Millions of Low-latency State Insertions on ASIC Switches

TOMMASO CAIAZZI, Roma Tre University, Italy and KTH Royal Institute of Technology, Sweden
MARIANO SCAZZARIELLO, KTH Royal Institute of Technology, Sweden
MARCO CHIESA, KTH Royal Institute of Technology, Sweden

Key-value data structures are an essential component of today's stateful packet processors such as load balancers, packet schedulers, firewalls, and more. Supporting key-value data structures entirely in the data plane of an ASIC switch would result in high throughput, low-latency, and low energy consumption. Yet, today's key-value implementations on ASIC switches are ill-suited for stateful packet processing as they support only a limited amount of flow-state insertions per second into these data structures. In this paper, we present SWITCHAROO, a mechanism for realizing key-value data structures on programmable ASIC switches that supports both high-frequency insertions and fast lookups *entirely* in the data-plane. We show that SWITCHAROO can be realized on state-of-the-art programmable ASICs, supporting millions of flow-state insertions per second with only a limited amount of packet recirculation.

CCS Concepts: • **Networks** → **Programmable networks**; **In-network processing**; *Middle boxes / network appliances.*

Additional Key Words and Phrases: stateful NFs, programmable switches, cuckoo hashing, ASIC switches, flowlet routing

ACM Reference Format:
Tommaso Caiazzi, Mariano Scazzariello, and Marco Chiesa. 2023. Millions of Low-latency State Insertions on ASIC Switches. *Proc. ACM Netw.* 1, CoNEXT3, Article 22 (December 2023), 23 pages. https://doi.org/10.1145/3629144

1 INTRODUCTION

Stateful packet processing is an essential part of any modern network system. A stateful packet processor stores, updates, and fetches that state that is needed to correctly process network traffic. Load balancers [3, 12, 17], NATs [12], anomaly detection systems [33], and many other network functionalities rely on the ability of the data plane to support stateful packet processors. The state needed to process packets is typically stored using *key-value* data structures, which map traffic class identifiers (*e.g.,* the 5-tuple of a connection) to a value (*e.g.,* the number of received packets).

Supporting key-value data structures on ASIC hardware devices has been challenging because of the complexity of realizing key-value data structures at data-plane speed. Key-value data structures typically support either constant-time lookups or constant-time insertions into its data structure. For instance, cuckoo-hash tables [22] support constant-time lookups yet worst-case linear time insertions whereas chained hash tables support the exact opposite [16]. Implementing complex non-constant-time operations at data-plane speed on ASIC hardware devices is cumbersome due to the strict time constraints set for processing a single packet [4]. Today's ASIC switches support frequent lookup operations at data-plane speed, although they depend on the slower CPU-based control

Authors' addresses: Tommaso Caiazzi, caiazzi@kth.se, Roma Tre University, Italy and KTH Royal Institute of Technology, Sweden; Mariano Scazzariello, marianos@kth.se, KTH Royal Institute of Technology, Sweden; Marco Chiesa, mchiesa@kth.se, KTH Royal Institute of Technology, Sweden.

2834-5509/2023/12-ART22
https://doi.org/10.1145/3629144

plane for managing insertions [17]. Even next-generation ASICs [10] rely on the control-plane CPU to perform table modifications from the data plane.

Delegating insertions to the control-plane presents two drawbacks. First, control-plane insertions are slow with up to one millisecond of latency [17]. Such latencies are unsuitable for datacenter environments where end-to-end latencies are confined to a few microseconds [13]. Second, the control-plane cannot perform many insertions per second. To the best of our knowledge, a multi-Tbps switch supports at most around 100 K insertions per second [34]. However, even a small 10 Gbps Internet traffic trace today requires >100 K insertions per second into a key-value structure to keep per-flow state [6]. Unfortunately, many network functions today require both low-latency and high-frequency insertions that go well beyond the performance of control-plane-based approaches. For instance, we show that a Flowlet datacenter load balancer [30] may require more than 100 million insertions per second in order to load balance terabits per seconds of traffic. Even worse, we also show that implementing Flowlet using probabilistic data structures, as opposed to an exact key-value data structure, may result in up to 65× higher memory consumption to achieve similar flow completion times.

In this paper, we present SWITCHAROO[1], a mechanism to support lookups and insertions into a key-value data structure entirely in the data plane. SWITCHAROO supports millions of insertions per second as well as sub-microsecond insertions into the key-value data structure. SWITCHAROO consists of a cuckoo hash mechanism that implements the insertion operation *entirely* in the *data plane* of a programmable switch. As insertions into a cuckoo hash are not constant-time operations, we rely on "recirculating" packets within the switch until the insertions succeed.

This simple idea comes with a large number of challenges that are unique to a data-plane implementation. First, the amount of computational resources that can be used to implement the cuckoo hash logic on an ASIC switch is constrained. Second, recirculating packets may lead to simultaneous insertions from multiple flows that results in *inconsistent* flow state and packets being transmitted *out-of-order*. Recirculating packets also increases bandwidth consumption. We present a cuckoo-hash table design that is amenable to ASIC switch targets and supports both consistent and in-order packet processing with limited packet recirculation overheads.

We evaluate SWITCHAROO on a multi-Tbps ASIC switch, *i.e.,* Intel Tofino [11], showing that our solution supports millions of insertions per second as well as μs-insertion latencies with limited levels of packet recirculation.

To summarize, our contributions are:

- We show a quantitative analysis of the benefits of using a key-value data structure in the data plane for implementing a common network function such as a stateful load balancer. We focus on the Flowlet load balancer use case, showing the need for >100 million low-latency insertions per second. We also show a 65× higher memory requirement for implementing Flowlet using probabilistic data structures.
- To the best of our knowledge, we are the first ones to support key-value lookups & insertions in the data plane while guaranteeing state consistency and packet ordering.
- We present SWITCHAROO, a key-value mechanism that preserves the order of packets and prevents inconsistencies during transient updates in a provable manner.
- We demonstrate through the implementation on a multi-Tbps ASIC switch the ability of SWITCHAROO to support millions of low-latency insertions in the data plane.
- We publicly release all the code for running SWITCHAROO on the Tofino programmable switch [5].

[1]"Switcharoo" is a colloquial term derived from 'switcheroo', informally referring to a situation where two things swap.

2 BACKGROUND AND MOTIVATION

We now provide a minimal background on key-value data structures and we motivate the need to support high-frequency and low-latency insertions.

Minimal background on key-value data structures. Key-value data structures are an abstract data type that stores pairs of (k, v) elements, where k is a key and v is a value associated with the key. They support four main operations: insertion of a pair (k, v) into the data structure, deletion of an element given a key, the lookup of the value v of an element given a key k, or the update of the value v for a key k. Key-value data structures can be realized through different implementations that strike different trade-offs with respect to the computational time of those four operations.

Chained hash tables support fast insertions in $O(1)$ time but slow lookups & deletion & updates in worst-case $O(n)$ time [18]. Conversely, cuckoo-hash tables support fast lookups & deletions & updates in $O(1)$ while slow insertions in $O(n)$ [18, 22]. Key-value data structures are a key component of stateless and stateful network functions.

Stateless network functions run on ASIC switches with fast lookups and slow insertions. Today's networks rely on a large variety of network functions to steer, process, and analyze network traffic. Basic network functions are *stateless*, such as IP routing, static transport port filtering, or ECMP load balancing. In stateless network functions, the control plane pre-installs into the data plane a set of forwarding rules that are matched by the packets. These rules are installed into either a key-value data structure (for exact matches) or a TCAM (for wildcard matches). Importantly, the action applied to a packet does *not* depend on the previously forwarded packets, *e.g.,* it does not matter for IP routing what previous packets have been forwarded.

Stateless network functions have long been running on ASIC switches, which operate at tens of terabits per second and are orders of magnitudes more energy-efficient than general-purpose CPUs [3]. These switches support billions of lookups per second and around 100 K insertions & deletions & updates per second [34] through their control planes in response to updates from the data plane, *e.g.,* in case of link failures, BGP withdrawals, and more. One common type of key-value data structure that is used for exact matches on ASIC is a cuckoo-hash table as it supports fast (more common) lookups and slow (more rare) insertions from the slower control-plane.

Stateful network functions handle state at data-plane speed. More advanced network functions are *stateful*, such as power-of-two-choices load balancers, per-flow counters, NATs, intrusion detection systems, per-flow packet schedulers, and more. With stateful functions, the data plane stores and updates a data structure (typically a key-value) that keeps the state needed to process packets. The processing of a packet action in this case *depends* on the previously processed packets. For example, consider a network function that load balances incoming connections towards a pool of servers in a round-robin manner. When the load balancer receives the first packet of a flow, it assigns it a server, stores this assignment in a key-value data structure through a (possibly fast) *insertion*, and forwards the packet. Whenever the load balancer receives again a packet belonging to the same connection, it will *lookup* into the key-value data structure and forward the packet to the assigned server.

Stateful network functions are hard to realize on ASIC. ASIC implementations must abide to strict timing budget constraints, which makes it difficult to implement operations that have unbounded worst-case time complexity (*i.e.,* non-constant time). Unfortunately, all existing key-value data structures cannot support both lookups and insertions in constant time. Performing insertions from the CPU of the switch may take up to 1 ms and only 100 k insertions per second may be supported [17, 34]. This latency penalty is not acceptable for a large variety of use cases, including intra- and inter-datacenter communication, 5G, and more. Some works rely on bloom

filters to store state while the insertions take place through the CPU [17]. While this technique avoids the 1 ms penalty, it still suffers from the limited number of insertions per second that can be performed on an ASIC switch (around 100 thousand [34]). The three main challenges to support a key-value data structure in the data plane are therefore *i)* the high-volume of flow-state insertions required to keep track of all flows on a high-speed switch, *ii)* the low latency required to insert the flow state, and *iii)* the constraints on the available memory on the switch, which may not suffice to store the state for all flows (even assuming the switch could support high-frequency insertions). In the following, we discuss these aspects using Flowlet routing [30] as an example.

