ApproSync: Approximate State Synchronization for Programmable Networks

Technical Report, 16 Pages

ABSTRACT

Programmable switches empower stateful packet processing, in which incoming packets update states in the data plane, while applications in the control plane read and/or 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 without involving switch operating systems. To achieve high accuracy, ApproSync leverages the resources in switch ASICs to avoid potential state loss, which bounds the state divergence between the data plane and control plane. In particular, ApproSync employs two different approximate synchronization strategies for two types of operations, i.e., state read and state write, given their unique patterns. We prototype ApproSync in P4, and enhance three real-world use cases using ApproSync. The experimental results indicate that compared to state-of-the-art methods, ApproSync achieves order-of-magnitude latency reduction while maintaining high accuracy.

1 INTRODUCTION

Recent advances in programmable networks empower network administrators to customize the packet processing logic of network devices. For example, with the P4 language [19], administrators are able to define new network protocols and specify packet processing behaviors in programmable switches. Programmable switches often expose a collection of stateful memory (e.g., registers), which can be leveraged to store the processing *state*. State is a set of historical packet processing values (e.g., packet counts) that influence future processing decisions. By manipulating state values, administrators can build *stateful* network management applications, such as real-time traffic monitoring [48, 49] and network function redundancy [28, 30, 64].

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, applications manipulate states via the control plane. 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 [25] 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 consistency is achieved via an operating system (switch OS) installed atop 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 [1, 4, 11, 51]. Both PCIe channels and TCP connections could be the bottleneck to keep pace with state updates incurred by high-speed traffic [53, 58]. Our experiments indicate that the OS-based approach even spends several seconds to transfer a state with a normal size of 2^{16} values, which is slow and unacceptable (see §2.2). Some approaches [55, 58, 70] bypass the switch OS via traffic mirroring, which directly transfers state updates between the switch ASIC and the control plane, to achieve low latency. However, traffic mirroring fails to achieve high reliability like rate control and loss recovery. Thus, it suffers from high state loss rate when traffic rate exceeds the link capacity.

In this paper, we propose ApproSync, a low-latency and accurate state synchronization framework. To achieve low latency, ApproSync bypasses the switch OS to eliminate the performance overheads. However, it is challenging to entirely handle state loss in the switch ASIC due to switch resource restrictions. In response, ApproSync incorporates approximate strategies to achieve high accuracy. The notion behind 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 and *eventually* makes the states in both planes consistent. Such design significantly alleviates resource requirements, making ApproSync readily deployable in programmable switches. Specifically, ApproSync adopts two approximate strategies to support two types of state operations, i.e., *state read* and *state write*, respectively. For state read, ApproSync monitors the state divergence in the switch ASIC and synchronizes state updates only when the state divergence exceeds a threshold. In particular, ApproSync

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 bound the state divergence into a negligible level. Thus, it keeps the emitted rate of state updates just below link capacity, such that it achieves maximum accuracy under link capacity. For state write, ApproSync retries all failed operations to mitigate state loss. It ensures the atomicity of state write by suspending any state update in the data plane during the state write. It guarantees that all suspended updates are eventually performed.

In summary, we make the following contributions.

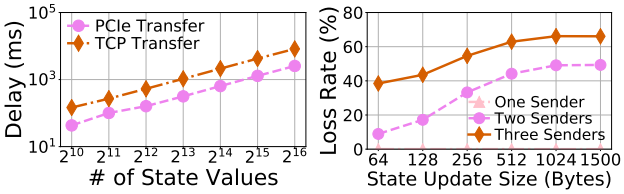
- We design ApproSync based on approximate techniques to offer low-latency and accurate state synchronization. ApproSync is the first system that addresses the state consistency in programmable networks.
- We present three use cases enhanced by ApproSync: low-latency result collection for sketch-based measurement, state migration among switches, and timely prevention for UDP flood attacks.
- We implement a prototype of ApproSync with Barefoot Tofino switches [2] and commodity servers. We extensively evaluate ApproSync with 16 stateful P4 applications. The experimental results indicate that ApproSync achieves order-of-magnitude latency reduction against state-of-the-art works and maintains high accuracy.

2 MOTIVATION

2.1 Problem

This paper targets state synchronization for programmable networks (e.g., data center networks [27, 48, 65]), 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 perform control actions by modifying states in the data plane. For instance, stateful firewall [29] requires to efficiently collect state values from switches to rapidly detect attacks. Also, it dynamically updates the detection thresholds recorded in switches to adjust to traffic dynamics.

Such distributed processing motivates a state synchronization mechanism to keep a consistent view of states in both planes. In the bottom-up direction, state updates incurred by data plane packets must be synchronized to the control plane. In the top-down direction, state modifications in the control plane should also be reflected in the data plane. In particular, applications require the state synchronization to achieve both *low latency* and *high accuracy*.



(a) High latency of OS-based approach. **(b) State loss rate of traffic mirroring.**

Figure 1: Benchmark of existing state transfers.

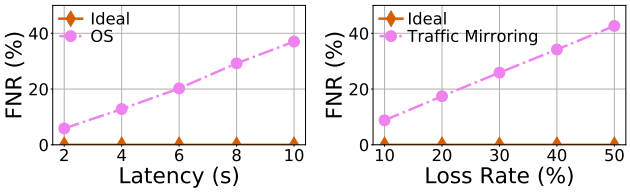
- **Low latency**: we aim to minimize the latency of state synchronization in both directions (i.e., from data plane to control plane and vice versa). This is critical to keep pace with high-speed network traffic and meet the tight latency requirements raised by applications. For example, network anomaly detection requires to rapidly detect and react to suspect events [21, 37, 39, 63].
- **High accuracy**: we aim to minimize the state divergence between the data plane and control plane, which guarantees the correctness of application operations. For instance, the attack detector may raise a false alarm if received states are highly noisy. Moreover, if a state modification is not properly synchronized to the data plane, the data plane behaviors may be wild (e.g., a firewall policy fails to work).

2.2 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.

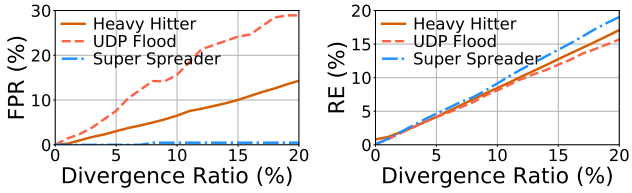
High latency in OS-based approach. The OS-based approach utilizes the switch OS to transfer the state between the switch ASIC and the control plane. For the switch ASIC, the OS manipulates the state via PCIe channels. For the control plane, the OS establishes TCP connections [1, 4, 11, 51] to transfer the state. However, this approach incurs high latency in two aspects. First, due to limited bandwidth, PCIe channels could be the performance bottleneck [58]. Second, TCP connections incur long delay 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 [2]. 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. Such high latency compromises many applications that require sub-second state collection (e.g., network measurement [23, 37, 60], attack mitigation [38, 42, 73], and data center network management [41, 56, 74]).

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 dedicated mirroring port [58, 70, 72, 82]. However, these approaches suffer from serious state loss when the emitted rate



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

Figure 2: Impacts on applications.



(a) False positive rate. **(b) Relative error.**

Figure 3: Impact of small state divergence.

of state updates exceeds the port capacity [58]. In Figure 1(b), we measure the loss rate in a Tofino switch. We allocate one mirroring port and vary the number of traffic ports. We inject traffic to each traffic port to reach its maximum rate (i.e., 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 conduct correct operations.

