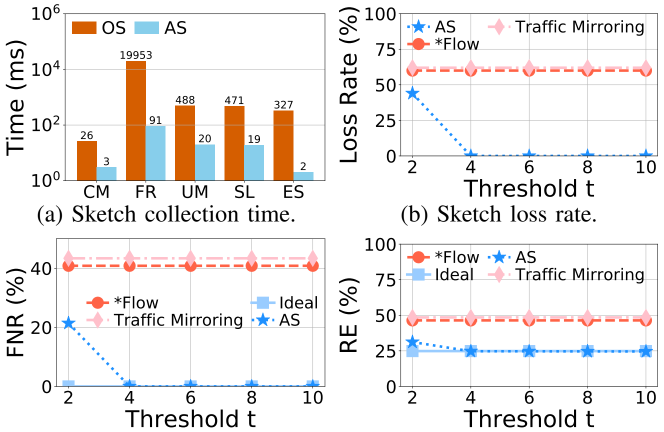
(b) State write.

Fig. 12: (Exp#7) Application-perceived latency.

(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 I, and measure the latency of reading states from the counters used by applications and resetting counters. We compare ApproSync with the OS-based approach in Figure 12. 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.

(Exp#8) Fast and accurate sketch collector. Sketch is a widely-used family of algorithms in network measurement [2, 3, 21, 36, 37, 59, 60, 61, 62]. 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 ApproSync, which contains two steps: (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 latest sketch values to the control plane.

We first evaluate the performance of sketch collector. We implement five sketch algorithms, Count-Min (CM) [40], FlowRadar (FR) [3], UnivMon (UM) [2], SketchLearn (SL) [21] and ElasticSketch (ES) [37]. We set these sketch algorithms with the following realistic settings [21, 37]. (1) Count-



(a) Sketch collection time.

(b) Sketch loss rate.

(c) Sketch false negative.

(d) Sketch relative error.

Fig. 13: (Exp#8) Fast and accurate sketch collector.

Min uses three hash functions and 2^{16} 4-byte counters. (2) FlowRadar uses three hash functions in both the Bloom filter [63] and the invertible Bloom lookup table (IBLT) [64]. 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 13(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 13(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.

Finally, we qualify the accuracy of heavy hitter detection, whose settings are the same as that in §II-B. We only present the results of Count-Min due to space limitation. We also build an original version of Count-Min without state loss (“Ideal”) as the baseline. Figure 13(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 closed to “Ideal”.

VIII. Related Work

State read. Prior solutions [31, 32, 33, 34, 11, 35] exploit packet sampling to reduce bandwidth consumption. However, sampling techniques inevitably degrade accuracy. TurboFlow [65] 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 [13] 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 [42] aggregates packets based on packet processing behaviors to reduce overhead. MAFIA [43] proposes intent-based primitives to ease the deployment of measurement tasks on programmable switches. Sonata [66] pre-processes packets in switches to reduce workloads in the control plane. ApproSync is complementary to these solutions with its state synchronization mechanisms.

State write. Commodity controllers modify states via TCP-based protocols such as OpenFlow [6], Thrift [7] and gRPC [8], which require a switch OS for complicated processing. Applications built on these protocols (e.g., P4NFV [67]) suffer from high latency. Swing State [68] directly migrates state values in the data plane to achieve rapid state migration. P4Sync [69] offers strong authenticity guarantees when migrating state values among switches. P4State [70] 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 state write.

Approximate systems. Approximation is a well-studied topic in distributed systems, including database [71], machine learning systems [72, 73, 74], and stream processing systems [75, 76, 77, 78]. BlinkDB [71] samples data to dynamically estimate the response time and error of a database query. JetStream [75] uses data aggregation and adaptive filtering to achieve trade-offs between accuracy and resource efficiency. AF-Stream [77] provides approximate fault tolerance in the content of stream processing. In contrast, ApproSync exploits approximate techniques to achieve low-latency and accurate state synchronization in programmable networks.

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 during state synchronization. It offers two types of operations, state read and state write. We implement ApproSync on Barefoot Tofino switches. Our evaluation indicates that ApproSync outperforms existing solutions with order-of-magnitude latency reduction and higher accuracy.