Example of network functions with high-volume insertions: Flowlet routing [30]. We argue that ASIC switches should support significantly higher rates of insertions into their key-value data structures. Consider a 25.6 Tbps Broadcom Tomahawk 4 switch that processes 16 billion packets per second [23, 32]. Let us assume we want to implement Flowlet on that switch. Flowlet [30] is a network function that selects an outgoing port on the switch for a burst of packets belonging to the same connection, where a burst is defined by a time expiration threshold set to 50 μs. Each burst of the same flow may be forwarded to a different port. The number of packets received within a 50 μs time interval at 25.6 Tbps is 160 k (assuming 1 kB average packets). Based on datacenter traffic statistics [35], each flow on average generates around 300 packets, which translates to roughly 10 million insertions per second. The size of the key-value data structure would be roughly $\frac{160\,k}{300} = 533$ elements, occupying roughly 10 KB of memory on the switch (13 B for the 5-tuple and 4 B for the Flowlet state), which is almost negligible [10]. When each flow contains on average fewer packets in a burst, the number of insertions increases. For instance, a recent work has shown that a cloud service receives on average 6 packets of the same flow within 64μs [8]. In this case, the number of insertions raises to 533 million per second. This amount of insertions represents 3% of a 25.6-Tbps switch forwarding throughput in bits and 16% in packets per second.

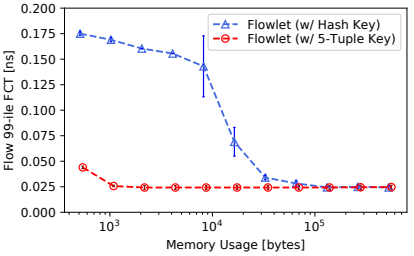


Fig. 1. FCTs using a hash table or a key-value data structure for Flowlet.

Case study: a key-value data structure in the data plane would reduce the memory requirements for implementing Flowlet routing by 65×. One may wonder whether there is a need to use an exact key-value data structure to implement Flowlet instead of relying on an hash table that embrace collisions, *i.e.,* if two flows collide on the same entry, they are treated as the same flow. To understand what is the impact of using a key-value data structure instead of an hash table, we use ns3 [21, 36] to simulate a small 2-layer datacenter with 8 servers and 4 leaf/core switches (as in Hermes [35]) in which each switch performs Flowlet routing. Fig. 1 shows the results of five runs. We perform two experiments, the first (blue line) uses a hash table indexed using the 5-tuple hash to store the state associated to flows, while the second (red line) uses a key-value data structure indexed using the full 5-tuple. For each experiment, we compute the Flow Completion Time (FCT) as a function of the size of the respective data structure (in bytes), showing that with a key-value of about 1 KB we have the same FCT as using a hash table that occupies 65 KB, leading to a memory reduction of 65×.

The number of insertions per second depends on the amount of memory on the switch and the duration of the state. Many stateful network functions would benefit from switches that support millions of low-latency insertions per second including packet schedulers, transport optimizers for TCP/UDP/QUIC, network telemetry, in-network computing, 5G, and beyond.

Clearly, there is a trade-off between the amount of memory available on the switch and the amount of insertions that makes sense to support. Consider a switch that has memory to store at most m key-value pairs for a network function that needs to store state for exactly t seconds. Then, the maximum amount of state insertions that makes sense to support is given by $\frac{m}{t}$. For instance, consider a switch that has 20 MB of memory and two load balancer functions that store state (of 18 bytes) for 1 and 30 seconds respectively. In the worst-case, the load balancer storing state for 30 seconds would need to support at most 37 K insertions per second while the load balancer storing state for 1 second would need to support 666 K insertions per second. Clearly, the shorter the duration for which the state must be stored, the higher the potential number of insertions a switch needs to support.

There are a large amount of network functions that store state for shorts amounts of time. Advanced packet schedulers such as Reframer [8] need to schedule packets in batches of 100μs at the per-flow level. QUIC optimizers, such as jumbo frame builders, need to merge packets of the same flow into larger packets. Network load balancers (*e.g.,* Flowlet [30]) as well as cloud load balancers are other examples that require higher number of insertions per second. Distributed storage systems such as Pegasus [15] require fast key-value data structures for load balancing among replicas located on different servers. Finally, in-network aggregation of data is another function that requires to store state for short amounts of time. This is the case for machine learning (*e.g.,* SwitchML [26]), Map-Reduce (*e.g.,* DAIET [25]) or sensor IoT networks [31].

Goal. In this work, we ask the following question:

> *"Can we support the four operations (i.e., lookup, insert, update, delete) of a key-value data structure entirely in the fast data plane of an ASIC switch?"*

3 SYSTEM DESIGN

We now present Switcharoo, a mechanism to implement key-value data structures entirely in the data plane of a programmable ASIC. We set the following requirements:

- **Fast lookups**, as forwarding packets is essential on a switch, requiring top-speed support.
- **High-frequency insertions**, as required by advanced network functions, *e.g.,* packet schedulers.
- **Fast deletion of entries**, to make efficient utilization of the constrained switch memory.
- **Packets ordering**, as packets of the same flow should not get re-ordered when leaving the switch.

3.1 Challenges

We now outline the key research challenges Switcharoo tackles to meet these system requirements.

Challenge #1: ASIC switch constraints. Realizing complex logic on an ASIC switch is challenging. In this work, we tailor our mechanism to fit real-world constraints that exist for P4-enabled ASIC programmable switches [9]. The closest data structure resembling a key-value data structure on such switches is a *table*, which is typically implemented as a cuckoo hash where insertions are performed by the slower CPU of the switch. A different data structure available on programmable ASIC switches is an *array register*, where the elements of the array can be accessed only through an index of the array (*i.e.,* no lookup based on a key). Register arrays implement a transactional memory that can be read/written at data-plane speed, *i.e.,* the next process packet will see the updated state. There exist three main constraints on what can be implemented using registers on programmable ASIC switches: (i) *single-element-access*: a packet can only access one element of the entire array; (ii) *single-register-access*: a packet can access a register only once; (iii) *registers-dependency*: access to the registers must define a partial order for all packets, *i.e.,* if register r_1 must be accessed before

register r_2, then r_2 cannot be accessed before r_1 by any other packet. Any logic not respecting the above constraints require to *recirculate* a packet through the switch and be processed a second time (which represents a bandwidth overhead).

We start from cuckoo-hash tables and tackle the slow insertion problem. The main challenge tackled within this work is how to build a key-value data structure using register arrays. Lookups are the most important operation and should be supported without recirculating packets. Chained hash tables [18] implement lookups by iterating over a list, which may potentially require to recirculate packets, so we do not rely on them. Linear-probing hash tables [7, 18] perform a lookup by accessing two consecutive elements in an array, which violates a constraint on registers and therefore requires recirculation. Conversely, cuckoo-hash tables perform a lookup by accessing two independent arrays using two distinct hash functions, which is possible on programmable switches. We therefore focus on *realizing cuckoo-hash tables on ASIC programmable switches.*

Additional background on cuckoo-hash tables. A minimal cuckoo-hash table consists of two arrays accessed using two hash functions. A *lookup* operation of a key accesses the elements pointed by the two hashes in the two arrays, respectively, and verifies if it finds the searched key. An *insertion* of a key-value (k, v) computes the two hashes of key k and checks if there is an empty element in any of the two arrays where it can insert the key-value pair. If this is the case, the insertion operation terminates. If this is not the case, the cuckoo-hash table performs a *swap* operation: it inserts (k, v) in one of the two arrays and extracts the key-value pair (k', v') that was stored there. It then tries to insert the extracted (k', v') into the other array. If the insertion succeeds, the insertion terminates. A new swap operation is performed otherwise.

Challenge #2: Handling transient states during insertions. Multiple swapping operations may be performed to insert a single element. This means that array registers may be accessed multiple times and, therefore, packets must be recirculated to implement insertions. A key challenge introduced by recirculating a packet is that this operation is not instantaneous. A switch may receive other data packets while recirculating a swapping operation, which may lead to inconsistent transient states. For example, the state of an existing flow will not be present in the key-value data structure while it is swapped with a recirculation. Packets belonging to that flow not matching any state should not assume that a new state must be created. This may result in security problems or performance degradation. Some existing switches lock the data structure while performing updates, which may lead to high packet drops [19]. Some other systems, *e.g.,* Lucid [28] implement cuckoo-hash tables insertions by recirculating packets, however, do not guarantee state consistency during insertions, swaps, or state updates (see the related work in § 6). *We need to devise the design of cuckoo-hash tables to carefully handle these switch-specific transient states*, described in § 3.3.

Challenge #3: Packets may leave the switch out-of-order. Another challenge introduced by recirculating packets is that they might be reordered while being processed by the switch. In fact, the switch guarantees that packets are processed sequentially in the pipeline, but recirculation may break this property since recirculated packets are considered as new packets by the receiving port. Avoiding packet reordering is a crucial property for any packet processor, since it leads to retransmissions, increased latency, and jitter on the end hosts. *We need to design a mechanism to guarantee that packets within a flow are sent in the same order as they are received*, described in § 3.4.