Impacts on applications. We study the impacts of existing approaches on heavy hitter detection [54]. We define a heavy hitter as a 2-tuple flow whose number of packets exceeds 1K, and set the time interval of collecting flow records to 1s. First, we study the impacts of the OS-based approach, which spends several seconds to pull states from switch ASICs. Figure 2(a) shows that false negative rate significantly increases as the latency spent by the OS-based approach increases. Second, we evaluate the impacts of traffic mirroring, which suffers from high loss rate. In Figure 2(b), false negative rate rapidly increases as the loss rate increases. To summarize, due to high latency or state loss, existing approaches significantly drop application-level accuracy.

3 APPROSYNC DESIGN

We design ApproSync to provide timely and accurate state synchronization. For low latency, ApproSync realizes synchronization entirely in the switch ASIC to bypass the switch OS. For high accuracy, ApproSync fully utilizes the resources of switch ASICs to avoid state loss.

Challenge. 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 [14, 61]. 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 [20]) to achieve high throughput and ultra-low latency. Such architectures impose several restrictions on their resource models due to the concern of chip footprints and heat consumptions. Here, we summarize three types of restrictions [20, 66]: (1) each switch is equipped with few memory (at most 10 MB [20, 52]); (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 the above restrictions, it is challenging to entirely avoid state loss in switch ASICs.

Key idea. We base ApproSync on *approximate state synchronization* that exploits approximate techniques to achieve high accuracy. We observe that although large errors seriously degrade application accuracy as shown in §2.2, it is acceptable for many applications to maintain a small state divergence. In fact, many applications are already built on approximate algorithms such as sampling [10, 24, 32, 58, 62, 80] and sketches [35, 37, 49, 78]. Thus, a small divergence is tolerable for these applications in practice. To justify this, we evaluate the impact of state divergence on the accuracy of three applications, heavy hitter detection [54], UDP flood mitigation [25], and super-spreader detection [54]. We vary the relative divergence from 0% to 20%. Figure 3 shows that the application-level error is negligible when the divergence is small, which validates our observation. Note that different from the benchmarks in §2.2 where we measure high state divergence (10%~50%), here we address application accuracy under small divergence (<5%).

To bound the divergence to a negligible level, ApproSync performs synchronization on demand while taking switch resources into account. Such approximate synchronization brings two-fold benefits. First, it greatly relaxes the resource requirement, and hence mitigates the above challenges incurred by switch restrictions and makes ApproSync readily deployable. Second, the approximate design imposes an upper bound on the state divergence, in which applications remain confident on the synchronization results.

Note that approximate techniques have been widely adopted in distributed systems (see §7 for details). Although ApproSync follows similar ideas, it carefully identifies the specific challenges of adopting approximation in state synchronization for programmable networks. Also, it offers customized solutions to address the identified challenges rather than simply bundling existing techniques.

Architecture. As shown in Figure 4, ApproSync synchronizes state in two directions, i.e., from the data plane to control plane and vice versa, which correspond to two types of operations, *state read* and *state write*, respectively. It offers two synchronization strategies, *bottom-up synchronization*

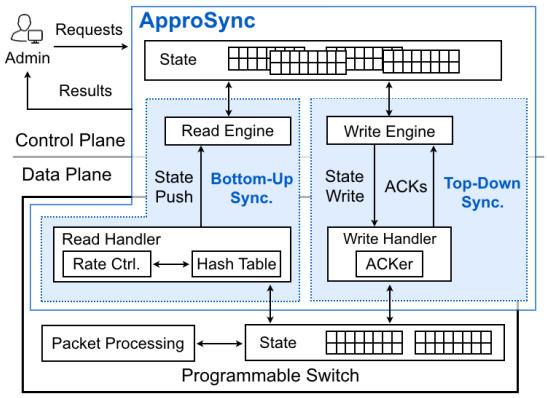


Figure 4: Overview of ApproSync framework.

for state read, and *top-down synchronization* for state write. Each strategy is realized by a read/write handler in the switch ASIC and an read/write engine in the control plane, which collectively synchronize states.

- **Bottom-up synchronization (§3.1)**. This strategy makes read operations keep pace with the state updates incurred by data plane packets. The read handler monitors every state update and *selectively* pushes updates to the read engine, which extracts state values from received updates in the control plane. To mitigate state loss, the read handler offers a hash table that aggregates state updates for each memory location and monitors the state divergence. Once the state divergence exceeds a threshold, it pushes aggregated updates to the control plane to make the state in both planes consistent. Meanwhile, it adaptively tunes the threshold to keep the emitted rate of state updates smaller than link capacity instead of traffic mirroring. Such rate control (1) offers error-free state read when incoming traffic rate is low, and (2) avoids link saturation and state loss while minimizing the state divergence even with high incoming traffic rate. Further, the state divergence will also be provisioned such that users can examine the quality of states. On the other hand, the read engine enables applications to obtain the latest state updates from the read handler, which reduces the state divergence.
- **Top-down synchronization (§3.2):** This strategy enforces the state modifications raised by applications on programmable switches for write operations. To guarantee zero loss of modifications, the write engine exploits an acknowledgement mechanism since write operations are not invoked so frequently. To ensure atomicity during synchronization, the write handler suspends the state updates incurred by data plane packets: it recirculates new state updates for a second-pass processing. These updates are eventually performed after the synchronization. This sacrifices a limited portion of ordering. However, this approach incurs no data loss while greatly improving the overall resource efficiency.