Acknowledgement

We thank our shepherd Prof. Gianni Antichi, and the anonymous reviewers for their constructive comments. This work is supported by the National Key R&D Program of China (2019YFB1802600), the National Natural Science Foundation of China (61802365), FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications (No. LZC0019), the Industrial Internet innovation and development project (No. TC190A449), the Key R&D Program of Zhejiang Province (2020C01021), and Major Scientific Project of Zhejiang Lab (2018FD0ZX01).

Reference

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

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

[3] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: a better netflow for data centers," in *NSDI*, 2016, pp. 311–324.

[4] Q. Huang, H. Sun, P. P. Lee, W. Bai, F. Zhu, and Y. Bao, "Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy," in *SIGCOMM*, 2020, pp. 404–421.

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

[6] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[7] Apache Software Foundation, Thrift. http://thrift.apache.org/.

[8] grpc. https://www.grpc.io/.

[9] Zeromq. http://zeromq.org/.

[10] J. C. Mogul and P. Congdon, "Hey, you darned counters!: get off my asic!" in *HotNet*. ACM, 2012, pp. 25–30.

[11] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, "Planck: Millisecond-scale monitoring and control for commodity networks," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 407–418.

[12] 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 *ATC*. USENIX, 2018, pp. 823–835.

[13] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *SIGCOMM*. ACM, 2017, pp. 85–98.

[14] Barefoot Network. Barefoot Tofino. https://www.barefootnetworks.com/technology/\#tofino.

[15] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 183–197.

[16] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian, "Drill: Micro load balancing for low-latency data center networks," in *SIGCOMM*. ACM, 2017, pp. 225–238.

[17] M. G. Gouda and A. X. Liu, "A model of stateful firewalls and its properties," in *DSN*. IEEE, 2005, pp. 128–137.

[18] R. Joshi, T. Qu, M. C. Chan, B. Leong, and B. T. Loo, "Burstradar: Practical real-time microburst monitoring for datacenter networks," in *APSys*. ACM, 2018, p. 8.

[19] D. Shan, F. Ren, P. Cheng, R. Shu, and C. Guo, "Micro-burst in data centers: Observations, analysis, and mitigations," in *ICNP*. IEEE, 2018, pp. 88–98.

[20] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, and O. Rottenstreich, "Catching the microburst culprits with snappy," in *SelfDN*. ACM, 2018, pp. 22–28.

[21] Q. Huang, P. P. Lee, and Y. Bao, "Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference," in *SIGCOMM*. ACM, 2018, pp. 576–590.

[22] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao *et al.*, "Packet-level telemetry in large datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 479–491.

[23] O. Tilmans, T. Bühler, S. Vissicchio, and L. Vanbever, "Mille-feuille: Putting isp traffic under the scalpel," in *HotNet*. ACM, 2016, pp. 113–119.

[24] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for sdn," in *HotSDN*. ACM, 2014, pp. 61–66.

[25] The caida 2018 anonymized internet traces. http://www.caida.org/data/overview/.

[26] M. Allman and V. Paxson, "On estimating end-to-end network path properties," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4, pp. 263–274, 1999.

[27] P. Sarolahti, M. Kojo, and K. Raatikainen, "F-rto: an enhanced recovery algorithm for tcp retransmission timeouts," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 2, pp. 51–63, 2003.

[28] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[29] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *SOSR*. ACM, 2017, pp. 164–176.

[30] 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 *SIGCOMM*. ACM, 2017, pp. 15–28.

[31] sflow. http://sflow.org/about/index.php.

[32] V. Sekar, M. K. Reiter, and H. Zhang, "Revisiting the case for a minimalist approach for network flow monitoring," in *SIGCOMM*. ACM, 2010, pp. 328–341.

[33] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 254–265.

[34] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *NSDI*, 2014, pp. 71–85.

[35] C. Zhang, J. Bi, Y. Zhou, J. Wu, B. Liu, Z. Li, A. B. Dogar, and Y. Wang, "P4db: On-the-fly debugging of the programmable data plane," in *ICNP*. IEEE, 2017, pp. 1–10.

[36] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *SIGCOMM*. ACM, 2017, pp. 113–126.

[37] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *SIGCOMM*. ACM, 2018, pp. 561–575.

[38] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *IMC*, 2009, pp. 202–208.