Challenge #4: The risk of infinite loops. Recirculating packets may lead to forwarding loops for two main reasons: *i)* infinite-loop swap insertions or *ii)* packet drops. In the first case, an insertion in a cuckoo-hash table may lead to a sequence of swap operations that is a loop, *i.e.,* the insertion never succeeds. Cuckoo-hash tables solve this problem by changing the hash functions and trying to re-insert all elements. The key observation here is that in a data structure with millions of

insertions per second and state that is kept for a few microseconds, the content of the hash table is constantly changing (on average every hundreds of nanoseconds). Therefore, *we expect that loops would likely break while performing insertions*. In the second case, packet drops may easily make any auxiliary data structure for implementing a cuckoo hash inconsistent. For instance, if a swap packet gets dropped, its state will be lost but existing packets may believe that the state is still recirculating and would recirculate forever. Similarly, if a data packet gets dropped, the subsequent packet in the flow may be recirculated to respect packet ordering (as the auxiliary data structures may be unaware of the packet drop). *We need to prevent or terminate forwarding loops to avoid a cascade-effect of packet drops.*

3.2 SwitchAroo Cuckoo Hashing

We now present a high-level overview of SwitchAroo: a system that implements a cuckoo-hash table at data-plane speed while supporting millions of insertions (and deletions) per second, guaranteeing packet ordering within each flow. SwitchAroo consists of the following data structures: (*i*) a cuckoo-hash table with two arrays that stores states associated to flows, (*ii*) a countable bloom filter to handle transient states, and (*iii*) three counter arrays that guarantee packet ordering within a flow.

We index the two arrays in SwitchAroo using two distinct hash functions that are fed with the flow identifier of a packet (*e.g.,* a TCP/IP 5-tuple). Each element in the table contains the flow identifier, the value of the state associated to that flow, and an expiration timeout that we use to determine when the switch can remove the state from the data structure.

We now describe how we implement the main operations of SwitchAroo on a cuckoo-hash table: lookups, update, insertions (including swaps), and deletions.

Lookup/Update. We implement lookups exactly as defined in any cuckoo-hash table. When a packet arrives, we check whether the state for that packet exists in any of the two arrays using two distinct hash functions. If the state exists, we read the state and forward the packet. If the state does not exist, two cases are possible: we either (*i*) need to generate a new state for the flow or (*ii*) we are currently inserting/swapping the state that should be applied to this packet. An update is like a lookup but also modifies the value.

Insertions. We implement insertions using special ad-hoc packets that contain metadata about the key-value pair that needs to be inserted (see Fig. 2). The switch inserts the state in the first array of the cuckoo-hash table. If the location is not empty (there is already some state), the switch *swaps* the existing element with the one that must be inserted and repeats an insertion operation for the swapped element into the second array. If there is again a swap operation in the second array, the switch performs a new insertion in the first array by recirculating the packet. These insertion operations are repeated (eventually with recirculation) until the insertion does not require to swap an element (*i.e.,* the location in the cuckoo table is either empty or there is an expired element).

Supporting selective flow insertions. In certain scenarios, one may want to keep track of states only for specific/important flows. SwitchAroo accommodates this requirement through a selective flow insertion mechanism. It is possible to specify which flows to monitor (*e.g.,* configuring a match-action table), allowing to reduce the size of the deployed cuckoo-hash table and lowering both switch memory and recirculation bandwidth utilization.

Supporting different state deletion policies. There are two main ways to remove elements from the table using expiration timeouts: (*i*) based on when the state has been generated (*e.g.,* useful for packet schedulers), or (*ii*) based on the last received packet (*e.g.,* useful for Flowlet). Our system supports both ways of updating expiration timeouts. To remove an entry, we can either rely on

existing packets that match the entry in the cuckoo hash or we can implement a timer-wheel data structure that is a simple queue, which is realizable on programmable ASICs [20].

3.3 Handling Transient States During Insertions

We now discuss the main challenge of handling transient states during state insertions/swaps. When a packet does not match any entry in the cuckoo-hash data structure, we may need to generate a new state (*i.e.,* it is the first packet of a flow) or not (*i.e.,* we have already triggered an insertion/swap for that flow, which is in progress through a recirculation). To determine this condition, we introduce an additional data structure, called Transient-States, which keeps track of the flows whose state is currently being inserted or swapped. We implement Transient-States using a countable bloom filter with a *single* hash (see Fig. 2). A bloom filter returns no false negatives: if the bloom filter says that there is not an insertion in progress for a flow, it means that we can generate a new state without causing inconsistencies, *i.e.,* two states for the same flow existing at the same time. If instead, the bloom filter returns a value different from zero, then it means that the state associated to the incoming flow *may* currently be inserted or swapped. We conservatively recirculate the packet with the expectation that the packet will match this state after it has been recirculated. If the state for processing that packet has never been created, recirculating a packet will simply delay the generation of this state, which will be created only when the packet does not match any state in the cuckoo table and the bloom filter returns zero (*i.e.,* there is not an ongoing insertion of the state for the flow of the packet).

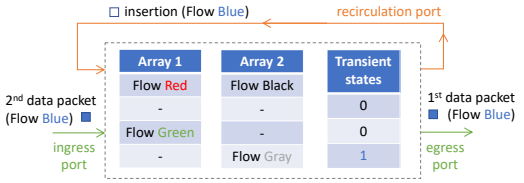


Fig. 2. Handling transient states.

Keeping track of transient states. To signal that a state is being swapped from the second to the first array, each "swap" packet increases the bloom filter for the swapped flow by 1. After the swap is performed, we decrease the Transient-States bloom filter by 1 for that specific flow. When a packet fails the lookup in the cuckoo hash, we need to generate a new state through an insertion. But, if the value of Transient-States is greater than zero, the insertion operation is reverted to a lookup since the entry associated to the flow is being swapped from the second to the first array. To give an example, in Fig. 2, the first packet of a blue flow has not matched an entry, which triggers an insertion (realized through an ad-hoc packet) that is currently in progress (by being recirculated). If a second data packet arrives, it will not generate a new state since the Transient-State bloom filter has been set to 1 by the ongoing insertion. Thus, the second packet will be recirculated. When the switch performs the insertion, Transient-States will be decreased by 1 and the second packet will match the inserted entry.

3.4 Guaranteeing Packet Ordering

Consider a scenario in which the switch receives three consecutive packets of the same flow. If the first packet fails the lookup, then it triggers an insertion and it is recirculated. The second packet is processed before the first recirculated packet performs the insertion and it is also recirculated. Now, the first packet re-enters the processing pipeline and performs the insertion. If the third packet arrives before the second packet, the third one would hit a match and be forwarded before the second one.

To guarantee packet ordering, Switcharoo implements a simple idea. The switch starts to sequentially numbering all packets of the same flow that arrive at the switch. The switch then

enforces that packets to leave the switch in the correct order by checking what is the next packet that needs to be sent out to preserve packet ordering.

To implement this mechanism, we rely on three additional array-based data structures (with the same size). The arrays are indexed using a single hash of the flow identifier of a packet. The data structures are Ordering, Packet-Counter, and Next-Packet. Each element entry in Ordering keeps track the number of currently recirculated packets that are hashed to that entry. Each element entry in Packet-Counter counts the number of packets that have matched that entry and is used to assign identifiers to newly incoming packets. Each element entry in Next-Packet stores the identifier of the next packet matching that entry that should leave the switch. The three data structures implement three countable bloom filters with a *single* hash, which means that the switch will preserve the ordering of packets for any set of packets that have a collision on the same entry of the array. Again, we rely on a conservative approach to guarantee that states related to packet ordering are handled correctly (at the cost of enforcing some unnecessary inter-flow ordering guarantees). All the arrays have the same size. We add the aforementioned data structures after the Transient-States bloom filter processing logic.

There are three types of packets that are processed by the ordering algorithm: packets that did not match an entry in the cuckoo hash (and must be recirculated), packets that matched an entry in the cuckoo hash, and recirculated packets. Note that, recirculated packets that are swapping an entry must not be sent out (since these are control packets, not data packets). Hence they simply bypass the ordering algorithm after the transient bloom filter.

(Non-recirculated) packets that did not match an entry. Assume the hash of the (non-recirculated) packet is such that it accesses the data structures at a specific *index*. We omit the index in writing, *e.g.,* we write Transient-States instead of Transient-States [*index*]. When a packet fails a lookup in the cuckoo hash, it passes through the Transient-States. Here, the packet either finds a value (*i*) equal to zero (no insertions or swaps in progress) or (*ii*) greater than zero (indicating an insertion or swap in progress).

(i) If the Transient-States value equals zero, the switch needs to signal to other packets of the same flow that the current packet will recirculate for inserting the state. So, it increments the Transient-States value by one. Moreover, it sets Ordering to 1, it sets Packet-Counter to 1, it assigns Packet-Counter as the identifier of the packet, and sets Next-Packet to 1. The packet is then marked with extra metadata to indicate the need to perform an insertion and the packet is recirculated.

(ii) If the Transient-States value is greater than zero, it means that there is an insertion in progress. The switch needs to recirculate the packet, increase Ordering by one and assign it a packet identifier based on Packet-Counter.

The switch uses the identifier of the packet to preserve packet ordering within a flow. In fact, a packet is sent out only when the identifier equals the entry in Next-Packet.

(Non-recirculated) Packets that matched an entry. When a (non-recirculated) packet successfully performs a lookup, it skips the check on Transient-States and it directly verifies the state of the Ordering value. If the value equals zero, it means that no other packet of the same flow is recirculating, so the packet is sent out. Otherwise, the packet must wait its turn. The switch increments Ordering and assigns it an identifier based on Packet-Counter. Finally, the packet is recirculated.

Recirculated packets. The recirculated packets processed by the ordering data structures can be of two types: packets that performed an insertion and packets that are waiting to be sent out. Both types have the metadata carrying the identifier of the packet in the flow, so they are managed in the same way. The switch reads Next-Packet. If the value of Next-Packet is equal to the one

in the metadata, the packet can be sent out preserving the order. Hence, the switch decrements Ordering and increments Next-Packet (now equal to the counter of the next packet to send out). Otherwise, the packet is recirculated waiting for its turn (skipping the cuckoo-hash).