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 ≥ 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 ≠ l
13:         Push (H[p], t) to the control plane
14:         H[p].loc = l, H[p].val = v, H[p].old = 0 ▷ Modify H[p]
15:     end if
16: end function
```

3.1 Bottom-Up Synchronization for State Read

For bottom-up state read, ApproSync offers (1) a read handler in the switch ASIC that aggregates state updates, selectively pushes state updates, and adaptively controls the emitted rate of state updates, and (2) a read engine in the control plane that enables applications to directly read the latest state updates. We elaborate how the two components collectively provide highly accurate state read as follows.

Data structure. The read handler maintains a hash table H to monitor 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) $H[p].val$ denotes the current state value in location $H[p].loc$, and (3) $H[p].old$ records the corresponding state value in the control plane. Since the switch memory is scarce, we restrict the size of H. According to Exp#1 and Exp#3 in §6, a size of 2^{16} entries is sufficient for applications to retain high accuracy with a small portion of switch resources. Moreover, when hash collisions happen, H evicts the old entries to the control plane so as to make room for new keys. Here, one concern is that frequent hash collisions could exhaust link bandwidth. However, in real-world workloads, most traffic is contributed by a small portion of flows due to the skewness of network traffic [40]. Thus, most state updates are incurred by a few flows, making the probability of hash collisions small in practice. According to our experiments, when using 2^{16} entries, the probability of hash collisions is below 5% for a one-hour CAIDA trace. This results in a peak rate of evicting entries of 9.18×10^5 pps, which occupies at most 1.2% capacity of a 40 Gbps link. Such a small rate is affordable.

Algorithm. Algorithm 1 details how the read handler processes state updates. It is invoked with respect to every tuple

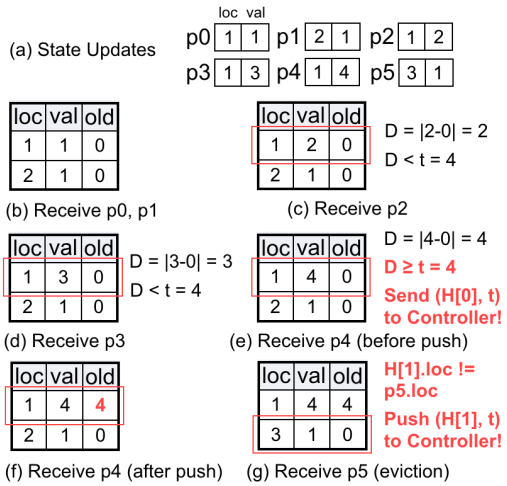


Figure 5: Example of hash table in the read handler.

(l, v), which indicates that the value in location l has been updated to v. The read handler 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 pre-defined threshold t, both the entry $H[p]$ and t are emitted to the control plane (lines 8-9). Meanwhile, $H[p].old$ is changed to reflect the updated control plane value (line 10). If the stored location is different from the new location (line 12), indicating a hash collision, the read handler pushes the existing entry and the threshold t to the control plane (line 13), and then modifies the entry to store the new one (line 14).

Example. Figure 5 presents an example of Algorithm 1. Suppose that the threshold t is four and the hash table H has two buckets, and there are six state updates (Figure 5(a)). For the first two updates 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)).

Rate control. The threshold t controls the trade-off between accuracy and bandwidth consumption. Our intuition is that we can employ a smaller threshold to minimize state divergence as long as the 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.

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 = 1ms, the rate controller consumes 6.4×10^{-2} Mbps to read Δ, which is negligible compared to Gbps-level switch bandwidth. Moreover, w is determined by applications. For instance, the detection of low-rate TCP denial-of-service attacks [45, 48] 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 $\frac{\Delta}{w}$. Also, the maximum emitted rate $M = \frac{c}{s}$ supported by a link can be calculated based on link capacity c and the size s of each state update (l, v), which are known a priori. Note that $\frac{\Delta}{w}$ is a metric that reflects incoming traffic rate: when incoming traffic rate is low, Δ is small, making $\frac{\Delta}{w}$ small; otherwise, the reverse holds true.

Thereafter, the rate controller determines 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 exceeds link capacity so that the rate controller needs to set a non-zero t for the sake of 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 $t = \left\lceil \frac{\Delta}{wM} \right\rceil$. In practice, the rate controller sets a $t = \left\lceil \beta \frac{\Delta}{wM} \right\rceil$ for some $\beta \geq 1$ to handle unexpected traffic bursts. Our experience is that a small β closed to one is sufficient for most applications. We summarize how the rate controller sets t as follows.

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

We further illustrate the rate control of ApproSync via an example. We assume that (1) ApproSync uses a 10 Gbps link to transfer 16-byte state updates, so the link capacity $c = 10^{10}$ bps and the size of a state value s = 128 bits; (2) the time interval w = 1ms; (3) no microbursts happen so a $\beta = 1$ is sufficient. Thus, M is calculated as $M = \frac{c}{s} = 7.8125 \times 10^7$ pps. In the first time interval, the read controller obtains a $\Delta = 10^4$. Since $\frac{\Delta}{w} = 10^7 \leqslant M$, the read controller sets the threshold t to zero, such that every state update can be transfered to the control plane without exhausting link capacity. In the second time interval, Δ is changed to 10^5,

which makes $\frac{\Delta}{w} > M$. In this case, the rate controller sets t to $\left\lceil \beta \frac{\Delta}{wM} \right\rceil = 2$. Thus, the maximum emitted rate of state updates is $\frac{\Delta}{wt} \approx 5 \times 10^7$ pps, which is below M.

Case study. As the threshold t dynamically varies, the collect handler sends the current t with each state update to the controller (Line 9 in Algorithm 1). Applications can utilize the received t to examine the quality of the state and fix the results. Here, we use an example of Count-Min (CM) [22] sketch to demonstrate the usability of t. A CM sketch maintains several counter arrays to estimate flow sizes. In the data plane, a packet selects one counter in each 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. Since a counter could accommodate multiple flows, the estimate is usually larger than the actual count. Previous study shows that the error of CM sketch can be bounded with sufficient number of counters [22].

When we deploy a CM sketch in ApproSync, we can include the divergence t caused by ApproSync into the original error of CM sketch, 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 [22]). 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. □

Entry read. As ApproSync records state updates in a hash table, one concern may be that some state updates can stay in the hash table for a long time when no evictions happen, which may affect accuracy. To this end, ApproSync offers a control plane interface, namely entry read. When entry read is activated, the read engine in the control plane generates some packets, the number of which equals to the size of hash table, and sends these packets to the switch ASIC. Each packet is a query that targets a specific hash table entry. When receiving a packet from read engine, the read handler in the switch ASIC searches for the target hash table entry, and appends the entry on packet headers. Then it sends the packet back to the control plane. In this way, ApproSync supports applications to obtain the latest state updates and alleviates the above concern.