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

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

[41] P4 Language Consortium, "The p4-14 language specification, version 1.0.4," 2017. [Online]. Available: https://p4lang.github.io/p4-spec/p4-14/v1.0.4/tex/p4.pdf

[42] Y. Zhou, J. Bi, T. Yang, K. Gao, C. Zhang, J. Cao, and Y. Wang, "Keysight: Troubleshooting programmable switches via scalable high-coverage behavior tracking," in *ICNP*. IEEE, 2018, pp. 291–301.

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

[44] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the abyss: An evaluation of concurrency control with one thousand cores," *PVLDB*, vol. 8, no. 3, 2014.

[45] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker, "An evaluation of distributed concurrency control," *PVLDB*, vol. 10, no. 5, pp. 553–564, 2017.

[46] 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," *PVLDB*, vol. 12, no. 13, pp. 2325–2338, 2019.

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

[48] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 2, pp. 213–226, 1981.

[49] D. Skeen, "Nonblocking commit protocols," in *SIGMOD*. ACM, 1981, pp. 133–142.

[50] ——, "A quorum-based commit protocol," Cornell University, Tech. Rep., 1982.

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

[52] Intel corporation. Data Plane Development Kit. http://dpdk.org.

[53] Redis. http://redis.io/.

[54] Hiredis. http://redis.io/.

[55] Pktgen. https://pktgen-dpdk.readthedocs.io/.

[56] P4 Tutorials. https://github.com/p4lang/tutorials/tree/sigcomm_17.

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

[58] K. Borders, J. Springer, and M. Burnside, "Chimera: A declarative language for streaming network traffic analysis," in *Security*. USENIX, 2012, pp. 365–379.

[59] Q. Huang and P. P. Lee, "Ld-sketch: A distributed sketching design for accurate and scalable anomaly detection in network data streams," in *INFOCOM*. IEEE, 2014, pp. 1420–1428.

[60] ——, "A hybrid local and distributed sketching design for accurate and scalable heavy key detection in network data streams," *Computer Networks*, vol. 91, pp. 298–315, 2015.

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

[62] ——, "Spreadsketch: Toward invertible and network-wide detection of superspreaders," in *INFOCOM*. IEEE, 2020, pp. 1608–1617.

[63] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[64] M. T. Goodrich and M. Mitzenmacher, "Invertible bloom lookup tables," in *Allerton*. IEEE, 2011, pp. 792–799.

[65] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "Turboflow: Information rich flow record generation on commodity switches," in *EuroSys*. ACM, 2018, p. 11.

[66] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *SIGCOMM*. ACM, 2018, pp. 357–371.

[67] M. He, A. Basta, A. Blenk, N. Deric, and W. Kellerer, "P4nfv: An nfv architecture with flexible data plane reconfiguration," in *CNSM*. IEEE, 2018, pp. 90–98.

[68] S. Luo, H. Yu, and L. Vanbever, "Swing state: Consistent updates for stateful and programmable data planes," in *SOSR*. ACM, 2017, pp. 115–121.

[69] J. Xing, A. Chen, and T. E. Ng, "Secure state migration in the data plane," in *SPIN*, 2020, pp. 28–34.

[70] M. He, A. Blenk, W. Kellerer, and S. Schmid, "Toward consistent state management of adaptive programmable networks based on p4," in *NEAT*, 2019, pp. 29–35.

[71] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "Blinkdb: queries with bounded errors and bounded response times on very large data," in *EuroSys*. ACM, 2013, pp. 29–42.

[72] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *OSDI*, 2014, pp. 583–598.

[73] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Managed communication and consistency for fast data-parallel iterative analytics," in *SoCC*. ACM, 2015, pp. 381–394.

[74] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A new platform for distributed machine learning on big data," *IEEE Transactions on Big Data*, vol. 1, no. 2, pp. 49–67, 2015.

[75] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, "Aggregation and degradation in jetstream: Streaming analytics in the wide area," in *NSDI*, 2014, pp. 275–288.

[76] S. Agarwal and K. Zeng, "Blinkdb and g-ola: Supporting continuous answers with error bars in sparksql," *Spark Summit*, 2015.

[77] Q. Huang and P. P. Lee, "Toward high-performance distributed stream processing via approximate fault tolerance," *VLDB*, vol. 10, no. 3, pp. 73–84, 2016.

[78] Z. Cheng, Q. Huang, and P. P. Lee, "On the performance and convergence of distributed stream processing via approximate fault tolerance," *The VLDB Journal*, vol. 28, no. 5, pp. 821–846, 2019.