Example: packet ordering and consistency. We now clarify how packet ordering is guaranteed with an example. Note that only packets that have the same identifier as the same value of Next-Packet can be sent out. This ensures that the order within a flow (or all flows using the same index, to be precise) is preserved. Moreover, to guarantee the consistency among states, our mechanism allows packets to read the state only if their identifier matches the value of Next-Packet. We clarify this with an example. Consider Fig. 3 in which four packets have been processed by the switch as follows. The first packet of a Blue flow has triggered an insertion. When this happened, the switch increased the Transient-States by 1, it set the Packet-Counter value to 1, the packet identifier to 1, and the Next-Packet to 1. When the second packet arrives, it does not match any entry. It checks the Transient-States entry and detects an ongoing insertion. The switch increases Packet-Counter to 2, stores the packet identifier 2 in the packet metadata, and recirculates the packet. The same happens for the third and fourth packet. At this point, Packet-Counter is 4 and Next-Packet is still 1. Assume the insertion is now executed, which decreases Transient-States to 0. This is exactly the state represented in the figure. Now, if the fifth packet arrives, it would perform a successful lookup (the first packet already performed the insertion) reading the state before the second packet. To handle this case, since the Packet-Counter is 2 and not 5, the switch assigns 5 as the new identifier, increases Ordering, and recirculates the new incoming packet disregarding the state that has been read. When the second packet re-enters the pipeline, it matches the new state and since Next-Packet is equal to 2, it is forwarded. Additionally, the switch increments Next-Packet (now equals to 3) and decreases Ordering. The algorithm is repeated until the fifth packet is forwarded when Ordering is reset to zero.

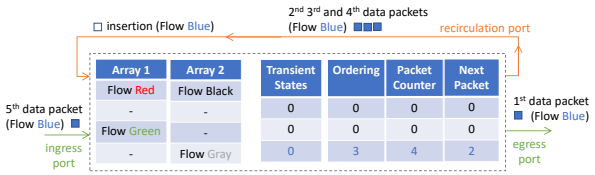


Fig. 3. Guaranteeing packet ordering.

Supporting dynamic updates to the values. Note that, the design of Switcharoo allows states to be updated at runtime, ensuring the consistency. This capability is fundamental to implement a variety of use cases that require to update the states associated to flows during the processing (*e.g.,* NFs based on state machines [28]). For example, a jumbo frame builder needs to update the state every time a packet is merged. Such property is not guaranteed by other systems, *e.g.,* Lucid [28].

3.5 Handling Infinite Forwarding Loops

There are two main reasons that create forwarding loops: swaps of insertions and packet drops. In the case of swaps or insertions, loops would naturally break in Switcharoo since each entry has an associated expiration timeout, which is very small for the NFs that can be supported in the limited memory of an ASIC switch. Moreover, such expiration timeout is only updated by insertion, lookup, or update packets, ensuring that a loop of swapping entries (that cannot update timers) ends when one of the involved entries expires.

We now discuss the case where the forwarding loop is caused by packet drops within the switch. In this scenario, the additional data structures required to handle both transient states and packet ordering may misalign, causing a stall in the forwarding of *any* packet. Consider again the example depicted in Fig. 3. The first packet fails the lookup, and enters the transient state logic to signal that it needs to perform an insertion. The switch increments Transient-States, it sets Packet-Counter

to 1, stores this packet identifier in the packet metadata, and recirculates the packet (other data structures updates are omitted for simplicity). Assume now the first packet is dropped by the switch (*e.g.,* due to congestion). The second packet arrives, it does not match any entry, it checks the Transient-States and detects an ongoing insertion. After assigning a packet identifier (equal to 2), the packet recirculates and fails the lookup again, it reads the Transient-States which is still equal to 1, it skips the ordering logic (since it already has a packet identifier), and it recirculates again. It is clear that the packet will recirculate indefinitely, waiting for the packet with identifier 1 to decrease the Transient-States value. In order to prevent this scenario, we set a configurable threshold on packets recirculating in the Transient-States. If the number of recirculations reaches the threshold, the value of the Transient-States is reset to zero, and the packet is treated as a new insertion.

3.6 Proof of Correctness

In this section, we prove the correctness of the Switcharoo mechanism (all the detailed proofs can be found in Appendix A). First, we propose a set of lemmas that is of general interest for analyzing P4 programs targeting PISA switches. Then, we prove that Switcharoo guarantees three key properties in the absence of packet drops:

- **Ordering**: after processing a packet, its order within a flow is preserved.
- **Consistency**: when a state s for a flow f is created, then all the subsequent packets of f will match s until the expiration-timeout expires.
- **Termination**: considering an input sequence of packets P, all the packets of P that enter the switch are forwarded to the final destination, *i.e.*, no infinite forwarding loops.

We introduce two preliminary lemmas to simplify the modeling of a PISA switch processing.

Lemma 3.1. *Consider a multi-stage switch and a sequence of packets. We can model the packet processing as if the switch processes one packet at a time.*

Lemma 3.2. *A packet processed at the switch can be either routed to an output port or recirculated back to the switch.*

By Lemma 3.1, we can simply analyse the processing of one packet at a time (through all the stages). By Lemma 3.2, a packet p that enters the switch can only be a new packet or a recirculated packet. Considering these properties, we can model the input packets on the switch as either newly incoming packets or recirculated packets.

We now prove that Switcharoo satisfies the three aforementioned properties: ordering, consistency, and termination. Before starting, we introduce some notation. In the next, we call t_1 and t_2 the first and second array of the cuckoo-hash data structure. We denote with h_1 the hash function used to index t_1, and with h_2 the hash function used to index t_2. We use the same notation of § 3.3 and § 3.4 for the transient bloom filter and the ordering data structures. The only difference is that, to account the *single-register-access* constraint of programmable ASICs, we split Transient-States in two distinct arrays, namely Swapping (counts packets that are recirculating for a state swap) and Swapped (counts successful state swaps). Moreover, we denote with (i) h_{ts} the hash function used to index both Swapping and Swapped, and (ii) h_{ord} the hash function used to index Ordering, Packet-Counter and Next-Packet. In the next, when we refer to an operation on these registers (*e.g.,* lookup, insert, update), we consider that the operation is performed at the index computed by the corresponding hash function on the 5-tuple $\mathcal{T}_p$ of the packet. Considering a packet p, we denote by $p.op \in$ {INSERT,LOOKUP,DELETE,WAIT,SWAP} the processing operation associated to it. We denote by f_p the flow identifier of a packet p.

Ordering. We prove that SWITCHAROO algorithm guarantees packet ordering within a flow. We first state in Lemma 3.3 that a packet leaves the switch only when it is its turn, and then use that lemma to prove the main theorem about packet ordering.

LEMMA 3.3. *A recirculated packet p exits the switch iff it has read a state from the tables and $p.idx$ = NEXT-PACKET at $h_{ord}(\mathcal{T}_p)$. Moreover, the value of NEXT-PACKET is incremented only when the packet with $p.idx$ = NEXT-PACKET is sent out.*

THEOREM 3.4 (ORDERING). *SWITCHAROO does not change the order of packets within a flow.*

PROOF. Consider a sequence of packets of a flow $[p^1, \ldots, p^n]$. Suppose that p^n exits the switch before a packet p^{n-i}, with $i > 0$. If p^n exists, it means that p^n reads the identifier of p^n in NEXT-PACKET. However, by Lemma 3.3, NEXT-PACKET is increased to $p^n.idx$ only when $p^{n-1}.idx$ leaves the switch. With a simple induction, this means that also packet p^{n-i} with $i > 0$ must have been forwarded, which proves the theorem by contradiction. □

Consistency. We now prove with the help of the following lemmas that SWITCHAROO guarantees the consistency of the states read by packets within a flow. The first two lemmas guarantee that a packet does not generate and insert a new state if a non-expired state exists in any of the two arrays of the cuckoo hash table.

LEMMA 3.5. *A packet p cannot overwrite an already-inserted non-expired state for f_p on t_1.*

LEMMA 3.6. *A packet p cannot insert a state for f_p in t_1 if a non-expired state for f_p exists in t_2.*

The next lemma guarantees that a packet always leaves the switch after reading the correct state.

LEMMA 3.7. *Consider a state s_f associated to a flow f, stored in t_1 or t_2. A packet $p^n \in f$ matching the state s_f does not disregard it iff the previous packet p^{n-1} had already been sent out.*

The following lemma guarantees that an insertion of a state happens only when there are no non-expired states in the arrays of the cuckoo hash table and that only the next packet that should be forwarded will be responsible to insert this state. After that, we prove the main theorem.

LEMMA 3.8. *A packet p of a flow f can perform an insertion iff there is not an associated state to f in t_1 and t_2, and it is the next packet of f to exit (i.e., $p.idx$ = NEXT-PACKET at $h_{ord}(\mathcal{T}_p)$).*

THEOREM 3.9 (CONSISTENCY). *SWITCHAROO does not create state inconsistencies in the cuckoo tables.*

PROOF. Lemma 3.5 guarantees that a valid state associated to a flow f_p cannot be overwritten by a packet belonging to f_p. Lemma 3.6 guarantees that a flow f_p can have at most one state in the cuckoo-hash tables. Lemma 3.7 ensures that lookup operations are performed respecting the order of the packets within a flow. Finally, Lemma 3.8 ensures that insertions, where needed, are performed only by the next packet to send out within a flow. These four properties ensure that the state read from a packet p^n of a flow f is consistent with the one read by p^{n-1}. □

Termination. We now prove that SWITCHAROO guarantees the termination property. There are only two types of packets that may recirculate. We prove in the following two lemmas that such packets always leave the switch, which prove the main theorem 3.12.

LEMMA 3.10. *A packet p of a flow f can perform an insertion iff there is not an associated state to f in t_1 and t_2, and it is the next packet of f to exit (i.e., $p.idx$ = NEXT-PACKET at $h_{ord}(\mathcal{T}_p)$).*

LEMMA 3.11. *A packet p with $p.op$ = LOOKUP, that failed the lookup on both tables and finds SWAPPING > SWAPPED, continues to recirculate until it reads (inserts) a state from (into) t_1 or t_2, then it always exits the switch.*

THEOREM 3.12 (TERMINATION). *SWITCHAROO ensures that each input packet p exits the switch in a finite amount of time.*

4 IMPLEMENTATION

We implemented SWITCHAROO in P4_16 language, and compiled it on an Intel Tofino 1 ASIC [11]. We publicly release all the P4 code for running SWITCHAROO [5].