The entry read is also low-overhead. Consider a general case where the hash table has 2^{16} entries and each entry records a 10-byte state update. Thus, the read engine will generate 2^{16} 64-byte packets for entry read, while the read handler will piggyback a 10-byte entry on each packet. Given a time interval of 1s, the bandwidth consumption is 3.88 Mbps, which below 0.01% capacity of a 40-Gbps link.

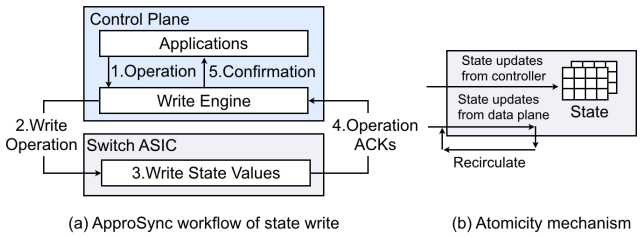


Figure 6: State write mechanism.

Summary. The state read of ApproSync avoids state loss via its rate control. In the interest of high accuracy, it essentially bounds the maximum per-state-value divergence between two planes by adaptively tuning the threshold t.

3.2 Top-Down Synchronization for State Write

For state write, ApproSync offers top-down synchronization driven by the write engine in the control plane. The write engine acknowledges every write operation to mitigate state loss. For atomicity, it suspends the state updates incurred by data plane packets during state write, and eventually performs them after state write. We present the details of state write in the following.

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 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 write operation. Specifically, it performs the write operation 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 need to reset the entire counter array in the switch ASIC to prevent legacy traffic statistics from disturbing ongoing measurement [46, 81]. However, during state write, data plane packets also continuously update the state in the switch ASIC, which harms atomicity. One strawman solution is to involve concurrency control methods [16, 33, 79] to avoid conflicts between state write and updates incurred by

data plane packets. However, existing concurrency control methods cannot be implemented on programmable switches. The reason is that these methods either need excessive memory (e.g., 2PL [17], TO [17], OCC [44], 2PC [67], and 3PC [68]) or require complicated queue scheduling (e.g., DS [59]), which are not feasible in switch ASICs.

To this end, ApproSync offers a hardware-compatible state write that guarantees atomicity during state write. As depicted in Figure 6(b), the basic idea is to lock the *entire* state in the data plane and suspend state updates during state write. It handles the suspended updates by using the *recirculation* mechanism in switches. Specifically, packets arriving during state write are normally forwarded. However, the state updates incurred by these packets are then recorded in dedicated metadata fields. These metadata fields are recirculated for a second-pass processing, which eventually performs state updates. The recirculation continues until state write is fully completed and the lock is free.

The motivation behind is three-fold: (1) For most applications, the priority of state write is higher than data plane updates. This is because state write operations usually change processing strategies (e.g., reset a data structure or modify control policies after observing a specific event). Thus, by prioritizing state write operations, ApproSync guarantees that merely using one lock is sufficient for resolving concurrency conflicts. It also naturally avoids deadlocks since only state write operations can obtain the lock; (2) ApproSync does not involve complicated operations or require excessive memory, such that it can be easily implemented in programmable switches; (3) ApproSync bypasses the switch OS to achieve low latency. Thus, the number of recirculated state updates is small, making the overhead on performance and accuracy negligible. The limitation of ApproSync is that the recirculation compromises the update ordering. However, most applications (e.g., measurement [48, 49] and load balancing [52]) are insensitive to such reordering.

Summary. ApproSync retries all failed operations during state write, such that it mitigates state loss and maintains high accuracy. It preserves atomicity by preventing data plane packets from updating states during state write.

4 USE CASES

In this section, we present three cases that enhance real applications via the state synchronization of ApproSync.

4.1 Sketch Collector

Sketch is a family of algorithms that has been extensively adopted in network measurement to fit switch memory limitations [35, 37, 48, 49, 78]. A sketch algorithm maintains a compact data structure. Every packet updates several counters in the data structure. The control plane collects the data structure at the end of each time interval and obtains various statistics for network management such as anomaly detection. Backing by sound theoretical guarantees, sketch algorithms achieve both high resource efficiency and high accuracy. However, existing approaches collect sketch structures inefficiently. They utilize the switch OS to serialize the values of the counters used by sketches. This incurs high latency overhead and hinders fine-grained deployment (e.g., in tens of milliseconds) of sketch algorithms, which eventually hurts the responsiveness of network management.

To this end, we build a low-latency sketch collector on ApproSync. Its workflow contains two steps: (1) When a packet updates sketch values, the read handler inserts those updates to the hash table via Algorithm 1; (2) It monitors the divergence between the sketch values in the data plane and those in the control plane. When the divergence exceeds the threshold, it pushes latest sketch values to the control plane, which reconstructs sketch data structures. Our experiments show that the collector significantly reduces the collection latency while maintaining high accuracy (see Exp#8 in §6).

4.2 State Migration

State migration transfers state values from one switch to another. It forms building blocks of many management tasks ranging from network redundancy [28, 30, 43, 64] to scalable network functions [71, 75]. Existing state migration techniques, such as OpenNF [26] and P4NFV [34], interact with switches using TCP connections to migrate states in a loss-free manner. Take P4NFV as an example. It demands the source switch to collect state values from the switch ASIC to the switch OS via PCIe channels, and send them to the control plane via TCP. Then it sends state values to the switch OS of a destination switch, which updates the local state resided in the switch ASIC. However, P4NFV suffers from high latency overhead incurred by PCIe channels and TCP connections, which compromises migration timeliness.

In response, we build a fast state migration framework atop the low-latency and accurate state synchronization of ApproSync. The framework performs two steps: (1) The read engine continuously receives state updates from the source switch. It extracts values from state updates and records them in a state storage; (2) The write engine sends state updates to the write handler in the destination switch to update the state. The experimental results indicate that the state migration framework offers low-latency (a few milliseconds) and accurate (no state loss) state migration (see Exp#9 in §6).

4.3 UDP Flood Prevention

A UDP flood attack is a type of denial of service attacks. The attackers randomly generate numerous UDP packets that target random UDP ports of innocent hosts. In this way, they

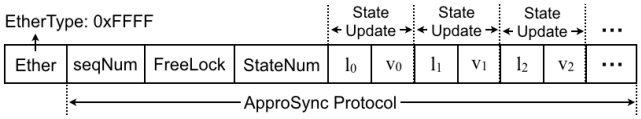


Figure 7: Protocol format of state transfer.