On a Tofino 1 switch, our implementation spans two pipes connected in a series, while it would be possible to deploy the entire processing pipeline on a single pipe on next-generation programmable ASICs [1, 10]. The first pipe implements the cuckoo hash with two arrays based on the design described in § 3. To perform hash indexing on the tables, we use the 4-tuple composed of *SrcIP, DstIP, SrcPort, DstPort* (we could support a 5-tuple as well). Each table is implemented using five registers. Three 32-bit registers store the 4-tuple composed of *SrcIP, DstIP, Src+DstPort* (ports are collapsed in a single 32-bit value). Another 16-bit register is the value associated to the key tuple. A 32-bit register is used to store the timestamp of the entry, which is used to implement the deletion policy described in § 3. The second pipe implements the SWAPPING and SWAPPED bloom filters using two 16-bit registers, as described in § 3.6. The data structures responsible for packet ordering use three 16-bit registers. In this Tofino implementation, we only support values of 32 bits. The value size ultimately depends on the specific P4 target used to realize SWITCHAROO.

Consistency guarantees. Due to the data plane resources constraints on Tofino 1, the current implementation does not guarantee Lemma 3.7, which enforces that lookup operations can only be performed if a packet p has $p.idx =$ NEXT-PACKET. We relax this constraint by allowing any packet p with $p.idx \neq$ NEXT-PACKET to perform a lookup and store the state as metadata when recirculated. Hence, our implementation satisfies a subset of the consistency properties that we call *weak consistency*. This choice leads to a simpler implementation that is feasible on a Tofino 1 ASIC. However, the full consistency property could fit within the resources provided by next-generation ASICs [1, 10]. We do not observe any inconsistency in our evaluation, even if we only support weak consistency.

ASIC Resources Usage. Table 1 shows the additional ASIC resources consumed by SWITCHAROO on both pipes when compiling the program with the maximum possible amount of entries in the registers, *i.e.,* 65 K entries.

Resource	Pipe 1	Pipe 2
Stages	12	6
SRAM	18.23%	10.54%
TCAM	0.69%	1.79%
VLIW Instructions	12.50%	12.50%
Exact Match Crossbar	19.34%	9.49%
Ternary Match Crossbar	0.51%	1.08%

Table 1. Required ASIC resources.

5 EVALUATION

In this section, we demonstrate that SWITCHAROO is capable of supporting millions of insertions in the data plane, illustrating the overheads introduced by the system. Both the P4 code and all the evaluation scripts, including documentation for full reproducibility, will be made available.

In the following, we aim to answer the following questions:

Q1: *"How many insertions per second are supported by SWITCHAROO?"*
Q2: *"How does the cuckoo table size impact the recirculation bandwidth?"*
Q3: *"How does the cuckoo table size impact the amount of swaps that require a recirculation?"*
Q4: *"How does the expiration timeout impact the amount of recirculation?"*
Q5: *"How does the ordering structures size impact the recirculation?"*
Q6: *"Does SWITCHAROO impact the end-to-end latency?"*
Q7: *"Is any packet delivered out of order, i.e., not satisfying packet ordering guarantees?"*
Q8: *"How many times are data packets recirculated (i.e., increased latency)?"*
Q9: *"Does the recirculation impact the packets' latency?"*

The evaluated implementation leverages SWITCHAROO to implement Flowlet routing with 50μs expiration timeouts (unless specified). Thus, it has to choose an output port for each incoming flow, and store this state for a certain amount of time. Our testbed is composed of a Tofino switch running SWITCHAROO, directly attached with four links to another Tofino switch that 4× multicasts

the traffic (with 1 KB packet sizes) coming from a traffic generator implemented in FastClick [2]. The testbed is wired with 100 Gbps links. The cuckoo table size is set to 32 K entries while the other data structures are set to 65 K entries, unless otherwise specified. Tests are repeated 10 times.

Traffic workload. In each experiment, we generate three types of synthetic traffic: (*i*) in 1-Pkt Flows, all packets belong to distinct flows, thus each packet results in an insertion, *i.e.,* 12.5 M insertions per second at 100 Gbps, (*ii*) in 2-Pkt Flows, flows consist of two non-consecutive packets within 50 μs, thus on average an insertion every two packets, requiring 6.25 M of insertions per second at 100 Gbps, and (*iii*) in 8-Pkt Flows, flows consist of 8 non-consecutive packets within 50 μs (which is in line with the average number of packets per flow in a cloud datacenter [8]), hence the switch performs 1.25 M of insertions per second at 100 Gbps.

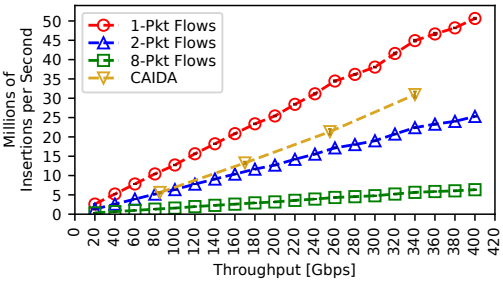


Fig. 4. Insertions per second w.r.t input throughput.

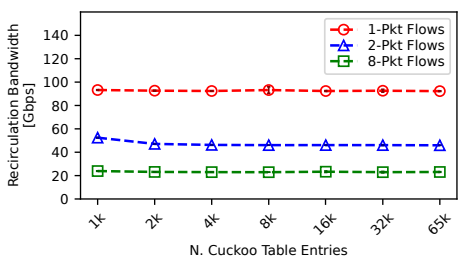
Q1: Million of insertions per second. Fig. 4 shows the number of insertions per second (y-axis) as a function of the input throughput (x-axis). The result shows that insertions scale linearly with respect to the input traffic. Switcharoo effectively performs insertions at line rate, reaching 50 M insertions per second at 400 Gbps in the 1-Pkt Flows case. Comparing this result with solutions that perform insertions from the control plane (in the order of 100 K [34]), the level of achieved performance is 50.000× higher, fundamentally pushing the barrier of what is possible to offload into ASIC devices. Note that we use more than one recirculation port to perform experiments that need more than 100 Gbps of recirculation bandwidth. The number of insertions per second is limited by our testbed, not the implementation. For this experiment, we also consider a real-world CAIDA trace. The original trace achieves a maximum throughput of 4.5 Gbps, which is well below the throughput of our switch. We therefore increase the throughput of the trace to 85 Gbps without altering the flow-size distribution. We then replay the trace from our traffic generator, multicasting it on multiple output ports on the Tofino to further increase the throughput. To avoid increasing the size of each flow when multicasting packets, we re-write the 5-tuple of each flow when we multicast it on a different port. This mechanism increases the throughput and the number of flows in the trace without increasing the size (in packets) of the individual flows. The experiment confirms again the performance of Switcharoo, even on real-world traffic scenarios.[2]

Q2: The cuckoo table size has a limited impact on the recirculation bandwidth. Fig. 5(a) shows the bandwidth of the recirculation ports (y-axis) as a function of the cuckoo table sizes (x-axis). The input traffic is 100 Gbps. The 1-Pkt Flows line requires a recirculation bandwidth equal to the input throughput since each packet recirculates at least once for the insertion, after performing a lookup (which fails). In more realistic scenarios, the recirculation bandwidth is lower and ranges from 20 Gbps (8-Pkt Flows case) to 50 Gbps (2-Pkt Flows case) for all the table sizes.

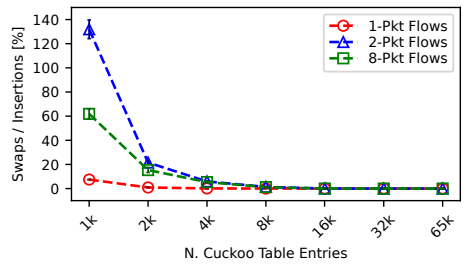
Q3: Limited amount of swaps. Fig. 5(b) shows the percentage of swap operations that require recirculating a packet over the total number of insertions (y-axis) as a function of the cuckoo table sizes (x-axis). The input traffic is 100 Gbps. As expected, by reducing the table size, the percentage increases. Due to the flows composition, the 2-Pkt Flows case has the highest ratio since it presents the highest number of non-expired entries, generating about 1.2 swaps for each insertion. However, by enlarging the table (*e.g.,* 4 K entries), the number of required swaps drops, until reaching a

[2]Given the similarity in results between the CAIDA trace and the 2-Pkt Flows synthetic trace at 100 Gbps, in the next experiments we will only consider the latter to better control input flows.

value of zero, *i.e.,* the cuckoo-hash table is large enough to perform insertions without requiring recirculation.



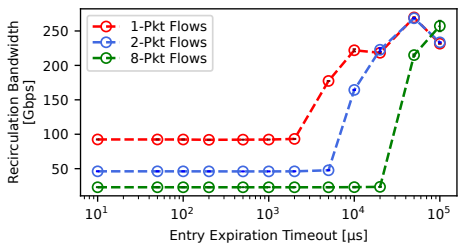
(a) Recirculation bandwidth varying table size.



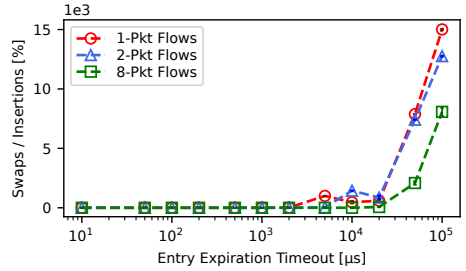
(b) Percentage of swaps that require a recirculation over the total insertions varying table size.

Fig. 5. Recirculation bandwidth and swaps.