can easily exhaust the available resources in victims. Existing approaches use per-flow counters resided in switches to detect such attacks [15, 25]. They rely on the OS-based approach to transfer counter values to the control plane for attack detection and periodically reset counters to mitigate measurement errors. However, the OS-based approach incurs non-trivial latency overhead that delays attack detection. Also, due to the lack of atomicity, the reset operations will be affected by data plane packets, leading to false alarms.

To overcome the above problems, we develop an application, namely UDP flood prevention (UFP), based on ApproSync. UFP employs FlowRadar [48] on the switch ASIC to record flow statistics. At runtime, UFP uses a three-step procedure to detect attacks: (1) The read handler transfers values of FlowRadar counters to the control plane; (2) The read engine receives counter values and inputs them to UFP to extract flow statistics and detect attacks; (3) The write engine periodically resets FlowRadar counters with atomicity guarantees. Our experiments indicate that UFP can efficiently detect attacks with high accuracy (see Exp#10 in §6).

5 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.

Protocol. Figure 7 shows the format of state transfer, which follows the Ethernet header with a reserved protocol 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 design can support arbitrary validate sizes of state updates.

Data plane handlers. Figure 8 details the P4 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

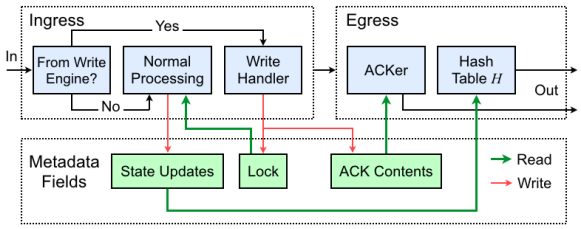


Figure 8: Workflow of the switch ASIC.

egress pipeline. We implement the hash table with match-action tables (MATs) and registers. The egress pipeline reserves a dedicated port, which pushes state updates to the control plane based on the processing results of hash table. For the write handler, ApproSync employs an MAT in the start 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 in the format as Figure 7, the write handler is invoked to handle it. We implement both state lock signal and ACK components with P4 metadata fields and registers.

Note that we use both an ingress pipeline and an egress pipeline rather than a single pipeline. This is because 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 controller. 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.

Control plane engines. We implement both read engine and write engine in C. Our implementation provides the interfaces for both the data plane (southbound APIs) and applications (northbound APIs). We implement the southbound APIs with DPDK [6] to eliminate the latency incurred by kernel stacks. We employ Redis [9] as the state storage and use HiRedis [5] to manage the storage. For the 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.

6 EVALUATION

In this section, we conduct experiments to evaluate our ApproSync prototype. We highlight our results as follows.

- ApproSync incurs less than 15% usage of memory and computational resources in switches (Exp#1).
- Compared to the OS-based approach, ApproSync achieves order-of-magnitude latency reduction in state read (Exp#2).
- Compared to *Flow [70], 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 degrade the 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).
- ApproSync completes loss-free state migration within 10 ms (Exp#9).
- ApproSync prevents UDP flood attacks in a timely and highly accurate manner (Exp#10).

6.1 Methodology

Platforms. We build a testbed comprising two 32×100 Gbps Barefoot Tofino switches [2] and six servers, each of which 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 of ApproSync on a dedicated server. In the data plane, we connect the two switches to compose a linear topologic, while using the remaining five servers as traffic testers. In our testbed, 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 [3] with 38M packets. We use PktGen [8] to replay the trace. In addition to the applications in §4, we consider another 16 stateful P4 applications listed in Table 1. In each experiment, we present the average after 100 runs.

Parameters. In our experiments, the hash table H has 2^{16} entries by default. We fix the time window w to 1 ms and use a $\beta = 1$, which are sufficient for all applications. For each application in Table 1, we obtain the size s of a state update and calculate the maximum emitted rate M as $\frac{c}{s}$, where c is a constant implying the capacity of a 40 Gbps link. Moreover, unless specified otherwise, ApproSync will automatically tune t to adapt to the rate of input traffic.

[1] "size" indicates the size (in bytes) of a state update.

Table 1: Stateful P4 applications used in §6.

Name	P4 LoC	# of counters	Size[1]
Packet counter (PC) [7]	265	2^{16}	6
Flowlet switching (FL) [7]	251	2^{14}	10
Malicious DNS domain detection (MD) [15]	358	3×2^{16}	4
Snort flowbits (FB) [15]	296	3×2^{16}	4
Affine LB (AL) [15]	345	2^{17}	4
DNS TTL change tracking (TC) [18]	357	3×2^{16}	6
DNS tunnel detection (TD) [18]	530	3×2^{16}	4
Stateful firewall (FW) [54]	349	2^{17}	4
FTP monitoring (FM) [54]	303	2^{16}	4
Heavy hitter detection (HH) [54]	310	2^{17}	6
Super-spreader detection (SS) [54]	313	2^{17}	4
Sampling based on flow size (FS) [54]	560	5×2^{16}	4
SYN flood detection (SF) [25]	313	2^{17}	4
DNS amplification mitigation (AM) [25]	360	2^{16}	4
UDP flood mitigation (UF) [25]	309	2^{17}	4
Elephant flows detection (EF) [25]	553	5×2^{16}	4

6.2 Microbenchmarks

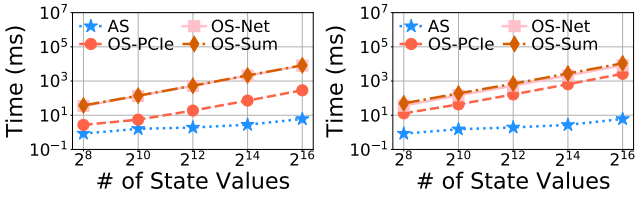
(Exp#1) Switch resource usage. This experiment measures the total usage of switch resources, including memory resources, computational resources, and match-action stages. Here, memory includes both SRAM and TCAM, while computational resources include meter ALUs (mALUs), stateful ALUs (sALUs), and very 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 all primitives. Table 2 shows that ApproSync uses less than 15% resources even when using all 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. We present more experimental results on Appendix A.

(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. We build the OS-based approach based on the interfaces exposed by our switches [2] and ZeroMQ [11]. 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 sum of the former two (OS-Sum). Figure 9 presents that the latency of OS-based approach exceeds one second when a state has 2^{16} values. When using 64-bit state, even reading state via PCIe channels takes at least tens of milliseconds. In contrast, the latency of ApproSync remains below 10 ms in all cases.

(Exp#3) Accuracy of state read. We evaluate the accuracy of ApproSync in state read. First, we validate that ApproSync can avoid link saturation via its rate control. Recall that a state update is transferred when the state divergence reaches

Table 2: (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%



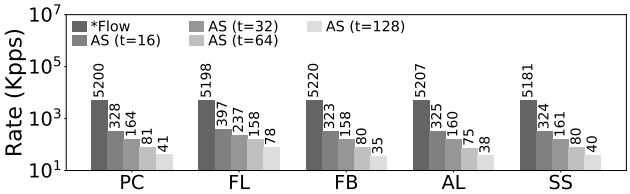
(a) 16-bit state value. **(b) 64-bit state value.**

Figure 9: (Exp#2) Performance of state read.

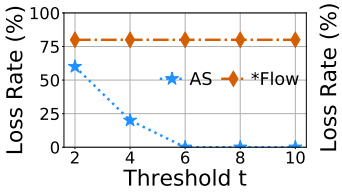
the threshold t. We replay our trace at 40 Gbps and employ t ranging from 16 to 128. Since the emitted rate of state updates depends on how an application updates state, we select five applications, in which every packet triggers an update, from Table 1. We compare ApproSync with *Flow [70], a traffic mirroring-based system that uses an LRU cache for rate control. We configure the LRU cache with the same configuration as the hash table of ApproSync. Figure 10(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. In contrast, ApproSync significantly reduces bandwidth consumption via its rate control. Its benefits increase as the growth of t. For example, it only incurs 35 Kpps for *Snort flowbits* (SF) [15] when t is 128.

Second, we validate that ApproSync can offer accurate state read via its rate control. 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 manually configure the read handler by varying its threshold t from 2 to 10. Figure 10(b) shows that ApproSync gradually reduces emitted rate as t increases, and avoids state loss when t exceeds 5. Then we repeat the experiment without any manual settings. Figure 10(c) shows that ApproSync (AS-Dyn) offers loss-free state read because its read handler 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.

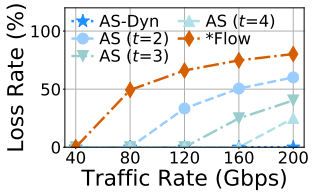
Third, we study the impact of hash table size on accuracy. We vary the size from 2^{14} to 2^{18} entries. We conduct the same experiment as above. Figure 10(d)-(e) show that ApproSync loses a few state updates (3.22%) when using 2^{14} entries. This is because hash collisions happen frequently given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



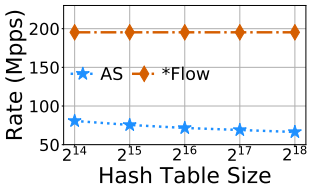
(a) Emitted rate of state updates.



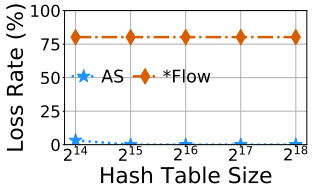
(b) Loss rate under different threshold t.



(c) Loss rate under different incoming traffic rates.

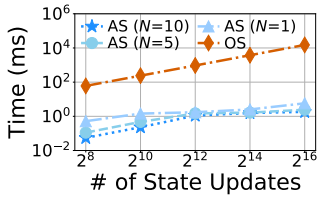


(d) Emitted rate with different table sizes.

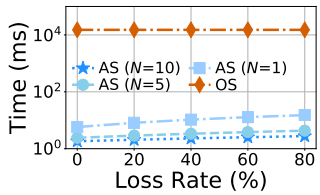


(e) Loss rate with different table sizes.

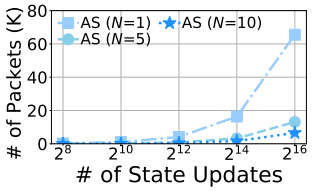
Figure 10: (Exp#3) Accuracy of state read.



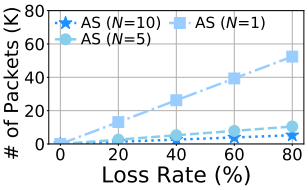
(a) Latency of state write without state loss.



(b) Latency of state write with state loss.



(c) # of pkts for state write.

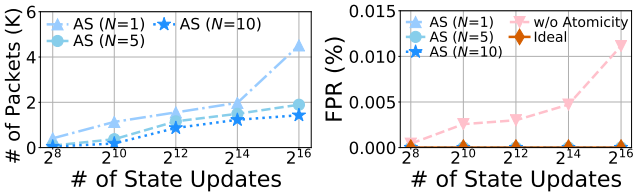


(d) # of retransmitted pkts.

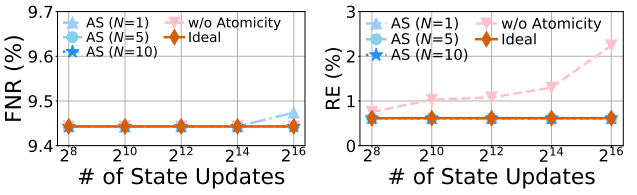
Figure 11: (Exp#4) Performance of state write.

can be totally avoided by setting a higher table size, e.g., 2^{16} entries are sufficient for ApproSync to avoid state loss for all the applications in Table 1 even with 200 Gbps traffic.

(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 11(a) shows that ApproSync



(a) # of out-of-order updates. (b) Impact on false positive.

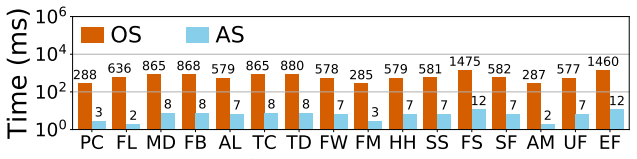


(c) Impact on false negative. (d) Impact on relative error.

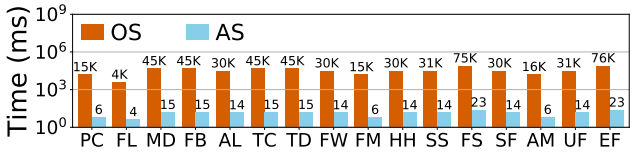
Figure 12: (Exp#5) Accuracy of state write.

Table 3: (Exp#6) Impact on packet forwarding.

Name	Thpt.	Latency	Thpt. cost	Latency cost
NoApproSync	39.99 Gbps	1073 ns	-	-
Only Read	39.99 Gbps	1123 ns	+0.0%	+4.6%
Only Write	39.99 Gbps	1101 ns	+0.0%	+2.6%
Overall	39.99 Gbps	1141 ns	+0.0%	+6.3%



(a) State read.



(b) State write.

Figure 13: (Exp#7) Application-perceived latency.

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 11(b) presents that even with 80% loss rate, ApproSync completes state write in a few milliseconds. In addition, 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).

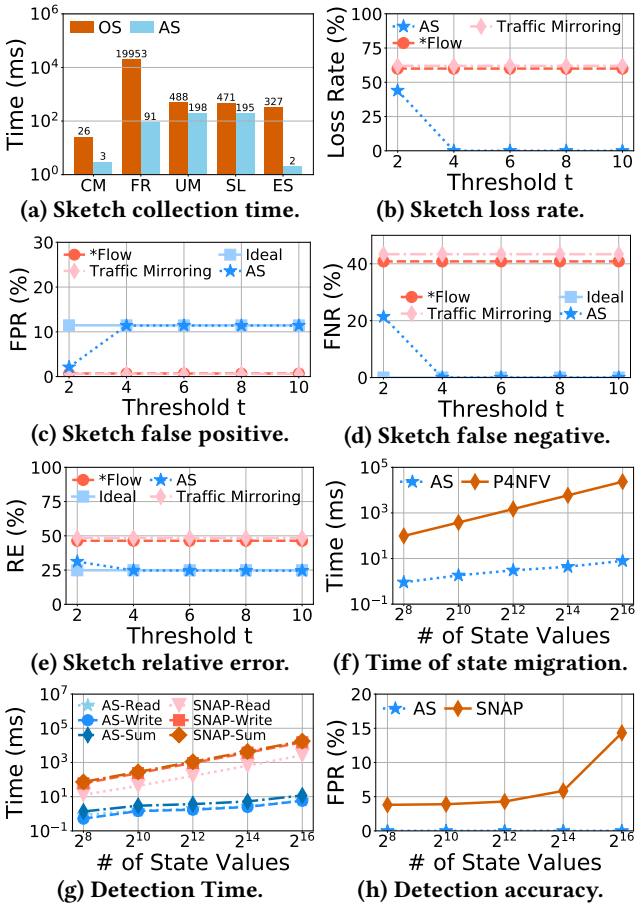
(Exp#5) Accuracy of state write. We measure the accuracy of ApproSync in state write. ApproSync recirculates state updates during state write to maintain atomicity, which brings two types of overheads: (1) the recirculation consumes a portion of switch bandwidth; (2) the recirculation affects the original order between state updates that may reduce accuracy. To quantify these overheads, we first count the number of out-of-order updates during state write. We deploy HashPipe [66], 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 and bandwidth consumption are small. Next, we examine the impact on HashPipe accuracy. Figure 12(b)-(d) show that: (1) the out-of-order updates only increase false negative rate and relative error by at most 0.2% and 0.03%, which is negligible; (2) compared to the OS-based approach, ApproSync offers highly accurate state write with atomicity guarantees.

(Exp#6) Impact on packet forwarding. Table 3 examines how ApproSync affects the 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 nearly zero throughput drop while adding the per-packet processing latency by at most 6.3%.

6.3 Real Applications

(Exp#7) Latency for stateful P4 applications. This experiment measures the application-perceived latency of state read and write operations of ApproSync. We implement the 16 stateful P4 applications in Table 1 with various complexity, resource consumption and state management patterns in Table 1. We measure the time of ApproSync when reading states from the counters used by applications and resetting counters. We compare ApproSync with the OS-based approach. Figure 13(a) shows that ApproSync completes state read within 12 ms, while the OS-based approach takes at least 285 ms. ApproSync brings more benefits for state write, shown in Figure 13(b). It requires at most 23 ms, while the OS-based approach requires several seconds.

(Exp#8) Performance and accuracy of sketch collector. We evaluate the performance and accuracy of the sketch collector built atop ApproSync. We consider five sketch-based solutions, Count-Min (CM) [22], FlowRadar (FR) [48], UnivMon (UM) [49], SketchLearn (SL) [37] and ElasticSketch (ES) [78]. We deploy these solutions on a switch and use the sketch collector to periodically collect sketch values from the



(a) Sketch collection time. (b) Sketch loss rate.
(c) Sketch false positive. (d) Sketch false negative.
(e) Sketch relative error. (f) Time of state migration.
(g) Detection Time. (h) Detection accuracy.

Figure 14: (Exp#8-10) Trade-offs in real cases.

switch. Figure 14(a) indicates that compared to the OS-based approach, ApproSync reduces the collection time by orders of magnitude. This implies future deployment of sketch-based solutions to identify network events with ApproSync.

Next, we inject traffic at 120 Gbps to examine the loss rate of state updates. Figure 14(b) presents the loss rate with respect to the threshold t that determines the emitted rate of state updates. We compare ApproSync against 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 its threshold $t = 2$. However, the state loss is completely eliminated as the threshold increases.

Finally, we qualify the accuracy of heavy hitter detection. In the interest of space, we only present the results of CM. We measure its false positive rate, false negative rate, and relative error in Figure 14(c)-(e), respectively. In addition to traffic mirroring and *Flow, we build an original version of CM without any state loss ("Ideal") as ground truth. We observe that traffic mirroring and *Flow achieve relatively small false positive rate. The reason is that false positives are caused by overestimating flows. Clearly, high loss rate mitigates overestimates so that traffic mirroring and *Flow have low false positive rate. However, state loss seriously improves false negative rate and relative error. In contrast, ApproSync achieves near-optimal accuracy closed to "Ideal".