Q4: SWITCHAROO supports up to 100 *ms* of entry expiration timeout. We now demonstrate that SWITCHAROO is able to support a wide range of entry expiration timeouts without incurring in additional overheads. Fig. 6(a) shows the bandwidth of the recirculation ports (y-axis) with a varying expiration timeout (x-axis) that ranges from 10 μs up to 100 *ms*. The input traffic is 100 Gbps. Up to 2 *ms*, all the cases are able to sustain the input traffic without additional recirculation bandwidth overheads. Starting from 5 *ms*, 1-PKT FLOWS and 2-PKT FLOWS cases start to require additional bandwidth, until they begin to drop packets at 100 *ms*. The 8-PKT FLOWS case reaches 20 *ms* without additional requirements, being also able to sustain up to 100 *ms* without packet drops. The trend is confirmed in Fig. 6(b), where we plot the percentage of swap operations that require recirculating a packet over the total number of insertions.



(a) Recirculation bandwidth varying the entry expiration timeout.

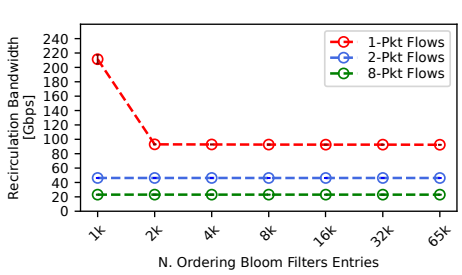


(b) Percentage of swaps that require a recirculation over the total insertions varying the entry expiration timeout.

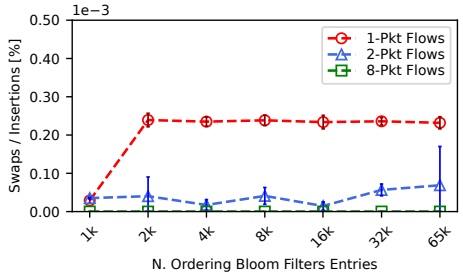
Fig. 6. Recirculation bandwidth and swaps varying the entry expiration timeout.

Q5: Smaller ordering structures can still guarantee limited recirculations. Fig. 7(a) shows the bandwidth of the recirculation ports (y-axis) as a function of the ordering structures size (x-axis). The input traffic is 100 Gbps. The results are in line with the ones depicted in Fig. 5(a), with the 2-PKT FLOWS and 8-PKT FLOWS cases having a recirculation bandwidth of 20 Gbps and 50 Gbps for all the sizes, respectively. It is worth noticing that the 1-PKT FLOWS case has a higher overall recirculation bandwidth with 1 K entries. In this case, the data structures cannot handle the total amount of flows, which collide very frequently on the same entries, introducing unnecessary inter-flow ordering that causes a dramatic increase in the number of recirculated packets. However, results with bigger data structures guarantee the same level of performance depicted in Fig. 5(a). This finding is further

confirmed in Fig. 7(b), where we plot the percentage of swap operations that require recirculating a packet over the total number of insertions, varying the ordering structures size. In the 1-Pkt Flows case, 1 K entries are not enough to handle the 100 Gbps input traffic, leading to packet drops. However, bigger data structures achieve the same results shown in Fig. 5(b).



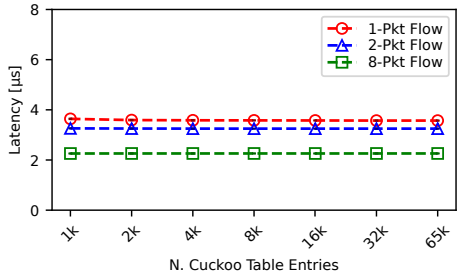
(a) Recirculation bandwidth varying ordering structures size.



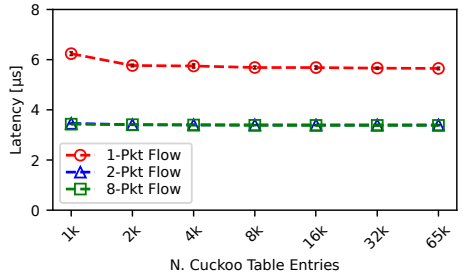
(b) Percentage of swaps that require a recirculation over the total insertions varying ordering structures size.

Fig. 7. Recirculation bandwidth and swaps varying ordering structures size.

Q6: Small impact on the end-to-end latency. Fig. 8 shows the end-to-end latency (y-axis) as a function of the cuckoo tables size (x-axis). The input throughput is 90 Gbps. For each traffic type, we compute the median and the tail (99'th perc.) latency depicted in Fig. 8(a) and Fig. 8(b), respectively. Analyzing the figure, we notice that the curves are all below 10μs. This result demonstrates that, even considering recirculations, Switcharoo has a small impact on the end-to-end latency.



(a) Median latency.



(b) 99-percentile tail latency.

Fig. 8. End-to-end latency varying table size.

Q7: Switcharoo is effective in guaranteeing packet ordering. To verify that Switcharoo guarantees packet ordering in the absence of drops, we measure in Fig. 9 the percentage of out-of-order packets over the total number of packets (y-axis) for different cuckoo table sizes (x-axis) with an input traffic of 100 Gbps. We run Switcharoo with and without the ordering data structures, focusing on the 8-Pkt Flows case, since it is the only traffic pattern containing enough packets that could be transmitted out of order while performing an insertion. The empirical results demonstrate that Switcharoo

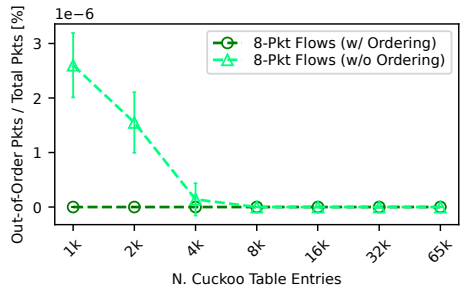
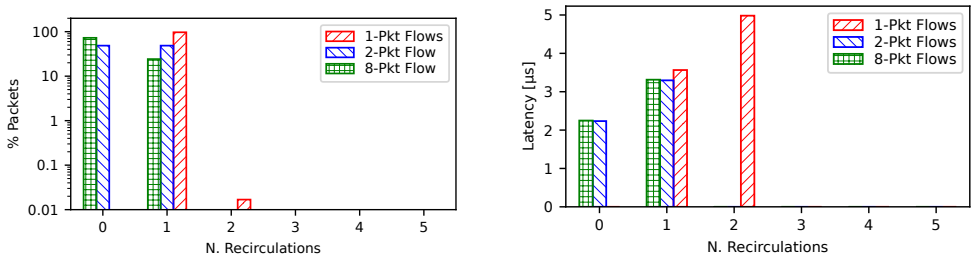


Fig. 9. Out-of-order packets w.r.t table size.

(with the ordering data structures) effectively guarantees packet ordering, confirming the theoretical results of § 3.6. It is worth noticing that even if the percentage of out-of-order packets is negligible, and decreases to zero when the table is large enough, it depends on the workload type and could be higher with different traffic patterns and different expiration timeouts.

Q8: SwitchAroo supports low-latency insertions by limiting packet recirculation. Fig. 10(a) shows the percentage of packets (y-axis) that recirculate a specific amount of times (x-axis). The input traffic in 100 Gbps. Analyzing the 1-Pkt Flows case (red bar) we see that almost all the packets recirculate one time, with a negligible percentage that recirculate two times due to collisions of different flows in the ordering data structures, which force some packets to recirculate waiting for their turn. Instead, considering 2-Pkt Flows and 8-Pkt Flows cases (blue and green bars, respectively), we observe that more than 50% of packets perform a successful lookup, exiting the switch without recirculating, while the remaining part recirculates only once.



(a) Percentage of packets that recirculated a certain number of times.

(b) Median end-to-end latency of packets that recirculated a certain number of times.

Fig. 10. Recirculated packets and end-to-end latency.

Q9: Recirculating packets adds a reasonable latency overhead. To understand the impact of the recirculation induced by swapping entries from the second to the first table, we measure the median end-to-end latency of packets that recirculate a certain number of times. The input traffic in 100 Gbps. Fig. 10(b) highlights that each recirculation adds about 1.5 μs to the overall latency. Moreover, consider that the tested SwitchAroo implementation is deployed on two pipes of the Tofino 1 ASIC, meaning that some extra overhead is introduced by internal recirculations to pass packets between the pipes. We expect that a single-pipe implementation would introduce a much lower latency overhead.

6 RELATED WORK

To the best of our knowledge, we are the first to design a key-value mechanism that is entirely implemented using programmable ASICs data-plane primitives and that theoretically guarantees packet ordering and state consistency. This section summarizes some related works.

Stateful processing in the data plane. We focus on techniques for stateful processing on hardware devices and omit both work related to general-purpose CPUs and work that have already been discussed in § 2. TEA extends the memory of a switch but still relies on insertions from a CPU through a stash data structure, incurring high latency and low frequency. Cheetah [3] is a stateful load balancer that performs both lookups and insertions in the data plane of a programmable switch. To achieve this, the authors rely on cookies that are sent at the connection establishment to the endpoints (containing the index where to fetch the state in an array). However, this requires to trick the TCP timestamp option, which is a hack, and it is not supported by Windows machines [3].

Moreover, state can only be inserted on the first packet of a connection, meaning Flowlet and other NFs cannot be supported. Conversely, we do not want any involvement from the endpoints (because of security issues and lack of compatibility) and we want to insert state also outside connection establishment (as in the case of packet schedulers). Both these requirements cannot fundamentally be supported in Cheetah. Tiara [34] is a load balancer that extends an ASIC switch with eight FPGAs and CPUs to support high-throughput and high-volume insertions. Conversely, we focus only on ASIC (programmable) switches, which are significantly cheaper (8 high-end FPGAs cheaper, which we estimate is in the 50K USD range) and more energy-efficient than FPGAs.