(Exp#9) Performance and accuracy of state migration. We evaluate the state migration framework based on ApproSync. We measure the latency of migrating state values, the number of which varies from 2^8 to 2^{16}. Figure 14(f) compares our framework with P4NFV [34], which migrates states via the OS-based approach. We see that P4NFV takes nearly 23s for the migration of 2^{16} state values, while our framework completes within 10 ms. Also, ApproSync achieves loss-free state migration in all cases, which indicates its high accuracy.

(Exp#10) Performance and accuracy of UDP flood prevention. We evaluate UDP flood prevention (UFP) running on ApproSync. 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 [15] based on OS-based approach. Figure 14(g)-(h) 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.

7 RELATED WORK

State read. To shed bandwidth consumption in state read, prior solutions employ sampling techniques [10, 24, 32, 58, 62, 80]. However, sampling techniques inevitably degrade accuracy. Some recent studies exploit the processing capability of programmable switches to optimize state read. TurboFlow [69] treats state values as flow records, and then processes flow records in the switch OS. However, a switch OS fails to process multi-Tbps state updates and incurs high latency. In contrast, ApproSync bypasses the switch OS to avoid performance overhead in the switch OS. Marple [55] caches flow records in the switch ASIC before mirroring states to remote servers. However, it does not consider the actual impact of state updates. A state is evicted even its update is insignificant, seriously saturating link capacity (see Exp#3 in §6). ApproSync improves in-switch caching by adaptively controlling the trade-off between accuracy and bandwidth consumption. KeySight [81] aggregates packets based on packet processing behaviors to reduce overhead. Sonata [31] pre-processes packets in switches to reduce workloads in the control plane. However, these systems are designed to build independent applications, while ApproSync aims to complement arbitrary user-implemented applications.

State write. Commodity controllers modify states via TCP-based protocols such as OpenFlow [51], Thrift [1] and gRPC

[4], which require a switch OS for complicated processing. Applications built on these protocols (e.g., P4NFV [34]) suffer from high latency. Swing State [50] directly migrates state values in the data plane to achieve rapid state migration. However, its write operations 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 [12], machine learning systems [47, 76, 77], and stream processing systems [13, 36, 57]. BlinkDB [12] samples a subset of data to dynamically estimate the response time and error of a database query. JetStream [57] uses data aggregation and adaptive filtering to achieve trade-offs between accuracy and resource efficiency. AF-Stream [36] provides approximate fault tolerance in the content of stream processing. On the contrary, ApproSync exploits approximate techniques to achieve low-latency and accurate state synchronization between programmable switches and control plane.

8 CONCLUSION

We propose ApproSync, a state synchronization framework, with a primary goal of achieving state consistency between the data plane and control plane. ApproSync exploits approximate techniques to alleviate resource consumption and bound errors during state synchronization. It offers two types of synchronization for state read and state write while achieving low latency and high accuracy. We implement a ApproSync prototype atop Barefoot Tofino switches. Experiments with real-world traces and stateful P4 applications indicate that ApproSync outperforms existing solutions with order-of-magnitude latency reduction and higher accuracy.

REFERENCE

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

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

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

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

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

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

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

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

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

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

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

[12] 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*, pages 29–42. ACM, 2013.

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

[14] M. Allman and V. Paxson. On estimating end-to-end network path properties. *ACM SIGCOMM Computer Communication Review*, 29(4):263–274, 1999.

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

[16] 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*, 12(13):2325–2338, 2019.

[17] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

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

[19] 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*, 44(3):87–95, 2014.

[20] 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*, 43(4):99–110, 2013.

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

[22] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[23] Í. Cunha, R. Teixeira, N. Feamster, and C. Diot. Measurement methods for fast and accurate blackhole identification with binary tomography. In *IMC*, pages 254–266. ACM, 2009.

[24] 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*, volume 41, pages 254–265. ACM, 2011.

[25] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Bohatei: Flexible and elastic ddos defense. In *Security*, pages 817–832. USENIX, 2015.

[26] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. Opennf: Enabling innovation in network function control. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 163–174. ACM, 2014.

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

[28] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *SIGCOMM*, pages 350–361. ACM, 2011.

[29] M. G. Gouda and A. X. Liu. A model of stateful firewalls and its properties. In *DSN*, pages 128–137. IEEE, 2005.

[30] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *SIGCOMM*, pages 58–72. ACM, 2016.

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

[32] 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*, pages 71–85, 2014.

[33] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker. An evaluation of distributed concurrency control. *PVLDB*, 10(5):553–564, 2017.

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

[35] 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*, pages 113–126. ACM, 2017.

[36] Q. Huang and P. P. Lee. Toward high-performance distributed stream processing via approximate fault tolerance. *VLDB*, 10(3):73–84, 2016.

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

[38] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: a highly-scalable software-based intrusion detection system. In *CCS*, pages 317–328. ACM, 2012.

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

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

[41] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *SoCC*, page 9. ACM, 2012.

[42] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. Moon. Nba (network balancing act): A high-performance packet processing framework for heterogeneous processors. In *EuroSys*, page 22. ACM, 2015.

[43] S. G. Kulkarni, G. Liu, K. Ramakrishnan, M. Arumaithurai, T. Wood, and X. Fu. Reinforce: achieving efficient failure resiliency for network function virtualization based services. In *CoNEXT*, pages 41–53. ACM, 2018.

[44] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

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

[46] P. Laffranchini, L. Rodrigues, M. Canini, and B. Krishnamurthy. Measurements as first-class artifacts. In *INFOCOM*, pages 415–423. IEEE, 2019.

[47] 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*, pages 583–598, 2014.

[48] Y. Li, R. Miao, C. Kim, and M. Yu. Flowradar: a better netflow for data centers. In *NSDI*, pages 311–324, 2016.

[49] 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*, pages 101–114. ACM, 2016.

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

[51] 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*, 38(2):69–74, 2008.

[52] 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*, pages 15–28. ACM, 2017.

[53] J. C. Mogul and P. Congdon. Hey, you darned counters!: get off my asic! In *HotNet*, pages 25–30. ACM, 2012.

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

[55] 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*, pages 85–98. ACM, 2017.

[56] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized zero-queue datacenter network. *ACM SIGCOMM Computer Communication Review*, 44(4):307–318, 2015.

[57] 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*, pages 275–288, 2014.

[58] 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*, volume 44, pages 407–418. ACM, 2014.

[59] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *PVLDB*, 7(10):821–832, 2014.

[60] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren. Passive realtime datacenter fault detection and localization. In *NSDI*, pages 595–612, 2017.

[61] P. Sarolahti, M. Kojo, and K. Raatikainen. F-rto: an enhanced recovery algorithm for tcp retransmission timeouts. *ACM SIGCOMM Computer Communication Review*, 33(2):51–63, 2003.

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

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

[64] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, et al. Rollback-recovery for middleboxes. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 227–240. ACM, 2015.

[65] 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*, volume 45, pages 183–197. ACM, 2015.

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

[67] D. Skeen. Nonblocking commit protocols. In *SIGMOD*, pages 133–142. ACM, 1981.

[68] D. Skeen. A quorum-based commit protocol. Technical report, Cornell University, 1982.

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

[70] 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*, pages 823–835. USENIX, 2018.

[71] C. Sun, J. Bi, Z. Meng, T. Yang, X. Zhang, and H. Hu. Enabling nfv elasticity control with optimized flow migration. *IEEE Journal on Selected Areas in Communications*, 36(10):2288–2303, 2018.

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

[73] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. Midea: a multi-parallel intrusion detection architecture. In *CCS*, pages 297–308. ACM, 2011.

[74] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. In *ACM SIGCOMM computer communication review*, volume 39, pages 303–314. ACM, 2009.

[75] Y. Wang, G. Xie, Z. Li, P. He, and K. Salamatian. Transparent flow migration for nfv. In *ICNP*, pages 1–10. IEEE, 2016.

[76] 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*, pages 381–394. ACM, 2015.

[77] 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*, 1(2):49–67, 2015.

[78] 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*, pages 561–575. ACM, 2018.

[79] 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*, 8(3), 2014.

[80] 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*, pages 1–10. IEEE, 2017.

[81] 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*, pages 291–301. IEEE, 2018.

[82] 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*, volume 45, pages 479–491. ACM, 2015.

APPENDIX A: MORE EXPERIMENTS

(Exp#11) Switch resource consumption of ApproSync. This experiment measures the resource consumption of ApproSync. We deploy the sixteen applications in Table 1 and the seven applications used by Exp#8-#10 in §6.3, including the sketches used by the sketch collector, the state migration framework (SM), and UDP flood prevention (UFP), on a programmable switch, respectively. We quantify the resource consumption of ApproSync by measuring the switch resource usage with and without ApproSync. As shown in Table 4, ApproSync consumes less than 15% computation resources and no more than 7% memory resources. This indicates that ApproSync is resource-efficient and adds negligible overheads to the deployment of data plane programs.

Table 4: (Exp#11) Resource usage of ApproSync applications.

Name	SRAM		TCAM		mALU	
	w/o ApproSync	w/ ApproSync	w/o ApproSync	w/ ApproSync	w/o ApproSync	w/ ApproSync
PC	1.88%	6.88% (+5%)	0%	0% (+0%)	2.08%	16.66% (+12.50%)
FL	0.94%	7.81% (+6.87%)	0%	0% (+0%)	2.08%	16.67% (+14.59%)
MD	2.81%	9.69% (+6.88%)	0%	0% (+0%)	6.25%	20.83% (+14.58%)
FB	1.88%	8.75% (+6.87%)	0%	0% (+0%)	4.17%	18.75% (+14.58%)
AL	1.88%	8.75% (+6.87%)	0%	0% (+0%)	4.17%	18.75% (+14.58%)
TC	3.65%	10.52% (+6.87%)	0%	0% (+0%)	6.25%	20.83% (+14.58%)
TD	2.81%	10.52% (+7.71%)	0%	0% (+0%)	6.25%	20.83% (+14.58%)
FW	1.04%	8.12% (+7.08%)	0%	0% (+0%)	4.17%	18.75% (+14.58%)
FM	0.94%	7.81% (+6.87%)	0%	0% (+0%)	2.08%	16.67% (+14.59%)
HH	1.88%	8.75% (+6.87%)	0%	0% (+0%)	4.17%	18.75% (+14.58%)
SS	1.88%	8.75% (+6.87%)	0%	0% (+0%)	4.17%	18.75% (+14.58%)
FS	4.69%	11.56% (+6.87%)	0%	0% (+0%)	10.42%	25.00% (+14.58%)
SF	1.88%	8.75% (+6.87%)	0%	0% (+0%)	4.17%	18.75% (+14.58%)
AM	0.94%	7.81% (+6.87%)	0%	0% (+0%)	2.08%	16.67% (+14.59%)
UF	1.88%	8.75% (+6.87%)	0%	0% (+0%)	4.17%	18.75% (+14.58%)
EF	2.81%	9.68% (+6.87%)	0%	0% (+0%)	6.25%	20.83% (+14.58%)
CM	0.31%	7.18% (+6.87%)	0%	0% (+0%)	2.08%	16.66% (+14.58%)
FR	39.38%	46.25% (+6.87%)	0%	0% (+0%)	50.00%	64.58% (+14.58%)
UM	38.85%	45.72% (+6.87%)	0%	0% (+0%)	68.75%	83.33% (+14.58%)
SL	38.85%	45.72% (+6.87%)	0%	0% (+0%)	68.75%	83.33% (+14.58%)
ES	1.98%	8.85% (+6.87%)	0%	0% (+0%)	18.75%	33.33% (+14.58%)
SM	2.81%	9.68% (+6.87%)	0.35%	0.35% (+0%)	4.17%	18.75% (+14.58%)
UFP	39.38%	46.25% (+6.87%)	0%	0% (+0%)	50.00%	64.58% (+14.58%)

Name	sALU		VLIW		Stage	
	w/o ApproSync	w/ ApproSync	w/o ApproSync	w/ ApproSync	w/o ApproSync	w/ ApproSync
PC	0%	0% (+0%)	0.52%	4.69% (+4.17%)	8.33%	100% (+91.67%)
FL	0%	0% (+0%)	0.52%	4.69% (+4.17%)	16.66%	100% (+83.34%)
MD	0%	0% (+0%)	1.30%	4.69% (+3.39%)	33.33%	100% (+66.67%)
FB	0%	0% (+0%)	0.78%	4.95% (+4.17%)	16.66%	100% (+83.34%)
AL	0%	0% (+0%)	1.30%	4.69% (+3.39%)	25%	100% (+75%)
TC	0%	0% (+0%)	1.30%	5.21% (+3.91%)	25%	100% (+75%)
TD	0%	0% (+0%)	2.86%	5.21% (+2.35%)	50%	100% (+50%)
FW	0%	0% (+0%)	1.56%	4.69% (+3.13%)	25%	100% (+75%)
FM	0%	0% (+0%)	0.26%	4.69% (+4.43%)	8.33%	100% (+91.67%)
HH	0%	0% (+0%)	0.78%	4.69% (+3.91%)	25%	100% (+75%)
SS	0%	0% (+0%)	0.78%	4.69% (+3.91%)	25%	100% (+75%)
FS	0%	0% (+0%)	3.65%	6.25% (+2.60%)	50%	100% (+50%)
SF	0%	0% (+0%)	0.78%	4.69% (+3.91%)	25%	100% (+75%)
AM	0%	0% (+0%)	1.82%	4.95% (+3.13%)	41.66%	100% (+58.34%)
UF	0%	0% (+0%)	1.04%	4.95% (+3.91%)	25%	100% (+75%)
EF	0%	0% (+0%)	2.60%	5.47% (+2.87%)	50%	100% (+50%)
CM	0%	0% (+0%)	1.04%	3.87% (+2.83%)	25%	100% (+75%)
FR	0%	0% (+0%)	3.12%	6.51% (+3.39%)	91.66%	100% (+8.34%)
UM	0%	0% (+0%)	3.39%	5.76% (+2.37%)	100%	100% (+0%)
SL	0%	0% (+0%)	3.39%	5.76% (+2.37%)	100%	100% (+0%)
ES	0%	0% (+0%)	3.91%	6.51% (+2.60%)	100%	100% (+0%)
SM	0%	0% (+0%)	2.34%	6.25% (+3.91%)	41.66%	100% (+58.34%)
UFP	0%	0% (+0%)	3.12%	6.51% (+3.39%)	91.66%	100% (+8.34%)