Cuckoo hashing in the data plane. Several works have tackled the problem of implementing cuckoo-hash tables in the data plane. FlowBlaze [24] is a system for building high-speed stateful NFs using FPGAs. FlowBlaze leverages on a cuckoo-hash table to store flow states, with consistency and ordering guarantees (similarly to Switcharoo). However, our ASIC switch mechanism is different from FlowBlaze due to different hardware constraints. Moreover, FPGAs are more expensive and less energy-efficient than ASICs. Some work envisioned the possibility to realize cuckoo-hash tables in ASIC (*e.g.,* [14]). However, these works do not implement and test a real-world implementation and, to the best of our knowledge, currently high-speed datacenter ASIC switches do not support key-value structures as a primitive in their chips. Lucid [28] proposes a high-level abstraction language for programming packet-processing pipelines of programmable ASIC switches. In the evaluation, authors demonstrate the feasibility of implementing a stateful firewall using a cuckoo-hash table. However, (*i*) Lucid cannot guarantee consistency when inserting or swapping elements as it creates a copy of the state that is stored in a stash and not synchronized back with the recirculated state (see line 150 of [29]). Lucid only guarantees consistency when inserting the state on the first packet of a flow (as in the simple firewall that they evaluate). This limitation implies that with Lucid it is not possible to build anything relying on a packet counter or a jumbo frame builder (which need to update the state associated to a flow at every packet). Dart [27] is a data-plane monitoring system for Round-Trip-Times (RTTs) that leverages a data structure similar to a cuckoo-hash table to store states. Dart performs state modifications on copies of original packets, thus not requiring packet ordering. Also, there are no guarantees about state consistency during insertions. Moreover, there is no evaluation showing millions of state insertions per second. Finally, the data structures in Dart are specific to RTT monitoring, while we propose a general-purpose cuckoo-hash table.

7 CONCLUSION

In this work, we show that is fundamentally possible to support hundreds of millions of μs-latency insertions per second on a programmable switch by realizing the entire logic with data plane available primitives. We guarantee that the insertions preserve packet ordering and state consistency. The level of achieved performance is *orders of magnitude higher* than existing solutions based on slow insertion into ASIC switches from a CPU-based control plane, fundamentally pushing the barrier of what is possible today to offload into ASIC devices. We plan to further investigate the possibility opened by FPGA-equipped ASIC switches and stash-based approaches.

ACKNOWLEDGEMENTS

We would like to thank our shepherd, the anonymous reviewers for their insightful comments and suggestions on this paper. This work has been partially supported by the Swedish Research Council (agreement No. 2021-04212) and KTH Digital Futures. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 770889).

REFERENCES

[1] Anurag Agrawal and Changhoon Kim. 2020. Intel Tofino2 – A 12.9Tbps P4-Programmable Ethernet Switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)*. 1–32. https://doi.org/10.1109/HCS49909.2020.9220636

[2] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast Userspace Packet Processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (Oakland, California, USA) *(ANCS '15)*. IEEE Computer Society, Washington, DC, USA, 5–16. https://doi.org/10.1109/ANCS.2015.7110116

[3] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. 2020. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 667–683. https://www.usenix.org/conference/nsdi20/presentation/barbette

[4] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *ACM SIGCOMM 2013* (acm sigcomm 2013 ed.). ACM. https://www.microsoft.com/en-us/research/publication/forwarding-metamorphosis-fast-programmable-match-action-processing-in-hardware-for-sdn/

[5] Tommaso Caiazzi, Mariano Scazzariello, and Marco Chiesa. 2023. Switcharoo GitHub Repository. https://github.com/orgs/Switcharoo-P4/repositories.

[6] CAIDA . 2019. Trace Statistics for CAIDA Passive OC48 and OC192 Traces. https://www.caida.org/catalog/datasets/trace_stats/.

[7] Philippe Flajolet, Patricio Poblete, and Alfredo Viola. 1997. *On the Analysis of Linear Probing Hashing*. Research Report RR-3265. INRIA. https://hal.inria.fr/inria-00073424

[8] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. 2022. Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 807–827. https://www.usenix.org/conference/nsdi22/presentation/ghasemirahni

[9] Intel. 2021. $P4_{16}$ Intel Tofino Native Architecture – Public Version. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf.

[10] Intel. 2023. Intel Tofino 2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[11] Intel. 2023. Intel Tofino Series. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[12] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. 2020. TEA: Enabling State-Intensive Network Functions on Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 90–106. https://doi.org/10.1145/3387514.3405855

[13] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 514–528. https://doi.org/10.1145/3387514.3406591

[14] Gil Levy, Salvatore Pontarelli, and Pedro Reviriego. 2017. Flexible Packet Matching with Single Double Cuckoo Hash. *IEEE Communications Magazine* 55, 6 (2017), 212–217. https://doi.org/10.1109/MCOM.2017.1700132

[15] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. 2020. Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 387–406. https://www.usenix.org/conference/osdi20/presentation/li-jialin

[16] Kurt Mehlhorn and Peter Sanders. 2008. *Algorithms and Data Structures: The Basic Toolbox* (1 ed.). Springer Publishing Company, Incorporated.

[17] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/3098822.3098824

[18] Pat Morin. 2013. *Open Data Structures: An Introduction.* Athabasca University Press., CAN.

[19] Felicián Németh, Marco Chiesa, and Gábor Rétvári. 2019. Normal Forms for Match-Action Programs. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (Orlando, Florida) *(CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 44–50. https://doi.org/10.1145/3359989.3365417

[20] NSLab @ KTH. 2021. The Cheetah Load Balancer. https://github.com/cheetahlb/.
[21] nsnam. 2023. ns-3. https://www.nsnam.org/.
[22] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo Hashing. In *Algorithms — ESA 2001*, Friedhelm Meyer auf der Heide (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 121–133.
[23] Ivan Pepelnjak. 2022. Data Center Switching ASICs Tradeoffs. https://blog.ipspace.net/2022/06/data-center-switching-asic-tradeoffs.html.
[24] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. 2019. FlowBlaze: Stateful Packet Processing in Hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 531–548. https://www.usenix.org/conference/nsdi19/presentation/pontarelli
[25] Amedeo Sapio, Ibrahim Abdelaziz, Marco Canini, and Panos Kalnis. 2017. DAIET: A System for Data Aggregation inside the Network. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 626. https://doi.org/10.1145/3127479.3132018
[26] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 785–808. https://www.usenix.org/conference/nsdi21/presentation/sapio
[27] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. 2022. Continuous In-Network Round-Trip Time Monitoring. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 473–485. https://doi.org/10.1145/3544216.3544222
[28] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. 2021. Lucid: A Language for Control in the Data Plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 731–747. https://doi.org/10.1145/3452296.3472903
[29] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. 2023. Lucid - Stateful Firewall (Github). https://github.com/PrincetonUniversity/lucid/blob/main/examples/publications/sigcomm21/orig/stateful_fw.dpt#L150.
[30] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. 2017. Let It Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 407–420. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/vanini
[31] Shie-Yuan Wang, Jun-Yi Li, and Yi-Bing Lin. 2020. Aggregating and disaggregating packets with various sizes of payload in P4 switches at 100Gbps line rate. *Journal of Network and Computer Applications* 165 (2020), 102676. https://doi.org/10.1016/j.jnca.2020.102676
[32] Bob Wheeler. 2019. Tomahawk 4 switch first to 25.6 Tbps. https://docs.broadcom.com/doc/12398014.
[33] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-Wide Measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 561–575. https://doi.org/10.1145/3230543.3230544
[34] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei Geng, Tao Feng, Feng Ning, Kai Chen, and Chuanxiong Guo. 2022. Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/nsdi22/presentation/zeng
[35] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. 2017. Resilient Datacenter Load Balancing in the Wild. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 253–266. https://doi.org/10.1145/3098822.3098841
[36] Junxue Zhang. 2023. ns3-load-balance. https://github.com/snowzjx/ns3-load-balance.

A PROOF OF CORRECTNESS

This appendix contains the proofs of all the lemmas stated in § 3.6.

Lemma 3.1. *Consider a multi-stage switch and a sequence of packets. We can model the packet processing as if the switch processes one packet at a time.*

Proof. The PISA architecture is a feed-forward architecture, hence it enforces each packet to traverse all the stages of the pipeline. This ensures that, even if many packets are being processed

simultaneously by the switch, they are processed sequentially, and that a packet p^n, that enters the switch right after p^{n-1}, finds all the changes applied by p^{n-1}. □

Lemma 3.2. *A packet processed at the switch can be either routed to an output port or recirculated back to the switch.*

Proof. When a packet p is processed, sw must assign an egress port to p, although p is dropped. If the assigned port is a recirculation port, the packet re-enters the switch and it is processed again (as a normal packet). The packet exits the switch otherwise. □

Ordering. We demonstrate the lemmas regarding packet ordering guarantees within a flow.

Lemma A.1. *A packet $p \in Q_n$ exits sw without recirculating iff p performs a successful lookup on one of the tables, and* Ordering $= 0$ *at $h_{ord}(\mathcal{T}_p)$.*

Proof. When a packet $p \in Q_n$ of a flow f enters sw, it performs a lookup on t_1 and t_2. (i) If the lookup fails, p reads the value of Ordering. If the value is 0, sw sets Next-Packet $\leftarrow 0$, Packet-Counter $\leftarrow 1$ and $p.idx \leftarrow 0$. Otherwise, sw assigns $p.idx \leftarrow$ Packet-Counter and increments Packet-Counter. p is recirculated with $p.op \leftarrow$ INSERT (if Transient-States = Swapped) or with $p.op \leftarrow$ LOOKUP (if Transient-States > Swapped). (ii) If the lookup succeeds, p checks the value of Ordering. A value greater than 0 implies that some packets of f are recirculating through sw, so sw increments the value of Ordering, sets $p.idx \leftarrow$ Packet-Counter and increments Packet-Counter. p is recirculated with $p.op \leftarrow$ LOOKUP. A value equals to 0 implies that no other packet of f is recirculating, so p can exit preserving the ordering. □

Lemma 3.3. *A recirculated packet p exits the switch iff it has read a state from the tables and $p.idx =$ Next-Packet at $h_{ord}(\mathcal{T}_p)$. Moreover, the value of Next-Packet is incremented only when the packet with $p.idx =$ Next-Packet is sent out.*

Proof. A packet $p \in Q_r$ of a flow f that enters sw can have four different values of $p.op$. (i) A packet with $p.op$ = INSERT performs an insertion on t_1 and reads Next-Packet. If Next-Packet $= p.idx$, sw increments Next-Packet and decrements Ordering. p exits sw. Otherwise, p is recirculated with $p.op \leftarrow$ WAIT. This behaviour does not change even if p causes a swap. (ii) A packet with $p.op$ = LOOKUP performs a lookup on t_1 and t_2. If the lookup fails, p is recirculated with $p.op \leftarrow$ INSERT (if Transient-States = Swapped) or with $p.op \leftarrow$ LOOKUP (if Swapping > Swapped). If the lookup succeeds, p checks the value of Next-Packet. If Next-Packet $= p.idx$, sw increments the value of Next-Packet and p exits sw. Else p is recirculated with $p.op \leftarrow$ WAIT. (iii) A packet in the SWAP state cannot exit sw. (iv) A packet with $p.op$ = WAIT has already taken a state and only checks the value of Next-Packet. If Next-Packet $= p.idx$, sw increments the value of Next-Packet and p exits sw. Else p is recirculated with $p.op \leftarrow$ WAIT. □

Consistency. We now demonstrate the lemmas that guarantee the consistency of the states read by packets within a flow.

Lemma A.2. *A value of Swapping for a flow f greater than the value of Swapped for the same flow f implies that an entry for f is being swapped from t_2 to t_1.*

Proof. The values of Swapping and Swapped are updated only by packets that carry an entry to swap from t_2 to t_1. The Swapping entries are increased by 1 when a packet passes through the bloom filter before the swap. The values of Swapped are increased by 1 when it passes through the

bloom filter after the swap. This implies that if SWAPPING > SWAPPED for a flow f, there is a swap packet that has to be processed. □

LEMMA 3.5. *A packet p cannot overwrite an already-inserted non-expired state for f_p on t_1.*

PROOF. Consider a flow f_p and a state s_{f_p} associated to f_p and inserted into t_1. Suppose that a packet p of f_p enters sw with $p.op$ = INSERT state. At this point, sw computes a state for p and overwrites the state s_{f_p} in t_1. To have $p.op$ = INSERT, p in the previous processing must have been a new packet or a LOOKUP packet. In addition, p must have failed the lookup in the hash tables and must have read SWAPPING = SWAPPED. Since s_{f_p} is a state in t_1, this implies that p failed the lookup because s_{f_p} was recirculating while p is processed. But, if s_{f_p} was still recirculating, for Lemma A.2 the value of SWAPPING must have been greater than the one of SWAPPED, which is a contradiction. □

LEMMA 3.6. *A packet p cannot insert a state for f_p in t_1 if a non-expired state for f_p exists in t_2.*

PROOF. Consider a flow f_p and a state s_{f_p} associated to f_p and inserted into t_2. Suppose that a packet p of f_p enters sw in the INSERT state. At this point, sw computes a state $s^1_{f_p}$ for p and writes it in t_1. To be in the INSERT state, p in the previous processing must have been a new packet or a packet in the LOOKUP state. In addition, p must have failed the lookup in the hash tables and must have read SWAPPING = SWAPPED. Since s_{f_p} is a state in t_2, this implies that p failed the lookup because s_{f_p} was recirculating while the processing of p. But, if s_{f_p} was still recirculating, for Lemma A.2 the value of SWAPPING must have been greater than the one of SWAPPED, which is a contradiction. □

LEMMA 3.7. *Consider a state s_f associated to a flow f, stored in t_1 or t_2. A packet $p^n \in f$ matching the state s_f does not disregard it iff the previous packet p^{n-1} had already been sent out.*

To guarantee Lemma 3.7, we introduce two additional arrays to store the index of the last packet that accessed an entry, namely LAST-LOOKUP-T1 and LAST-LOOKUP-T2. Such arrays are accessed only by packets that performed a successful lookup, and they are conceptually placed after the cuckoo-hash tables and before the auxiliary data structures.

PROOF. When a packet $p \in Q_r$ of a flow f with $p.op$ = LOOKUP performs a successful lookup on t_1, it reads the value of LAST-LOOKUP-T1. If LAST-LOOKUP-T1 $\neq p.idx - 1$, p performs a lookup on t_2. If LAST-LOOKUP-T1 $= p.idx - 1$, p increments LAST-LOOKUP-T1 and reads the state from t_1. Then it follows the standard processing, exiting from sw when $p.idx$ = NEXT-PACKET. When a packet $p \in Q_r$ of a flow f with $p.op$ = LOOKUP performs a successful lookup on t_2, it reads the value of LAST-LOOKUP-T2. If LAST-LOOKUP-T2 $\neq p.idx - 1$, p is recirculated with $p.op \leftarrow$ LOOKUP (not its turn). If LAST-LOOKUP-T2 $= p.idx - 1$, p increments LAST-LOOKUP-T2 and reads the state from t_2. Then it follows the standard processing, exiting from sw when $p.idx$ = NEXT-PACKET. When a packet $p \in Q_n$ of a flow f performs a successful lookup it reads the value of ORDERING. If ORDERING = 0, p exits sw. Else, a value of ORDERING > 0 implies that other packets of f are recirculating. To ensure that p respects the lookup ordering, sw assign NEXT-PACKET $\leftarrow p.idx$ and recirculates p with $p.op \leftarrow$ LOOKUP. In this way, p re-performs the lookup considering the value of LAST-LOOKUP-T1 and LAST-LOOKUP-T2. □

LEMMA 3.8. *A packet p of a flow f can perform an insertion iff there is not an associated state to f in t_1 and t_2, and it is the next packet of f to exit (i.e., $p.idx$ = NEXT-PACKET at $h_{ord}(\mathcal{T}_p)$).*

Proof. Suppose that a packet p^n with $p^n.op = INSERT$ of a flow f enters sw while a packet p^{n-1} of f is recirculating with $p^{n-1}.op = LOOKUP$. At this point, p^n performs an insertion on t_1 before p^{n-1}. To enter sw with $p.op = INSERT$, p^n must have failed a lookup in the previous processing.

If in the previous processing $p^n \in Q_r$ three scenarios are possible. (i) p^n fails the lookup and finds Swapping > Swapped, so p^n recirculates with $p^n.op \leftarrow$ LOOKUP waiting for the swap. (ii) p^n fails the lookup and finds Swapping = Swapped & Next-Packet $\neq p^n.idx$, so it is recirculated with $p^n.op \leftarrow$ LOOKUP since it is not the next packet that should exit the switch. (iii) p^n fails the lookup and finds Swapping = Swapped & Next-Packet = $p^n.idx$, so it is recirculated with $p^n.op \leftarrow$ INSERT since it is the next packet that should exit the switch and the one that has to perform the insertion. But, if p^{n-1} is still recirculating, Next-Packet $\leq p^{n-1}.idx$. (Absurd)

If in the previous processing $p^n \in Q_n$ three scenarios are possible. (i) p^n fails the lookup and sw assigns the current Packet-Counter to p^n and updates the value of Packet-Counter and Ordering by 1. p^n is recirculated to Q_r with $p^n.op \leftarrow$ LOOKUP. (ii) p^n fails the lookup and finds Swapping > Swapped, so it is recirculated with $p^n.op \leftarrow$ LOOKUP waiting for the swap. (iii) p^n fails the lookup and finds Swapping = Swapped & Ordering = 0, so, since p^n is the only packet of the flow in sw, it is recirculated to Q_r with $p^n.op \leftarrow$ INSERT. However, if p^{n-1} is still recirculating, Ordering > 0. (Absurd) □

Termination. We demonstrate the lemmas that guarantee the termination property.

Lemma 3.10. *A packet p of a flow f can perform an insertion iff there is not an associated state to f in t_1 and t_2, and it is the next packet of f to exit (i.e., $p.idx$ = Next-Packet at $h_{ord}(\mathcal{T}_p)$).*

Proof. Consider a packet p^i of a flow f with $p.op$ = WAIT. p^i is recirculating in sw since Next-Packet $\neq p^i.idx$. If so, each packet p^{i-k}, with $p^{i-k}.idx \geq$ Next-Packet & $1 \leq k \leq i$, is also recirculating. Suppose that $k = i$, this implies that p^0, the first packet of f, is recirculating. Hence, Next-Packet = 0 by construction. Consequently, p^0 will exit sw the next time it is processed, and it will also increase Next-Packet by 1, causing the exiting of p^1. This process is repeated until p^i exits from sw. □

Lemma 3.11. *A packet p with $p.op = LOOKUP$, that failed the lookup on both tables and finds Swapping > Swapped, continues to recirculate until it reads (inserts) a state from (into) t_1 or t_2, then it always exits the switch.*

Proof. Consider that a packet p of a flow f_p fails a lookup and finds Swapping > Swapped. After re-entering sw in Q_r with $p.op$ = LOOKUP, four conditions are possible: (i) the lookup succeeds and p is recirculated with $p.op \leftarrow$ WAIT. For Lemma 3.10, p exits sw; (ii) p fails the lookup and Swapping = Swapped. p recirculates with $p.op \leftarrow$ INSERT, re-enters sw from Q_r, performs an insertion on t_1 and recirculates with $p.op \leftarrow$ WAIT. For Lemma 3.10, p exits sw; (iii) p fails a lookup and finds Swapping > Swapped re-entering the initial state. If so, the entry e_{f_p} associated to f_p is being swapped while p is performing the lookup. This condition can repeat until the swap of e_{f_p} succeeds, leading to the second case, or until e_{f_p} expires and is dropped. This implies that e_{f_p} does not increase Swapping and hence p will read Swapping = Swapped, leading to the third case; (iv) the lookup fails since p finds a value of the Last-Lookup entry different from $p.idx - 1$, so p is recirculated with $p.op \leftarrow$ LOOKUP waiting for its turn. □

Received July 2023; accepted October 2023